O'REILLY®

# Async Rust

Unleashing the Power of Fearless Concurrency

Early Release

RAW & UNEDITED

Maxwell Flitton
& Caroline Morton

# Async Rust

Unleashing the Power of Fearless Concurrency

Maxwell Flitton and Caroline Morton

Beijing • Boston • Farnham • Sebastopol • Tokyo

# Async Rust

by Maxwell Flitton and Caroline Morton

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

- Acquisitions Editor: Brian Guerin
- Development Editor: Melissa Potter
- Production Editor: Jonathon Owen
- Interior Designer: David Futato
- Cover Designer: Karen Montgomery
- Illustrator: Kate Dullea

- December 2024: First Edition

## Revision History for the Early Release

- 2023-12-15: First Release
- 2024-02-07: Second Release
- 2024-03-14: Third Release
- 2024-04-22: Fourth Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098149093 for release details.

978-1-098-14903-1

# Chapter 1. Introduction to Async

**A NOTE FOR EARLY RELEASE READERS**

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mpotter@oreilly.com.

For years software engineers have been spoiled with the relentless increase in power of hardware. Phrases like "just chuck more computing power at it", or "write time is more expensive than read time" have become popular one-liners when justifying using a slow algorithm, rushed approach, or slow programming language. At the time of writing this book, multiple microprocessor manufactures have reported that the semiconductor advancement has slowed since 2010 leading to the controversial statement from Nvidia CEO Jensen Huang in 2022 stating that "Moore's law is dead". With the increased demand on software and increasing number of IO network calls with systems such as microservices, there is a need for being more efficient with the resources we have. This is where async programming comes in. With async programming, we do not need to add another core to the CPU to get performance gains. Instead, with async we can effectively juggle multiple tasks on a single thread if there is some deadtime in those tasks such as waiting for a response from a server.

We live our lives in an async way. For instance, when we put the laundry into the washing machine, we do not sit still doing nothing until the machine has finished. Instead, we do other things. If we want our computer and programs to live an efficient life, we need to embrace async programming. However, before we roll up our sleeves and dive into the weeds of async programming we need to understand where async programming sits in the context of our computer. This chapter provides an overview of how threads and processes work demonstrating the effectiveness of async programming in I/O operations and why/where we would apply them to async programming.

After reading this chapter you should be able to understand what async programming is at a high level without knowing the intricate details of an async program. You will also understand some basic concepts around threads and Rust as these concepts will pop up in async programming due to async runtimes using threads to execute async tasks. You should be ready to explore the details of how async programs work in the following chapter. If you are familiar with processes, threads and sharing data between them, feel free to skip this chapter. In the next chapter, we cover async specific concepts like futures, tasks, and how an async runtime executes tasks.

## What is Async?

When we use a computer we expect our computer to perform multiple tasks at the same time. It would be a pretty bad experience otherwise. However, think about all the tasks that our computer does at one time. At the time of writing this book, we've clicked onto the activity monitor of a laptop. The laptop at one point was running 3,118 threads and 453 processes, while only using 7% of the CPU. The number of processes and threads is down to multiple applications, browser tabs, and background processes. So, how does the laptop keep all these threads and processes running at the same time? Here's the thing, the computer is not running all 3,118 threads and 453 processes at the same time. The computer needs to schedule resources.

To demonstrate the need for scheduling resources, we can run some computationally expensive code to see how the activity monitor changes. To stress our CPU, we employ a recursion calculation like the fibonacci number calculation below:

```rust
fn fibonacci(n: u64) -> u64 {
    if n == 0 || n == 1 {
        return n;
    }
```

```
        fibonacci(n-1) + fibonacci(n-2)
    }
```

We can then spawn eight threads and calculate the 4000th number with the following code:

```
use std::thread;


fn main() {
    let mut threads = Vec::new();


    for i in 0..8 {
        let handle = thread::spawn(move || {
            let result = fibonacci(4000);
            println!("Thread {} result: {}", i,
        });
        threads.push(handle);
    }
    for handle in threads {
        handle.join().unwrap();
    }
}
```

If we then run this code, we will see that our even though the number of threads and processes do not even come close to doubling, but our CPU usage jumps to 99.95%. Considering that the number of processes and threads did not double yet the CPU usage jumps from 7% to 99.95%, we can deduce that most of these processes and threads are not all using CPU resources all of the time.

There’s multiple nuances to modern CPU design. What we need to know is that a portion of CPU time is allocated when a thread or process is created. Our task in the created thread or process is then scheduled to run on one of the CPU cores. The process or thread runs until it is interrupted or it is yielded by the CPU voluntarily. Once the interpretation has occurred, the CPU saves the state of the process or thread, and then the CPU switches to another process or thread.

Now that we understand at a high level how the CPU interacts with processes and threads, let us see basic asynchronous code in action. The specifics of the asynchronous code will be covered in the following chapter, so right now, it’s not important to understand exactly how every line of code works, but appreciate what asynchronous code is giving us in terms of utilisation of CPU resources. First we need the following dependencies:

```
[dependencies]
reqwest = "0.11.14"
tokio = { version = "1.26.0", features = ["full"
```

Tokio is giving us a high level abstraction of an async runtime, and reqwest enables us to make async HTTP requests. HTTP requests are a good, simple real world example of utilising async because of the latency through the network when making a request to a server. The CPU doesn’t need to do anything when waiting on a network response. We can time how long it takes to make a simple HTTP request using Tokio as the async runtime with the code below:

```
use std::time::Instant;
use reqwest::Error;


#[tokio::main]
async fn main() -> Result<(), Error> {
    let url = "https://jsonplaceholder.typicode.c
    let start_time = Instant::now();

    let _ = reqwest::get(url).await?;


    let elapsed_time = start_time.elapsed();
    println!("Request took {} ms", elapsed_time.a

    Ok(())
}
```

Your time may vary but at the time of writing this book it took roughly 140ms to make the request. We can increase the number of requests by merely copying and pasting the request another three times like so:

```
let _ = reqwest::get(url).await?;
let _ = reqwest::get(url).await?;
let _ = reqwest::get(url).await?;
let _ = reqwest::get(url).await?;
```

Running our program again gave us 656ms. This makes sense as we have increased the number of requests by four. If our time was less than 140x4 then it would not make sense, as the increase in total time would not be proportional to increasing the number of requests by four. It must be noted there that although we are using async syntax we have essentially just written synchronous code. This means we are executing each request after the previous one has finished. To make our code truly asynchronous, we can join the tasks together and have them running at the same time with the code below:

```
let (_, _, _, _) = tokio::join!(
    reqwest::get(url),
    reqwest::get(url),
    reqwest::get(url),
    reqwest::get(url),
);
```

Running our code again gives us a duration time of 137ms. That’s a 4.7 times increase in the speed of our program without increasing the number of threads! This is essentially async programming. Using async programming, we can free up CPU resources by not blocking the CPU of tasks that can be waited on as seen in figure 1-1.

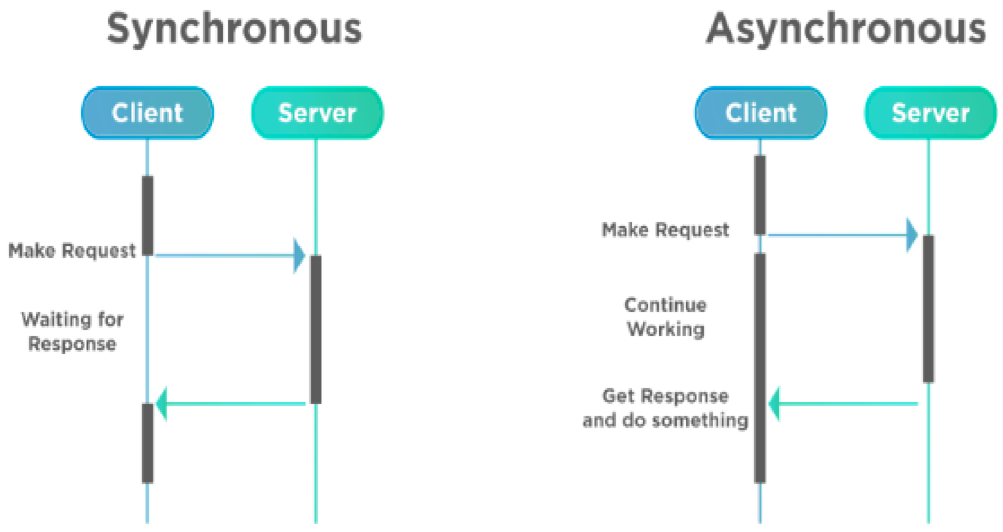


Figure 1-1. Blocking synchronous timeline compared to asynchronous timeline

However, it must be noted that asynchronous programming is not just juggling multiple tasks at the same time. If we were to use Tokio to join four Fibonacci computations we would not get an increase in speed as we are only using one thread. If we were going to run our Fibonacci function over multiple processes or threads at the same time, we would call it parallelism, because multiple computations are performed simultaneously. To understand the context around async programming, we need to briefly explore how processes and threads work. Whilst we will not be using processes in asynchronous programming, it is important to understand how processes work and communicate between each other to give us context of threads and asynchronous programming.

## Introduction to Processes

So far we have referenced threads and processes without drilling down on the specifics on what these are. Let's start with *processes*. A process is a program or application that is executed by the CPU. The instructions of the program are loaded into memory and the CPU executes these instructions in a sequence to perform a task or set of tasks. Processes can vary widely. For example, they can be simple or complicated. Processes can rely on external inputs like input from a user via a keyboard, or data from other processes, and can generate an output as seen in figure 1-2.

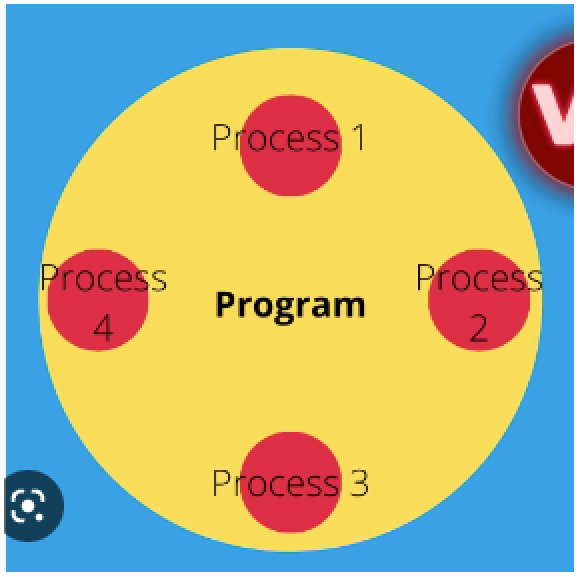


Figure 1-2. Diagram of how processes relate to the program

Each process consists of its own memory space and this is an essential part of how the CPU is managed, as it prevents data being corrupted or bleeding over into other processes. A process has its own ID called a *PID* (short for Process ID), and can be monitored and controlled by the computer's operating system. A PID is a unique identifier that the operating system assigns to a process. It allows the operating system to keep track of all the resources that are associated with the process such as memory usage and CPU time. We can also use the PID - if we need to stop a process, for example, if the process is not responding, we can use the `kill <PID>` command to send a signal to process with this PID to terminate.

Here we have an example of a process running on one core. We are simulating a program running and doing 10 tasks by adding in a sleep for each for 1 second.

First we will import the libraries we need:

```
use std::thread::sleep;
use std::time::{Duration, Instant};
```

Now we create a `task()` function that simply prints to the command line and sleeps for 1 second.

```
fn task() {
    println!("Running task...");
```

```
        sleep(Duration::from_secs(1));
}
```

Finally we run this task 10 times and time this.

```
fn main() {
    let start = Instant::now();


    for _ in 0..10 {
        task();
    }
    let elapsed = start.elapsed();
    println!("The whole program took: {:?}", ela
}
```

You can see that running this function `task()` 10 times took 10 seconds. The process contains 10 tasks but because it is one unit of activity, they run on the same core and all 10 tasks execute one after another.

An important consideration is that whilst a single-core CPU can only execute one process at a given time, most modern computers now have multiple core CPUs which means the number of processes running concurrently can be more. We can rewrite this code to split the 10 tasks across two different processes, hence halving the total time taken to 5 seconds. This is an example of parallelism as discussed above.

First we import the libraries that we need.

```
use std::env;
use std::process::{Command, exit};
use std::time::{Duration, Instant};
use std::thread::sleep;
```

Now we use the `std::process::Command` module to *spawn* new processes that run the same binary but execute different parts of the code. This is because Rust does not have in-built support for creating separate processes.

```
fn run_processes() {
    let mut process1 = Command::new(env::current_
        .arg("task")
        .arg("1")
        .spawn()
        .expect("Failed to start process1");

    let mut process2 = Command::new(env::current_
        .arg("task")
        .arg("6")
        .spawn()
```

```
          .expect("Failed to start process2");

    process1.wait().expect("Failed to wait for p
    process2.wait().expect("Failed to wait for p

    println!("Both processes have completed.");
}
```

We create a task that will execute and sleep for 1 second. We also grab the PID and print it to the terminal.

```
fn main() {
    let args: Vec<String> = env::args().collect(

    let start = Instant::now();

    if args.len() > 2 && args[1] == "task" {
        let start_task_number = args[2].parse::<
        task(start_task_number);
    } else {
        run_processes();
    }

    if args.len() <= 1 {
        let elapsed = start.elapsed();
        println!("The whole program took: {:?}",
    }
}
}
```

This code takes 5.014 seconds to run so you can see that we can half the time needed for 10 tasks that take 1 second each by splitting them across 2 processes.

We can examine the output below:

```
Task 1 completed in process: 22538
Task 6 completed in process: 22539
Task 7 completed in process: 22539
Task 2 completed in process: 22538
Task 8 completed in process: 22539
Task 3 completed in process: 22538
Task 9 completed in process: 22539
Task 4 completed in process: 22538
Task 10 completed in process: 22539
Task 5 completed in process: 22538
Both processes have completed.
The whole program took: 5.013870167s
```

Note that Task 1 to 5 occur in process with PID 22538, where as task 6 - 10 have a different PID, 22539. Now that we have created some processes, we should try and interact with these processes.

## Interacting With Processes

Processes are independent of each other and cannot directly interact with each other. To demonstrate a process’s isolation, we can build a simple program that spawns a process that is a continuous loop. Once we have created this, we will be able to interact directly with the process from another terminal. Before we write any code, we will need the following dependency:

```
[dependencies]
tempfile = "3.2.0"
```

This will enable us to create a temporary file for our process to run on. We can then move to our `main.rs` file and define the imports below:

```
use std::fs::File;
use std::io::Write;
use std::env;
use std::process::{Command, Stdio};
```

These imports are going to enable us to write Rust code to a temporary file, compile the Rust code in that temporary file, and then spawn a child process running that compiled code. Note that this approach is not recommended for solving real world problems. However, this approach does concrete how processes in Rust work, which is our goal here.

Inside our main function, we define the Rust code that is going to be run in our child process, and write that Rust code to the temporary file in a tempory directory that we created with the following code:

```
let child_process_code = r#"
    use std::env;
    use std::process;
    use std::thread;
    use std::time::Duration;

    fn main() {
        loop {
            println!("This is the child process s
            thread::sleep(Duration::from_secs(4)
            let pid = process::id();
            println!("Child process ID: {}", pid
        }
    }
"#;

// Create a temporary file

let mut temp_dir = env::temp_dir();
temp_dir.push("child_process_code.rs");
let mut file = File::create(&temp_dir).expect("Fa



// Write the Rust code to the temporary file
```

```
file.write_all(child_process_code.as_bytes())
    .expect("Failed to write child process code t
```

Here, can see that we are just looping and printing out the process ID of the child process. We print out the child process ID because we need the ID to interact with the child process from outside. We then compile our Rust file with the code below:

```
let compile_output = Command::new("rustc")
    .arg("-o")
    .arg("child_process")
    .arg(&temp_dir)
    .output()
    .expect("Failed to compile child process code


if !compile_output.status.success() {
    eprintln!(
        "Error during compilation:\n{}",
        String::from_utf8_lossy(&compile_output.s
    );
    return;
}
```

Once our Rust code is compiled, we can then spawn our child process and wait for the process to complete with the code below:

```
// Spawn the child process
let mut child = Command::new("./child_process")
    .stdout(Stdio::inherit())
    .spawn()
    .expect("Failed to spawn child process");

println!("Child process spawned with PID: {}", ch

// Wait for the child process to finish
let status = child.wait().expect("Failed to wait

println!("Child process terminated with status: 
```

Because our process is an infinite loop, we will never see the final `println!` statement in our main function unless we somehow interact with our child process. Running our program will give the following output:

```
Child process spawned with PID: 32065
This is the child process speaking!
Child process ID: 32065
This is the child process speaking!
Child process ID: 32065
```

We can then kill the child process by using the kill statement followed by the process ID like below:

```
kill 32065
```

This terminates the child process finishing our main Rust process with the following print out:

```
Child process terminated with status: ExitStatus(
```

This printout indicates that the child process received a signal with the number 15. We’re using a Mac so your signal might be a different number. In Unix-like systems, signal 15 is usually *SIGTERM*, which is a polite request to terminate the process allowing, for cleanup operations to complete before exiting.

With the program that we have just run, we can see that processes are isolated due to the fact that we could not just pass a function with an infinite loop directly into a child process and run it. Instead, we had to get the child process to run its own Rust program. We could also interact with the child process from outside of the program in a completely different terminal with the kill command.

However, just sending signals is not the only way we can communicate with our processes. We can communicate with our processes by using specific communication mechanisms.

## Communicating With Processes

The most basic form of communication is through the std in and out. We can demonstrate this basic communication using `stdin` with the following simple program:

```
use std::io::{self, BufRead};
use std::process;

fn main() {
    let pid = process::id();
    println!("process ID: {}", pid);

    let stdin = io::stdin();
    let mut lines = stdin.lock().lines();


    loop {
        let line = match lines.next() {
            Some(Ok(line)) => line,
            _ => {
                eprintln!("Failed to read from st
                break;
            }
        };
        println!("Received: {}", line);
    }
}
```

This is where we lock the `stdin` and then wait for a new line with the next function. It must be stressed that the next function is not asynchronous, it is a synchronous call that will block the process until the next line from `stdin` is read. `stdin` can be thought of as a file-like object that can be read like a file. However, the `stdin` is not a regular file stored on disk. `stdin` is a communication channel provided by the operating system that is reading data from that has been provided by another user or process.

Running our program will give us the following print out which by now you can deduce is *stdout*:

```
process ID: 38271
```

Different operating systems will have different approaches. To find the location of stdin for a Mac, we can run the following command:

```
lsof -p 38271 | grep 0u
```

Here we essentially get data on the process including the location of stdin for the process as seen below:

```
process_c 38271 homeuser  0u CHR  16,0  0t14667
```

The location of the stdin is on the right of the printout. We can then pipe our string to the stdin with the following command:

```
echo "Hello from the terminal" | dd of=/dev/ttys(
```

And we will get the following printout:

```
0+1 records in
0+1 records out
24 bytes transferred in 0.000016 secs (1500000 by
```

We also get a “Hello from the terminal” printout in the terminal of the process that is running our program. And with this we can conclude that we have transferred bytes from one terminal to our program to read. With Linux the process is simpler. Once the program is running you can pass bytes with the following command:

```
echo "Hello from the terminal" > /proc/[PID]/fd/(
```

In Windows there is not a direct equivalent. We recommend that readers using Windows install a Linux subsystem like WSL, as we would otherwise need to rewrite the code to support direct stdin for Windows with raw file handles.

While writing directly to stdin in the terminal is interesting, it is advised that you use pipes and sockets to communicate between processes. We can pipe data directly to the process without knowing the process ID by simply getting rid of the loop and having the following code in its place:

```
let line = match lines.next() {
    Some(Ok(line)) => line,
    _ => {
        panic!("Failed to read from stdin");
    }
};
```

We can then pipe data into our program with the command below:

```
echo "Hello from the terminal" | ./path/to/rust/
```

This will print out our string that we piped in and then our program will complete.

Processes can also communicate using *sockets*. This is where two processes establish a connection for exchanging data. Sockets do not only share data between processes in the same system but also across different systems connected via a network. A process can listen to a socket on a certain port whilst another process can send bytes over that socket by pointing to the same port. This is the basis of internet communication as the HTTP protocol is merely a protocol built on top of the TCP protocol. There are many high-level abstractions such as JSON serialisation crates, and TCP/HTTP listeners.

---

**NOTE**

We have covered enough ground on processes to give context to what processes are without going into coding examples of socket communication. There are multiple Rust web programming books that already cover this in more detail.

---

Spinning up new processes is expensive and memory cannot be directly shared between them. Processes can also interact with other processes outside of our main application. If we were to use sockets to communicate between processes, each process would bind to an individual port for listening, and this networking brings its own headaches. Remember what we want to do with asynchronous programming. We want to spin off light weight non-blocking tasks and wait for them to finish. In a lot of cases we would want to get the data from those tasks and use them. We also want the option of sending the task to the async runtime with data. Threads seem like a much better choice over processes for asynchronous programming. We will cover threads next.

## What Are Threads?

A thread of execution is the smallest sequence of programmed instructions that can be independently managed by a scheduler. Inside a process we can share memory across multiple threads as seen in figure 1-3.

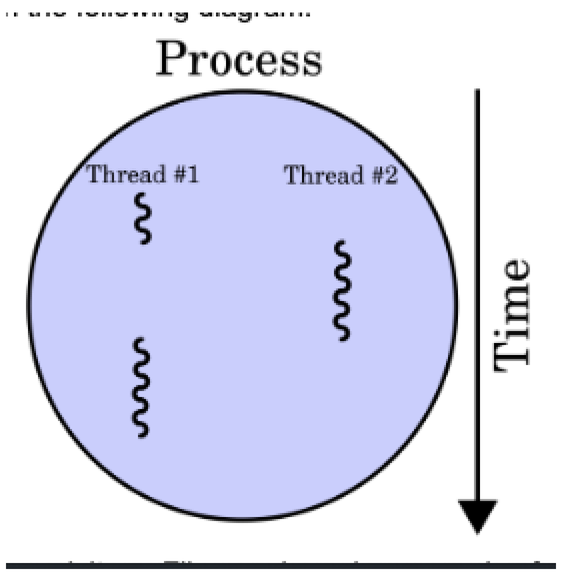


Figure 1-3. Diagram of how threads relate to a process

To spin up threads, we can revisit our Fibonacci number recursive function and spread the calculations of Fibonacci numbers over four threads. First we need to import the following:

```
use std::time::Instant;
use std::thread;
```

We can then time how long it takes to calculate the 50th Fibonacci number in the main function with the code below:

```
let start = Instant::now();
let _ = fibonacci(50);
let duration = start.elapsed();
println!("fibonacci(50) in {:?}", duration);
```

After this, we can reset the timer and calculate the time taken to calculate four 50th Fibonacci numbers over four threads. We achieve the multithreading by iterating four times to spawn four threads and attach the `JoinHandles` into a vector with the following code:

```
let start = Instant::now();
let mut handles = vec![];
for _ in 0..4 {
    let handle = thread::spawn(|| {
        fibonacci(50)
    });
    handles.push(handle);
}
```

`JoinHandles` are owned permissions to join the thread associated with that handle. Joining the thread means blocking the program until the thread is terminated. `JoinHandles` implement the `Send` and `Sync` traits which means that `JoinHandles` can be sent between threads. However, it must be noted that the `JoinHandle` does not implement the `Clone` trait. This is because we need a unique `JoinHandle` for each thread. If there were multiple `JoinHandles` for one thread, you can run the risk of multiple threads trying to join the running thread, leading to data races.

---

**GREEN THREADS**

You may have come across green threads if you have used other programming languages. Green threads are threads that are scheduled by something other than the operating system, for example, a runtime or a virtual machine. Rust originally implemented green threads before pulling them prior to version 1. The main reason for removing them and moving to a native thread with green threads in libraries is that in Rust threads and I/O operations are coupled, which forced native threads and green threads to need to have and maintian the same API. This resulted in various problems in using I/O operations and designating allocation.

---

Now that we have our vector of `JoinHandles`, we can wait for them to execute and then print the time taken with the code below:

```
for handle in handles {
    let _ = handle.join();
}
let duration = start.elapsed();
println!("4 threads fibonacci(50) took {:?}", du
```

Running our program shows gives the following output:

```
fibonacci(50) in 39.665599542s
4 threads fibonacci(50) took 42.601305333s
```

So we can see that when using threads in Rust, we can handle multiple CPU intensive tasks at the same time. Considering that multiple threads can handle CPU intensive tasks at the same time, we can also deduce that multiple threads can also handle waiting concurrently. Even though we do not use the results of the fibonscci calculations, we could use the results of the threads in our main program if we wanted to. When we are calling a join on a `JoinHandle` in this example, we are returning a `Result<u64, Box<dyn Any + Send>>`. The `u64` is the result of the calculated Fibonacci number from the thread. The `Box<dyn Any + Send>>` is a generic type that allows flexibility in handling different error types. These error types need to be sent over but there could be a whole host of reasons why a thread errors. There is some overhead to this however, because there needs to be dynamic downcasting and boxing as we do not know the size at compile time.

Threads can also directly interact with each other over memory within the program. For the last example of this chapter we will use channels, but for now, we can make do with an `Arc`, `Mutex`, and a `CondVar` to create the system depicted in figure 1-4.

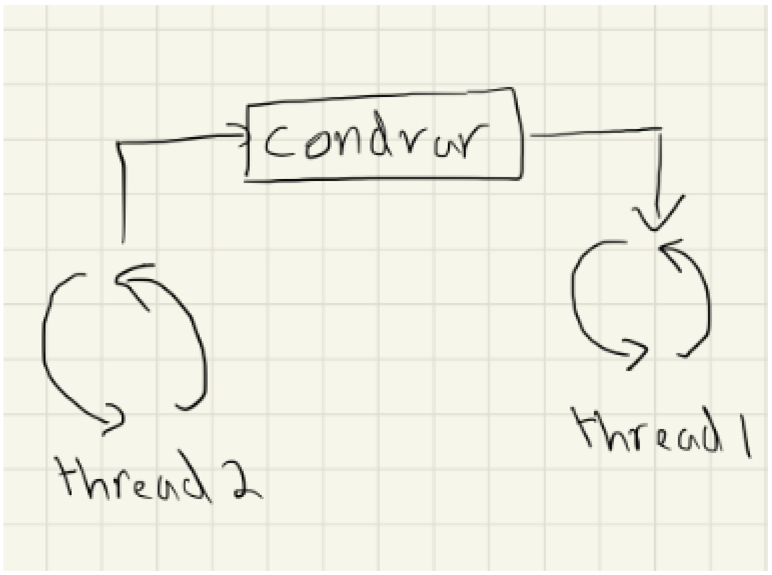


Figure 1-4. Condvar alerting another thread of a change

Here, we are going to have two threads. One thread is going to update the `Condvar`, and the other thread is going to listen to the `Condvar` for updates, and print out that there has been an update to the file the moment the update has occurred. Before we write any code however, we need to establish the following structs:

*`Arc`*
This stands for Atomic Reference Counting, meaning that Arc keeps count of the amount of references to the variable that is wrapped in an `Arc`. So, if we were to define an `Arc<i32>` and then reference that `Arc<i32>` over four threads, the reference count would increase to

four. The `Arc<i32>` would only be dropped when all four threads had finished referencing it resulting in the reference count being zero.

*`Mutex`*

Remember that Rust only allows us to have one mutable reference to a variable at any given time? A `Mutex` is a smart pointer type that provides *interior mutability* by having the value inside the `Mutex`. This means that we can provide mutable access to a single variable over multiple different threads. This is achieved by a thread acquiring the lock. When we acquire the lock, we get the single mutable reference to the value inside the `Mutex`. We then perform a transaction, and then give up the lock to allow other threads to perform a transaction. The lock ensures that only one thread will have access to the mutable reference at a time ensuring that Rust's rule of only one mutable reference at a time is not violated. We must note that there is an overhead of acquiring the lock as we might have to wait until it is released.

*`Condvar`*

This is short for "conditional variable". This allows our threads to sleep and be woken when a notification is sent through the `Condvar`. It must be noted that we cannot send variables through the `Convar`, but multiple threads can subscribe to a single Condvar.

Now that we have covered what we are using, we can build our system by initially importing the following:

```
use std::sync::{Arc, Condvar, Mutex};
use std::thread;
use std::time::Duration;
use std::sync::atomic::AtomicBool;
use std::sync::atomic::Ordering::Relaxed;
```

We can then define the data that we are going to share across our two threads with the code below:

```
let shared_data = Arc::new((Mutex::new(false), Co
let shared_data_clone = Arc::clone(&shared_data)
let STOP = Arc::new(AtomicBool::new(false));
let STOP_CLONE = Arc::clone(&STOP);
```

Here we have a tuple that is wrapped in `Arc`. We then have our boolean variable that is going to be updated wrapped in a `Mutex`. We then clone our data package so both threads have access to the shared data. Now that our data is available, we can define our first thread with the following code:

```
let background_thread = thread::spawn(move || {
    let (lock, cvar) = &*shared_data_clone;
    let mut received_value = lock.lock().unwrap(
    while !STOP.load(Relaxed) {
        received_value = cvar.wait(received_value
        println!("Received value: {}", *received_
    }
```

```
        }
    });
```

Here we can see that we merely wait on the `Condvar` notification. At the point of waiting, it is said that the thread is "parked". This means that the thread is blocked but also not executing. Once the notification comes in from the condvar, the thread then accesses the variable in the `Mutex` once the thread has been woken by the `Condvar`. We then merely print out the variable and the thread goes back to sleep. We must note that we are relying on the atomic bool being false for the loop to continue indefinitely. This enables us to stop the thread if we need.

In the next thread we only do four iterations before completing the thread as seen in the code below:

```
let updater_thread = thread::spawn(move || {
    let (lock, cvar) = &*shared_data;
    let values = [false, true, false, true];

    for i in 0..4 {
        let update_value = values[i as usize];
        println!("Updating value to {}...", updat
        *lock.lock().unwrap() = update_value;
        cvar.notify_one();
        thread::sleep(Duration::from_secs(4));
    }
    STOP_CLONE.store(true, Relaxed);
```

```
        println!("STOP has been updated");
        cvar.notify_one();
    });
    updater_thread.join().unwrap();
```

Here we see that we update the value and then notify the other thread that the value has changed. We then block the main program until the `updater_thread` has finished. Running the program will give us the following output:

```
Updating value to false...
Received value: false
Updating value to true...
Received value: true
Updating value to false...
Received value: false
Updating value to true...
Received value: true
```

We can see that our updater thread is updating the value of the shared data, and notifying our first thread which accesses it.

The values are consistent which is what we want, although admittedly it's a very crude implementation of what we could describe as async behaviour. The thread is stopping and waiting for updates. Adding multiple condvars for the `updater_thread` to cycle through and check would result in one thread keeping track of multiple tasks and acting on those tasks when changed. Whilst this will certainly spark a debate online on if this is truly async behavour or not, we can certainly say that this is not an optimum or standard way of implementing async programming. However, we can see how threads are a key building block for async programming. Async runtimes essentially accept futures, resulting in multiple async tasks being juggled in one thread. This thread is usually seperate from the main thread. Runtimes can also have multiple threads executing tasks. In the next section we will utilise standard implementations of async code whilst exploring where to utilise async.

## Where Can We Utilise Async?

We have introduced you to Asynchronous Programming and demonstrated some of its benefits in the examples such as multple HTTP requests. These have been toy examples designed to show you the power of async. In this section we will discuss some real life utilitsations of async and why you might want to include them in your next project.

Let’s first think about what we can use async for. Unsurprisingly the main use cases involve operations where there is a delay or potential delay in doing something or receiving something. For example, I/O calls to the file system, or network requests. Async allows the program that calls these operations to continue without blocking, which could cause the program to hang and become less responsive.

I/O operations like writing files are considered slow comparatively to in-memory operations because they usually rely on external devices such as hard-drives. Most hard-drives still rely on mechanical parts that need to physically move which is slower than electronic operations in RAM or the CPU. In addition, the speed at which data can be transferred from the CPU to the device may be limited for example by a USB connection.

We should mention now at the point of writing this book, async file reads are not actually sped up by async at the moment. This is because file I/O are still bound by disk performance so the bottleneck is in the disk write and read speed rather than CPU. What async can do however is make sure that whilst your file I/O is occurring, your program can continue and is not blocked by these operations.

We will now work through an example using async for a file I/O program. Imagine a situation in which we need to keep track of changes to a file and perform an action when a change in the file has been detected.

## Using Async For File I/O

To track file changes, we need to have a loop in a thread checking the metadata of the file, and then feeding back to the main loop in the main thread when the metadata of the file changes as depicted in figure 1-5.

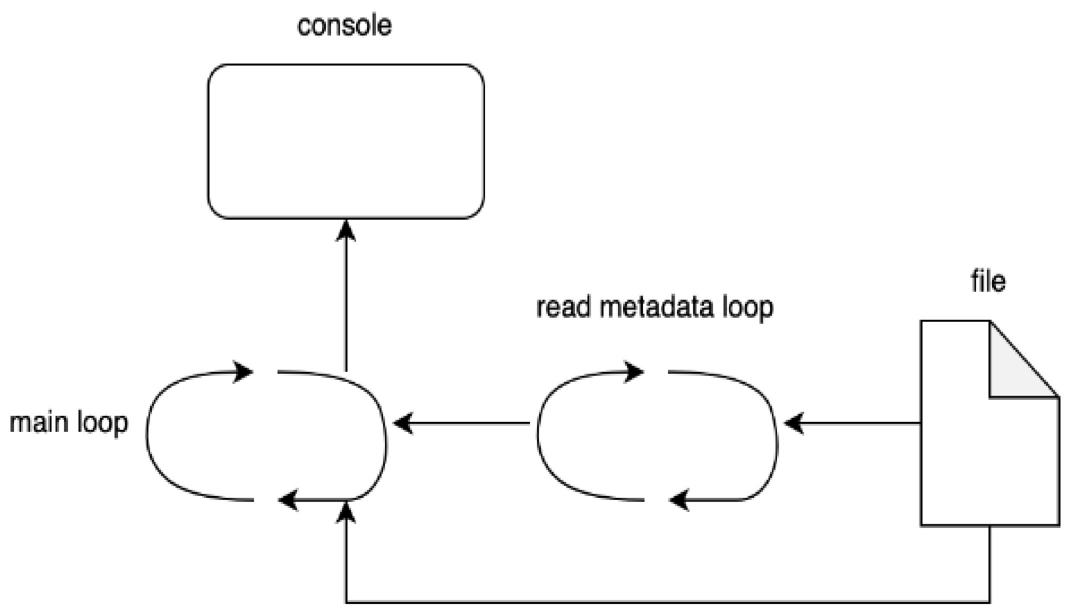


Figure 1-5. Overview of a system keeping track of changes in a file

We can do all manner of things once the change is detected but for the purpose of this exercise, we will just print the contents out to the console. Before we start tackling the components in figure 1-5, we need to import the following structs and traits:

```rust
use std::path::PathBuf;
use tokio::fs::File as AsyncFile;
use tokio::io::AsyncReadExt;
use tokio::sync::watch;
use tokio::time::{sleep, Duration};
```

We will cover how these structs and traits are used as we go along. Referring back to figure 1-5, it makes sense to tackle the file operations first and then the main loop later. Our simplest operation is just reading the file with a function as seen with the following code:

```
async fn read_file(filename: &str) -> Result<Stri
    let mut file = AsyncFile::open(filename).awai
    let mut contents = String::new();
    file.read_to_string(&mut contents).await?;
    Ok(contents)
}
```

Here we merely open the file, and read the contents to a string, returning that string. However, it must be noted that at the time of writing this, the standard implementation of async file reading is not async. Instead it is blocking, thus the file open operation is not truly async. The inconsistency of async file reading is down to the file API that the operating system supports. For instance, if you have Linux with a kernel version of 5.10 or higher, you can utilise the tokio-uring crate that will enable true asynchronous I/O calls to the file API. However, for now, our function does the job that we need.

We can now move onto our loop that periodically checks the metadata of our file with the following code:

```
async fn watch_file_changes(tx: watch::Sender<boo
    let path = PathBuf::from("data.txt"); ❶
    let mut last_modified = None; ❷
    loop { ❸
        if let Ok(metadata) = path.metadata() {
            let modified = metadata.modified().ur

            if last_modified != Some(modified) {
                last_modified = Some(modified);
                let _ = tx.send(true);
            }
        }
        sleep(Duration::from_millis(100)).await;
    }
}
```

Here we can see that we see that our function is an async function that carries out the following steps:

❶ We get the path to the file that we are checking.

❷ set the last modified time to none as we have not checked the file yet.

❸ We then have an infinite loop.

❹ In that loop we extract the time of last modified.

❺ If the extracted timestamp is not the same as our cached timestamp, we then update our cached timestamp, and send a message through a channel using the sender that we passed into our function. This message then alerts our main loop that the file has been updated.

❻ For each iteration we sleep a small amount of time just so that we are not constantly hitting the file that we are checking on.

---

**NOTE**

If we use a Tokio thread to run this function, the Tokio runtime will be able to switch context and execute another thread in the same process. If we use the standard library's sleep function, the thread will block. This is because the standard library's sleep will not send the task to the Tokio executor. We will go over executors in chapter 3, a deeper understanding of futures.

---

Now that our first loop is defined, we can move onto the loop that is run in the main. At this point, if you know how to spin up Tokio threads and channels, you could have an attempt at writing the main function yourself.

If you did attempt to write your own main function, hopefully it looks similar to the following code:

```
#[tokio::main]
async fn main() {
    let (tx, mut rx) = watch::channel(false); ❶

    tokio::spawn(watch_file_changes(tx)); ❷

    loop { ❸
        // Wait for a change in the file
        let _ = rx.changed().await; ❹

        // Read the file and print its contents t
```

```
            // Read the file and print its contents
            if let Ok(contents) = read_file("data.txt
                println!("{}", contents);
            }


        }
    }
```

Our main function carries out the following steps:

1. We create a channel that is a single producer multi consumer channel that only retains the last set value.
2. We then pass the transmitter of that channel into our watch file function which is being run in a tokio thread that we spin off.
3. Now our file watch loop is running, we then move onto our loop that essentially holds until the the value of the channel is changed.
4. Because we do not care about the value coming from the channel, we merely denote the variable assignment as an underscore. Our main loop will stay there until there is a change in the value inside the channel.
5. Once the value inside the channel changes due to the metadata of the file changing, the rest of the loop interaction executes, reading the file and printing out the contents.

Before we run this we do need a `data.txt` file in the root of our project next to our `Cargo.toml`. We can then run the system, open out the `data.txt` file in an IDE, and then type something into the file. Once you save the file, you will get the contents of the `data.txt` file printed out in the console!

Now that we have utilised async programmig locally, we can now go back to implementing async programming with networks.

## Improving HTTP Request Performance With Async

I/O operations do not just concern reading and writing files, but include getting information from an API, executing operations on a database or receiving information from a mouse or keyboard. What ties them altogether is that these operations are slower than the in-memory operations that can be performed in RAM. Async allows the program to continue without being blocked by the ongoing operation. Other tasks can be executed whilst we await the async operation.

In the following example, let’s imagine a user has logged into a website, and we want to display some data along with the time since their last login. To fetch the data, we’ll be using an external API that provides a specific delay. We need to process this data once it’s received, so we’ll define a `Response` struct and annotate it with the `Deserialize` trait to enable deserialization of the API data into a usable object.

To make the API calls, we'll use the reqwest package and since we'll be working with JSON data, we'll enable the json feature of reqwest by specifying `features=["json"]` in the dependency configuration. This allows us to conveniently handle JSON data when making API requests and processing the responses.

We will need to add these dependencies to our `Cargo.toml`:

```
[dependencies]
tokio = { version = "1", features = ["full"] }
reqwest = { version = "0.11", features = ["json"]
serde = { version = "1.0", features = ["derive"]
serde_json = "1.0"
```

Next we will import the libraries we need and define the Response struct.

```
use reqwest::Error;
use serde::Deserialize;
use tokio::time::sleep;
use std::time::Duration;
use serde_json;

#[derive(Deserialize, Debug)]
struct Response {
    url: String,
    args: serde_json::Value,
}
```

We’ll now implement the `fetch_data()` function. When called, it sends a GET request to “https://httpbin.org/delay/”, which will return a response after a specified number of seconds. In our example, we’ll set the delay to 5 seconds to emphasize the importance of designing a program capable of handling delays effectively in real-world scenarios.

```
async fn fetch_data(seconds: u64) -> Result<Respo
    let request_url = format!("https://httpbin.or
    let response = reqwest::get(&request_url).awa
    let delayed_response: Response = response.jso
    Ok(delayed_response)
}
```

Whilst this is occurring, we will create a function that calculates the time since you logged in. This would usually require a database check but we will simulate the time it takes to check this by doing a sleep for 1 second. This simplifies the example so we do not need to get into database set ups.

```
async fn calculate_last_login() {
    sleep(Duration::from_secs(1)).await;
    println!("Logged in 2 days ago");
}
```

Now we put it together:

```
#[tokio::main]
async fn main() -> Result<(), Error> {
    let data = fetch_data(5);
    let time_since = calculate_last_login();
    let (posts, _) = tokio::join!(data, time_sinc
    println!("Fetched {:?}", posts);
    Ok(())
}
```

Lets examine the output:

```
Logged in 2 days ago
Fetched Ok(Response { url: "https://httpbin.org/d
```

In the `main()` function, we initiate the API call using the `fetch_data()` function before calling the `calculate_last_login()` function. The API request is designed to take 5 seconds to return a response. Since `fetch_data()` is an asynchronous function, it is executed in a non-blocking manner, allowing the program to continue its execution. As a result, the `calculate_last_login()` function is executed and its output is printed to the terminal first. After the 5-second delay, the `fetch_data()` function completes, and its result is returned and printed. What this demonstrates which the initial http request example did not highlight, is how asynchronous programming allows concurrent execution of tasks without blocking the program’s flow resulting in network requests completing out of order. Therefore, we can utilise async for multiple network requests as long as the scope of where we call the `await` on the network requests in the order that we need the data.

## Summary

In this chapter we introduced the concept of async programming and how it relates to the computer system in terms of threads and processes. We then covered some basic high-level interactions with threads and processes to demonstrate that threads can have some utility to async programming. We then explored some basic high-level async programming improving the performance of multiple http calls by sending more requests whilst waiting for other requests to respond. We also used async principles to keep track of file changes. What this chapter has demonstrated is that async is a powerful tool for juggling multiple simultaneous tasks at the same time that do not need constant CPU time. Therefore, async enables us to have one thread handling multiple tasks at the same time. Now that you understand the context of where async programming sits in a computer system, we will explore basic async programming concepts in the next chapter.

# Chapter 2. Basic Async Rust

This chapter introduces the important components of using async in rust and gives an overview of tasks, futures, async and await. Here we cover context, pins, polling and closures which are important concepts for fully taking advantage of async programming in Rust. Once again, we must mention that we have chosen the examples in this chapter to demonstrate the learning points and they may not necessarily be optimal for efficiency, rather we seek to show practical example to demonstrate the concepts we’re covering. The chapter concludes with an example for building a async audit logger for a sensitive program which we hope pulls all the concepts together.

By the end of this chapter, you will be able to define a task and a future, and understand the more technical components of a future including context and pins.

## Understanding Tasks

In asynchronous programming, a task represents an asynchronous operation. The Task asynchronous programming model (TAP) provides an abstraction over asynchronous code. You write code as a sequence of statements. You can read that code as though each statement completes before the next begins. For instance, let us think about making a cup of coffee and toast which requires the following steps:

1. Put bread in toaster
2. Butter toasted bread
3. Boil kettle
4. Pour milk
5. Put in instant coffee granules (not the best but simplifies the example)
6. Pour boiled water

We can definitely apply async programming to speed this up, but before we do this, we need to break down all the steps into two big steps, “make coffee” and “make toast” with the following steps:

1. **Make coffee**

    1. Boil kettle
    2. Pour milk
    3. Put in instant coffee
    4. Pour boiled water
2. **Make toast**
    1. Put bread in toaster
    2. Butter toasted bread

Even though we have only one pair of hands, we can run these two steps at the same time. We could boil the kettle, and whilst the kettle is boiling we can put the bread in the toaster. There is a bit of dead time whilst we wait for the kettle and toaster, so if we really wanted to be efficient and we were comfortable with the risk that we could end up pouring the boiled water before the coffee and milk due to an instant boil, we could break the steps down even more with the following:

1. **prep coffee mug**
    1. Pour milk
    2. Put in instant coffee
2. **Make coffee**
    1. Boil kettle
    2. Pour boiled water
3. **Make toast**
    1. Put bread in toaster
    2. Butter toasted bread

If there is any time taken for the boiling of the water and toasting of the bread, we can execute the pouring of the milk and adding the coffee, reducing the deadtime. First of all, we can see that steps are not goal specific. When we walk into the kitchen our mind will think “make toast”, and “make coffee” which are two separate goals. But we have defined three steps for those two goals. Steps are about what you can run at the same time out of sync to achieve all your goals. We must also note that there is a trade off when it comes to assumptions and what we are willing to tolerate. For instance, it may be completely unacceptable to pour boiling water before adding milk and coffee. This is a risk if there is no delay in the boiling of the kettle. However, we can make the safe assumption that there will be a delay.

Now that we understand what steps are, we can concrete our example by using some high-level crate like Tokio so we can focus on the concepts of steps and how they relate to tasks. Do not worry, we will use other crates in later chapters when we go into lower-level concepts. First, we need to import the following:

```
use std::time::Duration;
use tokio::time::sleep;
use std::thread;
use std::time::Instant;
```

We use the Tokio sleep for steps that we can wait on, such as the boiling of the kettle and the toasting of the bread, as the Tokio sleep function is non-blocking so we can switch to another step when the water is boiling or the bread is toasting. We use the `thread::sleep` to simulate a step that we use both our hands for as we can't do anything else whilst pouring milk/water, or buttering toast. In general programming these will be CPU intensive steps. We can then define our prepping of the mug step with the following code:

```rust
async fn prep_coffee_mug() {
    println!("Pouring milk...");
    thread::sleep(Duration::from_secs(3));
    println!("Milk poured.");
    println!("Putting instant coffee...");
    thread::sleep(Duration::from_secs(3));
    println!("Instant coffee put.");
}
```

We then define the "make coffee" step with the code below:

```rust
async fn make_coffee() {
    println!("boiling kettle...");
    sleep(Duration::from_secs(10)).await;
    println!("kettle boiled.");
    println!("pouring boiled water...");
    thread::sleep(Duration::from_secs(3));
    println!("boiled water poured.");
}
```

And then we define our last step with the following code:

```
async fn make_toast() {
    println!("putting bread in toaster...");
    sleep(Duration::from_secs(10)).await;
    println!("bread toasted.");
    println!("buttering toasted bread...");
    thread::sleep(Duration::from_secs(5));
    println!("toasted bread buttered.");
}
```

You may have noticed that `await` is used on the Tokio sleep functions that represent the steps that are not intensive and that we can wait on. The `await` keyword is used to suspend the execution of our step until the result is ready. When the `await` is hit, the async runtime can switch to another async task.

Now that we have all of our step defined we can run all of them in an async manner with the code below:

```
#[tokio::main]
async fn main() {
    let start_time = Instant::now();
    let coffee_mug_step = prep_coffee_mug();
    let coffee_mug_step = prep_coffee_mug();
    let coffee_step = make_coffee();
    let toast_step = make_toast();

    tokio::join!(coffee_mug_step, coffee_step, to
    let elapsed_time = start_time.elapsed();
    println!("It took: {} seconds", elapsed_time
}
```

Here we define our steps which are called futures. We will cover futures in the next section. For now we can think of Futures as a placeholder for something that has not completed yet. We then wait for our steps to complete, and then print out the time taken. If we run our program we get the following:

```
Pouring milk...
Milk poured.
Putting instant coffee...
Instant coffee put.
boiling kettle...
putting bread in toaster...
kettle boiled.
pouring boiled water...
boiled water poured.
bread toasted.
buttering toasted bread...
```

```
toasted bread buttered.
It took: 24 seconds
```

This is a bit of a lengthy printout but it is important. We can see that it looks strange. If we are being efficient, we would not start pouring milk and adding coffee. Instead, we would get the kettle boiling and put the bread in the toaster, and then go to pour milk. We can see that preparing the mug was first passed into the `tokio::join macro`. If we run our program again and again it will always be the case that the preparation of the mug is the first future to be executed. Now, if we go back to the mug preparation function, we simply add a non-blocking sleep function before the rest of the processes as seen below:

```
async fn prep_coffee_mug() {
    sleep(Duration::from_millis(100)).await;
    . . .
}
```

This now gives us the following printout:

```
boiling kettle...
putting bread in toaster...
Pouring milk...
Milk poured.
Putting instant coffee...
```

```
Instant coffee put.
bread toasted.
buttering toasted bread...
toasted bread buttered.
kettle boiled.
pouring boiled water...
boiled water poured.
It took: 18 seconds
```

Ok, now the order makes sense, we are boiling the kettle, putting bread in the toaster, and then pouring milk, and as a result, we saved three seconds. However, the cause and effect is counter intuitive. Putting in an extra sleep function has reduced our overall time. This is because that extra sleep function allowed the async runtime to switch context to other tasks and execute them until their `await` line was executed, and so on. This insertion of an artificial delay in the future to get the call rolling on other futures is informally referred to as “cooperative multitasking”. More on this later.

When we pass our futures into the Tokio join macro, all the async expressions are evaluated concurrently in the same task. The join macro does not create tasks, it merely enables multiple futures to be executed concurrently within the task. For instance, we can spawn a task with the following code:

```
let person_one = tokio::task::spawn(async {
    prep_coffee_mug().await;
    make_coffee().await;
    make_toast().await;
});
```

Each future in the task is will block further execution of that task until the future is finished. So, if we ensure that the runtime has one worker with the annotation below:

```
#[tokio::main(flavor = "multi_thread", worker_thr
```

And we join on two tasks each representing a person, it will result in a 40 second runtime. We can redefine the task with a join as opposed to blocking futures with the following code:

```
let person_one = tokio::task::spawn(async {
    let coffee_mug_step = prep_coffee_mug();
    let coffee_step = make_coffee();
    let toast_step = make_toast();
    tokio::join!(coffee_mug_step, coffee_step, to
});
```

Joining on two tasks representing people will result in a 28 second runtime. If we are to join three tasks representing people, it would result in a 42 second runtime. Seeing as the total blocking time for each task is 14 seconds, the time increase makes sense. We can deduce from the linear increase in time that although there are three tasks sent to the async runtime and put on the queue, the executor is setting the task to idle when coming across an await and working on the next task in the queue whilst polling the idle tasks.

Async runtimes can have multiple workers and queues, and we will explore writing our own runtime in the next chapter. Considering what we have covered in this section, we can give the following definition of a task:

*“A task is a result of a series of futures”*

Now let us discuss what a Future is.

## Futures

One of the key features of async programming is the concept of a *future*. We mentioned above that a future is a placeholder object that represents the result of an asynchronous operation that has not yet completed. Futures allow you to start a task and continue with other operations while the task is being executed in the background.

To truly understand how a future works, we should cover the lifecycle of a future. When a future is created, the future is idle. It has yet to be executed. Once the future is executed, it can either yield a value, resolve, or go to sleep because the future is pending (awaiting on a result). When the future is polled again the poll can either return a pending or ready result. The future will continue to be polled until it is either resolved or cancelled. To concrete how futures work, let us build a basic counter future. The counter future will merely count up to 5 and then will be ready once the counter has reached 5. First we need to import the following:

```
use std::future::Future;
use std::pin::Pin;
use std::task::{Context, Poll};
use std::time::Duration;

use tokio::task::JoinHandle;
```

You should be able to understand most of the code above. We will cover context and Pin after building our basic future. Seeing as our future is a counter, the struct takes the following form:

```
struct CounterFuture {
    count: u32,
}
```

We then implement the Future trait with the following code:

```
impl Future for CounterFuture {
    type Output = u32;

    fn poll(mut self: Pin<&mut Self>, cx: &mut Co
        self.count += 1;
        println!("polling with result: {}", self
        std::thread::sleep(Duration::from_secs(1
        if self.count < 3 {
            cx.waker().wake_by_ref();
            Poll::Pending
        } else {
            Poll::Ready(self.count)
        }

    }
}
```

Again, let us not focus on the Pin or context just yet, we are just looking at the poll function as a whole. Everytime the future is polled, the count is increased by one. If the count is at three we then state that the future is ready. We introduce the `std::thread::sleep` function to merely exaggerate the time taken so it is easier to follow this example when running the code. To run our future, we simply need the code below:

```
#[tokio::main]
async fn main() {
    let counter_one = CounterFuture { count: 0 };
    let counter_two = CounterFuture { count: 0 };
    let handle_one: JoinHandle<u32> = tokio::tas
        counter_one.await
    });
    let handle_two: JoinHandle<u32> = tokio::tas
        counter_two.await
    });
    tokio::join!(handle_one, handle_two);
}
```

Running two of our futures in different tasks gives us the following printout:

```
polling with result: 1
polling with result: 1
polling with result: 2
polling with result: 2
polling with result: 3
polling with result: 3
```

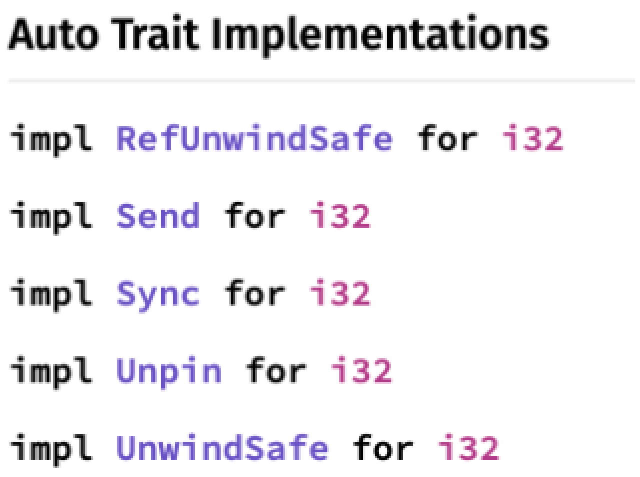
We can see that one of the futures was taken off the queue, polled, and then set to idle whilst another future was taken off the task queue to be polled. These futures were polled in alternate fashion. You may have noticed that our poll function is not async. This is because an async poll function would return a circular dependency as you would be sending a future to be polled in order to resolve a future being polled. With this, we can see that the future is the bedrock of the async computation.

We see that the poll function takes a mutable reference of itself, however, this mutable reference is wrapped in a Pin which we need to discuss.

## Pinning in Futures

In Rust, the compiler often moves values around in memory. For instance, if we move a variable into a function, the memory will be moved. It’s not just moving values that results in the moving of memory addresses. Collections can also change memory addresses. For instance, if a vector gets to capacity, the vector will have to be reallocated in memory changing the memory address.

Most normal primitives such as number, string, bools, structs, enum etc implement the `Unpin` trait enabling them to be moved around. If you are unsure if your data type implements the `Unpin` trait, run a doc command and check the traits your data type implements. For example, below is the auto-trait implementations on an `i32` in the standard docs as shown in figure 2-1.


Auto Trait Implementations

```
impl RefUnwindSafe for i32
impl Send for i32
impl Sync for i32
impl Unpin for i32
impl UnwindSafe for i32
```

Figure 2-1. Example of Auto Trait Implementations in documention shownning thread safety of a struct or primitive

So why do we concern ourselves with pinning and unpinning? We know that futures get moved as we use the async move in our code when spawning a task. However, moving can be dangerous. To demonstrate the data, we can build a basic struct that references itself with the following code:

```
use std::ptr;

struct SelfReferential {
    data: String,
    self_pointer: *const String,
}
```

The `*const String` is a raw pointer to a string. This means that the pointer directly references the memory address where the data is. The pointer offers no safety guarantees. This means that the reference does not update if the data being pointed to moves. We are using a raw pointer to demonstrate why pinning is needed. For this demonstration to take place, we need to define the constructor of the struct, and printing of the structs reference using the code below:

```
impl SelfReferential {
    fn new(data: String) -> SelfReferential {
        let mut sr = SelfReferential {
            data,
            self_pointer: ptr::null(),
        };
        sr.self_pointer = &sr.data as *const Str
        sr
    }
    fn print(&self) {
        unsafe {
            println!("{}", *self.self_pointer);
        }
    }
}
```

To then expose expose the danger of moving the struct by creating two instances of the `SelfReferential` struct, swap these instances in memory, and then print what data the raw pointer is pointing to with the following code:

```
fn main() {
    let mut first = SelfReferential::new("first"
    let mut second = SelfReferential::new("secon
    unsafe {
        ptr::swap(&mut first, &mut second);
    }
    first.print();
}
```

If you try and run the code, you will get a segmentation fault. The segmentation fault is an error caused by accessing memory that does not belong to the program. We can see that moving structs with references to itself can be dangerous. Pinning essentially ensures that the future remains at a fixed memory address. This is important because futures can be paused or resumed which can change the memory address. We have nearly covered all of the components in the basic future that we have defined. The only component is the context.

## Context in Futures

A Context only serves to provide access to a waker to wake a task. A waker is a handle that notifies the executor when the task is ready to be run. Lets look at a stripped down version of our poll function so we can focus on the path of waking up the future:

```
fn poll(mut self: Pin<&mut Self>, cx: &mut Contex
        . . .
        if self.count < 3 {
            cx.waker().wake_by_ref();
            Poll::Pending
        } else {
            Poll::Ready(self.count)
        }
    }
```

We can see that the waker is wrapped in the context, and is only utilised when the result of the poll is going to be pending. The waker is essentially waking up the future so it can be executed. If the future is completed then there is no need for any more execution to be done. If we were to remove the waker and run our program again, we would get the following printout:

```
polling with result: 1
polling with result: 1
```

We can see that our program would not have completed and the program hangs. This is because our tasks are still idle but there is no way to wake them up again to be polled and executed to completion. Futures need the `Waker::wake()` function so the wake function can be called when the future should be polled again. The process takes the following steps:

1. The poll function for a future is called and the result is that the future needs to wait for some async operation to complete before the future is able to return a value.
2. The future then registers its interest of being notified of the completion of the operation by calling a method that references the waker.
3. The executor then takes note of the interest in the future's operation and stores the waker in a queue.
4. At some later time the operation completes and the executor is notified. The executor retrieves the wakers from the queue and calls the `wake_by_ref` on each one waking up the futures
5. The `wake_by_ref` signals the associated task that should be scheduled for execution. The way in which this is done can vary depending on the runtime.
6. When the future is executed, the executor will call the poll method of the future again and the future will determine whether the operation has completed returning a value if completion is achieved.

We can see that Futures are used with an async/await function but let's have a think about how else they can be used. We can also use a timeout on a thread of execution. This means that the thread finishes when so much time has elapsed meaning that we do not end up in a situation where the program hangs indefinitely. This is useful when we have a function that can be slow to complete and we want to move on or error early. Remember that threads provide the underlying functionality for executing tasks. We import timeout from `tokio::time` and set up a slow task. In this case, we put this as a sleep for 10 seconds to exaggerate the effect.

```rust
use std::time::Duration;
use tokio::time::timeout;

async fn slow_task() -> &'static str {
    tokio::time::sleep(Duration::from_secs(10)).a
    "Slow Task Completed"
}
```

Now we set up our timeout, in this case, setting it to 3 seconds. This means that the thread will end if the Future is not completed within these 3 seconds. We match the result and print `“Task timed out”`.

```rust
#[tokio::main]
async fn main() {
    let duration = Duration::from_secs(3);
    let result = timeout(duration, slow_task()).a

    match result {
        Ok(value) => println!("Task completed suc
        Err(_) => println!("Task timed out"),
    }
}
```

For CPU-intensive work, we can also off-load work to a separate threadpool and the future resolves when the work is finished. We have now covered the context of futures. We now move onto sharing data between futures.

## Sharing data between Futures

Although it can complicate things, we can share data between futures. We may want to share data between futures for the following reasons:

- Aggregating Results
- Dependent Computations
- Caching Results
- Synchronization
- Shared State
- Task Coordination and supervision
- Resource Management
- Error Propagation

While sharing data between futures is useful, there are some things that we need to be mindful of when doing so. We can highlight them as we work through a simple example. First, we will be relying on the standard Mutex with the following import:

```
use std::sync::{Arc, Mutex};
use tokio::task::JoinHandle;
```

For our example, we will be using a basic struct that has a counter. One async task will be for increasing the count, and the other task will be decreasing the count. If both tasks hit the shared data the same number of times, the end result will be zero. Therefore, we need to build a basic enum to define what type of task is being run with the code below:

```
#[derive(Debug)]
enum CounterType {
    Increment,
    Decrement
}
```

We can then define our shared data struct with the following code:

```
struct SharedData {
    counter: i32,
}

impl SharedData {
    fn increment(&mut self) {
        self.counter += 1;
    }
    fn decrement(&mut self) {
        self.counter -= 1;
    }
}
```

Now that our shared data struct is defined, we can define our counter future with the code below:

```
struct CounterFuture {
    counter_type: CounterType,
    data_reference: Arc<Mutex<SharedData>>,
    count: u32
}
```

Here, we have defined the type of operation the future will perform on the shared data. We also have access to the shared data and a count to stop the future once the total number of executions of the shared data has happened for the future.

Now we can update the poll function inside our implementation of the `Future` trait. First we will cover getting access to the shared data with the following code:

```
    fn poll(mut self: Pin<&mut Self>, cx: &mut Co
        std::thread::sleep(Duration::from_secs(1
        let mut guard = match self.data_reference
            Ok(guard) => guard,
            Err(error) => {
                println!("error for {:?}: {}", se
                cx.waker().wake_by_ref();
                return Poll::Pending
            }
        };
}
```

We sleep to merely exaggerate the difference so it is easier for us to follow the flow of our program when running it. We then use a `try_lock`. This is because we are using the standard library `Mutex`. It would be nice to use the Tokio version of the `Mutex` but remember, our poll function cannot be async. Here lies a problem. If we acquire the `Mutex` using the standard lock function, we can block the thread until the lock is acquired. Remember, we could have one thread handling multiple tasks in our runtime. We would defeat the purpose of the async runtime if we locked the entire thread until the `Mutex` is acquired. Instead, the `try_lock` function attempts to acquire the lock, returning a result immediately in whether the lock was acquired or not. If the lock is not acquired, we print out the error to inform us for educational purposes, and then return a poll pending. This means that the future will be polled periodically until the lock is acquired so the future does not hold up the async runtime unnecessarily.

If we do get the lock we then move forward in our poll function to act on the shared data with the code below:

```
            let value = &mut *guard;
```

```
        let value = &mut *guard;

        match self.counter_type {
            CounterType::Increment => {
                value.increment();
                println!("after increment: {}", 
            },
            CounterType::Decrement => {
                value.decrement();
                println!("after decrement: {}", 
            }
        }
```

Now that the shared data has been altered, we can return the right response depending on the count with the following code:

```
        std::mem::drop(guard);
        self.count += 1;
        if self.count < 3 {
            cx.waker().wake_by_ref();
            return Poll::Pending
        } else {
            return Poll::Ready(self.count)
        }
```

We can see that we drop the guard before bothering to work out the return. This increases the time the guard is free for other futures, and enables us to update the `self.count`.

Running two different variants of our future can be done with the code below:

```
#[tokio::main]
async fn main() {
    let shared_data = Arc::new(Mutex::new(SharedI
    let counter_one = CounterFuture {
        counter_type: CounterType::Increment,
        data_reference: shared_data.clone(),
        count: 0
    };
    let counter_two = CounterFuture {
        counter_type: CounterType::Decrement,
        data_reference: shared_data.clone(),
        count: 0
    };
    let handle_one: JoinHandle<u32> = tokio::tasl
        counter_one.await
    });
    let handle_two: JoinHandle<u32> = tokio::tasl
        counter_two.await
    });
    tokio::join!(handle_one, handle_two);
}
```

Now we had to run the program a couple of times before we got an error that was printed out, but when an error acquiring the lock occurred, we got the following printout:

```
after decrement: -1
after increment: 0
error for Increment: try_lock failed because the
after decrement: -1
after increment: 0
after decrement: -1
after increment: 0
```

We can see that the end result is still zero so the error did not affect the overall outcome. The future just got polled again. While this has been interesting, we can mimic the exact same behaviour using a higher level abstraction from a third party crate such as Tokio for an easier/simplier implementation.

## High-level data sharing between futures

The future that we built in the previous section can be replaced with the following async function:

```
async fn count(count: u32, data: Arc<tokio::sync
                                counter_type: CounterTy
    for _ in 0..count {
            let mut data = data.lock().await;
            match counter_type {
                CounterType::Increment => {
                    data.increment();
                    println!("after increment: {}", c
                },
                CounterType::Decrement => {
                    data.decrement();
                    println!("after decrement: {}", c
                }
            }
            std::mem::drop(data);
            std::thread::sleep(Duration::from_secs(1)
        }
        return count
    }
```

Here we merely loop through the total number acquiring the lock in an async way and sleeping to enable the second future to operate on the shared data. This can simply be run with the code below:

```
    let shared_data = Arc::new(tokio::sync::Mutex::ne
    let shared_two = shared_data.clone();


    let handle_one: JoinHandle<u32> = tokio::task::sp
        count(3, shared_data, CounterType::Increment)
```

```
  });
  let handle_two: JoinHandle<u32> = tokio::task::s
      count(3, shared_two, CounterType::Decrement)
  });
  tokio::join!(handle_one, handle_two);
```

If we run this we get the exact same printout and behaviour as our futures in the previous section. However, it’s clearly simpler and easier to write. There are trade-offs to both approaches. For instance, if we just wanted to write futures that have the behaviour we have coded, it would make sense to use just an async function. However, if we needed more control over how a future was polled, or there we do not have access to an async implementation but we have a blocking function that tries, then it would make sense to write the poll function ourselves.

---

**HOW FUTURES IN RUST ARE DIFFERENT**

Other languages implement futures for async programming, and some of these languages rely on the callback model. The callback model uses a function that fires when when another function completes. This callback function is usually passed in as an argument to this function. This did not work for Rust because the callback model relies on dynamic dispatch, which means at runtime the exact function that was going to be called was determined at runtime as opposed to compile time. This produced additional overhead because the program had to work out what function to call at run-time. This violates the “zero-cost” approach and resulted in reduced performance.

Rust opted for an alternative approach with the aim of optimising runtime performance by using the `Future` trait which uses polls. The runtime is responsible for managing when to call polls. It does not need to schedule callbacks and worry about working out what function to call, instead it can use polls to see if the future is completed. This is more efficient because futures can be represented as a state machine in a single heap allocation, and the state machine captures local variables that are needed to execute the async function. This means there is one memory allocation per task, without any concern that the memory allocation will be the incorrect size. This decision is truly a testament to the Rust programming language, where the developers take the time to get the implementation right.

Often times we are not using async/await in isolation and we want to do something else when a task is complete. We can specify this with specific combinators like `and_then` or `or_else` which are provided by Tokio.

---

## How are Futures processed?

Let’s talk through how a Future gets processed by walking through the steps at a high level.

### Create a Future

We create a Future by defining an async function. When we call an async function, it returns a future. However, this future has not calculated anything yet, and the `await` is not called on the future yet.

**Spawn a Task**

We spawn a task with the future with `await`, which means we register with an executor. The executor then takes responsibility for taking the task to completion. To do this it maintains a queue of tasks.

**Polling the Task**

The executor processes the futures in the task by calling the poll method. This is a feature of the `Future` trait and will need to be implemented even if you are writing your own future. The future is either ready or not ready.

**Schedules the next execution**

If the Future is not ready, the executor places the task back into the queue to be executed in the future.

**Completion of Future**

At some point, all the future in the task will complete and the poll will return a ready. We should note that the result might be a Result or an Error. At this point, the executor can release any resources that it no longer needs and pass the results onwards.

**NOTE ON ASYNC RUNTIMES**

It must be noted that there are different variances of how async runtimes are implemented and Tokio's async runtime is much more complex and will be covered in chapter 7, Customising Tokio.

We have now covered why we pin futures to prevent undefined behaviour, context in futures, and data sharing between futures. To concrete what we have covered, we can move onto the next chapter where we implement what we covered in the tasks and futures sections in a practical project.

## Putting it all together

We have now covered tasks and futures and how they relate to async programming. To concrete what we have learned, we are now going to code a system that implements all that we have covered in this chapter. For our problem, we can conceive that we have a server or daemon that receives requests or messages. The data received needs to be logged to a file incase we need to inspect what happened. This problem means that we cannot predict when a log will happen. For instance, if we are just writing to a file in a single problem, our write operations can be blocking. However, receiving multiple requests from different programs can result in considerable overhead. It makes sense to send a write task to the async runtime and have the log written to the file when it is possible. It must be noted that this example is for educational purposes. Whilst async writing to a file might be useful for a local application, if you have a server that is designed to take a lot of traffic then you should explore database options.

In the example below, we are creating an audit trail for an application that logs interactions. This is an important part of many products that use sensitive data, for example in the medical field. We want to log the user’s actions but we do not want that logging action to hold up the program as we still want to facilitate a quick user experience. For this exercise to work you will need the following dependencies:

```
[dependencies]
tokio = { version = "1.0", features = ["full"] }
futures-util = "0.3"
```

Using these dependencies, we need to import the following:

```
use std::fs::{File, OpenOptions};
use std::io::prelude::*;
use std::sync::{Arc, Mutex};
use std::future::Future;
use std::pin::Pin;
use std::task::{Context, Poll};
use tokio::task::JoinHandle;
use futures_util::future::join_all;
```

At this stage pretty much all of the above should make sense and you should be able to work out what we are using them for. We will be referring to the handle throughout the program so we might as well define the type now with the line below:

```
type AsyncFileHandle = Arc<Mutex<File>>;
type FileJoinHandle = JoinHandle<Result<bool, St
```

Seeing as we do not want two tasks trying to write to the file at the same time it makes sense to ensure that only one task has mutable access to the file at one time.

We may want to write to multiple files. For instance, we might want to write all logins to one file, and error messages to another file. If you have medical patients in your system, you want to have a log file per patient (as you would probably inspect log files on a patient-by-patient basis), and you'd want to prevent unauthorized people looking at actions on a patient that they are not allowed to view. Considering there are needs for multiple files when logging, we can create a function that creates a file or obtains the handle of an existing file with the following code:

```
fn get_handle(file_path: &dyn ToString) -> Async
    match OpenOptions::new().append(true).open(f
        Ok(opened_file) => {
            Arc::new(Mutex::new(opened_file))
        },
        Err(_) => {
            Arc::new(Mutex::new(File::create(file
        }
    }
}
```

Now that we have our file handles, we need to work on our future that will write to the log. The fields of the future take the following form:

```
struct AsyncWriteFuture {
    pub handle: AsyncFileHandle,
    pub entry: String
}
```

We are now at a stage in our worked example where we can implement the `Future` trait for our `AsyncWriteFuture` struct and define the poll function. We will be using the same methods that we have covered in this chapter. Because of this, you can attempt to write the `Future` implementation and poll function yourself.

If you have attempted to write your own Future implementation, hopefully it will look similar to the implementation below:

```
impl Future for AsyncWriteFuture {
    type Output = Result<bool, String>;

    fn poll(self: Pin<&mut Self>, cx: &mut Contex
        let mut guard = match self.handle.try_loc
            Ok(guard) => guard,
            Err(error) => {
                println!("error for {} : {}", sel
                cx.waker().wake_by_ref();
                return Poll::Pending
            }
        };
        let lined_entry = format!("{}\n", self.en
        match guard.write_all(lined_entry.as_byte
            Ok(_) => println!("written for: {}",
            Err(e) => println!("{}", e)
        };
        Poll::Ready(Ok(true))
    }
}
```

The `Self::Output` type is not super important to get right. We just decided it would be nice to have a true value to say it was written but an empty bool or anything else works. The main focus of the above code is that we try to get the lock for the file handle. If we do not manage to get the lock we return a `Pending`. If we do get the lock we write our entry to the file.

When it comes to writing to the log, it is not very intuitive for other developers to construct our future and spawn off a task into the async runtime. They just want to write to the log file. Therefore we need to write our own `write_log` function that accepts the handle of the file and the line that is to be written to the log. Inside this function, we then spin off a tokio task and return the handle of the task. This is a good opportunity for you to attempt to write this function yourself.

If you attempted to write the `write_log` function yourself, it should take a similar approach to the code below:

```
fn write_log(file_handle: AsyncFileHandle, line:
    let future = AsyncWriteFuture{
        handle: file_handle,
        entry: line
    };
    tokio::task::spawn(async move {
        future.await
    })
}
```

It must be noted that even though the function does not have async in front of the function definition, it still behaves like an async function. We can call it and get the handle which we can then choose to await on later on in our program like so:

```
let handle = write_log(file_handle, name.to_stri
```

Or we can directly await on it like below:

```
let result = write_log(file_handle, name.to_stri
```

We can now run our async logging functions with the following main function:

```
#[tokio::main]
async fn main() {
    let login_handle = get_handle(&"login.txt");
    let logout_handle = get_handle(&"logout.txt"


    let names = ["one", "two", "three", "four", '
    let mut handles = Vec::new();


    for name in names {
        let file_handle = login_handle.clone();
        let file_handle_two = logout_handle.clone
        let handle = write_log(file_handle, name
        let handle_two = write_log(file_handle_tw
        handles.push(handle);
```

```
            handles.push(handle_two);
        }
        let _ = join_all(handles).await;
    }
```

If you look at the print out, you will see something similar to below. We have not included the whole printout for brevity. We can see that `six` can not be written to the file because of the `try_lock()` but `five` is written successfully.

```
. . .
error for six : try_lock failed because the opera
written for: five
error for six : try_lock failed because the opera
. . .
```

To make sure this has all worked in an async fashion, lets look at the `login.txt` file. Now your file may have a different order but mine looks like this:

```
one
four
three
five
```

```
two
six
```

---

**WARNING**

You can see here that the numbers which were in order in prior to entering the loop, have been logged out of order in an async way. This is an important observation to note. Because obtaining the lock is not deterministic, we cannot assume the order in which the log is written to. Locks are not just the only cause of this disorder. Delays in the reponse of any async operation can result in a disordered result, as when we are awaiting on one result, we process another. Therefore, when reaching for async solutions, we cannot rely on the results being processed in a certain order. If the order is essential, then keeping to one future and using data collections like queues will slow down the completion of all steps but will ensure that the steps are processed in the order you need them to be. In this case, if we needed to write to the file in order, we could wrap a queue in a `Mutex` and give one future the responsibility of checking the queue on every poll. Another future could then add to that queue. Increasing the number of futures with access to the queue on either side will compromise the assumption of order. While restricting the number of futures accessing the queue to one on each side reduces speed, we will still benefit if there are I/O delays. This is because the waiting of log inputs will not block our thread.

---

And there we have it! We have built an async logging function that is wrapped up in a single function making it easy to interface with. Hopefully this worked example has concreted the concepts that we have covered in this chapter.

## Summary

In this chapter, we've embarked on a journey through the landscape of asynchronous programming in Rust, highlighting the pivotal role of Tasks. These units of asynchronous work, grounded in futures, are more than just technical constructs; they are the backbone of efficient concurrency in practice. For instance, consider the everyday task of preparing coffee and toast. By breaking it down into async blocks, we have seen first-hand that multitasking in code can be as practical and timesaving as in our daily routines.

However, async is not deterministic, meaning the execution order of async tasks is not set in stone, which, while initially daunting, opens a playground for optimization. Cooperative multitasking isn't just a trick; it's a strategy to get the most out of our resources, something we've applied to accelerate our async operations.

We have also covered the sharing of data between tasks, which can be a double-edged sword. It's tempting to think that access to data is a nice tool for designing our solution, but without careful control, as demonstrated with our `Mutex` examples, it can lead to unforeseen delays and complexity. Here lies a valuable lesson: shared state must be managed, not just for the sake of order but for the sanity of our code's flow.

Finally, we looked into the `Future` trait was more than an academic exercise; it offered us a lens to understand and control the intricacies of task execution. It's a reminder that power comes with responsibility—the power to control task polling comes with the responsibility to understand the impact of each await expression. As we move forward, remember that implementing and utilizing async operations is not just about putting tasks into motion. It’s about grasping the underlying dynamics of each async expression. We can understand the underlying dynamics further by constructing our own async queues in the next chapter. There, you will gain the insights needed to define and control asynchronous workflows in Rust.

# Chapter 3. Building Our Own Async Runtime

While we have explored some basic async syntax and solved a problem using high-level async concepts, it can be understandable that you are still not completely sure what tasks and futures really are and how they flow through the async runtime, despite being able to manipulate async crates and solve problems using async concepts. Describing futures and tasks can be difficult to do, and hard to understand. In this chapter, we will concrete specifically what futures and tasks are, and how they run through an async runtime by building our own async queues with minimal dependencies. This async runtime will be customizable by choosing how many queues we have and how many consuming threads will be processing these queues. The implementation does not have to be uniform. For instance, we can have a low priority queue with two consuming threads, and a high priority queue having 5 consuming threads. We will then be able to choose which queue a future is going to be processed on. We will also be able to implement task stealing where consuming threads can steal tasks from other queues if their queue is empty. Finally, we will build our own macros to enable high level use of our async runtime.

By the end of this chapter, you will be able to implement custom async queues and fully understand how futures and tasks travel through the async runtime. You will also have the skills to customise async runtimes to solve problems that are specific to you that a standard out of the box runtime environment might not be able to handle. Even if you do not want to ever implement your own async queues ever again, you will have a deeper understanding of async runtimes so you can manipulate high-level async crates more effectively to solve problems. You will also understand the trade-offs of async code even when implementing async code at a high level. We will start our journey of building an async queues by defining the spawning of tasks as spawning a task is the entry point to the runtime.

We will start our journey of building an async runtime by defining the spawning of tasks as spawning a task is the entry point to the runtime.

## Building Our Own Async Queue

If we break down our implementation into steps, we will see first hand how our futures get converted into tasks and executed.

Before we write any code, we need the following dependencies:

```
[dependencies]
async-task = "4.4.0"
futures-lite = "1.12.0"
once_cell = "1.17.1"
flume = "0.10.14"
```

And we need to import the following into our `main.rs` file:

```
use std::{future::Future, panic::catch_unwind, th
use std::pin::Pin;
use std::task::{Context, Poll};
use std::time::Duration;

use async_task::{Runnable, Task};
use futures_lite::future;
use once_cell::sync::Lazy;
```

We will cover what we have imported when we use the imports as and when we use them in the code throughout the chapter so we understand the context of the imports.

We should start by building the task spawning function. This is where we pass a future into the function. The function then converts the future into a task and puts the task on the queue to be executed. At this point, it might seem like a complex function, so let us start with the signature below:

```
fn spawn_task<F, T>(future: F) -> Task<T>
where
    F: Future<Output = T> + Send + 'static,
    T: Send + 'static,
{
    . . .
}
```

Here we can see that this is a generic function. This makes sense as we do not want to be restricted to sending one type of future through our function. We can see that our future being passed in needs to have implemented the `Future` and `Send` trait. Recall that the `Future` trait denotes that our future is going to result in either an error or value `T`. Our future needs the `Send` trait because we are going to be sending our future into a different thread where the queue is based. The `Send` trait enforces constraints that ensure that our future can be safely shared between threads. The static means that our future does not contain any references that have a shorter lifetime than the static lifetime. This means that the future can be used for as long as the program is running. Ensuring this lifetime is essential as we cannot force programmers to await for a task to finish. If the developer never awaits for a task, the task could run for the entire lifetime of the program. Seeing as we cannot guarantee when a task is finished, we must ensure that the lifetime of our task is static. When browsing async code you may have seen async move utilised. This is where we move the ownership of variables used in the async closure to the task so we can ensure that the lifetime is static.

Now that we have defined our spawn task function signature, we move onto the first block of code in the spawn task function, which is defining the task queue with the following code:

```
static QUEUE: Lazy<flume::Sender<Runnable>> = La
    . . .
});
```

With the static we are ensuring that our queue is living throughout the lifetime of the program. This makes sense as we will want to send tasks to our queue throughout the lifetime of the program. The `Lazy` struct gets initialised on the first access of the struct. Once the struct is initialised, it is not initialised again. This is because we will be calling our task spawning function every time we send a future to the async runtime. If we initialise the queue every time we call the `spawn_task` function, we would be wiping the queue of previous tasks. Inside the `OnceCell`, we have the transmitting end of a channel which sends a `Runnable`.

A `Runnable` is a handle for a runnable task. Every spawned task has a single `Runnable` handle, and this handle only exists when the task is scheduled for running. The runnable handle essentially has the run function that polls the task's future once. Then the runnable is dropped. The runnable only appears again when the waker wakes the task in turn scheduling the task again. Recall from the previous chapter, if we do not pass the waker into our future, it would not be polled again. This is because the future cannot be woken to be polled again. We can build an async runtime that will poll futures no matter if a waker is present or not and we will explore this in chapter 10.

**NOTE**

It must be noted that we are using flume as opposed to standard library channels. This is because unbounded flume channels can hold an unlimited number of messages. The flume unbound channel also implements lock-free algorithms while at the time of writing this book, the standard library channels use a blocking mutex to synchronise access to the standard library channel's internal buffer. This means that the flume unbound channel is beneficial for highly concurrent programs as the channel to the queue could receive a large number of messages in parallel.

Now that we have defined our signature of the queue, we can look into the closure that we passed into the `Lazy` `OnceCell`. We need to create our channel, and create a mechanism for receiving futures send to that channel with the following code:

```
let (tx, rx) = flume::unbounded::<Runnable>();

thread::spawn(move || {
    while let Ok(runnable) = rx.recv() {
        println!("runnable accepted");
        let _ = catch_unwind(|| runnable.run());
    }
});
tx
```

We can see that after we have created the channel, we then spawn a thread which waits for incoming traffic. The waiting for the incoming traffic is blocking because remember, we are building the async queues to handle incoming async tasks, therefore we cannot rely on async in our thread. Once we have received our runnable, we run it in the `catch_unwind` function. We use the `catch_unwind` function because we do not know the quality of the code being passed to our async runtime. Ideally all Rust developers would handle possible errors properly, however, in case they do not, the `catch_unwind` function runs the code, and catches the error if it is thrown whilst the code is running returning a `Ok` or `Err` depending on the outcome. This is to prevent a badly coded future blowing up our async runtime. We then return the transmitter channel so we can send runnables to our thread.

We now have a thread that is running and waiting for tasks to be sent to that thread to be processed which we achieve with the code below:

```
let schedule = |runnable| QUEUE.send(runnable).ur
let (runnable, task) = async_task::spawn(future,
```

Here we have created closure that accepts a runnable and sends it to our queue. We then create the runnable and task by using the `async_task` spawn function. The spawn function essentially leads to an unsafe function that allocates the future onto the heap. The task and runnable returned from the spawn function essentially have a pointer to the same future.

**NOTE**

In this chapter we will not be building our own executor or code that creates a runnable or schedule the task. We will do this in chapter 10 where we build an async server completely from the standard library with no external dependencies.

Now that the runnable and task have pointers to the same future, we have to schedule the runnable to be run and return the task with the the following code:

```
runnable.schedule();
println!("Here is the queue count: {:?}", QUEUE.l
return task
```

When we schedule the runnable, we essentially put the task on the queue to be processed. If we did not schedule the runnable, the task would not be run, and our program would crash when we try to block the main thread to await on the task being executed because there is no runnable on the queue, but we still return the task. Remember the task and the runnable have pointers to the same future.

Now that we have scheduled our runnable to be run on the queue and returned the task, our basic async runtime is now complete. All we need to do is build some basic futures. We can construct our basic counter future with a sleep and print statement that we initially explored in the previous chapter with the code below:

```
struct CounterFuture {
    count: u32,
}
impl Future for CounterFuture {
    type Output = u32;

    fn poll(mut self: Pin<&mut Self>, cx: &mut Co
        -> Poll<Self::Output> {
        self.count += 1;
        println!("polling with result: {}", self
        std::thread::sleep(Duration::from_secs(1)
        if self.count < 3 {
            cx.waker().wake_by_ref();
            cx.waker().wake_by_ref();
        Poll::Pending
        } else {
            Poll::Ready(self.count)
        }
    }
}
```

We also create another future using the async function syntax with the following code:

```
async fn async_fn() {
    std::thread::sleep(Duration::from_secs(1));


    println!("async fn");
}
```

Before we progress we can take a detour to get an appreciation for how async sleep functions work. Throughout this chapter we are using sleep functions that block the executor. We are doing this for educational purposes so we can easily map how our tasks are processed in our runtime. However, if we want to build an efficient async sleep function, we need to lean into getting the executor to poll our sleep future, and return a pending if the time has not elapsed. First we need the instant to calculate the time elapsed and two fields to keep track of the sleep with the following struct:

```
use std::time::Instant;

struct AsyncSleep {
    start_time: Instant,
    duration: Duration,
}
impl AsyncSleep {
    fn new(duration: Duration) -> Self {
        Self {
            start_time: Instant::now(),
            duration,
        }
    }
}
```

We can then check the time elapsed between now and the start_time on every poll returning a pending if the time elapsed it not sufficient with the code below:

```
impl Future for AsyncSleep {
    type Output = bool;

    fn poll(self: Pin<&mut Self>, cx: &mut Contex
    -> Poll<Self::Output> {
        let elapsed_time = self.start_time.elapse
        if elapsed_time >= self.duration {
            Poll::Ready(true)
        } else {
            cx.waker().wake_by_ref();
            Poll::Pending
        }
    }
}
```

This will not block the executor with idle sleep time. Because sleep is only one part of a process, we can call the await on our future inside async blocks for our async sleep future as seen in the following implementation:

```
let async_sleep = AsyncSleep::new(Duration::from_
let asnyc_sleep_handle = spawn_task(async {

    async_sleep.await;
    . . .
});
```

---

**NOTE**

Like most things in programming, there is always a tradeoff. Imagine if there are a lot of tasks in-front of the sleep task, there is an increased chance that the async sleep task might effectively wait longer than the duration required before finishing, as it might have to wait for other tasks to complete before it can complete between every poll. If you have an operation where X seconds has to happen between two steps, then a blocking sleep might be a better option, but you are going to clog up your queues quickly if you have a lot of these tasks.

---

Going back to our blocking example, we can now run some futures in our runtime with the following main function:

```rust
fn main() {
    let one = CounterFuture { count: 0 };
    let two = CounterFuture { count: 0 };
    let t_one = spawn_task(one);
    let t_two = spawn_task(two);
    let t_three = spawn_task(async {
        async_fn().await;
        async_fn().await;
        async_fn().await;
        async_fn().await;
    });
    std::thread::sleep(Duration::from_secs(5));
    println!("before the block");
    future::block_on(t_one);
    future::block_on(t_two);
    future::block_on(t_three);
}
```

There is some repetition in this main function but it is needed in order for us to get a sense of how the async runtime we just built processes futures. We can see that we have multiple futures in our async for task three. We then wait 5 seconds and print before employing the `block_on` functions so we can get a sense of how our system runs before we call the `block_on` functions.

Running our program will give us the following lengthy but essential printout in the terminal:

```
Here is the queue count: 1
Here is the queue count: 2
Here is the queue count: 3
runnable accepted
polling with result: 1
runnable accepted
polling with result: 1
runnable accepted
async fn
async fn
before the block
async fn
async fn
runnable accepted
polling with result: 2
runnable accepted
polling with result: 2
runnable accepted
polling with result: 3
runnable accepted
polling with result: 3
```

Our printout essentially gives us a timeline of our async runtime. We can see that our queue is being filled up with the three tasks that we have spawned, and our runtime is processing them in order in an async manner before we call our block_on functions. Even after the first block_on function is called, which blocks on the first task we spawned, the two different counter tasks are being processed at the same time. It also must be noted that the async function that we built and called four times in our third task was essentially blocking. There was no await within the async function, so even though we use the await syntax like so:

```
async {
    async_fn().await;
    async_fn().await;
    async_fn().await;
    async_fn().await;
}
```

The stack of async_fn futures just blocked the thread processing the task queue until the entire task was completed. We can also see that when a poll resulted in pending, the task was then put back on the queue to be polled again.

Our async runtime can be summarised by the diagram in Figure 3-1:

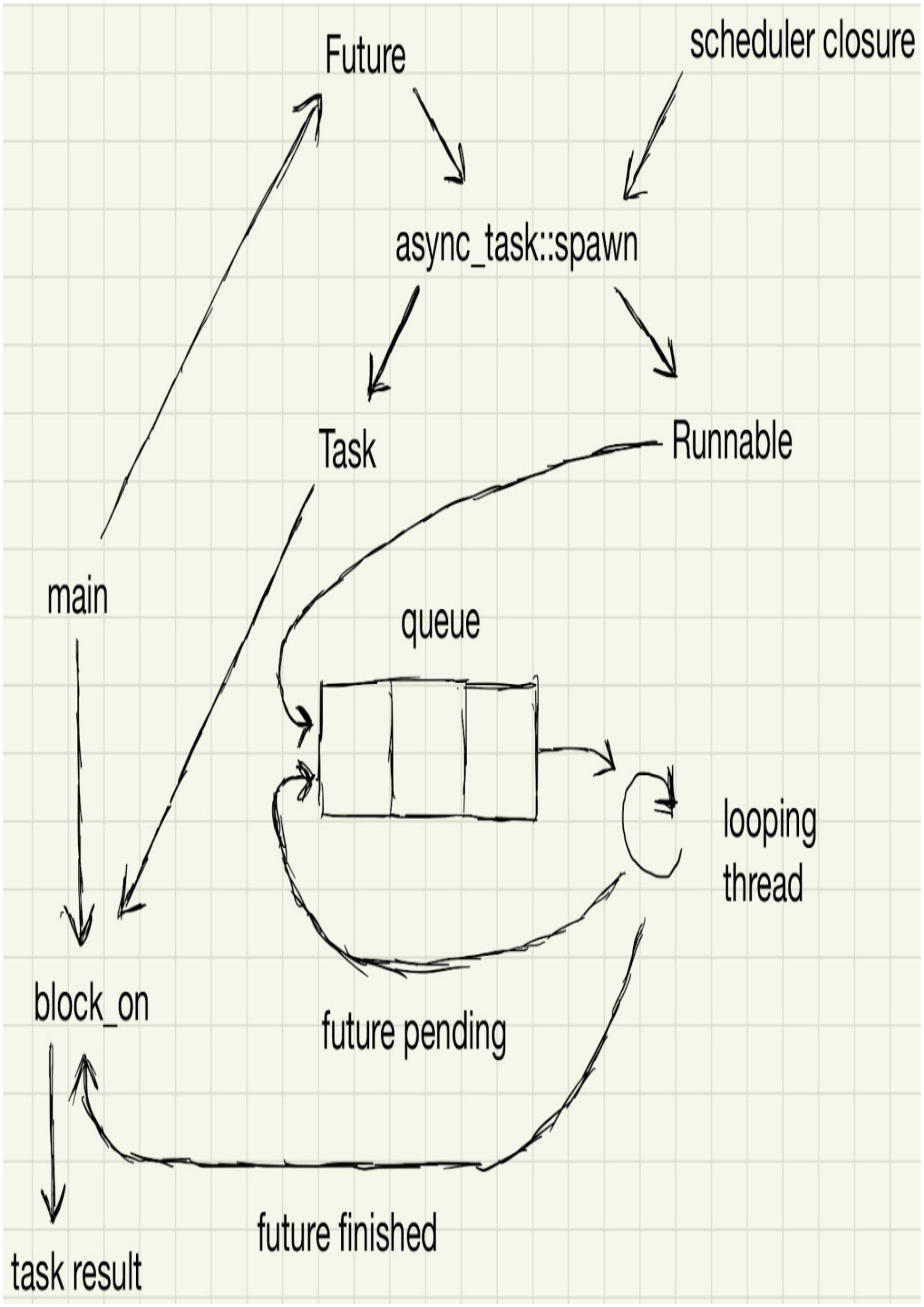

Future
scheduler closure
async_task::spawn
Task
Runnable
main
queue
looping
thread
block_on
future pending
future finished
task result

Figure 3-1. Overview of our async runtime

We can attempt to describe what is happening with an analogy. Let us say that we have a dirty coat that needs cleaning. The label inside the coat containing cleaning instructions and material content is essentially the future. We then walk into the dry cleaners and hand over the coat with the instructions. The worker at the cleaner makes the coat runnable by putting a plastic cover on the coat and giving it a number. The worker also gives you a ticket with the number which is essentially like the task that the main function gets. We then go about our day doing things whilst the coat is being cleaned. If the coat is not cleaned the first time round it keeps going through cleaning cycles until the coat is clean. We then come back with our ticket and hand it over to the worker. This is the same stage as the `block_on` function. If we have really taken our time before coming back, the coat might be clean already and we can take it and go on with our day. If we go to the cleaners too early, the coat will not be clean and we then have to wait until the coat is cleaned before taking it home. The clean coat is the result.

Right now our async runtime only has one thread processing the queue. This would be like us only insisting on one worker at the cleaners. This is not the most efficient use of resources available as most CPUs have multiple cores. Considering this, it would be useful to explore how to increase the number of workers and queues to increase our capacity to handle more tasks.

## Increasing Workers And Queues

To increase our number of threads working on the queue we can add another thread consuming from the queue with a cloned receiver of our queue channel with the code below:

```
let (tx, rx) = flume::unbounded::<Runnable>();

let queue_one = rx.clone();
let queue_two = rx.clone();

thread::spawn(move || {
    while let Ok(runnable) = queue_one.recv() {
        let _ = catch_unwind(|| runnable.run());
    }
});
thread::spawn(move || {
    while let Ok(runnable) = queue_two.recv() {
        let _ = catch_unwind(|| runnable.run());

    }
});
```

If we send tasks through the channel, the traffic will generally be distributed across both threads. If one thread is blocked with a CPU intensive task, the other thread will continue to work through tasks. Recall that our first chapter proved that CPU intensive tasks can be run in parallel using threads with our Fibonacci number example. We can have a more ergonomic approach to building a thread pool with the following code:

```
for _ in 0..3 {
    let receiver = rx.clone();
    thread::spawn(move || {
        while let Ok(runnable) = receiver.recv()
            let _ = catch_unwind(|| runnable.run
        }
    });
}
```

We could offload CPU intensive tasks to our threadpool and continue working through the rest of the program, blocking the program when we need a result from the task. Whilst this is not really the spirit of async programming as we use async programming to optimise the juggling of IO operations, it is a useful approach to remember if certain problems can be solved by offloading CPU intensive tasks early on in the program.

**WARNING**

You may have come across sayings along the lines of “async is not for computationally heavy tasks”. Async is merely a mechanism and you can use it for what you want as long as it makes sense. However, the warning that async code is not for computationally heavy tasks is not without merit as you have to be careful. For instance, if you are using your async runtime to handle income requests as most web frameworks do, then chucking computationally heavy tasks onto the async runtime queue could potentially block your ability to process incoming requests until those computations are done.

Now that we have explored multiple workers, we should really look into multiple queues.

## Passing Tasks To Different Queues

One of the reasons why we would want to have multiple queues, is that we might have different priorities for tasks. In this section, we are going to build a high priority queue with two consuming threads, and a low priority queue which has one consuming thread. To support multiple queues, we are going to need the following enum to classify the type of queue the task is destined for:

```
#[derive(Debug, Clone, Copy)]
enum FutureType {
    High,
    Low
}
```

We then need our futures to yield the future type when passed into our spawn function by utilising the trait:

```
trait FutureOrderLabel: Future {
    fn get_order(&self) -> FutureType;
}
```

We then need to add the future type for our future by adding an extra field as seen below:

```
struct CounterFuture {
    count: u32,
    order: FutureType
}
```

Our poll function stays the same so there is no need to revisit that. However, we do need to implement the FutureType trait for our Future with the following code:

```
impl FutureOrderLabel for CounterFuture {
    fn get_order(&self) -> FutureType {
        self.order
    }
}
```

Our Future is now ready to be processed, we now need to reformat our async runtime to utilise future types. The signature for our `spawn_task` function essentially stays the same apart from the additional trait as seen below:

```
fn spawn_task<F, T>(future: F) -> Task<T>
where
    F: Future<Output = T> + Send + 'static + Futu
    T: Send + 'static,
{
    . . .
}
```

We can now define our two queues. At this point in time, you can attempt to code these two queues yourself before moving forward as we have covered all that we need to build the two queues. If you attempted to build the queues, hopefully they took a form similar to the following code:

```
static HIGH_QUEUE: Lazy<flume::Sender<Runnable>>
    let (tx, rx) = flume::unbounded::<Runnable>(
    for _ in 0..2 {
        let receiver = rx.clone();
        thread::spawn(move || {
            while let Ok(runnable) = receiver.re
                let _ = catch_unwind(|| runnable
            }
        });
    }
    tx
});
static LOW_QUEUE: Lazy<flume::Sender<Runnable>> =
    let (tx, rx) = flume::unbounded::<Runnable>(
    for _ in 0..1 {
        let receiver = rx.clone();
        thread::spawn(move || {
            while let Ok(runnable) = receiver.re
                let _ = catch_unwind(|| runnable
            }
        });
    }

    tx
});
```

We can see that the low priority queue has one consuming thread, and the high priority queue has two consuming threads. We now need to route futures to the right queue. This can be done by defining an individual runner closure for each queue, and then passing the correct closure based on the future type with the code below:

```
let schedule_high = |runnable| HIGH_QUEUE.send(ru
let schedule_low = |runnable| LOW_QUEUE.send(runn

let schedule = match future.get_order() {
    FutureType::High => schedule_high,
    FutureType::Low => schedule_low
};
let (runnable, task) = async_task::spawn(future,
runnable.schedule();
return task
```

We can now create a future that can be inserted into the selected queue with the following code:

```
let one = CounterFuture { count: 0 , order: Futu
```

However, we now have a problem. Let us imagine that we are in a situation where loads of low priority tasks get created and it just so happens that there are no high priority tasks. We would have one consumer thread working on all the tasks whilst the other two consumer threads are just sitting idle. We would essentially be working at a third capacity. This is where task stealing comes in.

NOTE

It must be noted that we do not need to write our own async runtime queues to have control over the distribution of tasks. For instance, Tokio will allow you to have control over the distribution of tasks using `LocalSet`.

## Task Stealing

Task stealing is when consuming threads steal tasks from other queues when their own queue is empty. The figure 3-2 shows task stealing in relation to our current async system.

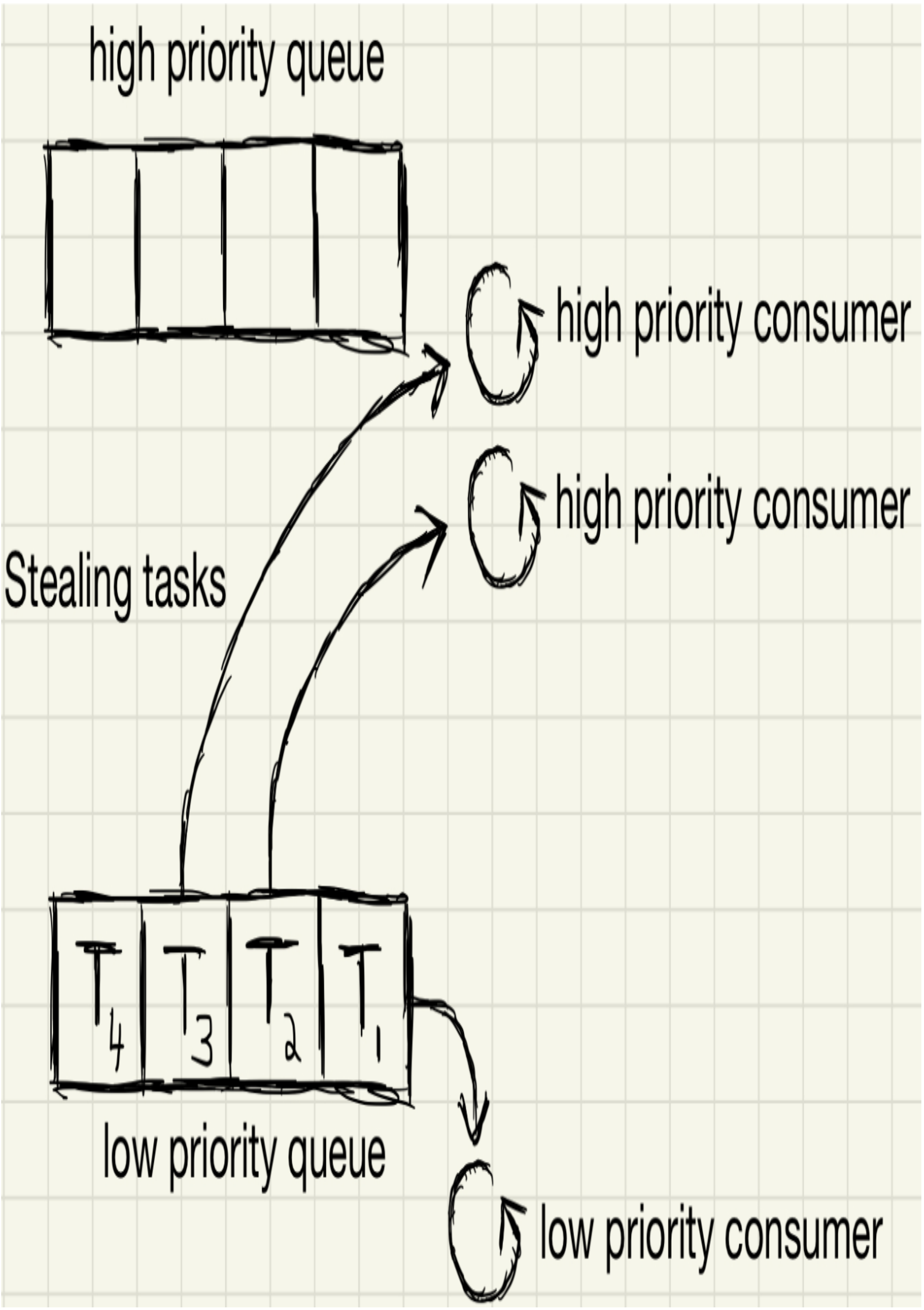

high priority queue
high priority consumer
high priority consumer
Stealing tasks
$T_4$
$T_3$
$T_2$
$T_1$
low priority queue
low priority consumer

Figure 3-2. Task stealing

We also must appreciate that stealing can go the other way. If the low priority queue is empty, we would want the low priority consumer thread to steal tasks from the high priority queue.

In order to achieve task stealing, we need to pass in channels for the high and low priority queues into both queues. Before we can define our channels, we need the import below:

```
use flume::{Sender, Receiver};
```

If we used the standard library for our Sender and Receiver we would not be able to send the Sender or Receiver over to other threads. With flume, we make both of the channels static that are lazily evaluated inside our spawn_task function with the following code:

```
static HIGH_CHANNEL: Lazy<(Sender<Runnable>, Rece
    {flume::unbounded::<Runnable>()}
);
static LOW_CHANNEL: Lazy<(Sender<Runnable>, Recei
    {flume::unbounded::<Runnable>()}
);
```

Now that we have our two channels, we need to define our high priority queue consumer threads to carry out the following steps for each iteration in an infinite loop:

1. Check the `HIGH_CHANNEL` for a message
2. If the `HIGH_CHANNEL` does not have a message, check the `LOW_CHANNEL` for a message
3. If the `LOW_CHANNEL` does not have a message, wait for 100 milliseconds for the next iteration

---

**NOTE**

We could park our threads if they are idle and wake these parked threads when they need to process incoming tasks. This can save excessive looping and sleeping when there are no tasks to be processed. We will cover thread parking in relation to async queues in chapter 10.

---

Our high priority queue can carry out these steps with the following code:

```
static HIGH_QUEUE: Lazy<flume::Sender<Runnable>>
    for _ in 0..2 {
        let high_receiver = HIGH_CHANNEL.1.clone
        let low_receiver = LOW_CHANNEL.1.clone()
        thread::spawn(move || {
            loop {
                match high_receiver.try_recv() {
                    Ok(runnable) => {
                        let _ = catch_unwind(||
                    },
                    Err(_) => {
                        match low_receiver.try_re
                            Ok(runnable) => {
                                let _ = catch_unw
                            },
                            Err(_) => {
                                thread::sleep(Du
                            }
                        }
                    }
                };
            }
        });
    }
    HIGH_CHANNEL.0.clone()
});
```

Our low priority queue would merely just swap steps one and two around, and return the `LOW_CHANNEL.0.clone()`. We now have both queues pulling tasks from their own queues first, and then pulling tasks from other queues when there are no tasks on their own queue. When there are no tasks left, we then have our consumer threads slow down.

**WARNING**

Remember that the queues and channels are lazy in their evaluation. This means that a task needs to be sent to the queue in order for the queue to start running. If you just send tasks to the low priority queue and never to the high priority queue, the high priority queue will never startup and therefore will never steal tasks from the low priority queue.

At this milestone, we can sit back and think about what we have done. We have created our own async runtime where we have defined different queues! We now have fine grain control over how our async tasks run. We must also note that we may not want task stealing. For instance, if we put CPU intensive tasks onto the high priority queue, and lightweight networking tasks on the low priority queue, we would not want the low priority queue stealing tasks from the high priority queue otherwise we run the risk of shutting down our network processing due to the low priority queue consumer threads being held up on CPU intensive tasks.

Whilst it was interesting to have the trait constraint and see how it could be implemented onto our future, we are now disadvantaged because we cannot pass in simple async blocks or async functions because they do not have the `FutureOrderLabel` trait implemented. Other developers will just want a nice interface to run their tasks. Could you imagine how bloated our code would be if we had to implement the `Future` trait for every async task and implement the `FutureOrderLabel` on all of them? We need to refactor our `spawn_task` function for a better developer experience.

## Refactoring Our Spawn Task Function

When it comes to allowing async blocks and async functions into our spawn task function, we need to remove the FutureOrderLabel trait constraint and add another argument for the order giving the following function signature:

```
fn spawn_task<F, T>(future: F, order: FutureType
where
    F: Future<Output = T> + Send + 'static,
    T: Send + 'static,
{
```

We then need to update the selection of the right scheduling closure in the spawn_task function with the following code:

```
let schedule = match order {
    FutureType::High => schedule_high,
    FutureType::Low => schedule_low
};
```

Still we do not want our developers stressing over the order, so we can create a macro for the `spawn_task` function with the code below:

```
macro_rules! spawn_task {
    ($future:expr) => {
        spawn_task!($future, FutureType::Low)
    };
    ($future:expr, $order:expr) => {
        spawn_task($future, $order)
    };
}
```

What this macro does is allow us to just pass in the future. If we only pass in the future, we then pass in the Low priority type meaning that this is the default type. If the order is passed in, then the order is passed into the `spawn_task` function. From this macro Rust works out that you need to at least pass in the future expression, and will not compile unless the future is supplied. We now have a more ergonomic way of spawning tasks as we can see with the following example:

```
fn main() {
    let one = CounterFuture { count: 0 };
    let two = CounterFuture { count: 0 };

    let t_one = spawn_task!(one, FutureType::Higl
    let t_two = spawn_task!(two);
    let t_three = spawn_task!(async_fn());
    let t_four = spawn_task!(async {
        async_fn().await;
```

```
            async_fn().await;
        }, FutureType::High);

        future::block_on(t_one);
        future::block_on(t_two);
        future::block_on(t_three);
        future::block_on(t_four);
    }
```

We can see that our macro is flexible, and a developer using it could casually spawn tasks without thinking about it, but they also have the ability to state that the task is high priority if needed. We can also see that we can pass in async blocks and async functions because these are just syntactic sugar for futures. However, we are repeating ourselves when blocking the main to wait on multiple tasks. We need to create our own join macro to prevent this repetition.

## Creating Our Own Join Macro

When it comes to creating our own join macro, we need to essentially accept a range of tasks, and call the block_on function. We can define our own join macro with the following code:

```
macro_rules! join {
    ($($future:expr),*) => {
        {
            let mut results = Vec::new();
            $(
                results.push(future::block_on($fu
            )*
            results
        }
    };
}
```

It is essential that we keep the order of the results the same as the order of the futures passed in otherwise the user will have no way of knowing which result belongs to which task. Also note that our join macro will only return one type, so, we use our join macro like the following:

```
let outcome: Vec<u32> = join!(t_one, t_two);
let outcome_two: Vec<()> = join!(t_four, t_three
```

The outcome is a vector of the outputs of the counters, and `outcome_two` is a vector of the outputs for the async functions that didn't return anything. As long as we have the same return type, this code will work.

We must remember that our tasks are being directly run. There could be an error in the execution of the task. To return a vector of `` Result`s, we `` `can create a `try_join` macro with the code below:

```
macro_rules! try_join {
    ($($future:expr),*) => {
        {
            let mut results = Vec::new();
            $(
                let result = catch_unwind(|| futu
                results.push(result);
            )*
            results
        }
    };
}
```

This is very similar to as our join! macro, but will return results of the tasks.

We now nearly have everything that we need to run async tasks on our runtime in an ergonomic way with task stealing and different queues. Although the spawning of our tasks is not ergonomic, we still need a nice interface to configure our runtime environment.

## Configuring Our Runtime

You may remember that the queue is lazy, meaning that it will not start until it is called. This directly affects our task stealing. The example we gave was that if no tasks were sent to the high priority queue then the high priority queue would not start and therefore would not steal tasks from the low priority queue if empty and vice versa. Configuring a runtime to get things going and refine the number of consuming loops is not an unusual way of solving this problem. For instance, we can look at the following Tokio example of starting their runtime:

```rust
use tokio::runtime::Runtime;

// Create the runtime
let rt = Runtime::new().unwrap();

// Spawn a future onto the runtime
rt.spawn(async {
    println!("now running on a worker thread");
});
```

At the time of writing this, the above example is in the Tokio documentation of the runtime struct. Tokio also uses procedural macros to set up the runtime but procedural macros are beyond the scope of this book. More on procedural macros can be found in the Rust Documentation. For our runtime, we can build a basic runtime builder where we can define the number of consuming loops on the high and low priority queues.

We first start with our runtime struct with the code below:

```
struct Runtime {
    high_num: usize,
    low_num: usize,
}
```

The high number is the number of consuming threads for the high priority queue, and the low number is the number of consuming threads for the low priority queue. We then implement the following functions for our runtime:

```
impl Runtime {
    pub fn new() -> Self {
        let num_cores = std::thread::available_pa

        Self {
            high_num: num_cores - 2,
            low_num: 1,
        }
    }
    pub fn with_high_num(mut self, num: usize) ->
        self.high_num = num;
        self
    }
    pub fn with_low_num(mut self, num: usize) ->
        self.low_num = num;
        self
    }
    pub fn run(&self) {
        . . .
    }
}
```

Here we have a standard way of defining the numbers based on the number of available cores on the computer that is running our async program. We then have the options to define the low and high numbers ourselves if we want. We then have a run function which defines the environment variables for the numbers and then spawn two tasks to both queues to setup the queues using the code below:

```
    pub fn run(&self) {
        std::env::set_var("HIGH_NUM", self.high_num.t
        std::env::set_var("LOW_NUM", self.low_num.to_

        let high = spawn_task!(async {}, FutureType:
        let low = spawn_task!(async {}, FutureType::L
        join!(high, low);
    }
```

We can see that we use our join, so that after the run function has been executed, both of our queues are ready to steal tasks.

Before we try our runtime, we need to utilise these environment variables to establish the number of consumer threads for each queue. In our spawn_task function, we just refer to the environment variable inside each queue definition like the following:

```
static HIGH_QUEUE: Lazy<flume::Sender<Runnable>>
    let high_num = std::env::var("HIGH_NUM").unw

    for _ in 0..high_num {
        . . .
```

Same goes for the low queue. We can then define our runtime with default numbers in our main function before anything else with the code below:

```
Runtime::new().run();
```

Or with custom numbers with the following:

```
Runtime::new().with_low_num(2).with_high_num(4).
```

We are now capable of running our spawn_task function and join macro whenever we want throughout the rest of the program.

We now have our own runtime that is configurable with two different types of queues and task stealing!

We now have nearly everything tied up. However, there is one last concept we need to cover before finishing the chapter and this is background processes.

## Running Background Processes

Background processes are tasks that execute in the background periodically for the entire lifetime of the program. These processes can be used for monitoring, maintenance tasks such as database cleanup, or log rotation, data updates to ensure that the program always has access to the latest information etc. Implementing a basic background process as a task in the async runtime will show us how to handle our long running tasks.

Before we handle the background task, we need to create a future that will never stop being polled. At this stage in the chapter you should be able to build this yourself and you should attempt to do this before moving on. If you have attempted to build your own future, it should take the following form if the process being carried out is blocking:

```
#[derive(Debug, Clone, Copy)]
struct BackgroundProcess;

impl Future for BackgroundProcess {
```

```
impl Future for BackgroundProcess {
    type Output = ();

    fn poll(self: Pin<&mut Self>, cx: &mut Conte
            -> Poll<Self::Output> {
        println!("background process firing");
        std::thread::sleep(Duration::from_secs(1
        cx.waker().wake_by_ref();
        Poll::Pending
    }
}
```

Your implementation might be different but the key takeaway is that we are always returning a pending.

What we need to acknowledge here is that if we drop a task in our main function, the task being executed in the async runtime will be cancelled and will not be executed, therefore our background task must be present throughout the entire lifetime of the program. We need to send the background task at the very beginning of the main function right after we have defined our runtime with the following code:

```
Runtime::new().with_low_num(2).with_high_num(4).
let _background = spawn_task!(BackgroundProcess{
```

And our background process will run periodically throughout the entire lifetime of our program.

However, this is not ergonomic. For instance, let us say that a struct or function could create a background running task. We do not need to try and juggle the task around the program so it does not get dropped cancelling the background task. We can remove the need for juggling tasks to keep the background task running by using the detach method as seen below:

```
Runtime::new().with_low_num(2).with_high_num(4).r
spawn_task!(BackgroundProcess{}).detach();
```

Detach essentially moves the pointer in the task into an unsafe loop that will poll the task and schedule it until it is finished. The pointer associated with the task in the main function is then dropped, dropping the need for keeping hold of the tasks in the main function.

## Summary

In this chapter we have now implemented our own runtime and learnt a lot in the process. We initially built a basic async runtime environment that accepted futures and created tasks and runnables. The runnables were put on the queue to be processed by consumer threads and the task was returned back to the main function where we can block the main function to wait for the result of the task. Here we spent some time concreting the steps that the futures and tasks go through in the async runtime. We then implemented different queues which had different numbers of consuming threads for the different queues and used this pattern to implement task stealing for situations when a queue was empty. We then created our own macros for users so they could easily spawn tasks and join them.

The nuance that task stealing introduces highlights the true nature of async programming. An async runtime is merely a tool that you use to solve your problems. There is nothing stopping you from having one thread on a queue for accepting network traffic and 5 threads processing long CPU intensive tasks if you have a program that has little traffic but that traffic requests the triggering of long running tasks. In this case you would not want your network queue to steal from the CPU intensive queue. Of course, your solutions should strive to be sensible. However, with deeper understanding of the async runtime you're using, comes the ability to solve complex problems in interesting ways. The async runtime we built certainly is not the best out there. Established async runtimes have teams of very smart people ironing out problems and edge cases. However, now you are at the end of this chapter, you should understand the need to read around your chosen runtime and its working when solving problems with that chosen runtime. It also must be noted that the simple implementation of the async_task queue with the flume channel can be used in production.

In the next chapter, we will cover integretting HTTP with our own async runtime.

# Chapter 4. Integrating Networking Into Our Own Async Runtime

In the previous chapter we built our own async runtime to understand how async runtimes run under the hood. However, we only did some basic sleep and print operations. Focusing on simple operations is useful initially as we do not have to split focus between how the async runtime works and another concept. However, simple sleep and print functions are limiting. In this chapter we build on the async runtime we defined in the previous chapter and integrate networking protocols so they can run on our async runtime.

By the end of this chapter you will be able to integrate the Hyper Crate for HTTP requests into our runtime using traits. This means you will be able to take this example and integrate other third party dependencies into our async runtime via traits after reading the documentation of that crate. Finally, we will go to a lower level by implementing the Mio crate to directly poll sockets in our futures. This will give you an understanding of how to utilize fine grained control over how the socket is polled, read, and written to in our async runtime. With this exposure and some further external reading, you will be able to implement our own custom networking protocols.

Before we progress through the chapter we will need the following additional dependencies alongside the dependencies we used in the previous chapter:

```toml
[dependencies]
hyper = { version = "0.14.26",
features = ["http1", "http2", "client", "runtime'
smol = "1.3.0"
anyhow = "1.0.70"
async-native-tls = "0.5.0"
http = "0.2.9"
tokio = "1.14.0"
```

As you can see we are going to be using Hyper for this example. This is to give you a different set of tools to what we have used in previous examples, and to demonstrate that tools like tokio are layered in other commonly used libraries.

## Building An Executor For Hyper

Hyper is an open-source library that implements HTTP in Rust. It allows the Client to talk to web services and a Server for building fast dynamic web services. If we look at Hyper official documentation or various online tutorials, we can get the impression that we can perform a simple get request using the Hyper crate with the following code:

```
use hyper::{Request, Client};

let url = "http://www.rust-lang.org";

let uri: Uri = url.parse().unwrap();

let request = Request::builder()
    .method("GET")
    .uri(uri)
    .header("User-Agent", "hyper/0.14.2")
    .header("Accept", "text/html")
    .body(hyper::Body::empty()).unwrap();

let future = async {
    let client = Client::new();
    client.request(request).await.unwrap()
};
let test = spawn_task!(future);
let response = future::block_on(test);
println!("Response status: {}", response.status(
```

However, if we run the tutorial code, we would get the following error:

```
thread '<unnamed>' panicked at 'there is no react
running, must be called from the context of a Tok
```

This is because under the hood, Hyper by default runs on the Tokio runtime. If you were going to use the Reqwest or other popular crates chances are you will get a similar error. This is not the end of the world, we are able to integrate a crate like Hyper into our own async runtime.

Before we connect our runtime to a Hyper client, we need to import the following into our program:

```
use std::net::Shutdown;
use std::net::{TcpStream, ToSocketAddrs};
use std::pin::Pin;
use std::task::{Context, Poll};
use anyhow::{bail, Context as _, Error, Result};
use async_native_tls::TlsStream;
use http::Uri;
use hyper::{Body, Client, Request, Response};
use smol::{io, prelude::*, Async};
```

We are going to need an executor and a connector. There are a few steps, but we are starting with the executor in this section. We can build our own executor with the code below:

```
struct CustomExecutor;

impl<F: Future + Send + 'static> hyper::rt::Execu
    fn execute(&self, fut: F) {
        spawn_task!(async {
            println!("sending request");
            fut.await;
        }).detach();
    }
}
```

In the above code, we define our custom executor, and the behavior of the execute function. Inside our execute function, we merely call our spawn task macro. Inside we essentially create an async block and await for the future that was passed into the execute function. We must note that we employ the detach function otherwise the channel will be closed and we will not continue with our request due to the task moving out of scope and simply being dropped. If we recall from the previous chapter, the detach function will send the pointer of the task to a loop to be polled until the task has finished before dropping the task.

We now have a custom executor that we can pass into the hyper client. However, our hyper client will still fail to make the request because the connection will still be looking for a Tokio runtime, therefore, we need to build our own async connection.

## Building An HTTP Connection

When it comes to networking requests the protocols are very well defined and standardized. For instance, a TCP connection has a three step handshake to establish a connection before sending packets of bytes through that connection. There is zero benefit in implementing the TCP connection from scratch unless you have very specific needs that the standardized connection protocols cannot provide. In Figure 4.1, we can see that HTTP and HTTPS are built on top of TCP:

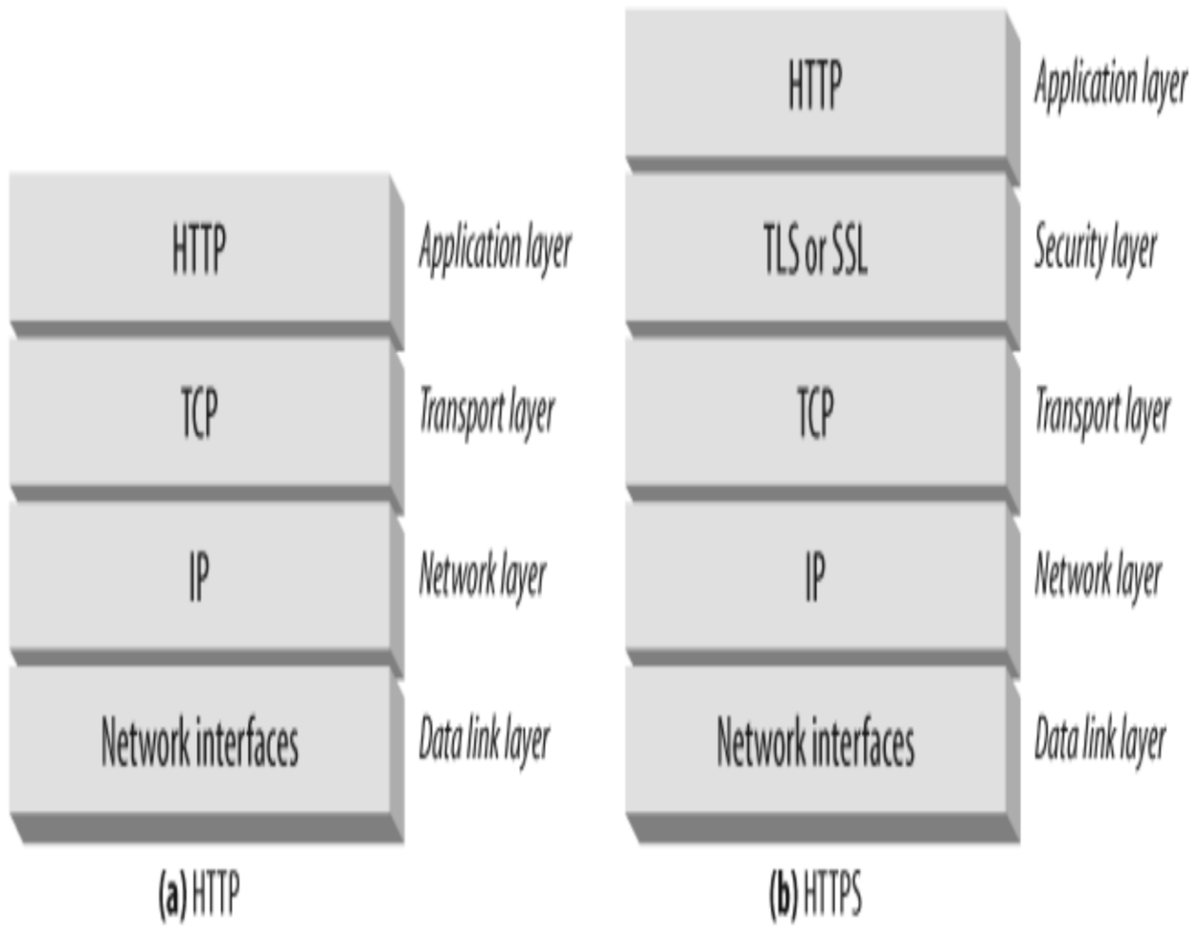


Figure 4-1. Networking protocol layers

With HTTP we are sending over a body, header etc. With HTTPS there are even more steps, where a certificate is checked and sent over to the client before the client starts sending over data. This is because the data needs to be encrypted. Considering all the back and forth in these protocols and waiting for responses, networking requests are a sensible target for async. We cannot get rid of the steps in networking without losing security and assurance that the connection is made. However, we can release the CPU from networking requests when waiting for responses with async.

For our connector, we are going to support HTTP and HTTPS so we are going to need the following enum:

```
enum CustomStream {
    Plain(Async<TcpStream>),
    Tls(TlsStream<Async<TcpStream>>),
}
```

The Plain variant is an async TCP stream. Considering Figure 4.1, we can deduce that the Plain variant supports HTTP requests. With the Tls variant we remember that HTTPS is merely a TLS layer between the TCP and the HTTP means that our Tls variant supports HTTPS.

We can now use this custom steam enum to implement the hyper Service trait for a custom connector strut with the code below:

```
#[derive(Clone)]
struct CustomConnector;

impl hyper::service::Service<Uri> for CustomConne
    type Response = CustomStream;

    type Error = Error;
    type Future = Pin<Box<dyn Future<Output = Res
                        Self::Response, Self
                    >>;
    fn poll_ready(&mut self, _: &mut Context<'_>
        -> Poll<Result<(), Self::Error>> {
        . . .
    }
    fn call(&mut self, uri: Uri) -> Self::Future
        . . .
    }
}
```

The Service trait is essentially for defining the future for the connection. We can see our connection is a thread safe future that returns our stream enum. This enum is either an async TCP connection, or an async TCP connection that is wrapped in a TLS stream.

We can also see that our `poll_ready` function just returns a ready. The `poll_ready` function is used by Hyper to check if a service is ready to process requests. If we return a pending, then the task will be polled until the service becomes ready. We return an error when the service can no longer process requests. Because we are using the `Service` trait for a client call, we will always return ready for the `poll_ready`. If we were implementing the `Service` trait for a server, we could have the following `poll_ready` function:

```
fn poll_ready(&mut self, cx: &mut Context<'_>)
    -> Poll<Result<(), Error>> {
    Poll::Ready(Ok(()))
}
```

We can see that our `poll_ready` function merely returns that the future is ready. We could ideally not bother defining the `poll_ready` function as our implementation makes calling it redundant. However, the `poll_ready` function is a requirement for the `Service` trait.

We can now move onto the response function which is the `call` function. The `poll_ready` function needs to return an Ok before we can use the `call` function. Our `call` function has the following outline:

```
fn call(&mut self, uri: Uri) -> Self::Future {
    Box::pin(async move {
        let host = uri.host().context("cannot pa

        match uri.scheme_str() {
            Some("http") => {
                . . .
            }
            Some("https") => {
                . . .
            }
            scheme => bail!("unsupported scheme:
        }
    })
}
```

We remember the pin and the async block returns a future. So, whatever the return statement is of the async block will be our pinned future. For our HTTPS block, we have build a future with the code below:

```
let socket_addr = {
    let host = host.to_string();
    let port = uri.port_u16().unwrap_or(443);

    smol::unblock(move || (host.as_str(), port).t
        .await?
        .next()
        .context("cannot resolve address")?
};
let stream = Async::<TcpStream>::connect(socket_a
let stream = async_native_tls::connect(host, stre
Ok(CustomStream::Tls(stream))
```

The port is 443 because this is the standard port for HTTPS. We then pass a closure into the unblock function. The closure returns the socket address. The unblock function runs blocking code on a threadpool so we can have the async interface on blocking code. So while we are resolving the socket address, we can free up the thread to do something else. We then connect our TCP steam, and then connect our steam to our native TLS. Once our connection is achieved, we finally return the CustomStream enum.

When it comes to building our HTTP code it is exactly the same, however, the port is 80 instead of 443, and the TLS connection is not required resulting in us returning a `Ok(CustomStream::Plain(stream))`.

Our `call` function is now defined. However, if we try to make a HTTPS call to a website with our stream enum or connection struct at this point, we will get an error message stating that we have not implemented the `AsyncRead` and `AsyncWrite` `Tokio` traits for our steam trait. This is because hyper requires these traits to be implemented in order for our connection enum to be used.

## Implementing The Tokio AsyncRead Trait

The `AsyncRead` trait is similar to the `std::io::Read` trait but integrates with asynchronous tasks systems. When implementing our `AsyncRead` trait, we only have to define the `poll_read` function which returns a poll enum as a result. If we return a `Poll::Ready` we are saying that the data was immediately read and placed into the output buffer. If we return a `Poll::Pending`, we are saying that no data was read into the buffer that we provided. We are also saying that the I/O object is not currently readable, but may become readable in the future. The return og `Pending` results in the current future's task being scheduled to be unparked when the object is readable. The final `Poll` enum variant that we can return is a `Ready` but with an error which would usually be standard I/O errors.

Our implementation of our `AsyncRead` trait is defined in the code below:

```
impl tokio::io::AsyncRead for CustomStream {
    fn poll_read(
      mut self: Pin<&mut Self>,
        cx: &mut Context<'_>,
        buf: &mut tokio::io::ReadBuf<'_>,
    ) -> Poll<io::Result<()>> {
        match &mut *self {
            CustomStream::Plain(s) => {
                Pin::new(s)
                    .poll_read(cx, buf.initialize
                    .map_ok(|size| {
                        buf.advance(size);
                    })
            }
            CustomStream::Tls(s) => {
                Pin::new(s)
                    .poll_read(cx, buf.initialize
                    .map_ok(|size| {
                        buf.advance(size);
                    })
            }
            }
        }
    }
```

Our different streams essentially have the same treatment, we just pass in either the async TCP stream or an async TCP stream with TLS. We then pin this stream and execute the stream's `poll_read` function which essentially performs a read and returns a `Poll` enum with the size of how much the buffer grew due to the read. Once the `poll_read` is done, we then execute the `map_ok` which takes in an `FnOnce(T)` which is either a function or a closure that can only be called once. The `map_ok` also references itself which is the result from the `poll_read`. If the `Poll` result is ready but with an error, then the `Poll` ready with the error is merely returned. If the `Poll` result is `Pending` then pending is returned. We can see that we pass in the context into the `poll_read` so a waker is utilized if we have a pending result. If we have a `Ready` with a `Ok` result, then the closure is called with the result from the `poll_read` `Ready` `Ok` is returned from the `map_ok` function. Our closure passed into our `map_ok` function merely advances the buffer.

There is a lot going on under the hood, but essentially, our stream is pinned, a read is performed on the pinned stream, and if the read is successful, we advance the size of the filled region of the buffer as the read data is now in the buffer. The polling in the `poll_read`, and the matching of the poll enum in the `map_ok`, enable this read process to be compatible with an async runtime.

So, we can now read into our buffer in an async manner but we also need to write in an async way for our HTTP request to be complete.

## Implementing The Tokio AsyncWrite Trait

The `AsyncWrite` trait is a trait that is similar to the `std::io::Write` but interacts with asynchronous task systems. It write bytes in an asynchrnous manner, and like the `AsyncRead` we just implemented, comes from Tokio.

When implementing the AsyncWrite trait, we will need the following outline:

```
impl tokio::io::AsyncWrite for CustomStream {
    fn poll_write(
        mut self: Pin<&mut Self>,
        cx: &mut Context<'_>,
        buf: &[u8],
    ) -> Poll<io::Result<usize>> {
        . . .
    }
    fn poll_flush(mut self: Pin<&mut Self>, cx: &
    fn poll_flush(mut self: Pin<&mut Self>, cx: 
    -> Poll<io::Result<()>> {
        . . .
    }
    fn poll_shutdown(mut self: Pin<&mut Self>, c
    -> Poll<io::Result<()>> {
        . . .
    }
}
```

The `poll_write` function should not be a surprise, however, we can also note that we have `poll_flush` and `poll_shutdown` functions. We can see that all functions return a variant of the `Poll` enum and accept the context. Therefore, we can deduce that all functions are able to put the task to sleep to be woken again to check if the future is ready for shutting down, flushing, and writing to the connection.

We should start with our `poll_write` function which has the code body below:

```
match &mut *self {
    CustomStream::Plain(s) => Pin::new(s).poll_w
    CustomStream::Tls(s) => Pin::new(s).poll_writ
}
```

This here we can see is merely matching the stream, pinning the stream, and executing the `poll_write` function of the stream. At this point in the chapter it should not come as a surprise that the `poll_write` function tries to write bytes from the buffer into an object. Like the read, if the write is successful, the number of bytes written is returned. If the object is not ready for writing, we will get a `Pending`, and if we get a 0, then this usually means that the object is no longer able to accept bytes.

Inside the `poll_write` function of the stream, a loop is executed where the mutable reference of the I/O handler is obtained. The loop then repeatedly tries to write to the underlying I/O until all the bytes from the buffer are written. Each write attempt has a result which is handled. If the error of the write is a `io::ErrorKind::WouldBlock`, this means that the write could not complete immediately and the loop repeats until the write is complete. If the result is any other error, the loop waits for the resource to be available again by returning a pending for the future to be polled again at a later time.

Now that we have written the `poll_write`, we can define the body of the `poll_flush` function with the following code:

```
match &mut *self {
    CustomStream::Plain(s) => Pin::new(s).poll_fl
    CustomStream::Tls(s) => Pin::new(s).poll_flus
}
```

This has the same outline as our poll_write function. However, in this case we merely call the poll_flush function on our stream. A flush is like a write except we ensure that all the contents of the buffer immediately reaches the destination. The underlying mechanism of the flush is exactly the same as the write with the loop, however, the flush function will be called in the loop as opposed to the write function.

We can now move onto our final function which is the shutdown function which takes the form below:

```
match &mut *self {
    CustomStream::Plain(s) => {
        s.get_ref().shutdown(Shutdown::Write)?;
        Poll::Ready(Ok(()))
    }
    CustomStream::Tls(s) => Pin::new(s).poll_clos
}
```

We can see that there is a slight variation in the way we implement the different types of our custom stream. The plain stream is just shut down directly. Once the plain stream is shut down we return a `Poll` that is ready. However, the TLS stream is an async implementation by itself. Because the TLS is async, we need to pin it to avoid it being moved in memory as it could be put on the task queue a number of times until the poll is complete, and we call the `poll_close` function which will return a poll result by itself.

We have now implemented our async read and write traits for our hyper client. All we need to do now is connect and run HTTP requests to test our implementation.

## Connecting And Running Our Client

At this stage we are merely wrapping up what we have done and testing it. We can create our connection and send request function with the code below:

```
impl hyper::client::connect::Connection for Custo
    fn connected(&self) -> hyper::client::connect
        hyper::client::connect::Connected::new()
    }
}

async fn fetch(req: Request<Body>) -> Result<Resp
    Ok(Client::builder()
        .executor(CustomExecutor)
        .build::<_, Body>(CustomConnector)
        .request(req)
        .await?)
}
```

Now all we need to do is run our HTTP client on our async runtime in the main function with the following code:

```
fn main() {
    Runtime::new().with_low_num(2).with_high_num

    let future  = async {
        let req = Request::get("https://www.rust
                                         .body(Bo
                                         .unwrap

        let response = fetch(req).await.unwrap()


        let body_bytes = hyper::body::to_bytes(re
                                .await.unwrap();
        let html = String::from_utf8(body_bytes.t
        println!("{}", html);
    };
    let test = spawn_task!(future);
    let _outcome = future::block_on(test);
}
```

And here we have it, we can run our code and get the HTML code from the Rust website. We can now say that our async runtime can communicate to the internet in an async manner, but what about accepting requests? We have already covered implementing traits from other crates to get an async implementation. We would get diminishing educational returns if we filled the rest of this chapter with implementing such traits to get a TCP listener running in an async manner. Instead, we are going to go one step lower, and directly listen to events in sockets with the Mio crate.

## Introduction To Mio

When it comes to implementing async functionality with sockets, we cannot really get any lower than MIO without directly calling the OS. We can see in Figure 4.2, Tokio is built on top of Mio.

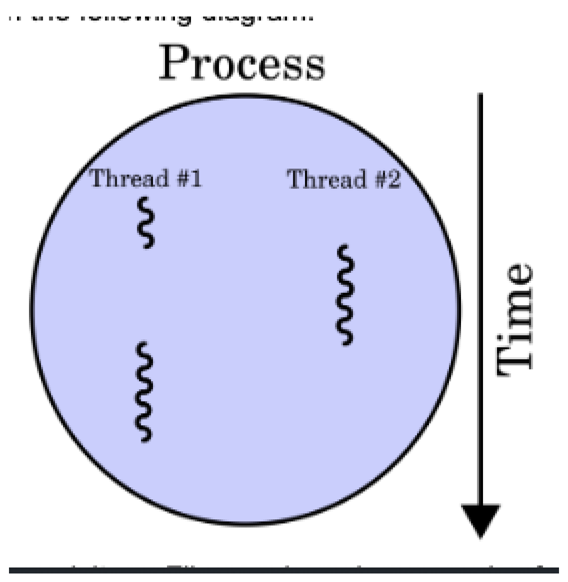


Figure 4-2. Layers of the Tokio async runtime

In the previous parts of the chapter we connected Hyper to our runtime. In order to get the full picture, it makes sense for us to now explore Mio and integrate this in our runtime. Before we proceed, we need the following dependency in the `Cargo.toml`:

```
mio = {version = "0.8.8", features = ["net"]}
```

We also need the imports below:

```
use mio::net::{TcpListener, TcpStream};
use mio::{Events, Interest, Poll as MioPoll, Toke
use std::io::{Read, Write};
use std::time::Duration;
use std::error::Error;

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

**WARNING**

It must be stressed that our exploration of Mio in this chapter is not an optimal approach to creating a TCP server. If you want to create a production server, you will need to take an approach similar to the hyper example below:

```
#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error + Send + S
    let addr = SocketAddr::from(([127, 0, 0, 1], 3000));
    let listener = TcpListener::bind(addr).await?;
    loop {
        let (stream, _) = listener.accept().await?;
        let io = TokioIo::new(stream);
        tokio::task::spawn(async move {
            if let Err(err) = http1::Builder::new()
                .serve_connection(io, service_fn(hello))
                .await
            {
                println!("Error serving connection: {:?}", err);
            }
        });
    }
}
```

Here we can see that the main thread is waiting for incoming data, and when incoming data arrives, a new task is spawned to handle that data. This keeps the listener ready to accept more incoming data. Whilst our Mio examples will get us to understand how polling TCP connections work, it is most sensible to use the listener that the framework or library gives you when building a web application. We will discuess some web concepts to give context to our example, but a comprehensive overview of web development is beyond the scope of this book.

Now that we have laid all the groundwork, we can move onto polling TCP sockets in futures.

## Polling Sockets in Futures

Mio is built for handling many different sockets (thousands). Therefore we need to identify which socket triggered the notification. Tokens enable us to do this. When we register a socket with the event loop, we pass it a token and that token is returned in the handler. The token is merely a struct tuple around `usize`. This is because every OS allows a pointer amount of data to be associated with a socket. So in the handler we can have a mapping function where the token is the key and we map it with a socket.

Mio is not using callbacks here because we want a zero cost abstraction and tokens were the only way of doing that. We can build callbacks, streams, and futures on top of Mio.

With tokens we now have the following steps:

1. register sockets with event loop
2. wait for socket readiness
3. lookup socket state using token
4. operate on the socket
5. repeat

In our example we are just going to define our tokens with the code below as our example is simple negating the need for mapping:

```
const SERVER: Token = Token(0);
const CLIENT: Token = Token(1);
```

Here, we just have to ensure that our tokens are unique. The integer passed into the `Token` is merely used to differentiate the `Token` from other tokens. Now that we have our tokens, we define the future that is going to poll the socket using the following struct:

```
struct ServerFuture {
    server: TcpListener,
    poll: MioPoll,
}
impl Future for ServerFuture {


    type Output = String;



    fn poll(mut self: Pin<&mut Self>, cx: &mut Co
        -> Poll<Self::Output> {
        . . .
    }
}
```

Here we are using the `TcpListener` to accept incoming data, and the `MioPoll` to poll the socket and tell the future when the socket is readable. Inside our future poll function we can define the events and poll the socket with the code below:

```
let mut events = Events::with_capacity(1);

let _ = self.poll.poll(
    &mut events,
    Some(Duration::from_millis(200))
).unwrap();


for event in events.iter() {
    . . .
}
cx.waker().wake_by_ref();
return Poll::Pending
```

The poll will extract the events from the socket into the events iterator. We also set the socket poll to timeout after 200 milliseconds. This means that if there are no events in the socket, we merely proceed without any events, and return a pending. This means that we will continue polling until we get an event.

When we do get events, we loop through them. In the preceding code we have set the capacity to one but we can increase the capacity to handle multiple events if needed. When processing an event, we need to clarify what type of event it is. For our future, we need to ensure that the socket is readable, and that the token is the SERVER token with the following code:

```
if event.token() == SERVER && event.is_readable(
    let (mut stream, _) = self.server.accept().u
    let mut buffer = [0u8; 1024];
    let mut received_data = Vec::new();

    loop {
        . . .
    }
    if !received_data.is_empty() {
        let received_str = String::from_utf8_los
        return Poll::Ready(received_str.to_strin
    }
    cx.waker().wake_by_ref();
    return Poll::Pending
}
```

The event is readable if there is data in the socket. If our event is the right event, we extract the `TcpStream`, define a `data_recieved` collection on the heap with the `Vec`, using the buffer slice to perform the reads. If the data is empty, we return a Pending so we can poll the socket again if the data is not there. We then convert it to a string and return that string with a `Ready`. This means that our socket listener is finished once we have the data.

---

**NOTE**

If we wanted our socket to be continuously polled throughout the lifetime of our program, we would spawn a detached task where we pass the data into an async function to handle the data as seen before.

```
if !received_data.is_empty() {
    spawn_task!(some_async_handle_function(&received_data)).detacl
    return Poll::Pending
}
```

---

In our loop, we read the data from the socket with the following code:

```
loop {
    match stream.read(&mut buffer) {
        Ok(n) if n > 0 => {
            received_data.extend_from_slice(&buf
        }
        Ok(_) => {
            break;
        }
        Err(e) => {
            eprintln!("Error reading from stream
            break;
            }
        }
    }
```

It does not matter if the received message is bigger than the buffer, our loop will continue to extract all the bytes to be processed, adding them onto our `Vec`. If there are no more bytes, we can stop our loop to process the data.

We now have a future that will continue to be polled until it accepts data from a socket. Once it has received the data it will then terminate. We can also make this future continuously poll the socket. Considering this, there is an argument that we would use this continuous polling future to keep track of thousands of different sockets if needed. We would have one socket per future and spawn thousands of futures into our runtime. Now that we have our `TcpListener` logic defined, we can move onto our client logic to send data over the socket to our future.

## Sending Data Over Socket

For our client, we are going to run everything in the main function which have the signature below:

```
fn main() -> Result<(), Box<dyn Error>> {
    Runtime::new().with_low_num(2).with_high_num
    . . .
    Ok(())
}
```

In our main, we initially create our listener and our stream for the client with the following code:

```
let addr = "127.0.0.1:13265".parse()?;
let mut server = TcpListener::bind(addr)?;
let mut stream = TcpStream::connect(server.local_
```

For our example we are just requiring one stream however, we can create multiple streams if we need them. We then register our server with a Mio poll and use the server and poll to spawn the listener task with the following code:

```
let poll: MioPoll = MioPoll::new()?;
poll.registry()
.register(&mut server, SERVER, Interest::READABLE

let server_worker = ServerFuture{
    server,
    poll,
};
let test = spawn_task!(server_worker);
```

Now our task is continuously polling the TCP port for incoming events. We then create another poll with the CLIENT token for writeable events. This means that if the socket is not full, we can write to it. If the socket is full, the socket is no longer writable and needs to be flushed. Our client poll takes the following form:

```
let mut client_poll: MioPoll = MioPoll::new()?;
client_poll.registry()
.register(&mut stream, CLIENT, Interest::WRITABLE
```

---

**NOTE**

With Mio we can also create polls that can trigger if the Socket is readable or writable with the example below:

```
.register(&mut server, SERVER, Interest::READABLE | Interest::WRI
```

---

Now that we have created our poll, we can wait for the socket to become writable before writing to it. We do this with the poll call below:

```
let mut events = Events::with_capacity(128);

let _ = client_poll.poll(
    &mut events,
    None
).unwrap();
```

It must be noted that there is a None for the timeout. This means that our current thread will be blocked until an event is yielded by the poll call. Once we have the event, we send a simple message to the socket with the following code:

```
for event in events.iter() {
    if event.token() == CLIENT && event.is_writal
        let message = "that's so dingo!\n";
        let _ = stream.write_all(message.as_bytes
    }
}
```

The message is sent, and therefore, we can block our thread and then print out the message with the code below:

```
let outcome = future::block_on(test);
println!("outcome: {}", outcome);
```

When running the code, you might get the following printout:

```
Error reading from stream: Resource temporarily u
outcome: that's so dingo!
```

It works but we get the initial error. This can be a result of non-blocking TCP listeners. Mio is non-blocking. The "Resource temporarily unavailable" error is usually down to no data being available in the socket. This can happen when the TCP stream is created but it is not a problem as we handle these errors in our loop and we returning, a `Pending` so the socket continues to be polled.

---

**NOTE**

With Mio's polling feature we have essentially implemented async communication through a TCP socket. We can also use Mio to send data between processes via a `UnixDatagram`. UnixDatagrams are sockets that are restricted to sending data on the same machine. Because of this, UnixDatagrams are faster, require less context switching, and UnixDatagrams do not have to go through the network stack.

---

## Summary

We have finally managed to get our async runtime to do something apart from sleep and print. In this chapter we have explored how traits can help us integrate third party crates into our runtime, and we have gone lower to poll TCP sockets using Mio. When it comes to getting a custom async runtime running there is not anything else standing in your way as long as you have access to trait documentation. If you really have to get a firm grip of your async knowledge so far, you are in the position to create a basic web server that handles different endpoints. It would be difficult to implement all your communication in Mio but using it just for async programming is much easier. Hyper's `HttpListener` will cover the protocol complexity so you can focus on how to pass the requests as async tasks, and the response to the client. For our journey in this book, we are exploring async programming as opposed to web programming. Therefore, we are going to move onto how we implement async programming to solve specific problems. We start in the next chapter with coroutines.

# Chapter 5. Coroutines

At this stage in the book you should hopefully be comfortable with async programming. When you see await syntax in code now, you know what is happening under the hood with futures, tasks, threads and queues. However, what about the building blocks of async programming? What if we can get our code to pause and resume without having to use an async runtime? Furthermore, what if we could use this pause and resume mechanism to test our code using normal tests. These tests can explore how code behaves under different polling orders and configurations. Our pause and resume mechanism can also be the interface between synchronous code and async. This is where coroutines come in.

By the end of this chapter, you should be able to define a coroutine and point to how they can be used. You should be able to integrate coroutines into your own programs to keep memory consumption low for tasks that would require large amounts of memory. You will also be able to mimic async functionality without an async runtime using coroutines, and implement a basic executor. This results in us essentially getting async functionality in our main thread without the need of an async runtime. You will also be able to gain fine-grained control over when and what order our coroutines get polled.

---

**NOTE**

At the time of writing this book we are using the coroutine syntax in nightly Rust. The syntax might have changed, or the coroutine syntax might have made its way to stable Rust. While changing syntax is annoying, it is the fundamentals of coroutines that we are covering in the book and syntax changes will not affect the overall implementation of coroutines and how we can use them.

---

## Introduction to Coroutines

Before we can explore the coroutines fully, you need to understand what a coroutine is first, and why you'd want to use them.

### What are Coroutines?

A coroutine is a special type of program that can pause its execution and then at a later point in time resume from where it left off. This is different to regular subroutines (like functions) which run to completion and typically return a value or throw an error.

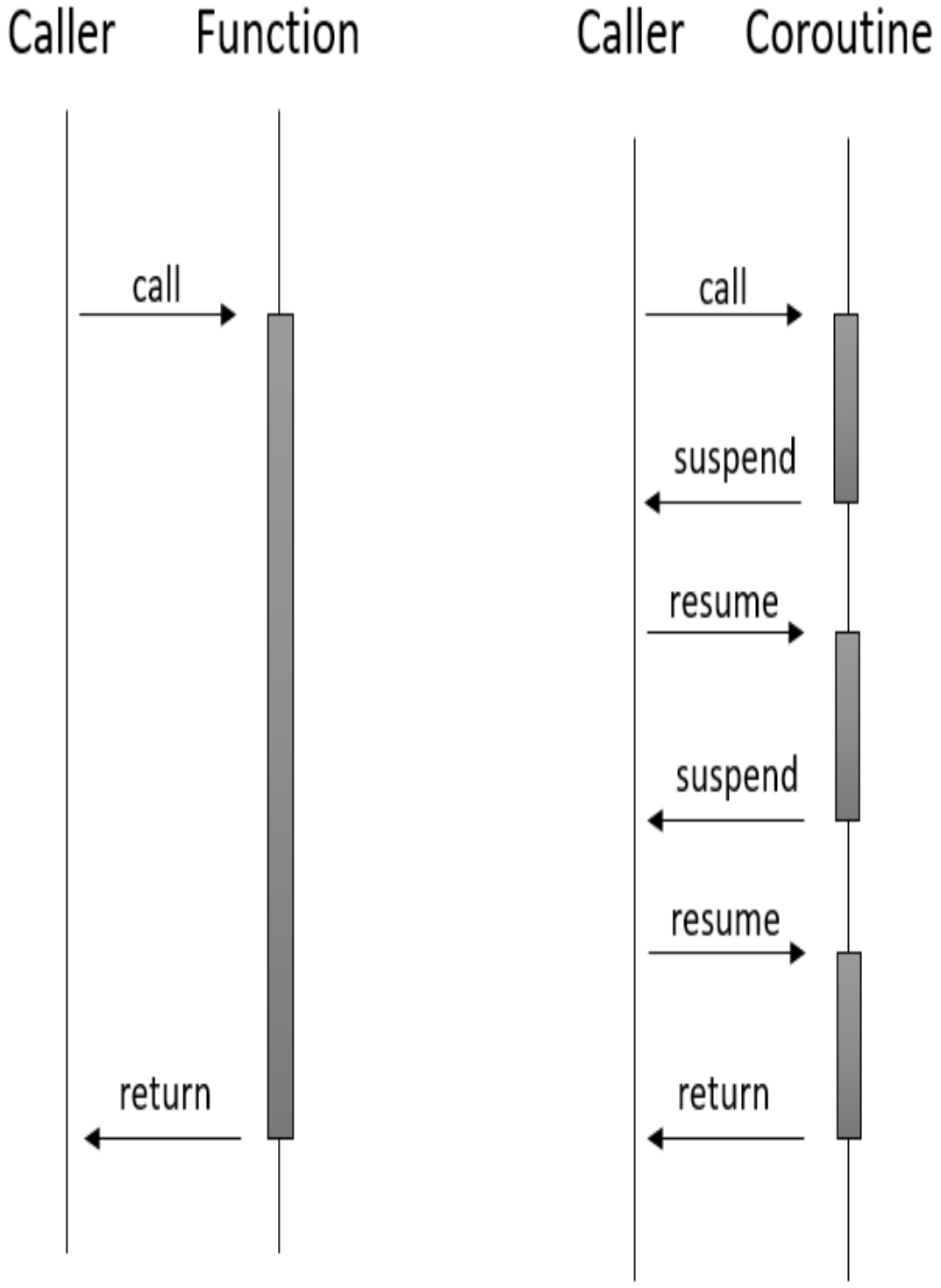

Caller
Function
call
return
Caller
Coroutine
call
suspend
resume
suspend
resume
return

Figure 5-1. Co-Routine (right) Regular Subroutine (left)

Let’s compare a coroutine to a subroutine. Once a subroutine starts, it executes from the start to the end and any particular instance of a subroutine returns just one time. A coroutine is different as it can exit in a number of ways. It can finish like a subroutine, but it can also exit by calling another coroutine (called yielding) and then returning later to the same point. This means that it keeps track of the state - i.e. stores the state when they are paused.

A coroutine is not unique to rust and there are many different implementations of coroutines in different languages. They all share the same basic features to allow for pausing and resuming execution:

- Non-blocking - coroutines must be non-blocking meaning that when they get paused, they do not block the thread of execution.
- Stateful - coroutines must be stateful meaning that they can store their state when they are paused and then resume from that state when they are resumed. There is no need to start from the beginning.
- Cooperative - coroutines are cooperative meaning that they are able to pause and be resumed at a later stage in a controlled manner.

Now let’s think about how coroutines are similar to threads. On the face of it, they seem quite similar, executing tasks, and pausing/resuming later on. The difference is in the scheduling. A thread is scheduled *pre-emptively* -

meaning that the task is interrupted by an external scheduler, with the aim that the task will be resumed later. Whereas a coroutine is cooperative. They are able to pause or yield to another coroutine, without a scheduler or the operating system getting involved. Using coroutines sounds great, so why bother with async/await at all? Like with anything in programming, there are trade-offs here. Let’s run through some pros to start with. Coroutines remove the need for syncing primitives like mutexes as the coroutines are running in the same thread. This can make it easier to understand and write the code, not an insignificant consideration. Switching back and forth between coroutines in one thread is much cheaper than switching between threads. It is particularly useful for tasks where you might spend a lot of time waiting. Imagine if you need to keep track of changes in 100 files. It would be a pain to have the OS schedule 100 threads that loop through and check each file. It is computationally expensive to context switch. It would be more efficient and easier instead to have 100 coroutines checking if the file they are monitoring has changed, and then sending that file into a thread pool to be processed when it has.

The major downside to just using coroutines in just one thread, is that you are not taking advantage of the power of your computer. Running it in a program in one thread, means you are not splitting tasks across multiple cores. Now we know what coroutines are, we should explore why we should use coroutines.

## Why use Coroutines?

At a high level, coroutines enable us to suspend an operation, yield control back to the thread that executed the coroutine, and then resume the operation when needed. This sounds a lot like async. An async task can yield control for another task to be executed through polling. With multithreading, we can merely send data to the thread and check in on the thread as and when we need through channels and data structures wrapped in `Sync` primitives. Coroutines, on the other hand, enable us to suspend an operation and resume with the waker without needing an async runtime or thread. This may seem a little abstract, but we should concrete the advantages of using a coroutine with a simple file write example. Let's imagine that we are getting a lot of integers as and when, and we need to write them to a file. Perhaps we are getting numbers from another program, and we can't wait for all the numbers to be received before we start writing as this would take up too much memory.

Before we write the code for our demonstration, we are going to need the following imports.

```
#![feature(coroutines)]
#![feature(coroutine_trait)]
#![feature(associated_type_bounds)]
use std::fs::{OpenOptions, File};
use std::io::{Write, self};
use std::time::Instant;
use rand::Rng;

use std::ops::{Coroutine, CoroutineState};
use std::pin::Pin;
```

We will also need the rand crate in our `Cargo.toml`. These imports might seem a little excessive for a simple write exercise but we will see how they are utilised when we move through our example. The macros are there to merely allow the experimental features. For our simple write file example, we have the following function:

```
fn append_number_to_file(n: i32) -> io::Result<(
    let mut file = OpenOptions::new()
        .create(true)
        .append(true)
        .open("numbers.txt")?;
    writeln!(file, "{}", n)?;
    Ok(())
}
```

This function opens the file and writes to it. We now want to test it and measure our performance, so we generate 200 thousand random integers and loop through these integers, writing to the file. We also time this operation with the following code:

```
fn main() {
    let mut rng = rand::thread_rng();
    let numbers: Vec<i32> = (0..200000).map(|_|

    let start = Instant::now();
    for &number in &numbers {
        if let Err(e) = append_number_to_file(nu
            eprintln!("Failed to write to file:
        }
    }
    let duration = start.elapsed();

    println!("Time elapsed in file operations is
}
```

At the time of writing this book, the total test took 3.8 seconds to complete. This is not very performant; however, this is because we are opening the file every time which has the overhead of checking permissions and updating the metadata of the file.

Now, we can now employ a coroutine to handle the writing of the integers. First, we define the struct that houses the file descriptor with the code below:

```
struct WriteCoroutine {
    pub file_handle: File,
}

impl WriteCoroutine {
    fn new() -> Self {
        Self {
            file_handle: OpenOptions::new()
                .create(true)
                .append(true)
                .open("numbers.txt")
                .unwrap(),
        }
    }
}
```

We then implement the Coroutine trait with the following code:

```
impl Coroutine<i32> for WriteCoroutine {
    type Yield = ();
    type Return = ();

    fn resume(mut self: Pin<&mut Self>, arg: i32
        -> CoroutineState<Self::Yield, Self::Retu
        writeln!(self.file_handle, "{}", arg).unw
        CoroutineState::Yielded(())
    }
}
```

We can see that our coroutine does not yield or return anything. We can cover this later, but what is important is that we return a `CoroutineState::Yielded`. This is essentially like returning a `Pending` in a `Future`, but the `Yield` type is returned. We can also return a `CoroutineState::Complete` which is like a `Ready` in a future.

In our test, we can then create our coroutine and loop through the numbers calling the resume function with the code below:

```
let mut coroutine = WriteCoroutine::new();

for &number in &numbers {
    Pin::new(&mut coroutine).resume(number);
}
```

Our updated test will give us a time of roughly 622.6ms. This is roughly 6 times faster. Sure, we could have just created the file descriptor and referenced it in the loop to get the same effect, but what this demonstrates is that there is a benefit to suspending the state of a coroutine and resuming it when needed. We managed to keep our write logic isolated, but we did not need any threads or async runtimes for that speed up. Coroutines can be building blocks within threads and async tasks to suspend resume computations.

NOTE

You may have noticed some similarities between implementing the `Future` trait and `Coroutine` trait. There are two possible states, you resume or finish depending on the two possible outcomes, and you suspend and resume both coroutines and async tasks. It could be argued that async tasks are coroutines in a general sense, with the difference being that they are sent to a different thread and polled by an executor.

Coroutines have a lot of uses. They could be used to handle network requests, big data processing or UI applications. They provide a simpler way of handling async tasks compared to using callbacks. Using coroutines, we can implement async functionality in one thread without the need for queues or a defined executor.

## Generating with Coroutines

You may have come across a concept of generators in other languages such as Python. Generators are a subset of coroutines - sometimes called “weak” coroutines. They are referred to this way as they always yield control back to the process that called them, rather than another coroutine.

Generators allow us to perform actions in a lazy manner. Lazy means that we act on it as and when we need it, meaning lazy operations “yield” output values only when needed. This could be performing a computation, making a connection over a network, or loading data. This “lazy” evaluation is particularly useful when having to deal with large datasets that may be inefficient or unfeasible.

Now we are going to put this theory into practice and write our first simple generator.

## Implementing a Simple Generator in Rust

Let's work through an example of using a generator. Let's imagine we want to pull in information from a large data structure that is contained in a CSV. The CSV is very large and ideally, we do not want to load it into memory all at once. For our example, we will use a very small CSV - with just 5 rows - to demonstrate streaming. Remember this is an educational example and in the real world, using a generator to read a 5-line CSV file would be considered overkill! You can make this yourself. We have saved a CSV with 5 rows, with a number on each row in our project called `test.csv`.

We need the coroutine features in the previous example, and we import the components we will need.

```rust
use std::fs::File;
use std::io::{self, BufRead, BufReader};
use std::ops::{Coroutine, CoroutineState};
use std::pin::Pin;
```

Now let's create our `ReadCoroutine` struct:

```
struct ReadCoroutine {
    lines: io::Lines<BufReader<File>>,
}

impl ReadCoroutine {
    fn new(path: &str) -> io::Result<Self> {
        let file = File::open(path)?;
        let reader = BufReader::new(file);
        let lines = reader.lines();

        Ok(Self { lines })
    }
}
```

Then we implement the Coroutine trait for this struct. Our input file has the numbers 1 to 5 in it so we are going to be yielding an `i32`.

```
impl Coroutine<()> for ReadCoroutine {
    type Yield = i32;
    type Return = ();

    fn resume(mut self: Pin<&mut Self>, _arg: ()
    -> CoroutineState<Self::Yield, Self::Return>
        match self.lines.next() {
            Some(Ok(line)) => {
                if let Ok(number) = line.parse::
                    CoroutineState::Yielded(numbe
                } else {
                } else {
                    CoroutineState::Complete(())
                }
            }
            Some(Err(_)) | None => CoroutineState
        }
    }
}
```

The coroutine contains a yield statement which allows us to “yield” a value out of the generator. The coroutine trait only has one required method which is resume. This allows us to resume execution, picking up at the previous execution point. In our case, the resume method reads lines from the file, parses them into integers, and yields them until there are no more lines left to yield, at which point it completes.

Now we will call our function on our test file.

```
fn main() -> io::Result<()> {
    let mut coroutine = ReadCoroutine::new("test

    loop {
        match Pin::new(&mut coroutine).resume(()
            CoroutineState::Yielded(number) => pr
            CoroutineState::Complete(()) => breal
        }
    }
    Ok(())
}
```

You should get a print out:

```
1
2
3
4
5
```

---

**WARNING**

It is worth noting that this was previously known as the `Generator` trait and `GeneratorState` retrospectively, with the name changes taking place in late 2023. If you have an older version of nightly already installed, you will need to update for the following code to work.

---

Now we are going to move on to how we can use two coroutines together, with them yielding between each other. This is where we will start to see some of the power of using coroutines.

## Stacking our Coroutines

We are now going to use the concept of a file transfer to demonstrate how 2 co-routines can be used sequentially. This is useful because we might want to transfer a file that is so big, we would not be able to fit all the data into memory. But transferring data bit by bit will enable us to transfer all the data without running out of memory. To enable this solution, one coroutine reads a file and yields values, while another coroutine receives values and writes them to a file. We will reuse our `ReadCoroutine`, and add in our `WriteCoroutine` from the first section of this chapter. In that example, we wrote 200,000 random numbers to a file called `numbers.txt`. Let's reuse this file as the file that we wish to transfer. We will read in numbers.txt and write to a file called output.txt.

We will rewrite the `WriteCoroutine` slightly so it is expecting a path, rather than hard-coding this in.

```
struct WriteCoroutine {
    pub file_handle: File,
}

impl WriteCoroutine {
    fn new(path: &str) -> Self {
        Self {
            file_handle: OpenOptions::new()
                .create(true)
                .append(true)
                .open(path)
```

```
                    .unwrap(),
            }
        }
    }
```

Now we are going to create a manager that has a reader and writer coroutine.

```
struct CoroutineManager{
    reader: ReadCoroutine,
    writer: WriteCoroutine
}
```

We need to create a function that sets off our file transfer. First, we will create the `new()` function that instantiates the coroutine manager. This sets read and write file paths.

Secondly, we will create a new function called run. We need to pin the reader and write in memory so that they can be used throughout the lifetime of the program. We then create a loop that incorporates both the reader and write functionality. The reader is matched to either `Yielded` or `Complete`. If it is `Yielded` (i.e. there is an output), the writer coroutine takes this in and writes it to the file.

If there are no more numbers left to read, we break the loop.

```
impl CoroutineManager {
    fn new(read_path: &str, write_path: &str) ->
        let reader = ReadCoroutine::new(read_patl
        let writer = WriteCoroutine::new(write_pa

        Ok(Self {
            reader,
            writer,
        })
    }
    fn run(&mut self) {
        let mut read_pin = Pin::new(&mut self.rea
        let mut write_pin = Pin::new(&mut self.wr

        loop {
            match read_pin.as_mut().resume(()) {
                CoroutineState::Yielded(number) =
                    write_pin.as_mut().resume(nur
                }
                CoroutineState::Complete(()) => l
            }
        }
    }
}
```

We can use this in the main:

```
fn main() {
    let mut manager = CoroutineManager::new(
        "numbers.txt", "output.txt"
    ).unwrap();
    manager.run();
}
```

Once you have run this, you can open your new `output.txt` file to double-check that you have the correct contents.

Let’s recap what we have done here. In essence, we have created a file transfer. One coroutine reads a file line by line and yields its values to another coroutine. This coroutine receives values and writes to the file. In both, the file handles are kept open for the whole execution which means we don’t have to keep contending with slow `I/O`. With this type of lazy loading and writing, we can queue up a program to work on processing a number of file transfers one at a time. Zooming out, you can see the benefit of this approach. We could use this to move 100 large files of multiple gigabytes each from one location to another, or even over a network.

## Calling a Coroutine from a Coroutine

In the previous example, we used a coroutine to yield a value which was then taken in by the writer coroutine. This process is managed by a manager. In an ideal situation, we would like to remove the need for a manager at all, and to allow the coroutines to call each other directly and pass back and forth. This is called symmetric coroutines and is used in other languages. It does not come as standard (yet) in Rust and in order to implement something similar to this, we need to move away from using the `Yielded` and `Complete` syntax.

We will create our own trait called `SymmetricCoroutine`. This has one function which is a `resume_with_input`. This takes in an input, and provides an output.

```
trait SymmetricCoroutine {
    type Input;
    type Output;

    fn resume_with_input(
        self: Pin<&mut Self>, input: Self::Input
    ) -> Self::Output;
}
```

We can now implement this trait for our `ReadCoroutine`. This outputs values of the type `i32`. Note we are not using `Yielded` here any more but are still using the line parser. This will output the values we need.

```
impl SymmetricCoroutine for ReadCoroutine {
    type Input = ();
    type Output = Option<i32>;

    fn resume_with_input(
        mut self: Pin<&mut Self>, _input: ()
    ) -> Self::Output {
        if let Some(Ok(line)) = self.lines.next()
            line.parse::<i32>().ok()
        } else {
            None
        }
    }
}
```

For the `WriteCoroutine`, we implement this trait as well:

```
impl SymmetricCoroutine for WriteCoroutine {
    type Input = i32;
    type Output = ();

    fn resume_with_input(
        mut self: Pin<&mut Self>, input: i32
    ) -> Self::Output {
        writeln!(self.file_handle, "{}", input).
    }
}
```

Finally, we put this together in main:

```rust
fn main() -> io::Result<()> {
    let mut reader = ReadCoroutine::new("numbers
    let mut writer = WriteCoroutine::new("output

    loop {
        let number = Pin::new(&mut reader).resume
        if let Some(num) = number {
            Pin::new(&mut writer).resume_with_in
        } else {
            break;
        }
    }
    Ok(())
}
```

The main function is explicitly instructing how the coroutines should work together. This involves manually scheduling so technically it does not meet the criteria for truly symmetrical coroutines. We are mimicking some of the functionality of symmetrical coroutines as an educational exercise. A true symmetrical coroutine would involve control being passed from the reader to the writer without having to return to the main function which is restricted by Rust's borrowing rules as both coroutines will need to reference each other. Despite this, it is still a useful example to demonstrate how you can get more functionality by writing your own coroutines.

We are now going to move on to looking at async behaviour and how we can mimic some of this functionality with simple coroutines.

## Mimicking Async Behaviour with Coroutines

In the introduction to this chapter, we discussed how similar coroutines are to async programming where execution is suspended and later resumed when certain conditions are met. A strong argument could be made that all async programming is a subset of coroutines. Async runtimes are essentially coroutines across threads.

We can demonstrate the pausing of executions with this simple example. We will set up a Coroutine that sleeps for one second.

```
struct SleepCoroutine {
    pub start: Instant,
    pub duration: std::time::Duration,
}
impl SleepCoroutine {
    fn new(duration: std::time::Duration) -> Sel
        Self {
            start: Instant::now(),
            duration,
        }
    }
}
impl Coroutine<()> for SleepCoroutine {
    type Yield = ();
    type Return = ();

    fn resume(
        self: Pin<&mut Self>, _: ())
    -> CoroutineState<Self::Yield, Self::Return>
        if self.start.elapsed() >= self.duration
            CoroutineState::Complete(())
        } else {
            CoroutineState::Yielded(())
        }

    }
}
```

We will set up 3 instances of `SleepCoroutine` that will run at the same time. Each instance sleeps for one second.

For this, we will use `VecDeque`. This will allow us to push each instance of the coroutines onto the `VecDeque`.

```
use std::collections::VecDeque;
```

We create a counter and use this to loop through the queue of coroutines - yielding or completing. Finally, we time the whole operation.

```
fn main() {
    let mut sleep_coroutines = VecDeque::new();
    sleep_coroutines.push_back(
        SleepCoroutine::new(std::time::Duration:
    );
    sleep_coroutines.push_back(
        SleepCoroutine::new(std::time::Duration:
    );
    sleep_coroutines.push_back(
        SleepCoroutine::new(std::time::Duration:
    );

    let mut counter = 0;
    let start = Instant::now();

    while counter < sleep_coroutines.len() {
        let mut coroutine = sleep_coroutines.pop_
        match Pin::new(&mut coroutine).resume(()
            CoroutineState::Yielded(_) => {
                sleep_coroutines.push_back(corout
            },
            CoroutineState::Complete(_) => {
                counter += 1;
            },
        }
    }
    println!("Took {:?}", start.elapsed());
}
```

This takes one second to complete, and yet we are carrying out three coroutines that each take one second to complete. We might expect it to therefore take three seconds in total. The shortened amount of time occurs precisely because they are coroutines - they are able to pause their execution and resume at a later time. We are not using Tokio or any other asynchronous runtime. All operations are running in a single thread. They are simply pausing and resuming.

In a way, we have written our own specific executor for this use case. We can even use the executor syntax to make this even clearer. Let’s create an `Executor` struct that uses `VecDeque`.

```
    struct Executor {
        coroutines: VecDeque<Pin<Box<
            dyn Coroutine<(), Yield = (), Return = (
        >>>,
    }
```

Now we will add the basic functionality of an `Executor`.

```
impl Executor {
    fn new() -> Self {
        Self {
            coroutines: VecDeque::new(),
        }
    }
}
```

We will define an add function which reuses the same code as we had before where coroutines can be returned to the queue.

```
fn add(&mut self, coroutine: Pin<Box<
    dyn Coroutine<(), Yield = (), Return = ()>>>
    {
        self.coroutines.push_back(coroutine);
    }
```

Finally, we will wrap our Coroutine State code into a function called poll.

```
fn poll(&mut self) {
    println!("Polling {} coroutines", self.corout
    let mut coroutine = self.coroutines.pop_front
    match coroutine.as_mut().resume(()) {
        CoroutineState::Yielded(_) => {
            self.coroutines.push_back(coroutine)
        },
        CoroutineState::Complete(_) => {},
    }
}
```

Our main function can now create the executor, add the coroutines and then poll them until they are all complete.

```
fn main() {
    let mut executor = Executor::new();

    for _ in 0..3 {
        let coroutine = SleepCoroutine::new(
            std::time::Duration::from_secs(1)
        );
        executor.add(Box::pin(coroutine));
    }
    let start = Instant::now();
    while !executor.coroutines.is_empty() {
        executor.poll();
    }
    println!("Took {:?}", start.elapsed());
}
```

That's it! We have built our first `Executor`. We will build on this in Chapter 11. Now that we have essentially achieved async functionality from our coroutines and executor, we should really drive home the relationship between async and coroutines by implementing the `Future` trait for our `SleepCoroutine` with the following code:

```
impl Future for SleepCoroutine {
    type Output = ();

    fn poll(mut self: Pin<&mut Self>, cx: &mut Co
        -> Poll<Self::Output> {
        match Pin::new(&mut self).resume(()) {
            CoroutineState::Complete(_) => Poll:
            CoroutineState::Yielded(_) => {
                cx.waker().wake_by_ref();
                Poll::Pending
            },
        }
    }
}
```

Just to recap, in this example, we are demonstrating that Coroutines pause and resume, in a similar manner to how async/await works. The difference is that we are using Coroutines in a single thread. The major drawback here is that you have to write the coroutine and if you want, your own executor. This means that they can be highly coupled to the problem you are trying to solve. We also lose the benefit of having a pool of threads. Defining your own coroutine may be justified in a situation in which having the async runtime might be overkill. We also may use coroutines when we want extra control. For instance, we do not really have much control over when an async task is polled in relation to other async tasks when the async task is sent to the runtime. This brings us to our next topic, controlling coroutines.

## Controlling Coroutines

Throughout this book, we have controlled the flow of the async task internally. For instance, when we implement the Future trait we get to choose when to return a Pending or Ready depending on the internal logic of the poll function. The same goes for our async functions where we choose when the async task might yield control back to the executor with the await syntax, and choose when the async task returns a Ready with return statements. We can control these async tasks with external Sync primitives such as atomic values and mutexes by getting the async task to react to changes and values in these atomic values and mutexes. However, the logic reacting to external signals has to be coded into the async task before sending the async task to the runtime. For simple cases this can be fine, however, it does expose the async tasks to being brittle if the system around the async task is evolving. This also makes the async task harder to use in other contexts. The async task might also need to know the state of other async tasks before reacting and this can lead to potential problems such as deadlocks.

**NOTE**

A deadlock can happen where task A requires something from task B to progress, and task B requires something from task A to progress resulting in an impasse. We cover deadlocking, livelocking, and other potential pitfalls with methods on how to test for them in Chapter 11 Testing.

This is where the ease of external control in coroutines can come in handy. To demonstrate how external control can simplify things, we are going to write a simple program that loops through a vector of coroutines that have a value, and a status of being alive or dead. When the coroutine gets called, a random number gets generated for the value, and this value is then yielded. We can accumulate all of these values and come up with a simple rule of when to kill the coroutine. For the random number generation we will need the dependency below:

```
[dependencies]
rand = "0.8.5"
```

Now we can build a random number coroutine with the following code:

```
use std::ops::{Coroutine, CoroutineState};
use std::pin::Pin;
use rand::Rng

struct RandCoRoutine {
    pub value: u8,
    pub live: bool,
}
impl RandCoRoutine {
    fn new() -> Self {
        let mut coroutine = Self {
            value: 0,
            live: true,
        };
        coroutine.generate();
        coroutine
    }
    fn generate(&mut self) {
        let mut rng = rand::thread_rng();
        self.value = rng.gen_range(0..=10);
    }
}
```

Considering that external code is going to be controlling our coroutine, we are going to use a simple generator implementation with the code below:

```
impl Coroutine<()> for RandCoRoutine {
    type Yield = u8;
    type Return = ();

    fn resume(mut self: Pin<&mut Self>, _: ())
        -> CoroutineState<Self::Yield, Self::Retu
        self.generate();
        CoroutineState::Yielded(self.value)
    }
}
```

We can use this generator all over the codebase as it just does what is said on the tin. External dependencies are not needed to run our coroutine and our testing is also simple. In our main function, we create a vector of these coroutines, calling them until all of the coroutines in the vector are dead with the following code:

```
let mut coroutines = Vec::new();
for _ in 0..10 {
    coroutines.push(RandCoRoutine::new());
}
let mut total: u32 = 0;

loop {
    let mut all_dead = true;
    for mut coroutine in coroutines.iter_mut() {
        if coroutine.live {
            . . .

        }
    }
    if all_dead {
        break
    }
}
```

```
println!("Total: {}", total);
```

If our coroutine in the loop is alive, we can then assume that all of the coroutines are not dead, setting the `all_dead` flag to false. We then call the resume on the coroutine, extract the result, and come up with a simple rule on whether to kill the coroutine or not with the code below:

```
all_dead = false;
match Pin::new(&mut coroutine).resume(()) {
    CoroutineState::Yielded(result) => {
        total += result as u32;
    },
    CoroutineState::Complete(_) => {
        panic!("Coroutine should not complete");
    },
}
if coroutine.value < 9 {
    coroutine.live = false;
}
```

If we reduce the cut off for killing the coroutine in the loop the end total will be higher as the cut off is harder to achieve. We are in our main thread so we have access to everything in the main thread. For instance, we could keep track of all dead coroutines and start reanimating coroutines if that number gets too high. We could also use the death number to change the rules of when to kill a coroutine. Now we could still achieve this toy example in an async task. For instance, a future can hold and poll another future inside it with the following simple example:

```
struct NestingFuture {
    inner: Pin<Box<dyn Future<Output = ()> + Senc
}
impl Future for NestingFuture {
    type Output = ();

    fn poll(mut self: Pin<&mut Self>, cx: &mut Cc
        -> Poll<Self::Output> {
        match self.inner.as_mut().poll(cx) {
            Poll::Ready(_) => Poll::Ready(()),
            Poll::Pending => Poll::Pending,
        }
    }
}
```

There is nothing stopping our `NestingFuture` from having a vector of other futures that update their own value field everytime they are polled and perpetually return a `Pending` as a result of that poll. The nesting future can then extract that value field and come up with rules on if the recently polled future should be killed or not. However, the `NestingFuture` would be operating in some thread in the async runtime, resulting in having limited access to data in the main thread.

Considering the ease of control over coroutines, we need to remember that it is not all or nothing. It’s not coroutines vs async. With the code below, we can prove that we can send coroutines over threads:

```
let (sender, reciever) = std::sync::mpsc::channel
let thread = std::thread::spawn(move || {
    loop {
        let mut coroutine = reciever.recv().unwra
        match Pin::new(&mut coroutine).resume(()
            CoroutineState::Yielded(result) => {
                println!("Coroutine yielded: {}",
            },
            CoroutineState::Complete(_) => {
                panic!("Coroutine should not comp
            },
        }
    }
});
std::thread::sleep(Duration::from_secs(1));
sender.send(RandCoRoutine::new()).unwrap();
sender.send(RandCoRoutine::new()).unwrap();
std::thread::sleep(Duration::from_secs(1));
```

Considering that coroutines are thread safe and easily map the results of coroutines, we can finish our journey of understanding coroutines. We can conclude that coroutines are a computational unit that can be paused and resumed. Furthermore, these coroutines also implement `Future` traits, which can call the resume function and map the results of that resume function to the results of the poll function as seen in Figure 5-2.

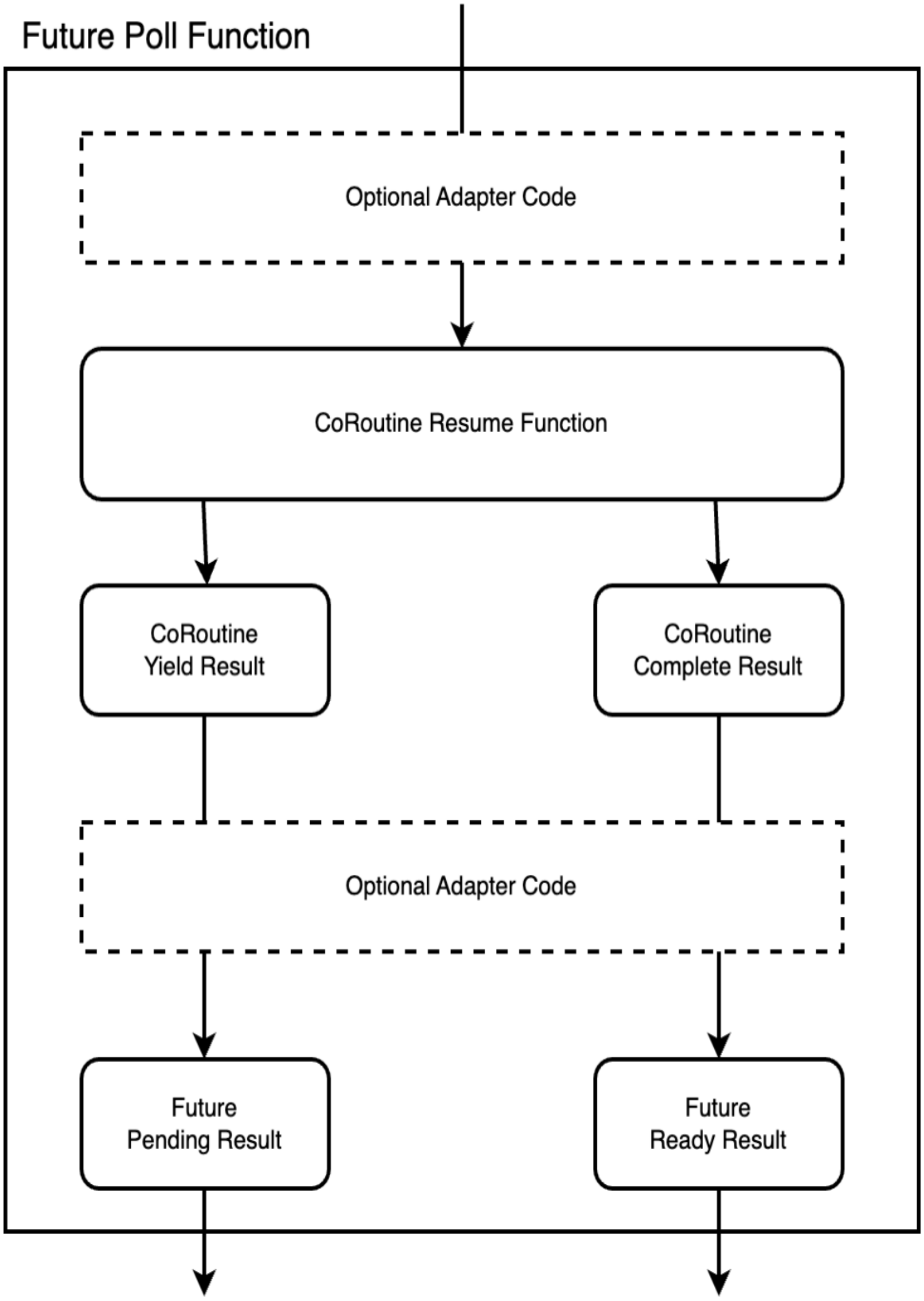

Future Poll Function
Optional Adapter Code
CoRoutine Resume Function
CoRoutine Yield Result
CoRoutine Complete Result
Optional Adapter Code
Future Pending Result
Future Ready Result

Figure 5-2. How Coroutines can be async adapters

What Figure 5-2 also shows us is that we can slot in optional adapter code between the coroutine functions and the future functions. With this, we can see coroutines as the fundamental computational building blocks. These coroutines can be suspended and resumed in synchronous code and therefore are easy to test in standard testing environments because you do not need an async runtime to test these coroutines. You can also choose as and when you can call the resume function of a coroutine so testing different orders in which coroutines interact with each other is also simple.

Once you are happy with your coroutine and how it works, you can essentially wrap one or multiple coroutines in a struct that implements the `Future` trait. The struct that implements the Future trait is essentially an adapter that enables coroutines to interact with the async runtime. This gives us ultimate flexibility and control over the testing and implementation of our computational processes, and a clear boundary between these computational steps and the async runtime, as the async runtime is basically a threadpool with queues. Anyone who is familiar with unit testing knows that we should not have to communicate with a threadpool to test the computational logic of a function or struct. With this in mind, let us wrap up our understanding of how coroutines fit in the world of async with testing.

## Testing Coroutines

For our testing example, we do not want to excessively bloat the chapter with complex logic, so we will be testing two coroutines that acquire the same mutex and increase the value in the mutex by one. With this, we can test on what happens when the lock is acquired, and the end result of the lock after the interaction.

---

**NOTE**

It must be noted that while testing using coroutines is simple and powerful, you are not completely on the rocks with regards to testing if you are not using coroutines. The chapter 11 is dedicated to testing and you will not see a single coroutine in chapter 11.

---

We start with our struct that has a handle to the mutex, and a threshold where the coroutine will be complete once the threshold is reached with the following definition:

```
use std::ops::{Coroutine, CoroutineState};
use std::pin::Pin;
use std::sync::{Arc, Mutex};

pub struct MutexCoRoutine {
    pub handle: Arc<Mutex<u8>>,
    pub threshold: u8,
}
```

We then implement the logic behind acquiring the lock and increasing the value by one with the code below:

```
impl Coroutine<()> for MutexCoRoutine {
    type Yield = ();
    type Return = ();

    fn resume(mut self: Pin<&mut Self>, _: ())
        -> CoroutineState<Self::Yield, Self::Retu
        match self.handle.try_lock() {
            Ok(mut handle) => {
                *handle += 1;
            },
            Err(_) => {
                return CoroutineState::Yielded((
            },
        }
        self.threshold -=1;
        if self.threshold == 0 {
            return CoroutineState::Complete(())
        }
        return CoroutineState::Yielded(())

    }
}
```

We can see that we are trying to get the lock, but if we cannot, we do not want to block so we will return a yield. If we get the lock, we increase the value by one, decrease our threshold by one, and then return a yield or complete depending on if our threshold is reached or not. And that’s it, we can test our coroutine in the same file with the following template:

```
#[cfg(test)]
mod tests {
    use super::*;
    use std::future::Future;
    use std::task::{Context, Poll};
    use std::time::Duration;

    // sync testing inteface
    fn check_yield(coroutine: &mut MutexCoRoutine
        . . .
    }
    // async runtime inteface

    impl Future for MutexCoRoutine {
        . . .
    }
    #[test]
    fn basic_test() {
        . . .
    }
    #[tokio::test]
    async fn async_test() {
            . . .
        }
    }
```

Here we are going to check how our code works directly, and then how our code runs in an async runtime. We have two interfaces. We do not want to have to alter our code to satisfy tests. Instead, we have a simple interface that returns a bool based on the type returned by our coroutine with the function definition below:

```
fn check_yield(coroutine: &mut MutexCoRoutine) ->
    match Pin::new(coroutine).resume(()) {
        CoroutineState::Yielded(_) => {
            true
        },
        CoroutineState::Complete(_) => {
            false
        },
    }
}
```

With our async interface we simply map the coroutine outputs to the equivalent async outputs with the following code:

```
impl Future for MutexCoRoutine {
    type Output = ();
    fn poll(mut self: Pin<&mut Self>, cx: &mut Co
        -> Poll<Self::Output> {
        match Pin::new(&mut self).resume(()) {
            CoroutineState::Complete(_) => Poll:
            CoroutineState::Yielded(_) => {
                cx.waker().wake_by_ref();
                Poll::Pending
            },
        }
    }
}
```

We are now ready to build our first basic test in the `basic_test` function. We initially define the mutex and coroutines with the code below:

```
let handle = Arc::new(Mutex::new(0));
let mut first_coroutine = MutexCoRoutine {
    handle: handle.clone(),
    threshold: 2,
};
let mut second_coroutine = MutexCoRoutine {
    handle: handle.clone(),
    threshold: 2,
};
```

We first want to acquire the lock ourselves, and then call both of the coroutines checking that they return a yield and that the value of the mutex is still zero because we have lock with the following code:

```
let lock = handle.lock().unwrap();
for _ in 0..3 {
    assert_eq!(check_yield(&mut first_coroutine),
    assert_eq!(check_yield(&mut second_coroutine
}
assert_eq!(*lock, 0);
std::mem::drop(lock);
```

**NOTE**

You may have noticed that we drop the lock after the initial testing of the first two yields. If we do not do this, the rest of the tests will fail as our coroutines will never be able to acquire the lock.

We do the loop to prove that the threshold is also not being altered when the coroutine fails to get the lock. If the threshold was altered, after 2 iterations, the coroutine would have returned a complete, and the next call to the coroutine will have resulted in an error. While the test would have picked this up without the loop, having the loop at the start removes any confusion as to what is causing the break in the test.

After we drop the lock, we then call the coroutines twice each to ensure that they return what we expect, and check on the mutex between all the calls to ensure that the state is changing in the exact way that we want it with the code below:

```
assert_eq!(check_yield(&mut first_coroutine), tru
assert_eq!(*handle.lock().unwrap(), 1);
assert_eq!(check_yield(&mut second_coroutine), tr
assert_eq!(*handle.lock().unwrap(), 2);
assert_eq!(check_yield(&mut first_coroutine), fal
assert_eq!(*handle.lock().unwrap(), 3);
assert_eq!(check_yield(&mut second_coroutine), fa
assert_eq!(*handle.lock().unwrap(), 4);
```

And our first test is complete.

In our async test, we create the mutex and coroutines in the exact same way. However, we are now testing that our behaviour end result is the same in an async runtime, and that our async interface is working the way we expect it to. Because we are using the Tokio testing feature we can just spawn our tasks, wait on them, and inspect the lock with the following code:

```
let handle_one = tokio::spawn(async move {
    first_coroutine.await;
});
```

```
    let handle_two = tokio::spawn(async move {
        second_coroutine.await;
    });
    handle_one.await.unwrap();
    handle_two.await.unwrap();
    assert_eq!(*handle.lock().unwrap(), 4);
```

If we run the command `cargo test`, we will see that both of our tests work. And here we have it! We have inspected the interaction between two coroutines and a mutex step by step, inspecting the state between each iteration. Our coroutines work in synchronous code. But, with a simple adapter, we can also see that our coroutines work in an async runtime exactly how we expect them to! We can see that we do not have the ability to inspect the mutex at each interaction of a coroutine with the async test. The async executor is doing its own thing.

## Conclusion

In this chapter, we have built our own coroutines by implementing the Coroutine trait and seeing how we can use Yield and Complete to pause and resume our coroutine. We have implemented a pipeline in which a file can be read by one coroutine which yields values, and those values can be used by a second coroutine and written to a file. Finally, we built our own executor and saw how Coroutines are truly pausing and resuming.

As you worked through the chapter you were likely struck by the similarities between Yield/Complete in Coroutines and Pending/Ready in Async. In our opinion, the best way to view this is that Async/Await is a subtype of a Coroutine. It is a Coroutine that operates across threads and makes use of queues. You can suspend something and come back to it in both Coroutines and Async Programming.

Coroutines enable us to structure our code, as they can act as the seam between async and synchronous code. With this, you can build synchronous code modules and then test them using standard tests. You can then build adapters that are coroutines where our synchronous code can connect with code that needs async functionality but that async functionality is represented as coroutines. Then, we can unit test our coroutines to see how our coroutines behave when they are polled in different orders and combinations. We can then inject those coroutines into Future trait implementations to integrate our code into an async runtime as we can call our coroutines in the Future poll function. Here, you just need to keep this async code isolated with interfaces. One async function can call your code, and then pass outputs into the third party async code, and vice versa.

A good way of isolating code is reactive programming where our units of code can consume data though subscriptions to broadcast channels. We'll explore this in the next chapter.

# Chapter 6. Reactive Programming

Reactive programming is essentially a programming paradigm where code reacts to changes in data values or events. Reactive programming enables us to build systems that respond dynamically to changes in real time. It is essential to underline that this chapter is written in the context of an asynchronous programming book. We cannot cover every aspect of reactive programming as entire books have been written on the topic. Instead, we are going to focus on async approaches to polling and reacting to changes in data by building a basic heater system, where futures react to changes in temperature. We will then build an event bus using atomics, mutexes, and queues to enable us to publish events to multiple subscribers.

By the end of this chapter, you will be familiar with enough async data sharing concepts to construct thread safe mutable data structures. These data structures can be manipulated safely by multiple different concurrent async tasks. You will also be able to implement the observer pattern. By the end of this chapter you’ll be equipped with the skills to build async Rust solutions to reactive design patterns that you’ll find in further reading.

We start our reactive programming journey with building a basic reactive system.

## Building a Basic Reactive System

When it comes to building a basic reactive system we are going to implement the observer pattern. With the observer pattern we have subjects, and then observers that subscribe to updates of that subject. When the subject releases an update, the observers generally react to this update depending on the specific requirements of the observer. For our basic reactive system, we can build a simplistic heating system. The system turns the heater on when the temperature goes below the desired temperature as seen in figure 6-1.

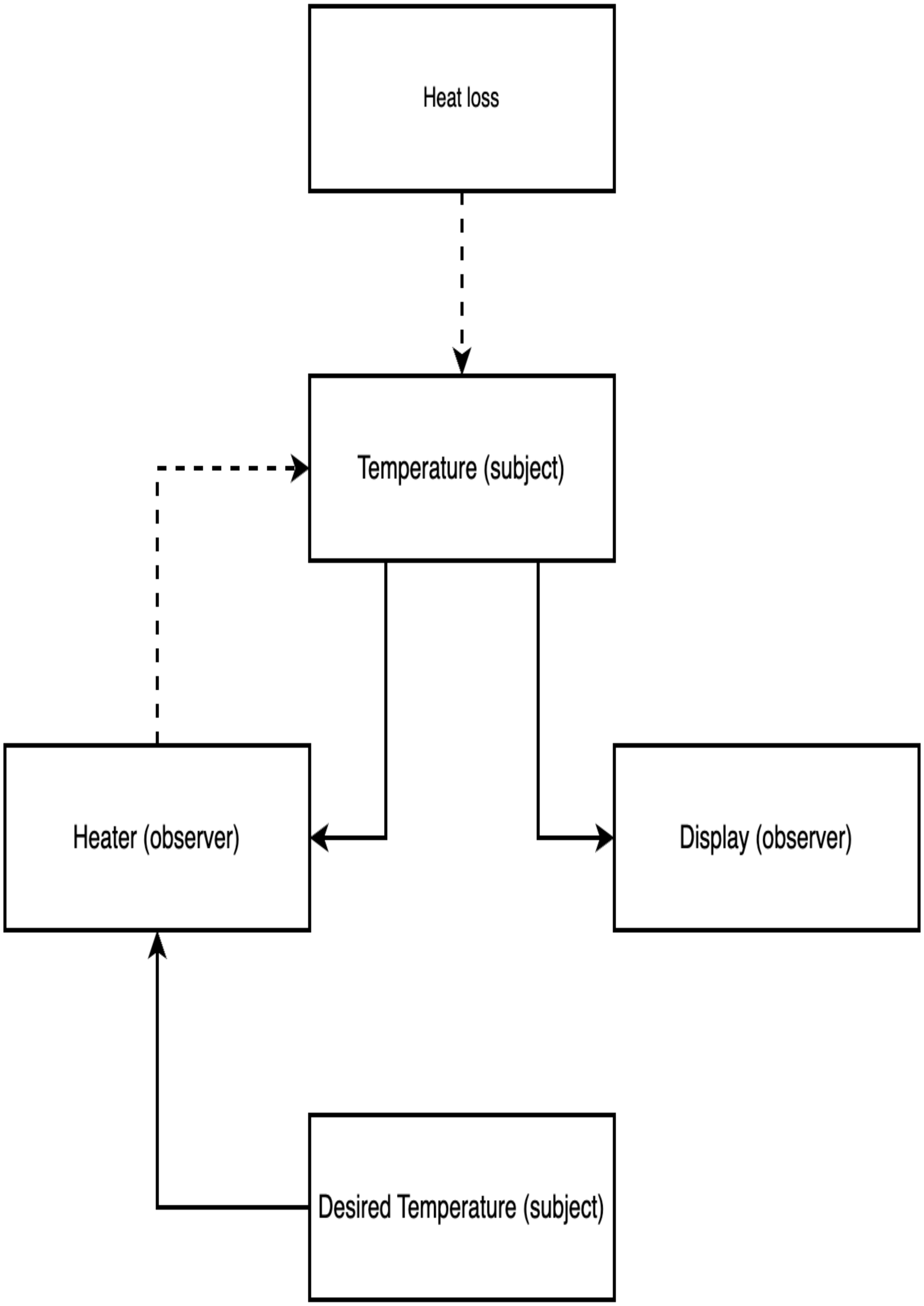

Heat loss
Temperature (subject)
Heater (observer)
Display (observer)
Desired Temperature (subject)

Figure 6-1. Our basic reactive heating system

Looking at figure 6.1 we can see the temperature and desired temperature are subjects. The heater and display are observers. Our heater will turn on if the temperature drops below the desired temperature. Our display will print out the temperature to the terminal if the temperature changes. It must be noted that in a real life system, we would just connect our system to a temperature sensor. However, because we are just using this example to explore reactive programming, we are skipping the detour into hardware engineering, and making do with coding the effect of the heater and the heat loss on the temperature directly. Now that we have our system laid out, we can move onto defining our subject.

## Defining Our Subjects

Our observers in the system are going to be futures with non-stop polling, as the observers are going to be polling the subjects continuously throughout the program to see if the subject has changed. Before we can start building our temperature system we need the following dependencies:

```
[dependencies]
tokio = { version = "1.26.0", features = ["full"]
once_cell = "1.18.0"
clearscreen = "2.0.1"
```

We are using the `clearscreen` to update the display of our system, the `once_cell` to enable our subjects to be referenced across the threads, and the `tokio` crate for an easy interface of async runtimes. With these dependencies we need the imports below to build our system:

```
use std::sync::Arc;
use std::sync::atomic::{AtomicI16, AtomicBool};
use core::sync::atomic::Ordering;
use once_cell::sync::Lazy;
use std::future::Future;
use std::task::Poll;
use std::pin::Pin;
use std::task::Context;
use std::time::{Instant, Duration};
```

We now have everything need, so we can define our subjects with the code below:

```
static TEMP: Lazy<Arc<AtomicI16>> = Lazy::new(||
    Arc::new(AtomicI16::new(2090))

});
static DESIRED_TEMP: Lazy<Arc<AtomicI16>> = Lazy
    Arc::new(AtomicI16::new(2100))
});
static HEAT_ON: Lazy<Arc<AtomicBool>> = Lazy::new
    Arc::new(AtomicBool::new(false))
```

```
        Arc::new(AtomicBool::new(false))
});
```

The subjects have the following responsibilities:

❶ The current temperature of the system

❷ The desired temperature that we would like the room to be

❸ Whether the heater should be on or off. If the bool is true, we instruct the heater to turn on. The heater will turn off if the bool is false.

---

**NOTE**

If you have Googled reactive programming or reactive systems before reading this, you may have read about passing messages and events. Messages and events are certainly part of reactive programming, but we need to remember that an important part of software development is not to over-engineer our system. The more complex our system is, the harder it is to maintain and change. Our system has very basic feedback needs. The heater gets turned on or off based on a number. If we look into locks and channels that send messages between threads, they essentially boil down to atomics for the locks and other data collections to handle the data. For our system for now, just using atomics is enough due to the simple requirements of the system.

---

We can see that subscribing to subjects with observers decouples our code. For instance, we can increase the number of observers easily by just getting the new observers to observe the subject. We do not have to alter any code in existing subjects.

We now have everything needed for our subjects, so the next thing to do is build an observer that is going to display our subjects and control our

`HEAT_ON` subject.

## Building Our Display Observer

Now that our subjects are defined, we can define our display future with the following code:

```rust
pub struct DisplayFuture {
    pub temp_snapshot: i16,
}


impl DisplayFuture {
    pub fn new() -> Self {
        DisplayFuture {
            temp_snapshot: TEMP.load(Ordering::Se
        }
    }
}
```

Here we can see that when we create the future, we load the value of the temperature subject and store it. We can then use this stored temperature to compare against the temperature at the time of polling with the code below:

```rust
impl Future for DisplayFuture {
    type Output = ();
    fn poll(mut self: Pin<&mut Self>, cx: &mut C
        -> Poll<Self::Output> {
        let current_snapshot = TEMP.load(Orderin
        let desired_temp = DESIRED_TEMP.load(Ord
        let heat_on = HEAT_ON.load(Ordering::Seq

        if current_snapshot == self.temp_snapsho
            cx.waker().wake_by_ref();
            return Poll::Pending
        }
        if current_snapshot < desired_temp && hea
            HEAT_ON.store(true, Ordering::SeqCst
        }
        else if current_snapshot > desired_temp
            HEAT_ON.store(false, Ordering::SeqCs
        }
        clearscreen::clear().unwrap(); ❺
        println!("Temperature: {}\nDesired Temp:
        current_snapshot as f32 / 100.0,
        desired_temp as f32 / 100.0,
        heat_on);
        self.temp_snapshot = current_snapshot; ❼
        cx.waker().wake_by_ref();
        return Poll::Pending
    }
}
```

The code above does the following:

1. Get a snapshot of the system as a whole
2. We check to see if there is a difference between the snapshot of the temperature that the future holds and the current temperature. If there is no difference, there is no point re rendering the display or making any heating decisions so we merely just return a pending ending the poll
3. We then check to see if the current temperature is below the desired temperature. If it is, then we turn the `HEAT_ON` flag to true
4. If the temperature is higher than the desired temperature we turn the `HEAT_ON` flag to `false`
5. We wipe the terminal for the update
6. Print out the current state of the snapshot
7. Update the snapshot that the future references

Here we can see that we initially get a snapshot of the entire system. This approach can be up for debate. Some people argue that we should be loading the atomic values at every step. This would get the true nature of the state everytime we make a decision on altering the state of the subject, or displaying it. This is a reasonable argument, but there are always trade-offs when it comes to these sorts of decisions. For our system, the display is going to be the only observer that is going to alter the state of the HEAT_ON flag, and the logic in our future is making the decision based on the temperature. However, there are two other factors that affect the temperature, and these factors could affect the temperature between the snapshot and print as seen in figure 6-2.

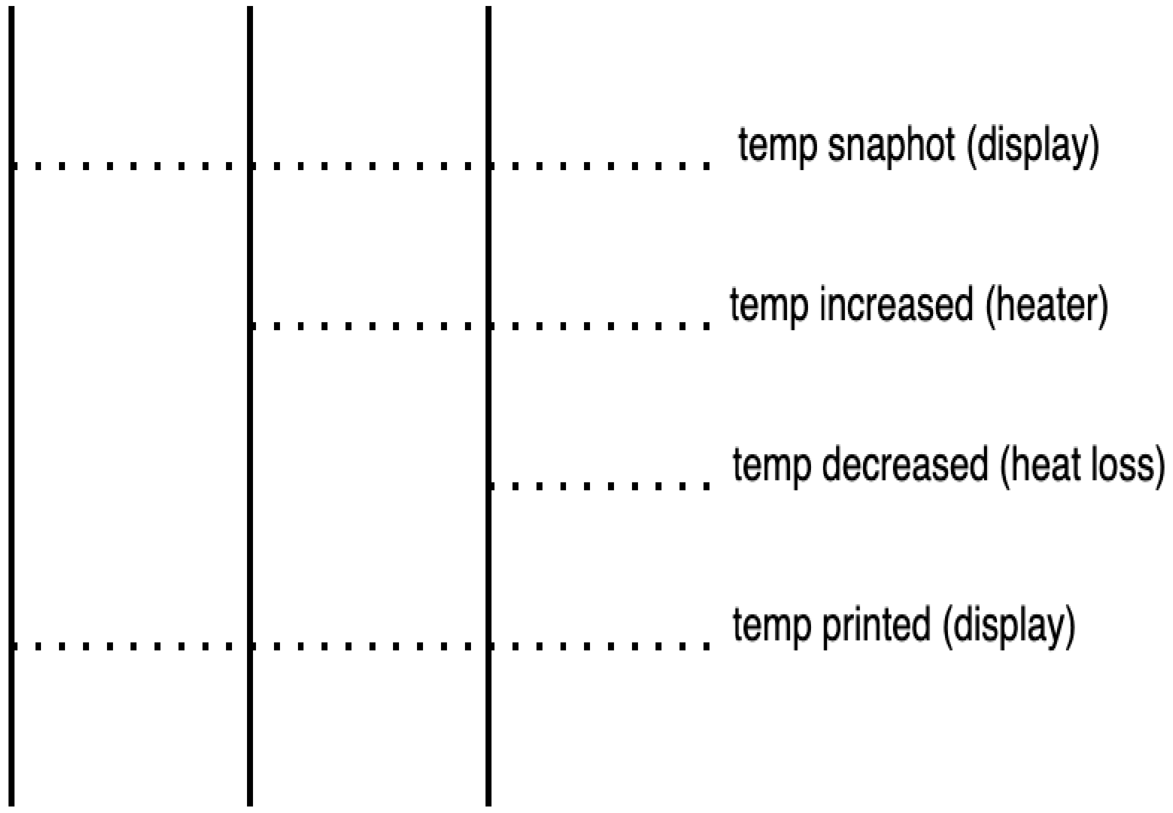


Figure 6-2. Futures affecting the temperature before the temperature snapshot was printed

In our system, it’s not the end of the world if the temperature display is slightly out for a split second. It could be argued that it is more important to take a snapshot, make decisions from that snapshot, and print that snapshot in order to see exactly what data was used to make a decision. This would also give us clear debugging information. We could also make the snapshot, alter the state of the `HEAT_ON` flag based on that snapshot, and then load every atomic for the print to the console so the display is always accurate the split second it is printed. Logging the snapshot for the decision and loading the atomics the moment we print is also an option. For our simple system, we are getting to the point of splitting hairs and we will stick to printing the snapshot so we can see how our system adapts and makes decisions. However, it is important to consider these trade-offs when building a reactive system. The data your observer is acting on could already be out of date.

For our simulation, we could remove the risk of operating on out of date data by restricting the runtime to just one thread. This would ensure that our snapshot would not be out of date as another future could not alter the temperature while our display future is being processed. Instead of restricting the runtime to one thread, we could wrap our temperature in a mutex which would also ensure that our temperature would not change between the snapshot and the print. However, our system is reacting to temperature. Temperature isn't a construct that our system just made up. Heat loss and the heater can be affecting our temperature in real time and we would only be lying to ourselves if we came up with tricks to avoid the changing of the temperature in our system while we had another process altering the state of our subjects.

While our system is simple enough that we do not worry about out of date data, we can utilize the compare and exchange functionality as shown in the code example from the standard library documentation below:

```
[illegible]
```

```
use std::sync::atomic::{AtomicI64, Ordering};


let some_var = AtomicI64::new(5);
assert_eq!(some_var.compare_exchange(5, 10,
                                     Ordering::Ac
                                     Ordering::Re
           Ok(5));
assert_eq!(some_var.load(Ordering::Relaxed), 10)


assert_eq!(some_var.compare_exchange(6, 12,
                                     Ordering::Se
                                     Ordering::Ac
           Err(10));
assert_eq!(some_var.load(Ordering::Relaxed), 10)
```

This is where we can appreciate why atomics are called atomics because their transactions are atomic. This means that no other transaction will happen on the atomic value while a transaction is being performed on the value. So what is happening in the `compare_exchange` function is that we are asserting that the atomic value is a certain value before we update it to the new value. If the value is not what we expect then we return an error, with the value of what the atomic value actually is. We can use the `compare_exchange` function to prompt observers to make another decision based on the value returned, and attempt to make another update on the atomic value based on the updated information. We have now covered enough to highlight the data concurrency issues with reactive programming and areas that provide solutions. We can now continue on with building our reactive system with the heater and heat loss observers.

## Building Our Heater and HeatLoss Observer

For our heater observer to function, we need to read the `HEAT_ON` bool, and not worry about the temperature. However, there is a time element to heaters. Sadly at the time of writing this book we live in a world where heaters are not instant, they take time to heat up the room. So, instead of a temperature snapshot, our heater future has a time snapshot, giving our heater future the following form:

```
pub struct HeaterFuture {
    pub time_snapshot: Instant,
}

impl HeaterFuture {
    pub fn new() -> Self {
        HeaterFuture {
            time_snapshot: Instant::now()
        }
    }
}
```

Now that we have a time snapshot, we can reference this snapshot, and increase the temperature after a certain duration with the poll function below:

```
impl Future for HeaterFuture {
    type Output = ();


    fn poll(mut self: Pin<&mut Self>, cx: &mut Co
        if HEAT_ON.load(Ordering::SeqCst) == fals
            self.time_snapshot = Instant::now();
            cx.waker().wake_by_ref();
            return Poll::Pending
        }
        let current_snapshot = Instant::now();
        if current_snapshot.duration_since(self.t
                                             Duration
            cx.waker().wake_by_ref();
            return Poll::Pending
        }
        TEMP.fetch_add(3, Ordering::SeqCst); ❸
        self.time_snapshot = Instant::now();
        cx.waker().wake_by_ref();
        return Poll::Pending
    }
}
```

In our heater future we carry out the following steps:

1. Exit as quickly as possible if the `HEAT_ON` flag is off because nothing is going to happen. We want to release the future from the executor as quickly as possible to avoid blocking other futures.
2. If the duration is not over three seconds we also exit because time has not elapsed for the heater to take effect.
3. Finally both time has elapsed and the `HEAT_ON` flag is on, so we increase the temperature by three.

Note that we update the time_snapshot at every exit opportunity apart from the `HEAT_ON` flag if `false` but not enough time has elapsed . If we did not update the `time_snapshot`, our heater future could be polled with the `HEAT_ON` flag as false until 3 seconds have elapsed. But as soon as the `HEAT_ON` flag is switched to `true`, the effect on the temperature would be instant. For our heater future, we need to reset the state between each poll.

When it comes to our heat loss future, the constructor method will be the same as the heater future as we are referencing time elapsed between each poll. However, with this poll we only reset the snapshot once the effect has taken place because heat loss is just a constant in this simulation. We recommend that you build this future yourself, following the form below:

```
impl Future for HeatLossFuture {
    type Output = ();
    fn poll(mut self: Pin<&mut Self>, cx: &mut Co
                                              Poll<Se
        let current_snapshot = Instant::now();
        if current_snapshot.duration_since(self.t
                                            Duration
            TEMP.fetch_sub(1, Ordering::SeqCst);
            self.time_snapshot = Instant::now();
        }
        cx.waker().wake_by_ref();
        return Poll::Pending
    }
}
```

We now have all our futures that will poll continuously as long as the program is running. Running all our futures with the code below will result in a display that will continuously update the temperature and note if the heater is on:

```
#[tokio::main]
async fn main() {
    let display = tokio::spawn(async {
        DisplayFuture::new().await;
    });
    let heat_loss = tokio::spawn(async {
        HeatLossFuture::new().await;
    });
```

```
        let heater = tokio::spawn(async {
            HeaterFuture::new().await;
        });
        display.await.unwrap();
        heat_loss.await.unwrap();
        heater.await.unwrap();
    }
```

You will notice that once the desired temperature is reached, you should see the temperature mildly oscillate between above and below the desired temperature.

---

**WARNING**

Oscillations are standard in classic systems theory. If we add a time snapshot to the display and delay the switching of the HEAT_ON flag the oscillations will get bigger. This is not controversial. However, oscillations need to be noted. If you get a delay in an observer acting, and then another observer is also delayed acting on the outcome of the initial observer, you can get a chaotic system that is very hard to understand or predict. In-fact, this was a big part of supply chain disruption during and after the pandemic. Meadows and Wright's book "thinking in systems" shows how reaction in demand to a supply chain is delayed and thus you get oscillations. Long supply chains had multiple parts oscillating. If the oscillations become too out of pace then you get this very chaotic system that is complex to resolve. This is partly why it took a long time post pandemic to recover supply chains. Luckily computer systems are fairly instantaneous. But it is worth keeping in mind the dangers of chaining delays and reacting to them.

---

Now that our system is working, we can move onto getting input from users using callbacks.

## Getting User Input By Callbacks

When it comes to getting user input from the terminal, we are going to use the device_query crate with the following version:

```
device_query = "1.1.3"
```

With this, we use the traits and structs below:

```
use device_query::{DeviceEvents, DeviceState};
use std::io::{self, Write};
```

The `device_query` crate uses callbacks which are a form of asynchronous programming. Essentially, callbacks are where we pass a function into another function. The function that is passed in is then called. We can code our own basic callback function with the following code:

```
fn perform_operation_with_callback<F>(callback: 
where
    F: Fn(i32),
{
    let result = 42;
    callback(result);
}

```

```
fn main() {
    let my_callback = |result: i32| {
        println!("The result is: {}", result);
    };
    perform_operation_with_callback(my_callback)
}
```

What we have just done is still blocking. We can make our callbacks non-blocking to the main thread by using an event loop thread that is a constant loop. This loop then accepts incoming events which are callbacks as seen in figure 6-3.

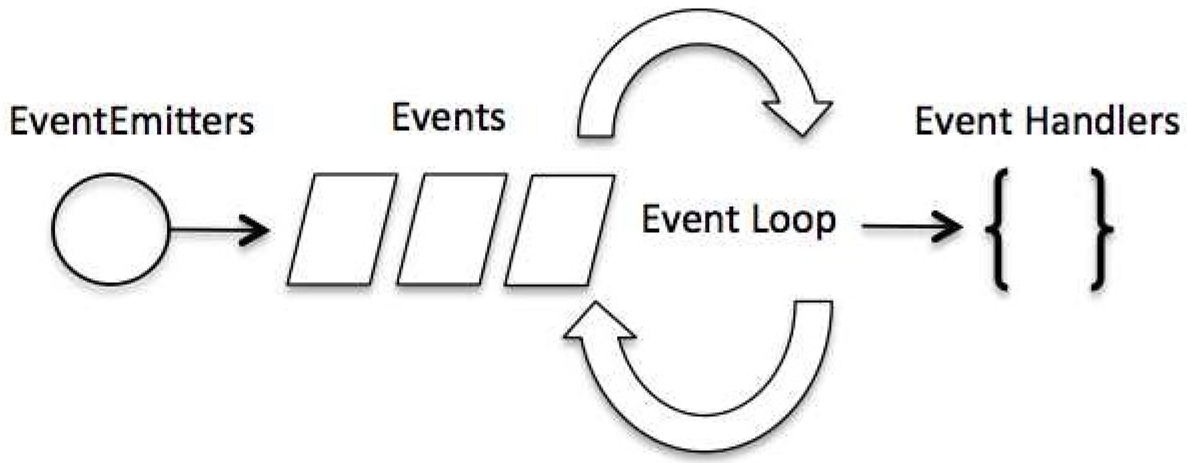


Figure 6-3. An event loop

For example, NodeJS servers usually have a thread pool that the event loop passes events to. If our callback has a channel back to the source of where the event was emitted, data can be sent back to the source of the event when convenient.

For our input, we must keep track of the device state and input with the following code:

```
static INPUT: Lazy<Arc<Mutex<String>>> = Lazy::ne
    Arc::new(Mutex::new(String::new()))
});
static DEVICE_STATE: Lazy<Arc<DeviceState>> = La
    Arc::new(DeviceState::new())
});
```

We now have to think about how our code is structured. Right now, our display is being updated when the display future checks the temperature, updating the display if the temperature has changed. However, this is no longer suitable when we have user input. If we think about it, it would not be a good application if the update of the user input is only displayed if the temperature changes. This would lead to users frustratingly pressing the same key multiple times, only to be dismayed to see their multiple presses executed when the temperature updates. Our system needs to update the display the moment the user presses the key. Considering this, we need our own render function that can be called in multiple places. Our render function takes the form below:

```
pub fn render(temp: i16, desired_temp: i16, heat_
    clearscreen::clear().unwrap();
    let stdout = io::stdout();
    let mut handle = stdout.lock();
    println!("Temperature: {}\nDesired Temp: {}\
    temp as f32 / 100.0,

    desired_temp as f32 / 100.0,
    heat_on);
    print!("Input: {}", input);
    handle.flush().unwrap();
}
```

Here we can see that it is similar to our display future but, we also print out the input. This means that the poll function for our DisplayFuture calls the render function as seen below:

```
#[tokio::main]
async fn main() {
    let _guard = DEVICE_STATE.on_key_down(|key|
        let mut input = INPUT.lock().unwrap();
        input.push_str(&key.to_string());
        std::mem::drop(input);
        render(
            TEMP.load(Ordering::SeqCst),
            DESIRED_TEMP.load(Ordering::SeqCst),
            HEAT_ON.load(Ordering::SeqCst),
            INPUT.lock().unwrap().clone()
        );
     });
    let display = tokio::spawn(async {
```

```
    let display = tokio::spawn(async {
        DisplayFuture::new().await;
    });
    let heat_loss = tokio::spawn(async {
        HeatLossFuture::new().await;
    });
    let heater = tokio::spawn(async {
        HeaterFuture::new().await;
    });
    display.await.unwrap();
    heat_loss.await.unwrap();
    heater.await.unwrap();
}
```

We must note the `_guard` which is the callback guard. The callback guard in the `device_query` crate is returned when adding a callback. If we drop the guard, the event listener is removed. Luckily for us our main thread is blocked until we exit the program due to our display, heat loss, and heater tasks will continuously poll until we force the program to exit.

The `on_key_down` function creates a thread, and runs an event loop. This event loop then has callbacks for mouse and keyboard movements. Once we get an event back from a keyboard press, we add it to our input state and re-render the display. We are not going to expend too much effort on mapping keys to different effects of the display, because that’s a bit too in the weeds for the goal of this book. Running the program now, you should be able to see the input get updated with a trace of the keys that you press down on.

We can see that callbacks are simple and easy to implement. There is also a predictable flow on how the callback is executed. However, you can fall down the trap of having nested callbacks, which can evolve into a situation called “callback hell”. This results in the code being hard to maintain and follow through.

You now have a basic system that takes input from the users. If you want to explore this system even further, alter the input code to handle a change in desired temperature. Note that our system only reacts to basic data types. What if our system requires complex data types to represent events? Also, our system might need to know the order of events and react to all events to function correctly. Not every reactive system is merely reacting to an integer value at the current time. For instance, if we were building a stock trading system, we would want to know the historical data of a stock, not just the current price once we got round to polling it. We also cannot guarantee when the polling happens in async, so when we do get round to polling stock price events, we would want access to all that had happened since the last poll and make a decision ourselves on what events are important. To do this, we need an event bus that we can subscribe to.

## Enabling Broadcasting With An Event Bus

An event bus is a system that enables parts of a system to send messages containing specific information. Unlike broadcast channels that have a simple pub/sub relationship, the event bus can stop at multiple different stops where only a select few people get off. This means that we can have multiple subscribers for updates from a single source, however, those subscribers can request that they only receive messages of a particular type, not every broadcasted message. With an event bus, we can essentially have a subject that publishes the event to an event bus. Multiple observers can then consume that event in the order it was published. In this section we are going to build our own event bus in order to understand the underlying mechanisms. However, broadcast channels are readily available in crates like Tokio.

---

**NOTE**

Broadcast channels are comparable to radio broadcasters. When a radio station emits a message, multiple listeners can listen to the same message as long as they all tune into the same channel. For a broadcast channel in programming, multiple listeners can subscribe and receive the same messages. Broadcast channels are different from regular channels. In regular channels a message is sent by one part of the program and is received by another part of the program. In broadcast channels, a message is sent by one part of the program and that same message is received by multiple parts of the program.

---

Using broadcast channels out of the box is preferable to building your own unless you have specific needs.

Before we build our event bus, we will need the following dependencies:

```
tokio = { version = "1.26.0", features = ["full"
futures = "0.3.28"
```

And the imports below:

```
use std::sync::{Arc, Mutex, atomic::{AtomicU32,
use tokio::sync::Mutex as AsyncMutex;
use std::collections::{VecDeque, HashMap};
use std::marker::Send;
```

We now have everything we need to build our event bus struct.

## Building Our Event Bus Struct

As async programming requires sending structs over threads for an async task to be polled, we are going to have to clone each event published, and distribute those cloned events to every subscriber to consume. The consumers also need to be able to access a backlog of events if for some reason the consumer has been delayed. Consumers also need to be able to unsubscribe to events. Considering all of these factors, our event bus struct takes the following form:

```
pub struct EventBus<T: Clone + Send> {
    chamber: AsyncMutex<HashMap<u32, VecDeque<T>>
    count: AtomicU32,
    dead_ids: Mutex<Vec<u32>>,
}
```

We can see that our events denoted by `T` need to implement the `Clone` trait so they can be cloned and distributed to each subscriber, and the `Send` trait to be sent across threads. Our chamber field is where subscribers with a certain id can access their queue of events. The count will be used to allocate ids, and the `dead_ids` will be used to keep track of consumers that have unsubscribed. It must be noted that the chamber mutex is async, and the `dead_ids` mutex is not async. The chamber mutex is async because we could have loads of subscribers looping and polling the chamber to access their individual queue. We do not want an executor to be blocked by an async task waiting for the mutex. This would slow down the performance of the system considerably. However, when it comes to our `dead_ids`, we will not be looping and polling this field. It will only be accessed when a consumer wants to unsubscribe. Having a blocking mutex also enables us to easily implement an unsubscribe process if a handle is dropped. We will cover the details for this when building our handle.

For our event bus struct, we can now implement the following functions:

```
impl<T: Clone + Send> EventBus<T> {

    pub fn new() -> Self {
        Self {
            chamber: AsyncMutex::new(HashMap::new
            count: AtomicU32::new(0),
            dead_ids: Mutex::new(Vec::new()),
        }
    }
    pub async fn subscribe(&self) -> EventHandle
        . . .
    }
    pub fn unsubscribe(&self, id: u32) {
        self.dead_ids.lock().unwrap().push(id);
    }
    pub async fn poll(&self, id: u32) -> Option<
        . . .
    }
    pub async fn send(&self, event: T) {
        . . .
    }
}
```

Here we can see that all of our functions have a `&self` reference, no mutable references. This is because we are exploiting interior mutability with the atomics and mutexes, as the mutable reference is inside the mutexes, getting round Rust's rule that we can only have one mutable reference at a time. The atomic also does not need a mutable reference, as we can perform atomic operations. This means that our event bus struct can be wrapped in an `Arc`, and cloned multiple times to be sent across multiple threads, enabling those threads to all perform multiple mutable operations on the event bus safely. For our subscribe function we merely push the id to the `dead_ids` field. We will cover the reasoning behind this when building our looping tasks at the end of the chapter.

The first operation that a consumer needs to do is to call the subscribe function of the bus which is defined with the code below:

```
pub async fn subscribe(&self) -> EventHandle<T> 
    let mut chamber = self.chamber.lock().await;
    let id  = self.count.fetch_add(1, Ordering::
    chamber.insert(id, VecDeque::new());
    EventHandle {
        id,
        event_bus: Arc::new(self),
    }
}
```

In the above code we return a `EventHandle` struct we will define the handle in the next subsection. We can see here that we are increasing the count by one, using the new count for the ID, and inserting a new queue under that id. We then return a reference to self which is the event bus wrapped in an `Arc`, coupled with the id in a handle struct to allow the consumer to interact with the event bus.

---

**WARNING**

While increasing the count by one and using that as the new ID is an easy way of allocating IDs, there is a risk for high throughput long running systems to merely run out of numbers eventually. If this risk is a serious consideration, you can add another field for IDs to be reclaimed once they have been cleared from the dead_ids field. You can then pull from the reclaimed IDs when allocating a new ID. Then the increase of the count only happens if there are no IDs in the reclaimed IDs.

---

Now that the consumer has subscribed to the bus, they can poll with the following bus function:

```rust
pub async fn poll(&self, id: u32) -> Option<T> {
    let mut chamber = self.chamber.lock().await;
    let queue = chamber.get_mut(&id).unwrap();
    if queue.is_empty() {
        return None
    }
    Some(queue.pop_front().unwrap())
}
```

We unwrap directly when getting the queue in relation to the ID because we will be interacting through a handle, and we can only get that handle when we subscribe to the bus. Thus, we know that the ID is certainly in the chamber. As each ID has their own queue, each subscriber can consume all the events published in their own time. While this is the simplest implementation of a poll, depending on the problem you are solving, you can alter the poll function by returning the entire queue, and putting a new empty queue into the chamber to reduce the amount of polls called on the bus, as the consumer would then be busy iterating through the queue they just extracted from a poll function call on the bus. Seeing as we are putting our own structs as the events, we could also create a timestamp trait and state that this is required for events being put on the bus. The timestamp would enable us to discard events that have expired when polling is only returning recent events.

Now that we have a basic poll function defined, we can build our send function for the bus with the code below:

```
pub async fn send(&self, event: T) {
    let mut chamber = self.chamber.lock().await;
    for (_, value) in chamber.iter_mut() {
        value.push_back(event.clone());
    }
}
```

We now have everything needed for our bus to function on its internal data structures. We now need to build our own handle.

## Building Our Event Bus Handle

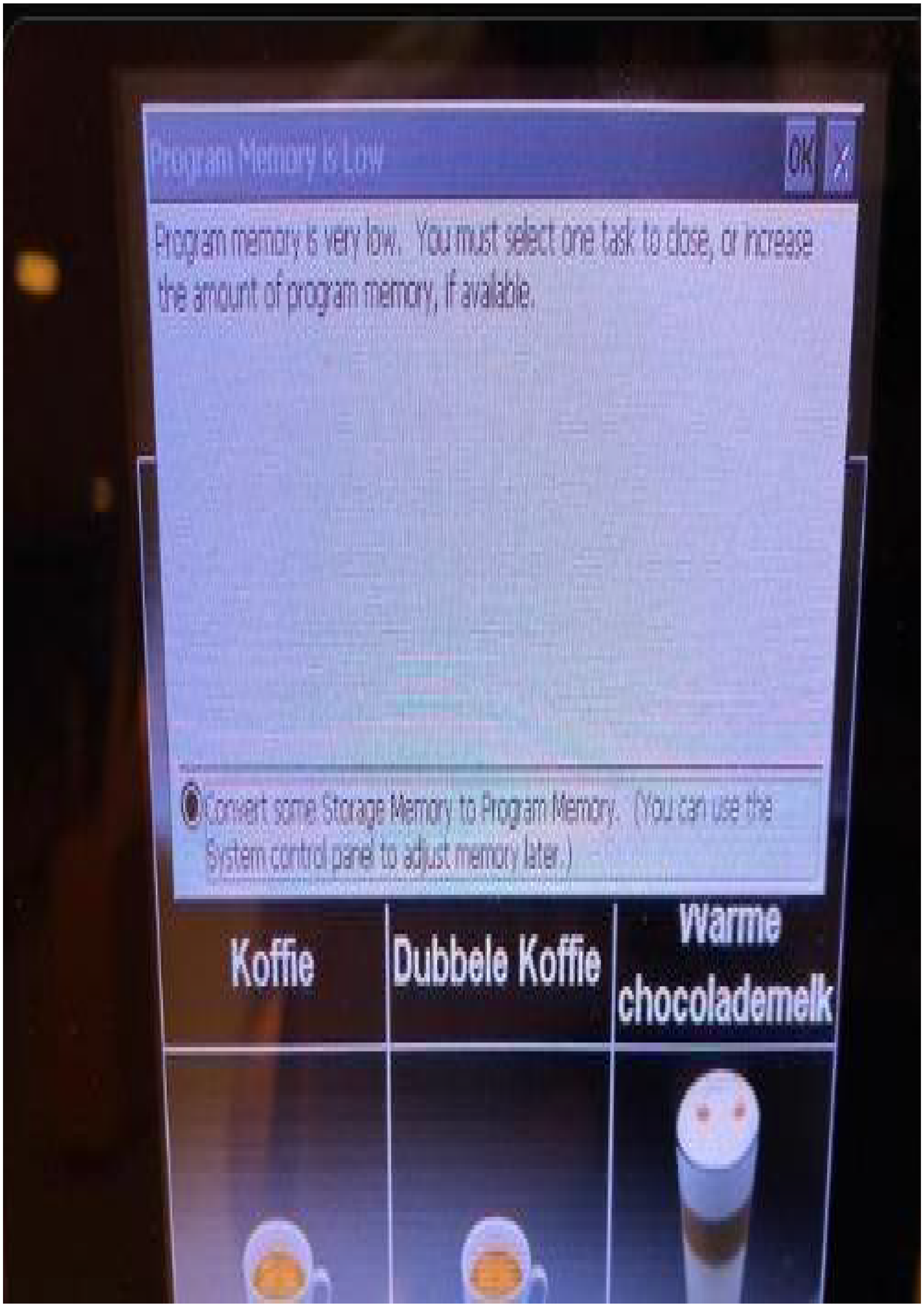
Our handle needs to have an ID and a reference to the bus so the handle can poll the bus. Our handle is defined with the following code:

```rust
pub struct EventHandle<'a, T: Clone + Send> {
    pub id: u32,
    event_bus: Arc<&'a EventBus<T>>,
}
impl <'a, T: Clone + Send> EventHandle<'a, T> {

    pub async fn poll(&self) -> Option<T> {
        self.event_bus.poll(self.id).await
    }
}
```

With the lifetime notation we can see that the handle lifetime cannot outlive the bus lifetime. We must note that `Arc` counts the references, and only drops the bus if there are no `Arcs` in our async system pointing to the bus. Therefore, we can guarantee that the bus will live as long as the last handle in our system, making our handle thread safe.

We also need to take care of dropping the handle. If the handle is removed from memory, there is no way in which we can access the queue relating to the ID of that handle as the handle stores the ID. However, events will keep getting sent to the queue of that ID. If a developer uses our queue and the handle is dropped in their code without explicitly calling the unsubscribe function, they will have an event bus that will fill up with multiple queues that don’t have any subscribers which would waste memory and even grow to the point where the computer runs out of memory depending on certain parameters. This is called a memory leak which is a very real risk. Figure 6-4 is a photograph of a coffee machine that is not suffering from a coffee leak, but from a memory leak.



Program Memory is Low

OK ×

Program memory is very low. You must select one task to close, or increase the amount of program memory, if available.

● Convert some Storage Memory to Program Memory. (You can use the System control panel to adjust memory later.)

Figure 6-4. Memory leak of coffee machine software causing the machine to run out of memory

To prevent memory leaks we must implement the `Drop` trait for our handle which will unsubscribe from the event bus when the handle is dropped using the code below:

```
impl<'a, T: Clone + Send> Drop for EventHandle<'a
    fn drop(&mut self) {
        self.event_bus.unsubscribe(self.id);
    }
}
```

Our handle is now complete and we can use it to safely consume events from the bus without the risk of memory leaks. We are now going to use our handle to build tasks that interact with our event bus.

## Interacting With Our Event Bus With Async Tasks

Throughout this chapter, our observers have been implementing the `Future` trait and comparing the state of the subject to the state of the observer. Now that we are having events directly streamed to our ID, we can easily implement a consumer async task using an async function as seen in the following code:

```
async fn consume_event_bus(event_bus: Arc<EventBu
    let handle = event_bus.subscribe().await;
    loop {
        let event = handle.poll().await;
        match event {
            Some(event) => {
                println!("id: {} value: {}", hanc
                if event == 3.0 {
                    break;
                }
            },
            None => {}
        }
    }
}
```

For our example, we are just streaming a float that breaks the loop if `3.0` is sent. This is just for educational purposes, but implementing logic to affect the `HEAT_ON` atomic bool would be trivial. We could also implement a Tokio async sleep function on the None branch if we did not want to loop to aggressively poll the event bus.

WARNING

There can be situations where the rate of creation of events can be bigger than the rate at which events can be processed. This results in a build up of events which is called backpressure. Backpressure can be solved by a range of approaches beyond the scope of this book as this is an async book, not a reactive programming book. Concepts such as buffering, flow control, rate limiting, batch processing, and load balancing can help reduce back pressure when pressure builds up. We will cover how to test for backpressure of channels in chapter 11.

We also need a background task to clean up the dead IDs as a batch once a certain amount of time has elapsed. This garbage collection task can also be defined using an async function as seen with the code below:

```
async fn garbage_collector(event_bus: Arc<EventBu
    loop {
        let mut chamber = event_bus.chamber.lock(
        let dead_ids = event_bus.dead_ids.lock()
        event_bus.dead_ids.lock().unwrap().clear(
        for id in dead_ids.iter() {
            chamber.remove(id);
        }
        std::mem::drop(chamber);
        tokio::time::sleep(std::time::Duration::f
    }
}
```

Note that we drop the chamber straight after the batch removal on the chamber. We do not want to block other tasks trying to access the chamber when we are not using it.

---

**NOTE**

In database systems, not deleting the record the moment the delete request is made is a common practice. This is called “tombstoning”. Instead, the database marks a record to indicate get requests to treat the record as if it has been deleted. Then, garbage collection processes periodically clean up the tombstoned records. Cleaning and reallocating storage on every delete request is an expensive choice because you want to keep handling async requests to the database.

---

We how have everything needed to interact with our event bus first we create our event bus and the references to the bus with the following code:

```
let event_bus = Arc::new(EventBus::<f32>::new())
let bus_one = event_bus.clone();
let bus_two = event_bus.clone();
let gb_bus_ref = event_bus.clone();
```

Now even if the `event_bus` is directly dropped, due to the `Arc`, the other references will keep the `EventBus<f32>` alive. All four references would have to be dropped. We then start our consumers and garbage collection process tasks with the code below:

```
let _gb = tokio::task::spawn(async {
    garbage_collector(gb_bus_ref).await
});
let one = tokio::task::spawn(async {
    consume_event_bus(bus_one).await
});
let two = tokio::task::spawn(async {
    consume_event_bus(bus_two).await
});
```

In this example, we run the risk of sending events before the two tasks have subscribed, so we wait for one second and then broadcast three events with the following code:

```
std::thread::sleep(std::time::Duration::from_secs
event_bus.send(1.0).await;
event_bus.send(2.0).await;
event_bus.send(3.0).await;
```

The third event is a `3.0` meaning that the consuming tasks will unsubscribe from the bus. We can print the state of the chamber, wait for the garbage collector to wipe the dead IDs, and then print the state again with the code below:

```
let _ = one.await;
let _ = two.await;
println!("{:?}", event_bus.chamber.lock().await)
std::thread::sleep(std::time::Duration::from_secs
println!("{:?}", event_bus.chamber.lock().await)
```

Running this would give us the following printout:

```
id: 0 value: 1
id: 1 value: 1
id: 0 value: 2
id: 1 value: 2
id: 0 value: 3
id: 1 value: 3
{1: [], 0: []}
{}
```

Here we can see that both subscribers receive the events, and the garbage collection works when they unsubscribe.

We can see how the event bus is the backbone of reactive programming. We can continue to add and remove subscribers in a dynamic fashion. We can have control over how the events are distributed and consumed, and implementing code that just hooks into an event bus is simple.

## Conclusion

While it was beyond the scope of this book to give a comprehensive view of reactive programming, we have covered the fundamental async properties of reactive programming such as polling subjects, and distributing data asynchronously through an event bus we wrote ourselves. With what we covered, you should be able to come up with async implementations of reactive programming.

Reactive programming is not just constrained to just one program with different threads and channels. Reactive programming concepts can be applied to multiple computers and processes under the title “reactive systems”. For instance, our message bus could be sending messages to different servers in a cluster. Event driven systems are also useful when scaling architecture. We have to remember with reactive programming there are more moving parts to a solution. Maxwell and Caroline only moved to event driven systems when the live system started to fail in performance. Reaching for reactive programming straight away can result in convoluted solutions that are hard to maintain so be careful.

You may have noticed that we have relied on Tokio for implementing our async code. In the next chapter we will be covering how to customize Tokio to solve more problems that have constraints and nuances. It could be connsidered controversial to dedicate an entire chapter to Tokio, but Tokio is actually the widest-used async runtime in the Rust ecosystem

# Chapter 7. Customising Tokio

---

---

Throughout this book we have been using Tokio for examples because not only is Tokio well established, but it also has a clean syntax, and you can get async examples running with just a single macro. Chances are, if you have worked on an async Rust codebase, you will have come across Tokio. However, so far we have only just used Tokio to build a standard Tokio runtime, and then send async tasks to that runtime. In this chapter, we will customise our Tokio runtimes to receive fine-grained control over how our tasks are processed in different threads of a set. We will also test to see if our unsafe access to thread state is actually safe in an async runtime.

Finally, we will cover how to enable graceful shutdowns when our async runtime finishes.

By the end of this chapter you will be able to take a Tokio runtime, and configure it to solve your specific problem. You will also be able to specify what thread the async task is exclusively processed on so your task can rely on the thread specific state, potentially reducing the need for locks to access data. Finally you will be able to specify how the program is shutdown when control+C or kill signals are sent to the program. So, let us get started with building the Tokio runtime.

## Building a Runtime

In chapter three, we got to understand how tasks are handled in an async runtime by implementing our own spawn task function. This gave us a lot of control over how the tasks were processed. Our previous Tokio examples have merely just used the `#[tokio::main]` macro. Whilst this macro was useful for implementing async examples with minimal code, just implementing the `#[tokio::main]` macro does not give us much control over how the async runtime is implemented. For us to explore Tokio, we can start with setting up a Tokio runtime that we can choose to call. For our configured runtime, we need the following dependencies:

```
tokio = { version = "1.33.0", features = ["full"]
once_cell = "1.18.0"
```

With these dependencies we need the following structs and traits:

```
use std::future::Future;
use std::time::Duration;
use tokio::runtime::{Builder, Runtime};
use tokio::task::JoinHandle;
use once_cell::sync::Lazy;
```

To build our runtime, we can lean on the `once_cell` for a lazy evaluation so our runtime is defined once, just like we did when building our runtime in chapter three with the code below:

```
static RUNTIME: Lazy<Runtime> = Lazy::new(|| {
    Builder::new_multi_thread()
        .worker_threads(4)
        .max_blocking_threads(1)
        .on_thread_start(|| {
            println!("thread starting for runtime
        })
        .on_thread_stop(|| {
            println!("thread stopping for runtime
        })

        .thread_keep_alive(Duration::from_secs(60
        .global_queue_interval(61)
        .on_thread_park(|| {
            println!("thread parking for runtime
        })
        .thread_name("our custom runtime A")
        .thread_stack_size(3 * 1024 * 1024)
        .enable_time()
        .build()
        .unwrap()
});
```

We can see that we get a lot of configuration out of the box along the following properties:

*worker_threads*

The number of threads processing async tasks

*max_blocking_threads*

The number of threads that can be allocated to blocking tasks. Blocking tasks is where a task does not allow switching due to having no await, or long periods of CPU computation between await statements. Therefore the thread is blocked for a fair amount of time processing the task. CPU intensive tasks or synchronous tasks are usually referred to as blocking tasks. If we block all our threads then no other tasks can be started. As mentioned throughout the book, this can be ok depending on what problem your program is solving. However, if for instance we are using async to process incoming network requests, we want to still process more incoming network requests. Therefore, with `max_blocking_threads`, we can limit the number of additional threads that can be spawned to process blocking tasks. We can spawn blocking tasks with the runtime's spawn_blocking function.

*on_thread_start/stop*
Functions that fire when the worker thread starts or stops. This can become useful if you want to build your own monitoring.

*thread_keep_alive*
Timeout for blocking threads. Once the time has elapsed for a blocking thread, the task that has overrun that timeout limit will be cancelled.

*global_queue_interval*
This is the number of "ticks" before a new task gets attention from the scheduler. A "tick" represents one instance when the scheduler polls a task to see if it can be run or if it needs to wait. In our configuration, once 61 ticks have elapsed, the scheduler will take on a new task that has been sent to the runtime. If there are no tasks to poll, the scheduler will take on new task sent to the runtime without waiting 61 ticks. There is a tradeoff between fairness and overhead. The lower the number of ticks, the quicker new tasks sent to the runtime receive attention. However, we will also be checking the queue for incoming new tasks more frequently which comes with overhead. Your system might become less efficient if we are constantly checking for new tasks instead of making progress

with existing ones. We also must acknowledge the number of await statements per task. If our tasks generally contain a lot of await statements, then the scheduler needs to work through a lot of steps, polling on each await statement to complete the task. However, if there is just one await statement in the task, then the scheduler will require less polling to progress the task. The tokio team has decided that the default tick number should be 31 for single threaded runtimes, and 61 for multithreaded runtimes. The multithreaded suggestion is a higher tick count as there are multiple threads consuming tasks, resulting in these tasks getting attention at a quicker rate.

*on_thread_park*
Functions that fire when the worker thread is parked. Worker threads are usually parked when there are no tasks for the worker thread to consume. The `on_thread_park` is useful if you want to implement your own monitoring.

*thread_name*
This merely names the threads that are made by the runtime. The default name is “tokio-runtime-worker”.

*thread_stack_size*
This allows us to determine the amount of memory in bytes that are allocated for the stack of each worker thread. The stack is a section of memory that stores local variables, function return addresses, and the

management of function calls. If you know that your computations are simple and you want to conserve memory, then reaching for a lower stack size makes sense. The default value for this stack size at the time of writing this book is 2MiB.

*enable_time*

This merely enables the time driver for tokio.

Now that we have built and configured our runtime, we can define how we call our runtime with the code below:

```
pub fn spawn_task<F, T>(future: F) -> JoinHandle<
where
    F: Future<Output = T> + Send + 'static,
    T: Send + 'static,
{
    RUNTIME.spawn(future)
}
```

We can see that we do not really need the function as we can just directly call our runtime, however, it is worth noting that the function signature is essentially the same as our `spawn_task` function in chapter three. The only difference is that we return a tokio `JoinHandle` as opposed to a `Task`.

Now that we know how to call our runtime, we can define a basic future with the following code:

```
async fn sleep_example() -> i32 {
    println!("sleeping for 2 seconds");
    tokio::time::sleep(Duration::from_secs(2)).aw
    println!("done sleeping");


    20
}
```

And then run our program with the code below:

```
fn main() {
    let handle = spawn_task(sleep_example());
    println!("spawned task");
    println!("task status: {}", handle.is_finishe
    std::thread::sleep(Duration::from_secs(3));
    println!("task status: {}", handle.is_finishe
    let result = RUNTIME.block_on(handle).unwrap(
    println!("task result: {}", result);
}
```

Here we can see that we spawn our task, and then wait for the task to finish using the `block_on` function from our runtime. We also check to see if our task has finished periodically. Running the code will give us the following printout:

```
thread starting for runtime A
thread starting for runtime A
sleeping for 2 seconds
thread starting for runtime A
thread parking for runtime A
thread parking for runtime A
spawned task
thread parking for runtime A
task status: false
thread starting for runtime A
thread parking for runtime A
done sleeping
thread parking for runtime A
task status: true
task result: 20
```

Although this printout is lengthy, we can see that our runtime starts creating worker threads, and also starts our async task before all the worker threads are created. Because we have only sent one async task, we can also see that the idle worker threads are being parked. By the time that we have gotten the result of our task, all of our worker threads have been parked. We can see that Tokio is fairly aggressive at parking its threads. This is useful because if we create multiple runtimes but we are not using one all the time, that unused runtime will quickly park their threads reducing the amount of resources being used.

Now that we have covered how to build and customise Tokio runtimes, we can recreate the runtime that we built in chapter three with the following code:

```rust
static HIGH_PRIORITY: Lazy<Runtime> = Lazy::new(
    Builder::new_multi_thread()
        .worker_threads(2)
        .thread_name("High Priority Runtime")
        .enable_time()
        .build()
        .unwrap()
});
static LOW_PRIORITY: Lazy<Runtime> = Lazy::new(|
    Builder::new_multi_thread()
        .worker_threads(1)
        .thread_name("Low Priority Runtime")
        .enable_time()
        .build()
        .unwrap()
});
```

This gives us the layout shown in figure 7-1.

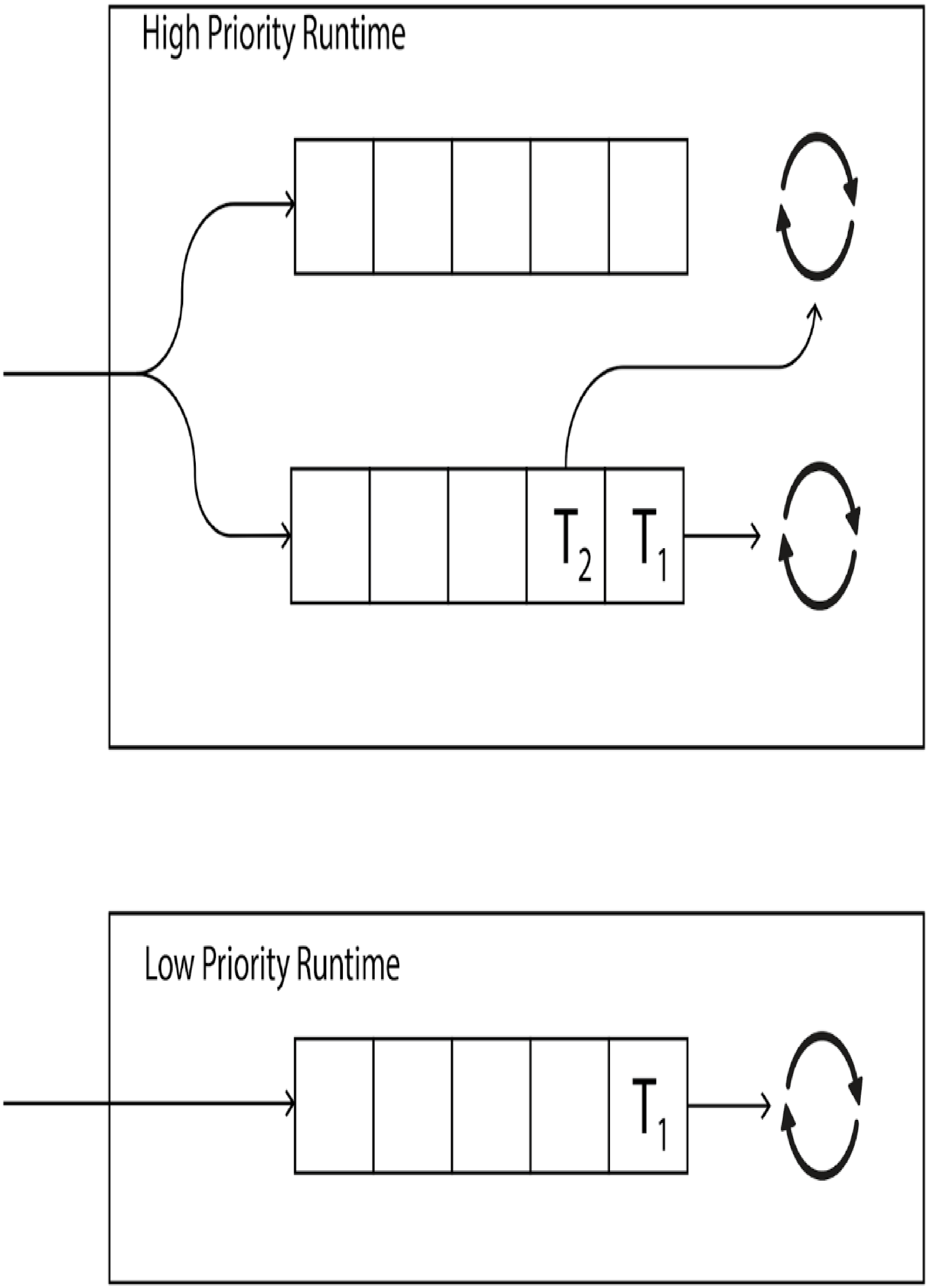

High Priority Runtime
$T_2$
$T_1$
Low Priority Runtime
$T_1$

Figure 7-1. Layout of our Tokio runtimes

The only difference between our two Tokio runtimes and our runtime that had two different queues with task stealing in chapter three, is that the threads from the high priority runtime will not steal tasks from the low priority runtime. Also the high priority runtime has two queues. The differences are not too pronounced as the threads task steal in the same runtime, so it is effectively one queue as long as we do not mind the exact order in which tasks are processed. We also must acknowledge that the threads get parked when there are no async tasks to be processed. If we have more threads than cores, then the OS will manage the resource allocation and context switching between these threads. Simply adding more threads past the number of cores will not result in a linear increase in speed. However, if we have three threads for the high priority runtime, and two threads for the low priority runtime, we can still distribute the resources effectively. If no tasks were to be processed in the low priority runtime, those two threads would be parked and the three threads in the high priority runtime would have more CPU allocation.

Now that we have defined our threads and runtimes, we need to interact with these threads in different ways. We can gain more control over the flow of the task using local pools.

## Processing Tasks With LocalPools

With local pools, we can have more control over the threads that are processing our async tasks. Before we explore local pools, we need to include the following dependency:

```
tokio-util = { version = "0.7.10", features = ["t
```

We also need the imports below:

```
use tokio_util::task::LocalPoolHandle;
use std::cell::RefCell;
```

When using local pools, we tie the spawned async task to the specific pool. This means we can use structs that do not have the Send trait implemented as we are ensuring that the task stays on on a specific thread. However, because we are ensuring that the async task runs on a particular thread, we will not be able to exploit task stealing meaning that we will not get the performance of a standard Tokio runtime out of the box.

To map how our async tasks map through our local pool, we first need to define some local thread data with the following code:

```
thread_local! {
    pub static COUNTER: RefCell<u32> = RefCell::n
}
```

This means that every thread will have access to a `COUNTER` variable for that specific thread. We then need a simple async task that blocks the thread for a second, increases the `COUNTER` of the thread that async task is operating in, and then printout the `COUNTER` and number with the code below:

```
async fn something(number: u32) -> u32 {
    std::thread::sleep(std::time::Duration::from_
    COUNTER.with(|counter| {
        *counter.borrow_mut() += 1;
        println!("Counter: {} for: {}", *counter
    });
    number
}
```

With this task, we will see how configurations of the local pool will process multiple tasks.

In our main function, we still need a tokio runtime as we still need to await on the spawned tasks which we do with the following code:

```
#[tokio::main(flavor = "current_thread")]
async fn main() {
    let pool = LocalPoolHandle::new(1);
    . . .
}
```

Our tokio runtime has a flavor of “current_thread”. The flavor at the time of writing this book is either `CurrentThread` or `MultiThread`. `MultiThread` executes tasks across multiple threads. `CurrentThread` executes all tasks on the current thread. There is one more flavor that is `MultiThreadAlt` which also claims to execute tasks across multiple threads but is unstable. So, the runtime that we have implemented will execute all tasks on the current thread, and the local pool has one thread in the pool.

Now that we have defined our pool, we can now use our pool to spawn our tasks with the code below:

```
let one = pool.spawn_pinned(|| async {
    println!("one");
    something(1).await
});
let two = pool.spawn_pinned(|| async {
    println!("two");
    something(2).await
});
let three = pool.spawn_pinned(|| async {
    println!("three");
    something(3).await
});
```

We now have three handles, so we can await on these handles and return the sum of these tasks with the following code:

```
let result = async {
    let one = one.await.unwrap();
    let two = two.await.unwrap();
    let three = three.await.unwrap();
    one + two + three
};
println!("result: {}", result.await);
```

When running our code we get the following printout:

```
one
Counter: 2 for: 1
two
Counter: 3 for: 2
three
Counter: 4 for: 3
result: 6
```

Here we can see that our tasks are processed sequentially, and that the highest `COUNTER` value is 4, meaning that all of the tasks were processed in one thread. Now, if we increase the local pool size to 3, we get the printout below:

```
one
three
two
Counter: 2 for: 1
Counter: 2 for: 3
Counter: 2 for: 2
result: 6
```

We can see that all three tasks started processing as soon as they were spawned. We can also see that the `COUNTER` has a value of 2 for each task. This means that our three tasks were distributed across all three threads.

We can also focus on particular threads. For example, we can spawn a task to a thread that has the index of zero with the following code:

```
let one = pool.spawn_pinned_by_idx(|| async {
    println!("one");
    something(1).await
}, 0);
```

If we spawn all of our tasks on the thread with the index of zero, we get the printout below:

```
one
Counter: 2 for: 1
two
Counter: 3 for: 2
three
Counter: 4 for: 3
result: 6
```

Here, we can see that our printout is the same as the single threaded pool even though we have three threads in the pool. If we were to swap the standard sleep to a tokio sleep, we get the following printout:

```
one
two
three
Counter: 2 for: 1
Counter: 3 for: 2
Counter: 4 for: 3
result: 6
```

We can see that because the tokio sleep is async, our single thread can juggle multiple async tasks, however, the `COUNTER` access is after the sleep. We can see the `COUNTER` value is `4`, meaning that although our thread juggled multiple async tasks at the same time, our async tasks never traversed over another thread.

With local pools, we can have fine grained control on where we send our tasks to be processed. Whilst we are sacrificing task stealing, we may want to use the local pool for the following advantages:

*Handling non Send futures*
: If the future cannot be sent between threads then we can process them with a local thread pool.

*Thread Affinity*
: Because we can ensure that a task is being executed on a specific thread, we can take advantage of the state of the thread. A simple example of this is caching. If we need to compute or fetch a value from another resource such as a server, we can cache this in a specific thread. All tasks in that thread then have access to the value so all tasks you send to that specific thread will not need to fetch or calculate the value.

*Performance for thread-local operations*
: You can share data across threads with Mutexes and atomic reference counters. However, there is some overhead with the synchronisation between threads. For instance, acquiring a lock that other threads are also acquiring is not free. As we can see in figure 7-2, if we have a standard Tokio async runtime with four worker threads and our counter is an `Arc<Mutex<T>>`, only one thread can access the counter at a time.

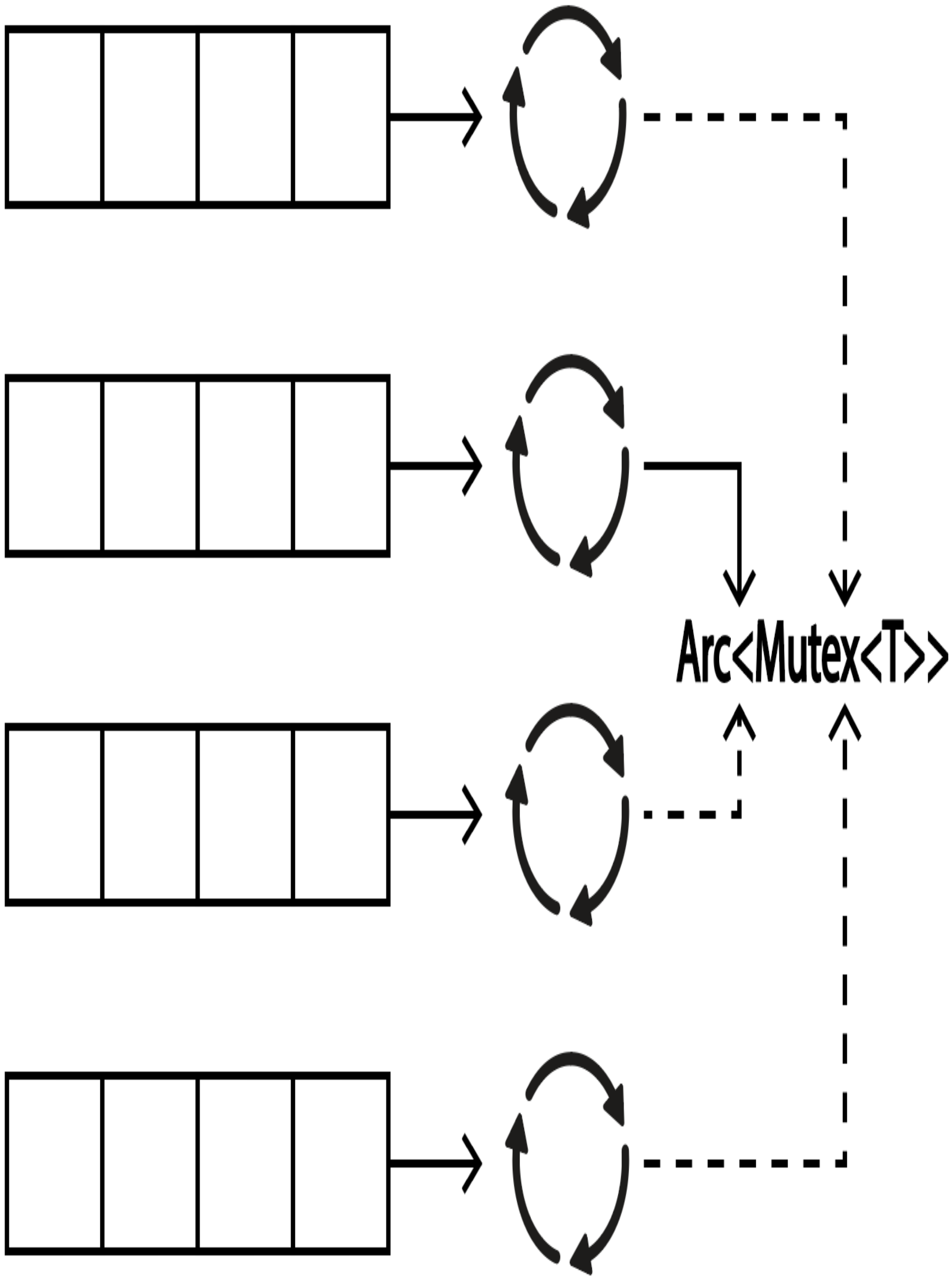

Arc<Mutex<T>>

Figure 7-2. A Mutex will only allow one Tokio thread to access it at a time

This means that the other three threads will have to wait to get access to the `Arc<Mutex<T>>`. Keeping the state of the counter local to each thread will remove the need for that thread to wait for access to a Mutex, speeding up the process. However, it must be noted that the local counters in each thread would not contain the complete picture. These counters do not know the state of the other counters in other threads. One approach for getting the entire state of the count can be sending an async task that gets counter to each thread, combining the results of each thread at the end. We will cover this approach when we cover graceful shutdowns, later in this chapter. The local access to data within the thread can also aid in the optimizations of CPU bound tasks when it comes to the CPU caching data to optimise operations.

> *Safe access to non Send Resources*
>
> Sometimes the data resource will not be thread safe. Keeping that resource in one thread and sending tasks into that thread to be processed is a way of getting around this.

---

**WARNING**

We have highlighted the potential for blocking a thread with a blocking task throughout the book. However, it must be stressed that the damage blocking can do on our local pool can be more pronounced as we do not have any task stealing. Using the tokio `spawn_blocking` function will prevent this.

---

So far we have been able to access the state of the thread in our async task using `RefCell`. `RefCell` enables us to access data with Rust checking the borrow rules at runtime. However, there is some overhead to this checking when borrowing the data in the `RefCell`. We can remove these checks and still safely access the data with unsafe code, which we will explore in the next section.

## Getting Unsafe With Thread Data

When it comes to removing the runtime checks for mutable borrows of our thread data, we need to wrap our data in an `UnsafeCell`. This means that we access our thread data directly without any checks. However, I know what you are thinking. If we are using an `UnsafeCell`, is that dangerous? Potentially yes, so we must be careful to ensure that we are safe. If we think about our system, we have a single thread that is processing async tasks that will not transfer to other threads. We must remember that whilst this single thread can juggle multiple async tasks at the same time through polling, the single thread can only actively process one async task at a time. Therefore, we can assume that whilst one of our async tasks is accessing the data in the `UnsafeCell` and processing it, no other async task is accessing the data as the `UnsafeCell` is not async. However, we need to make sure that we do not have an await when the reference to the data is in scope. If we do, then our thread could context switch to another task whilst the existing task still has a reference to the data. We can test this by exposing a hashmap in unsafe code to thousands of async tasks and increasing the value of a key in each of those tasks. In order to run this test, we need the following imports:

```
use tokio_util::task::LocalPoolHandle;
use std::time::Instant;
use std::cell::UnsafeCell;
use std::collections::HashMap;
```

We then define our thread state with the code below:

```
use std::cell::UnsafeCell;
use std::collections::HashMap;
thread_local! {
    pub static COUNTER: UnsafeCell<HashMap<u32, u
    (HashMap::new());
}
```

Now that we have our thread state defined, we can define our async task that is going to access and update the thread data using unsafe code with the following code:

```
async fn something(number: u32) {
    tokio::time::sleep(std::time::Duration::from_
    COUNTER.with(|counter| {
        let counter = unsafe { &mut *counter.get(
        let counter = unsafe { &mut *counter.get
        match counter.get_mut(&number) {
            Some(count) => {
                let placeholder = *count + 1;
                *count = placeholder;
            },
            None => {
                counter.insert(number, 1);
            }
        }
    });
}
```

Here we can see that we add in a Tokio sleep with the duration of the number put in to shuffle the async tasks around in terms of the order that the tasks are going to access the thread data. We then obtain a mutable reference to the data and perform an operation. We must note the `COUNTER.with` block where we access the data. This is not an async block, meaning that we cannot put an await operation whilst accessing the data. This means that we cannot context switch to another async task whilst accessing the unsafe data. Inside the `COUNTER.with` block we use unsafe code to directly access the data and increase the count.

Once our test is done we are going to need to print out the thread state, so we are going to need to pass an async task into the thread to perform the print operation which takes the form below:

```
async fn print_statement() {
    COUNTER.with(|counter| {
        let counter = unsafe { &mut *counter.get
        println!("Counter: {:?}", counter);
    });
}
```

We now have everything, so all we need to do is run our code in our main async function. First, we set up our local thread pool which is just a single thread and 100,000 sequences of 1 to 5 with the following code:

```
let pool = LocalPoolHandle::new(1);
let sequence = [1, 2, 3, 4, 5];
let repeated_sequence: Vec<_> = sequence.iter().
```

This gives us half a million async tasks with varying Tokio sleep durations that we are going to chuck into this single thread. We then loop through these numbers spinning off tasks that all our async function twice so the task sent to the thread makes the thread context switch between each function, and inside each function with the code below:

```
let mut futures = Vec::new();
for number in repeated_sequence {
    futures.push(pool.spawn_pinned(move || async
        something(number).await;
```

```
            something(number).await
        }));
    }
```

We are really encouraging the thread to context switch multiple times when processing a task. This context switching combined with the varying sleep durations and high number of tasks in total will lead to inconsistent outcomes in the counts if we have clashes when accessing the data. Finally, we loop through the handles joining them all to ensure that all of the async tasks have executed, and we print out the count with the following code:

```
for i in futures {
    let _ = i.await.unwrap();
}
let _ = pool.spawn_pinned(|| async {
    print_statement().await
}).await.unwrap();
```

The end result should have the results below:

```
Counter: {2: 200000, 4: 200000, 1: 200000, 3: 20(
```

No matter how many times we run them, the counts will always be the same. Here we did not have to perform atomic operations such as compare and swap, with multiple tries if there is an inconsistency. We also did not need to await on a lock. We didn’t even need to check to see if there were any mutable references before making a mutable reference to our data. Our unsafe code in this context is safe.

We can now utilise the state of a thread to affect our async tasks. However, what happens if our system is shut down? We might want to have a cleanup process so we can recreate our state when we spin up our runtime again. This is where graceful shutdowns come in.

## Graceful Shutdowns

Essentially, a graceful shutdown is where we catch when the program is shutting down, in order to perform a series of processes before the program exits. These processes can be sending signals to other programs, storing state, clearing up transactions, and anything else you would want before the program exits.

Our first exploration of this can be the `control+C` signal. Usually, when we run a rust program through the terminal, we can stop our program by pressing `control+C` prompting the program to exit. However, we can overwrite this preemptive exit with the `tokio::signal` module. To really prove that we have overwritten the `control+C` signal, we can build a simple program that has to accept the `control+C` signal three times before we exit our program. We can achieve this by building the background async task with the code below:

```rust
async fn cleanup() {
    println!("cleanup background task started");
    let mut count = 0;
    loop {

        tokio::signal::ctrl_c().await.unwrap();
        println!("ctrl-c received!");
        count += 1;
        if count > 2 {
            std::process::exit(0);
        }
    }
}
```

Now that we have our background task, we can run it and loop indefinitely with the following main function:

```rust
#[tokio::main]
async fn main() {
    tokio::spawn(cleanup());
    loop {
    }
}
```

When running our program, if we press control+C three times, we will get the printout below:

```
cleanup background task started
^Cctrl-c received!
^Cctrl-c received!
^Cctrl-c received!
```

Here we can see that our program did not exit until the signal was sent three times. Now we can exit our program on our own terms. However, before we move on, lets just add a blocking sleep to our loop in our background task before we await for the `control+C` signal giving the following loop:

```
loop {
    std::thread::sleep(std::time::Duration::from_
    tokio::signal::ctrl_c().await.unwrap();
    . . .
}
```

If we were to run our program again, pressing `control+C` before the 5 seconds is up will just result in the program exiting. With this, we can deduce that our program will only handle the control+C as we want when our program is directly awaiting the signal. We can get around this by spawning a thread that will manage an async runtime. We then use the rest of the main thread to listen for our signal with the code below:

```
#[tokio::main(flavor = "current_thread")]
async fn main() {
    std::thread::spawn(|| {
        let runtime = tokio::runtime::Builder::ne
            .enable_all()
            .build()
            .unwrap();

        runtime.block_on(async {
            println!("Hello, world!");

        });
    });
    let mut count = 0;
    loop {
        tokio::signal::ctrl_c().await.unwrap();
        println!("ctrl-c received!");
        count += 1;
        if count > 2 {
            std::process::exit(0);
        }
    }
}
```

Now, no matter what our async runtime is processing, our main thread is ready to act on our `control+C` signal, but what about our state? In our cleanup process, we can extract the current state, and then write the state to a file, so we can load the state when the program is started again. Writing and reading files is trivial, so we will focus on the extraction of the state from all of the isolated threads we built in the previous section. The main difference from the previous section, is that we are going to distribute the tasks over 4 isolated different threads. First, we can have our local thread pool wrapped in a lazy evaluation with the following code:

```
static RUNTIME: Lazy<LocalPoolHandle> = Lazy::new
    LocalPoolHandle::new(4)
});
```

We now need to define our async task that extracts the state of a thread with the code below:

```
fn extract_data_from_thread() -> HashMap<u32, u3
    let mut extracted_counter: HashMap<u32, u32>
    COUNTER.with(|counter| {
        let counter = unsafe { &mut *counter.get(
        extracted_counter = counter.clone();
    });

    return extracted_counter
}
```

We can then send this task through each thread which gives us a non-blocking way to sum the total number of counts for the entire system as seen in figure 7-3.

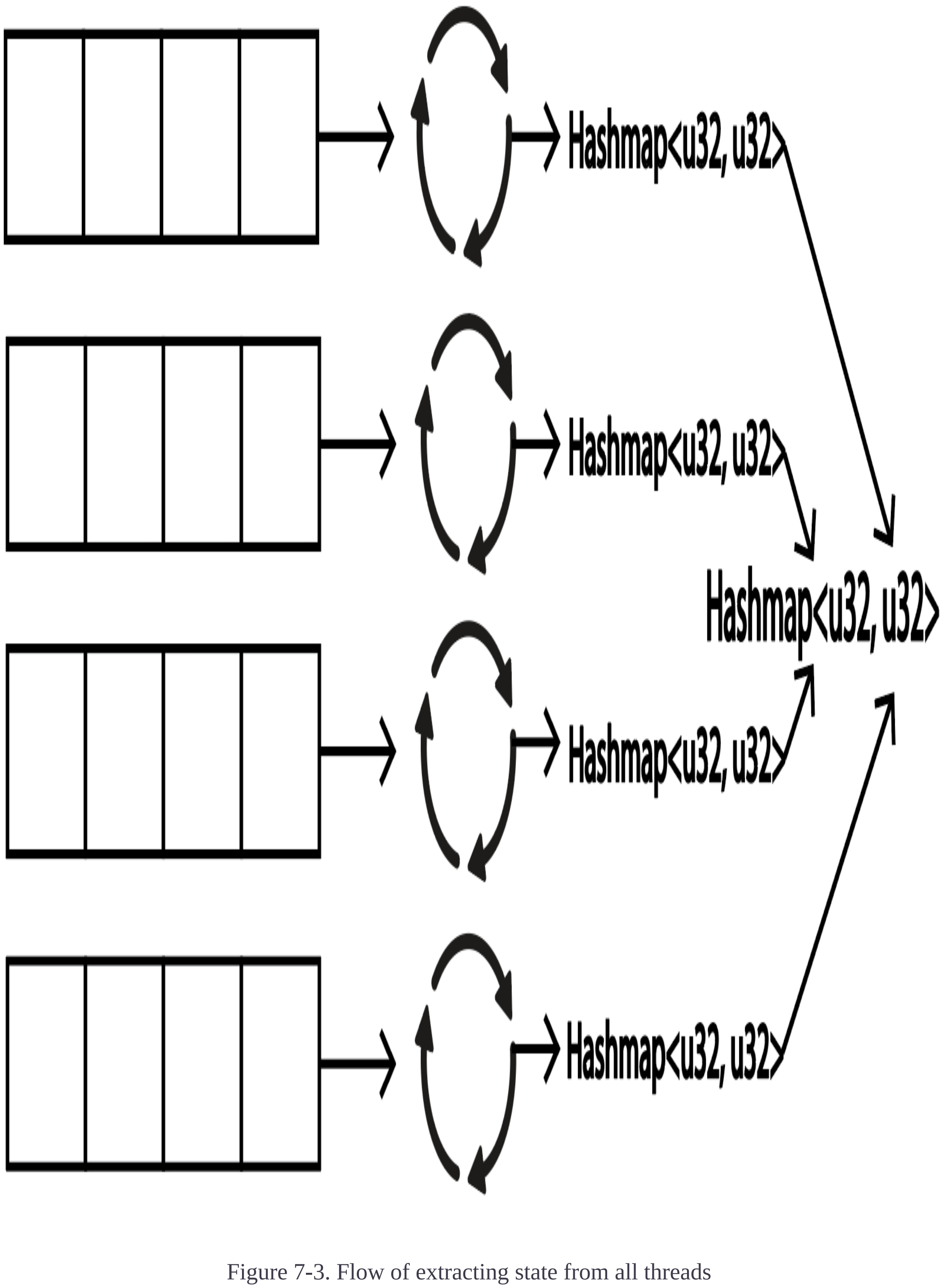


Figure 7-3. Flow of extracting state from all threads

We can implement the process mapped out in figure 7-3 with the following code:

```
async fn get_complete_count() -> HashMap<u32, u32
    let mut complete_counter = HashMap::new();
    let mut extracted_counters = Vec::new();
    for i in 0..4 {
        extracted_counters.push(RUNTIME.spawn_pin
            async move {
                extract_data_from_thread()
        }, i));
    }
    for counter_future in extracted_counters {
        let extracted_counter = counter_future.aw
                                         .unwrap_or_defa
        for (key, count) in extracted_counter {
            *complete_counter.entry(key).or_inser
        }
    }
    return complete_counter
}
```

We can see that we call the `spawn_pinned_by_idx` to ensure that we only send one `extract_data_from_thread` task to every thread.

We are now ready to run our system with the following main function:

```
#[tokio::main(flavor = "current_thread")]
async fn main() {
    let _handle = tokio::spawn( async {
        . . .
    });
    tokio::signal::ctrl_c().await.unwrap();
    println!("ctrl-c received!");
    let complete_counter = get_complete_count().a
    println!("Complete counter: {:?}", complete_c
}
```

Where we spawn tasks to increase the counts inside the `tokio::spawn` with the code below:

```
let sequence = [1, 2, 3, 4, 5];
let repeated_sequence: Vec<_> = sequence.iter().c
                                               .t
                                               .c
                                               .c
let mut futures = Vec::new();
for number in repeated_sequence {
    futures.push(RUNTIME.spawn_pinned(move || asy

        something(number).await;
        something(number).await
    }));
}
```

```
for i in futures {
    let _ = i.await.unwrap();
}
println!("All futures completed");
```

Our system is now ready to run. If you run the program the till you get the printout that all futures are completed before pressing the `control+C` we get the following printout:

```
Complete counter: {1: 200000, 4: 200000, 2: 20000
```

Because we know that we only sent one extract task to each thread using the `spawn_pinned_by_idx` function, and that our total count is the same as it was when we were running all our tasks through one thread, we can conclude that our data extraction is accurate. If we press `control+C` before the futures have finished we should get something similar to the printout below:

```
Complete counter: {2: 100000, 3: 32290, 1: 200000
```

Here we can see that we have exited the program before the program finishes, and get the current state. Our state is now ready to be written before we exit if we want.

Whilst our code facilitates a cleanup when we press `control+C`, this signal is not always the most practical method of shutting down our system. For instance, we might have our async system running in the background so our terminal is not tethered to the program. We can shutdown our program by sending a `SIGHUP` signal to our system. To listen for the `SIGHUP` signal, we need the following import:

```
use tokio::signal::unix::{signal, SignalKind};
```

We can then replace the control+C code at bottom of our main function with the code below:

```
let pid = std::process::id();
println!("The PID of this process is: {}", pid);
let mut stream = signal(SignalKind::hangup()).unw
stream.recv().await;
let complete_counter = get_complete_count().await
println!("Complete counter: {:?}", complete_count
```

Here we print out our PID so we know that we know which PID to send the signal to with the following command:

```
kill -SIGHUP <pid>
```

When running the kill command you should have similar results to when you were pressing the `control+C`. And with this, we can now say that we know how to customise Tokio in the way the runtime is configured, how the tasks are run, and how the runtime is shutdown.

## Conclusion

In this chapter we went into the specifics of setting up a Tokio runtime, and how the settings of the runtime affected how the runtime operates. With these specifics we really got to take control of the number of workers and blocking threads the runtime has, and how many ticks the number performs before accepting a new task to be polled. We also got to explore how to define different runtimes in the same program so we can choose which runtime to send the task on. Remember, the threads in the Tokio runtime get parked when they are not being used, so we will not be wasting resources if a Tokio runtime is not being constantly used.

We then controlled how our tasks were handled by threads with local pools. We even tested our unsafe access to our thread state of the threads in the Tokio runtime to show that accessing the thread state in a task is safe. Finally we covered graceful shutdowns. We can see that although we do not have to write our own boilerplate code, Tokio still gives us the ability to configure our runtime with a lot of flexibility. We have no doubt that in your async rust career you will come across a codebase that is using Tokio.

You should now be comfortable to customise the Tokio runtime in the codebase and manage how your async tasks are being processed. In the next chapter we will be implementing the actor model to solve problems in an async way that is modular.

# Chapter 8. The Actor Model

Actors are isolated pieces of code that communicate exclusively through message passing. Actors can also have state that they can reference and manipulate. Because we have async compatible non-blocking channels, our async runtime can essentially juggle multiple actors, only progressing these actors when they receive a message in their channel.

The isolation of actors enables easy async testing and simple implementation of async systems. By the end of this chapter, you will be able to build an actor system that has a router actor. This means that the actor system you build can easily be called anywhere in your program without having to pass a reference around for your actor system. You will also be able to build a supervisor heartbeat system that will keep track of other actors and force a restart of those actors if they fail to ping the supervisor past a time threshold. To start on this journey, you need to understand how to build basic actors.

## Building a Basic Actor

The most basic actor we can build is essentially an async function that is stuck in an infinite loop listening for messages as shown in the following code:

```
use tokio::sync::{
    mpsc::channel,
    mpsc::{Receiver, Sender},
    oneshot
};

struct Message {
    value: i32
}

async fn basic_actor(mut rx: Receiver<Message>) 
    let mut state = 0;

    while let Some(msg) = rx.recv().await {
        state += msg.value;
        println!("Received: {}", msg.value);
        println!("State: {}", state);
    }
}
```

Here we can see that the actor listens to incoming messages, updates the state, and then prints it out the state. We can test our actor with the code below:

```
#[tokio::main]
async fn main() {
    let (tx, rx) = channel::<Message>(100);

    let _actor_handle = tokio::spawn(async {
        basic_actor(rx).await;
    });
    for i in 0..10 {
        let msg = Message { value: i };
        tx.send(msg).await.unwrap();
    }
}
```

But what if we want to receive a response back? Right now we are just sending a message into the void and looking at the printout in the terminal. We can facilitate a response by packaging a `oneshot::Sender` in the message that we are sending to the actor. The receiving actor can then use that `oneshot::Sender` to send a response. We can define our responding actor by the following code:

```
struct RespMessage {
    value: i32,
    responder: oneshot::Sender<i32>
}

async fn resp_actor(mut rx: Receiver<RespMessage
    let mut state = 0;

    while let Some(msg) = rx.recv().await {
        state += msg.value;
        if msg.responder.send(state).is_err() {
            eprintln!("Failed to send response")
        }
    }
}
```

If we wanted to send a message to our responding actor, we would have to construct a oneshot channel, use it to construct a message, send over the message, and then await for the response. The code below depicts a basic example of how to achieve this:

```
let (tx, rx) = channel::<RespMessage>(100);
let _resp_actor_handle = tokio::spawn(async {
    resp_actor(rx).await;
});
for i in 0..10 {
    let (resp_tx, resp_rx) = oneshot::channel::<
    let msg = RespMessage {
        value: i,

        responder: resp_tx
    };
    tx.send(msg).await.unwrap();
    println!("Response: {}", resp_rx.await.unwra
}
```

Here we are using a oneshot channel because we only need the response to be sent once and then the client code can go about doing other things. This is the best choice for our use case as oneshot channels are optimised in terms of memory and synchronisation for the use case of just sending one message back and then closing.

Considering that we are sending structs over the channel to our actor, you can see that our functionality can increase in complexity. For instance, sending an enum that encapsulates multiple different messages could instruct the actor to do a range of different actions based on the type of message being sent. Actors can also create new actors, or send messages to other actors.

From the example we have shown, we could just use a mutex and acquire it for the mutation of the state. The mutex would be simple to code but how would it match up to the actor?

## Actors Vs Mutexes

When it comes to recreating what our actor in the previous section did with a mutex, we have a function that takes the following form:

```
async fn actor_replacement(state: Arc<Mutex<i32>>
    let mut state = state.lock().await;
    *state += value;
    return *state
}
```

While this is simple to write, how does this measure up in terms of performance? We can devise a simple test with the following code:

```
let state = Arc::new(Mutex::new(0));
let mut handles = Vec::new();

let now = tokio::time::Instant::now();

for i in 0..100000000 {
    let state_ref = state.clone();
```

```
        let future = async move {
            let handle = tokio::spawn(async move {
                actor_replacement(state_ref, i).awai
            });
            let _ = handle.await.unwrap();
        };
        handles.push(tokio::spawn(future));
    }
    for handle in handles {
        let _ = handle.await.unwrap();
    }
    println!("Elapsed: {:?}", now.elapsed());
```

We have spawned a lot of tasks trying to gain access to the mutex at once, and then waited on them. If we spawned one task at a time, we would not get the true effect that the concurrency of our mutex has on the outcome. Instead, we would just be getting how quick individual transactions are. We are running a large number of tasks because we want to see a statistically significant difference between the approaches. These tests take a long time to run but the results cannot be misinterpreted. At the time of writing this book on a M2 macbook with high specs, the time taken for all the mutex tasks to complete is 120 seconds.

To run the same test using our actor in the previous section, we need the code below:

```
    let (tx, rx) = channel::<RoomMessage>(100000000)
```

```
let (tx, rx) = channel::<RespMessage>(100000000);
let _resp_actor_handle = tokio::spawn(async {
    offload_resp_actor(rx).await;
});
let mut handles = Vec::new();


let now = tokio::time::Instant::now();
for i in 0..100000000 {
    let tx_ref = tx.clone();

    let future = async move {
        let (resp_tx, resp_rx) = oneshot::channel
        let msg = RespMessage {
            value: i,
            responder: resp_tx
        };
        tx_ref.send(msg).await.unwrap();
        let _ = resp_rx.await.unwrap();
    };
    handles.push(tokio::spawn(future));
}
for handle in handles {
    let _ = handle.await.unwrap();
}
println!("Elapsed: {:?}", now.elapsed());
```

Running this test gives 101 seconds at the time of writing this book. It must be noted that we ran the tests in `--release` mode to see what the compiler optimizations would do to the system. The actor is faster by 19 seconds. One reason is the overhead of acquiring the mutex. When placing a message in a channel, we just have to check to see if the channel is full or has been closed. When it comes to acquiring a mutex, the checks are more complicated. These checks typically involve checking if the lock is held by another task. If the lock is held by another task, the task trying to acquire the lock then needs to register interest and then wait to be notified.

---

**NOTE**

Generally, passing messages through channels can scale better than mutexes in concurrent environments as the senders do not have to wait for other tasks to finish what they are doing. They may have to wait to put a message on the queue of the channel, but waiting for a message to be put on a queue is quicker than waiting for an operation to finish what it is doing with the mutex, yield the lock, and then for the awaiting task to acquire the lock. As a result, channels can result in higher throughput.

---

To drive our point home, let us explore the scenario where the transaction is more complex than just increasing the value by one. Maybe there’s a few checks and a calculation before committing the final result to the state and returning the number. Us being efficient engineers may want to do other things while that process is happening. Because we are sending a message and waiting for the response, we already have that luxury with our actor code as you can see below:

```
let future = async move {
    let (resp_tx, resp_rx) = oneshot::channel::<
    let msg = RespMessage {
        value: i,
        responder: resp_tx
    };
    tx_ref.send(msg).await.unwrap();
    // do something else
    let _ = resp_rx.await.unwrap();
};
```

However, our mutex implementation would merely yield the control back to the scheduler. If we wanted to progress our mutex task while waiting for the complex transaction to complete, we would have to spawn another async task with the following code:

```
async fn actor_replacement(state: Arc<Mutex<i32>
    let update_handle = tokio::spawn(async move
        let mut state = state.lock().await;
        *state += value;
        return *state
    });
    // do something else
    update_handle.await.unwrap()
}
```

However, the overhead of spawning those extra async tasks, shoots the time elapsed in our test up to 174 seconds. That is 73 seconds more than the actor for the same functionality. This is not surprising as we are sending an async task to the runtime and getting a handle back just to allow us to progress a wait for our transaction result later on in our task.

With our test results in mind, you can see why we would want to use actors. Actors are more complex to write. You need to pass messages over a channel and package a oneshot channel for the actor to respond just to get the result. This is more complex than just acquiring a lock. However, the flexibility of choosing when to wait for the result of that message comes for free with actors. Mutexes on the other hand have a big penalty if that flexibility is desired. We can also argue that actors are easier to conceptualise. If we think about this, actors contain their state. If you want to see all interactions with that state, you look in the actor code. However, with mutex codebases, we do not know where all the interactions with the state are. The distributed interactions with the mutex also increases the risk of the mutex being highly coupled throughout the system making refactoring a headache.

Now that we have gotten our actors working, we need to be able to utilise them in our system. The easiest way to implement actors into the system with a minimal footprint is the router pattern.

## Implementing the Router Pattern

For our routing, we construct a router actor that accepts messages. These messages can be wrapped in enums to help our router locate the correct actor. For our example, we are going to implement a basic key value store. We must stress that although we are building the key value store in Rust, you should not use this educational example in production. Established solutions like RocksDB and Redis have put a lot of work and expertise into making their key value stores robust, and scalable.

For our key value store, we need to set, get, and delete keys. We can signal all of these operations with the message layout defined in Figure 8-1.

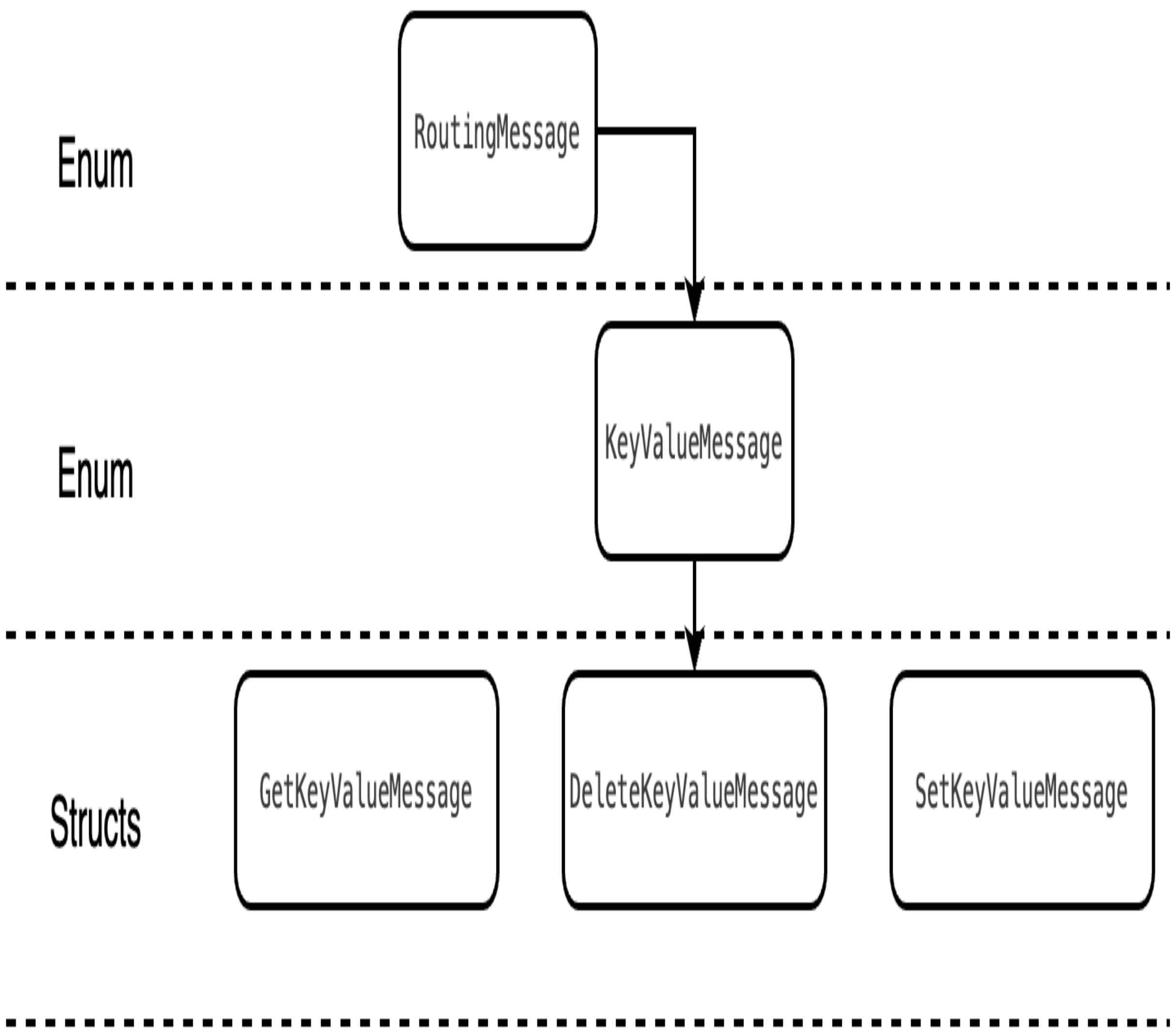


Figure 8-1. Enum structure of router actor message

Before we code anything, we need the imports defined by the code below:

```
use tokio::sync::{
    mpsc::channel,
    mpsc::{Receiver, Sender},
    oneshot,
};
use once_cell::sync::OnceCell;
```

We now need to define our message layout in figure 8-1 with the following code:

```
struct SetKeyValueMessage {
    key: String,
    value: Vec<u8>,
    response: oneshot::Sender<()>,
}
struct GetKeyValueMessage {
    key: String,
    response: oneshot::Sender<Option<Vec<u8>>>,
}
struct DeleteKeyValueMessage {
    key: String,
    response: oneshot::Sender<()>,
}
enum KeyValueMessage {
    Get(GetKeyValueMessage),
    Delete(DeleteKeyValueMessage),
    Set(SetKeyValueMessage),
}
enum RoutingMessage {
    KeyValue(KeyValueMessage),
}
```

We now have a message that can be routed to the key value actor, and this message signals the right operation with the data needed to perform the operation. For our key value actor, we merely accept the `KeyValueMessage`, match the variant, and perform the operation with the code below:

```
async fn key_value_actor(mut receiver: Receiver<K
    let mut map = std::collections::HashMap::new(
    while let Some(message) = receiver.recv().awa
        match message {
            KeyValueMessage::Get(
                GetKeyValueMessage { key, respons
            ) => {
                let _ = response.send(map.get(&ke
            }
            KeyValueMessage::Delete(
                DeleteKeyValueMessage { key, resp
            ) => {
                map.remove(&key);
                let _ = response.send(());
            }
            KeyValueMessage::Set(
                SetKeyValueMessage { key, value,
            ) => {
                map.insert(key, value);
                let _ = response.send(());
            }
            }
        }
    }
```

With our handling of the key value message, we need to connect our key value actor with a router actor using the following code:

```
    async fn router(mut receiver: Receiver<RoutingMes
        let (key_value_sender, key_value_receiver) =
        tokio::spawn(key_value_actor(key_value_receiv

        while let Some(message) = receiver.recv().awa
            match message {
                RoutingMessage::KeyValue(message) =>
                    let _ = key_value_sender.send(mes
                }
            }
        }
    }
```

As you can see, we create the key value actor in our router actor. Actors can create other actors. Putting the creation of the key value actor in the router actor ensures that there will never be a mistake in setting up the system. It also reduces the footprint of the setup of our actor system in our program.

Our router is our interface, so everything will go through the router to get to the other actors.

Now that the router is defined, we now must turn our attention to the channel for that router. All of the messages being sent into our actor system will go through that channel. It would not be very useful if we had to keep track of references to the sender of that channel. If a developer wants to send a message to our actor system and they are four levels deep, imagine the frustration that developer will feel if they have to trace the function they are using back to the main, opening up a parameter for the channel sender for each function leading to the function that they are working on. Making changes later on would also be equally frustrating. To avoid such frustrations, we define the sender as a global static with the code below:

```
static ROUTER_SENDER: OnceCell<Sender<RoutingMess
);
```

When we create the main channel for the router, we will set the sender. You might be wondering if it would be more ergonomic to construct the main channel and set the `ROUTER_SENDER` inside the router actor function. However, you could get some concurrency issues where functions are trying to send messages down the main channel before the channel is set. Remember, async runtimes can span over multiple threads so it’s possible that an async task could be trying to call the channel while the router actor is trying to set up the channel. Considering this, it is better to set up the channel at the start of the main function before spawning anything. Therefore, even if the router actor is not the first task to be polled on the async runtime, it can still access the messages sent to the channel before it was polled.

Our router actor is now ready to receive messages and route them to our key value store. We now need some functions that enable us to send key value messages. We can start with our set function which is defined by the following code:

```
pub async fn set(key: String, value: Vec<u8>)  -
    let (tx, rx) = oneshot::channel();
    ROUTER_SENDER.get().unwrap().send(
        RoutingMessage::KeyValue(KeyValueMessage
            SetKeyValueMessage {
        key,
        value,
        response: tx,
    }))).await.unwrap();
    rx.await.unwrap();
    Ok(())
}
```

There are a fair number of unwraps, but if our system is failing due to channel errors we have bigger problems. These unwraps merely avoid code bloat in the chapter. We will be covering handling errors later in the chapter when we discuss supervisors. We can see that our routing message is self explanatory. We know it is a routing message, and that the message is routed to the key value actor. We then know what method we are calling in the key value actor and the data being passed in. The routing message enums are just enough information to tell us the route intended for the function.

Now that our set function is defined, you can probably build the get function by yourself. Give that a try.

Hopefully your get function goes alone the same lines as the code below:

```
pub async fn get(key: String) -> Result<Option<Ve
    let (tx, rx) = oneshot::channel();
    ROUTER_SENDER.get().unwrap().send(
        RoutingMessage::KeyValue(KeyValueMessage
            GetKeyValueMessage {
        key,
        response: tx,
    }))).await.unwrap();
    Ok(rx.await.unwrap())
}
```

Our delete function is pretty much identical to our get function apart from the different route and the fact that the delete function does not return anything as depicted in the following code:

```
pub async fn delete(key: String) -> Result<(), st
    let (tx, rx) = oneshot::channel();
    ROUTER_SENDER.get().unwrap().send(
        RoutingMessage::KeyValue(KeyValueMessage
            DeleteKeyValueMessage {
        key,
        response: tx,
    }))).await.unwrap();
    rx.await.unwrap();
    Ok(())
}
```

And our system is ready. We can test our router and key value store with the main function below:

```
#[tokio::main]
async fn main() -> Result<(), std::io::Error> {
    let (sender, receiver) = channel(32);
    ROUTER_SENDER.set(sender).unwrap();
    tokio::spawn(router(receiver));

    let _ = set("hello".to_string(), b"world".to_
    let value = get("hello".to_string()).await?;
    println!("value: {:?}", String::from_utf8(va
    let _ = delete("hello".to_string()).await?;
    let value = get("hello".to_string()).await?;
    println!("value: {:?}", value);
    Ok(())
}
```

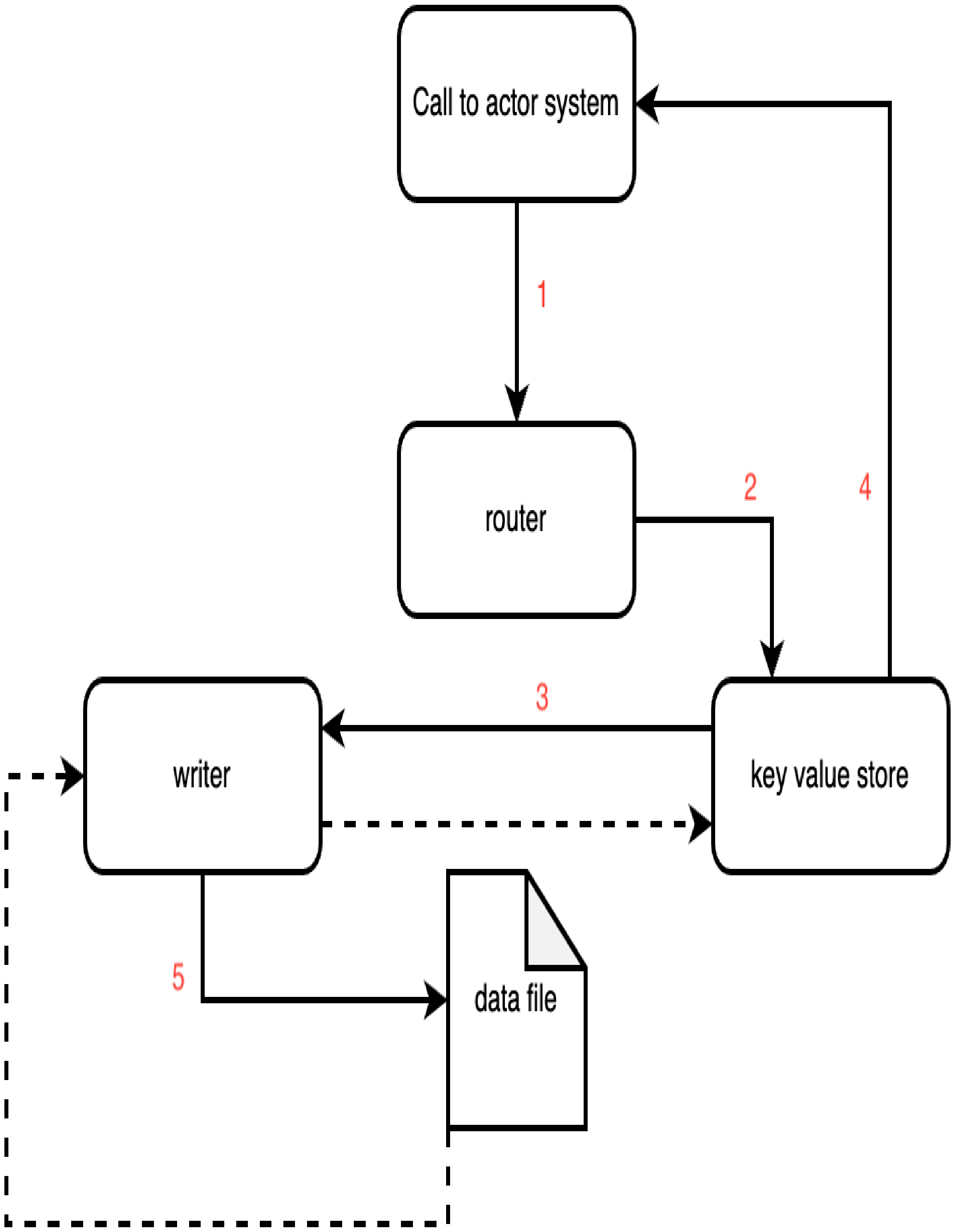
Which gives us the following printout:

```
value: Ok("world")
value: None
```

Our key value store is now working and operational. However, what happens when our system closes down or crashes? We need an actor that can keep track of the state, and recover the state when restarting the system.

## Actor State Recovery

Right now our system just has a key value store actor. However, our system might be stopped and started again, or an actor could crash. If this happens we could lose all of our data which is not good. To reduce the risk of data loss, we will create another actor that just writes our data to a file. The outline of our new system is defined in figure 8-2.



Load from file on startup

Figure 8-2. A writer backup actor system

From figure 8-2, we can see the following steps that are carried out:

1. A call is made to our actor system.
2. The router sends the message to the key value store actor.
3. Our key value store actor then clones the operation and sends that operation to the writer actor
4. The writer actor performs the operation on its own map and writes the map to the data file.
5. The key value store performs the operation on its own map and returns the result to the code that called the actor system.

When our actor system starts up, we will have the sequence below:

1. Our router actor starts, creating our key value store actor
2. Our key value store actor then creates our write actor
3. When our writer actor starts, it reads the data from the file, populates itself, and also sends the data to the key value store actor.

**NOTE**

We have given our writer actor exclusive access to the data file. This will avoid concurrency issues as the writer actor can only process one transaction at a time and there will be no other resources altering the file. The writer's exclusivity to the file can also give us performance gains as the writer actor can keep the file handle to the data file open for the entire duration of its lifetime instead of opening the file for each write. This drastically reduces the number of calls to the OS for permissions and checking if the file is available.

For this system, we are going to need to update the initialisation code for the key value actor. We also need to build the writer actor, and add a new message for the writer actor that can be constructed from the key value message.

Before we write any new code we need the following imports:

```
use serde_json;
use tokio::fs::File;
use tokio::io::{
    self,
    AsyncReadExt,
    AsyncWriteExt,
    AsyncSeekExt
};
use std::collections::HashMap;
```

For our writer message, we need the writer to also set and delete values. However, we also need our writer to return the full state that has been read from the file, giving us the following definition:

```
enum WriterLogMessage {
    Set(String, Vec<u8>),
    Delete(String),
    Get(oneshot::Sender<HashMap<String, Vec<u8>>>
}
```

We now need to construct this message from the key value message without consuming the key value message with the following code:

```
impl WriterLogMessage {
    fn from_key_value_message(message: &KeyValueM
        -> Option<WriterLogMessage> {
        match message {
            KeyValueMessage::Get(_) => None,
            KeyValueMessage::Delete(message) => S
                WriterLogMessage::Delete(message
            ),
            KeyValueMessage::Set(message) => Some
                WriterLogMessage::Set(
                    message.key.clone(),
                    message.value.clone()
                )
            )
```

```
                    ));
            }

        }
}
```

Our message definitions are now complete. We only need one more piece of functionality before we can write our writer actor and this is the loading of the state. We need both actors to load the state on startup, so our file loading is defined by the isolated function below:

```
async fn read_data_from_file(file_path: &str)
    -> io::Result<HashMap<String, Vec<u8>>> {
    let mut file = File::open(file_path).await?;
    let mut contents = String::new();
    file.read_to_string(&mut contents).await?;
    let data: HashMap<String, Vec<u8>> = serde_js
    Ok(data)
}
```

While this works, we need the loading of the state to be fault tolerant. It is nice to recover the state of the actors before they were shut down, however, it would not be a very good system if our actors failed to run at all if the actors could not load from the state file because the state file is missing or corrupted. Therefore, we wrap our loading in a function that will return an empty hashmap if there is a problem loading the state using the following code:

```
async fn load_map(file_path: &str) -> HashMap<St
    match read_data_from_file(file_path).await {
        Ok(data) => {
            println!("Data loaded from file: {:?
            return data
        },
        Err(e) => {
            println!("Failed to read from file:
            println!("Starting with an empty has
            return HashMap::new()

        }
    }
}
```

We print this out so we can check the logs of the system if we are not getting the results we expect.

We are now ready to build our writer actor. Our writer actor needs to load data from the file and then listen to incoming messages which is achieved with the code below:

```
async fn writer_actor(mut receiver: Receiver<Wri
    -> io::Result<()> {
```

```
    let mut map = load_map("./data.json").await;
    let mut file = File::create("./data.json").aw

    while let Some(message) = receiver.recv().awa
        match message {
            . . .
        }
        let contents = serde_json::to_string(&ma
        file.set_len(0).await?;
        file.seek(std::io::SeekFrom::Start(0)).aw
        file.write_all(contents.as_bytes()).awai
        file.flush().await?;
    }
    Ok(())
}
```

**NOTE**

You can see that we wipe the file and write the entire map between each message cycle. This is not an efficient way of writing to the file. However, this chapter is focused on actors and how to use them. Trade-offs around writing transactions to file is a big subject involving different file types, batch writing, and garbage collection around cleaning up data. If this stuff interests you, the O’Reilly book Database Internals provides comprehensive coverage on writing transactions to files.

Inside our matching of the message in the writer actor, we merely insert, remove, or clone and return the entire map with the code below:

```
match message {
    WriterLogMessage::Set(key, value) => {
        map.insert(key, value);
    }
    WriterLogMessage::Delete(key) => {
        map.remove(&key);
    },
    WriterLogMessage::Get(response) => {
        let _ = response.send(map.clone());
    }
}
```

While our router actor remains untouched, our key value actor needs to create the writer actor before it does anything else with the following code:

```
let (writer_key_value_sender, writer_key_value_re
tokio::spawn(writer_actor(writer_key_value_receiv
```

Our key value actor then needs to get the state of the map from our writer actor with the code below:

```
let (get_sender, get_receiver) = oneshot::channel
let _ = writer_key_value_sender.send(WriterLogMes
    get_sender
)).await;
let mut map = get_receiver.await.unwrap();
```

Finally, the key value actor can construct a writer message and send that message to the writer actor before handling the transaction itself with the following code:

```
while let Some(message) = receiver.recv().await 
    if let Some(
        write_message
    ) = WriterLogMessage::from_key_value_message(
    &message) {
        let _ = writer_key_value_sender.send(
            write_message
        ).await;
    }
    match message {
        . . .
    }
}
```

And with this, our system now supports writing and loading from a file while all the key value transactions are handled in memory. If you play around with your code in the main function, commenting bits out and inspecting the `data.json` file, you will see that it works. However, if your system is running on something like a server, you may not be manually monitoring the file to see what is going on. Now that our actor system has gotten more complex, our writer actor could have crashed and not be running, but we would be none the wiser because our key value actor could still be running. This is where supervision comes in as we need to keep track of the state of our actors.

## Actor Supervision

Right now we have two actors, the writer and key values store actors. In this section, we are going to build a supervisor actor that keeps track of every actor in our system. This is where we'll be grateful that we have implemented the router pattern. Creating a supervisor actor and then passing the sender of the supervisor actor channel through to every actor would be a headache. Instead, we can just send update messages to the supervisor actor through the router as every actor has direct access to the `ROUTER_SENDER`. The supervisor can also send reset requests to the correct actor through the browser as depicted in figure 8-3.

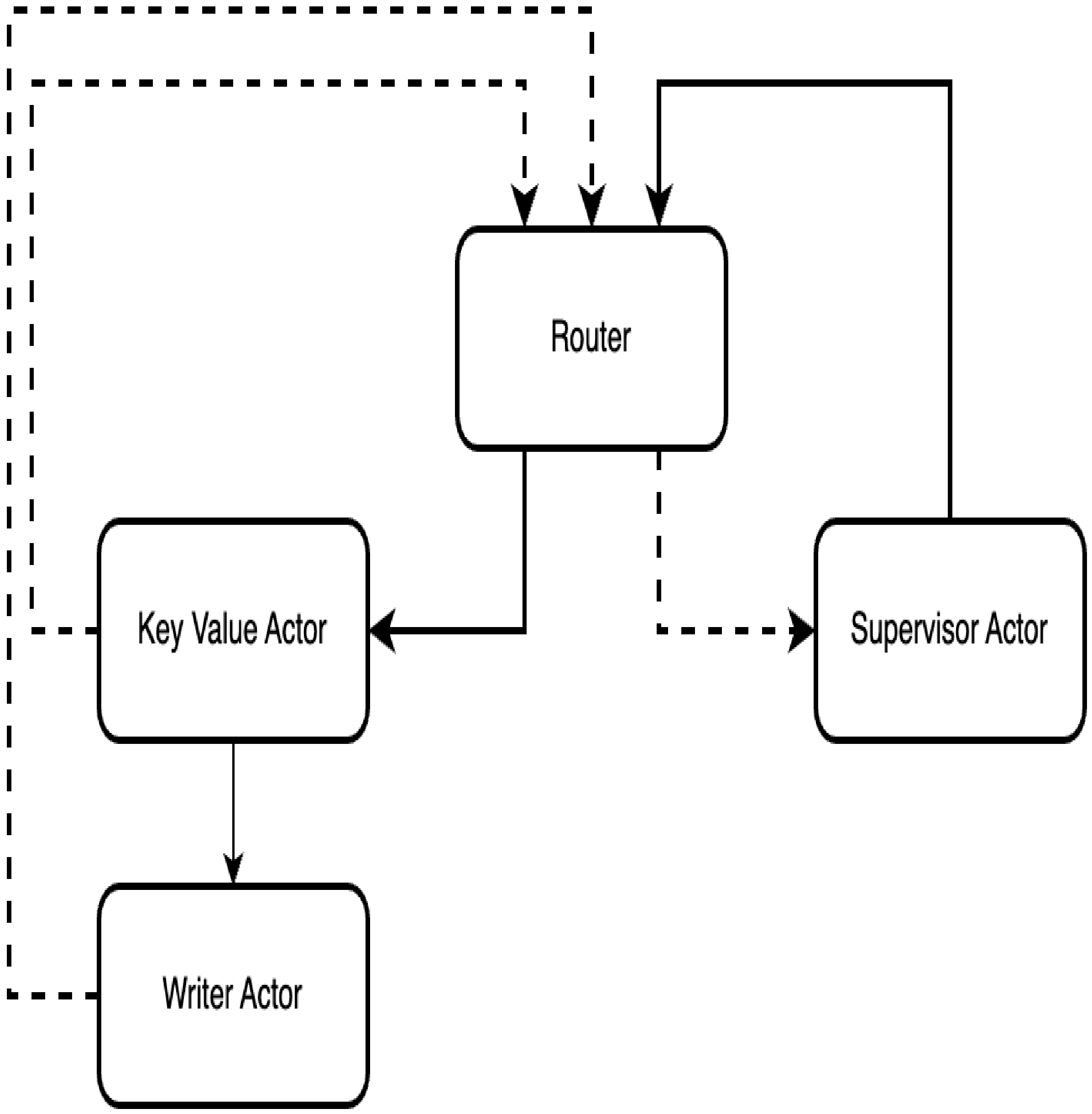


Figure 8-3. A writer backup actor system

You can see in figure 8-3 that if we do not get an update from either the key value actor or the writer actor, we can reset the key value actor. Because we can get the key value actor to hold the handle of the writer actor when the key value actor creates the writer actor, the writer actor will die if the key value actor dies. When the key value actor is created again, the writer actor will also be created.

To achieve this heartbeat supervisor mechanism, we must refactor our actors a little bit but this will show us how a little trade-off in complexity enables us to keep track and manage our long running actors. Before we code anything however, we do need the following import to handle the time checks for our actors:

```
use tokio::time::{self, Duration, Instant};
```

We also now need to support the resetting of actors and registering of heartbeats. Therefore, we must expand our `RoutingMessage` with the code below:

```
enum RoutingMessage {
    KeyValue(KeyValueMessage),
    Heartbeat(ActorType),
    Reset(ActorType),
}
#[derive(Clone, Copy, PartialEq, Eq, Hash, Debug
enum ActorType {
    KeyValue,
    Writer
}
```

Here, we can request a reset, or register a heartbeat of any actor that we want to declare in the `ActorType` enum.

Our first refactor can be our key value actor. First, we define a handle for the writer actor with the following code:

```
let (writer_key_value_sender, writer_key_value_re
let _writer_handle = tokio::spawn(
    writer_actor(writer_key_value_receiver
));
```

We still send a get message to the writer actor to populate the map, but then we lift our message handling code into an infinite loop so we can implement a timeout using the code below:

```
let timeout_duration = Duration::from_millis(200
let router_sender = ROUTER_SENDER.get().unwrap()

loop {
    match time::timeout(timeout_duration, receive
        Ok(Some(message)) => {
            if let Some(
                write_message
            ) = WriterLogMessage::from_key_value_
                let _ = writer_key_value_sender.
```

```
                        write_message
                    ).await;
                }
                match message {
                    . . .
                }
            },
            Ok(None) => break,
            Err(_) => {
                router_sender.send(
                    RoutingMessage::Heartbeat(ActorTy
                ).await.unwrap();
            }
        };
    }
```

Here, you can see that at the end of the cycle, we send a heartbeat message to the router to say that our key value store is still alive. We also have a timeout, so if 200 milliseconds passes, we still run a cycle as we do not want the lack of incoming messages to be the reason why our supervisor thinks that our actor is dead or stuck.

We need a similar approach for our writer actor. We encourage you to try and code this yourself, hopefully your attempt will be similar to the following code:

```
    let timeout_duration = Duration::from_millis(200
let router_sender = ROUTER_SENDER.get().unwrap()

loop {
    match time::timeout(timeout_duration, receive
        Ok(Some(message)) => {
            match message {

                . . .
            }
            let contents = serde_json::to_string
            file.set_len(0).await?;
            file.seek(std::io::SeekFrom::Start(0)
            file.write_all(contents.as_bytes()).a
            file.flush().await?;
        },
        Ok(None) => break,
        Err(_) => {
                router_sender.send(
                    RoutingMessage::Heartbeat(Act
                ).await.unwrap();
        }
    };
}
```

Our actors now support sending heartbeats to the router for the supervisor to keep track of. Next we need to build our supervisor actor. Our supervisor actor has a similar approach to the rest of the actors. It has an infinite loop where there is a timeout because the lack of heartbeat messages should not stop the supervisor actor from checking on the state of the actors it is tracking. In-fact, the lack of heartbeat messages would suggest that the system is in need of checking. However, instead of sending a message at the end of the infinite loop cycle, the supervisor actor loops through its own state to check if there are any actors that have not checked in. If the actor is out of date, then the supervisor actor sends a reset request to the router. The outline of this process is laid out in the code below:

```
async fn heartbeat_actor(mut receiver: Receiver<A
    let mut map = HashMap::new();
    let timeout_duration = Duration::from_millis(
    loop {
        match time::timeout(timeout_duration, rec
            Ok(Some(actor_name)) => map.insert(
                actor_name, Instant::now()
            ),
            Ok(None) => break,
            Err(_) => {
                continue;
            }
        };

        let half_second_ago = Instant::now() -
                                Duration::from_mill
        for (key, &value) in map.iter() {
            . . .
        }
```

```
            }
        }
    }
```

We have decided that we are going to have a cut off of half a second. The smaller the cut off, the quicker the actor is restarted after failure. However, this also increases work as the timeouts in the actors waiting for messages also have to be smaller to keep the supervisor satisfied.

When we are looping through our state keys to check the actors, we send a request for a reset if the cut off is exceeded with the following code:

```
    if value < half_second_ago {
        match key {
            ActorType::KeyValue | ActorType::Writer =
                ROUTER_SENDER.get().unwrap().send(
                    RoutingMessage::Reset(ActorType:
                ).await.unwrap();
                key_reset = true;
            }
        }
    }
    if key_reset {
        break
    }
    if key_reset {
        map.remove(&ActorType::KeyValue);
        map.remove(&ActorType::Writer);
    }
```

You might notice that we reset the key value actor even if the writer actor is failing. This is because the key value actor will restart the writer actor. We also remove the keys from the map because when the key value actor starts again, it will send a heartbeat message causing the keys to be checked again. However, the writer key might still be out of date causing a second unnecessary fire. We can start checking those actors once they have registered again.

Our router actor now must support all of our changes. First of all, we need to set our key value channel and handle to mutable with the code below:

```
    let (mut key_value_sender, mut key_value_receive
    let mut key_value_handle = tokio::spawn(

        key_value_actor(key_value_receiver)
    );
```

This is because we need to reallocate a new handle and channel if the key value actor is reset. We then spawn the heartbeat actor to supervise our other actors with the following code:

```
    let (heartbeat_sender, heartbeat_receiver) = chan
```

```
tokio::spawn(heartbeat_actor(heartbeat_receiver)
```

Now that our actor system is running, our router actor can handle incoming messages with the code below:

```
while let Some(message) = receiver.recv().await
    match message {
        RoutingMessage::KeyValue(message) => {
            let _ = key_value_sender.send(message
        },
        RoutingMessage::Heartbeat(message) => {
            let _ = heartbeat_sender.send(message
        },
        RoutingMessage::Reset(message) => {
            . . .
        }
    }
}
```

For our reset we must carry out a couple of steps. First we create a new channel. We then abort the key value actor, reallocate the sender and receiver to the new channel, and then spawn an new key value actor with the following code:

```
match message {
    ActorType::KeyValue | ActorType::Writer => {
        let (new_key_value_sender, new_key_value_
            32
        );
        key_value_handle.abort();
        key_value_sender = new_key_value_sender;
        key_value_receiver = new_key_value_recei
        key_value_handle = tokio::spawn(
            key_value_actor(key_value_receiver)
        );
        time::sleep(Duration::from_millis(100)).a
    },
}
```

You can see that we have a small sleep to ensure that the task has spawned and is running on the async runtime. You may worry that more requests to the key value actor might be being sent during this transition which may error. However, all requests go through the router actor. If these messages are being sent to the router for the key value actor, they will just queue up in the channel of the router. With this you can see how actor systems are very fault tolerant.

Since there are a lot of moving parts, let’s run this all together with the main function below:

WARNING

Before you run this, make sure that your `data.json` file has a set of empty curly brackets like the following:

```
{}
```

```
#[tokio::main]
async fn main() -> Result<(), std::io::Error> {
    let (sender, receiver) = channel(32);
    ROUTER_SENDER.set(sender).unwrap();
    tokio::spawn(router(receiver));
    let _ = set("hello".to_string(), b"world".to_
    let value = get("hello".to_string()).await?;
    println!("value: {:?}", value);
    let value = get("hello".to_string()).await?;
    println!("value: {:?}", value);
    ROUTER_SENDER.get().unwrap().send(
        RoutingMessage::Reset(ActorType::KeyValue
    ).await.unwrap();
    let value = get("hello".to_string()).await?;
    println!("value: {:?}", value);
    let _ = set("test".to_string(), b"world".to_v
    std::thread::sleep(std::time::Duration::from_
    Ok(())
}
```

Running our main gives us the printout below:

```
Data loaded from file: {}
value: Some([119, 111, 114, 108, 100])
value: Some([119, 111, 114, 108, 100])
Data loaded from file: {"hello": [119, 111, 114,
value: Some([119, 111, 114, 108, 100])
```

You can see that the data was loaded by the writer actor initially when setting up the system. Our `get` functions work after the setting of the `”hello”` value. We then forced a reset manually. Here you can see that the data is loaded again meaning that the writer actor is being restarted. We know that the previous writer actor died because the writer actor gets the file handle and keeps hold of it. We would get an error as the file descriptor would already be held. If you want to sleep soundly at night, you can add a timestamp before the loop of the writer actor, and print out the timestamp at the start of every iteration of the loop so the printout of the timestamp is not dependent on any incoming messages. This would give a printout like the following:

```
Data loaded from file: {}
writer instance: Instant { tv_sec: 1627237, tv_ns
value: Some([119, 111, 114, 108, 100])
writer instance: Instant { tv_sec: 1627237, tv_ns
writer instance: Instant { tv_sec: 1627237, tv_n
value: Some([119, 111, 114, 108, 100])
Starting key_value_actor
writer instance: Instant { tv_sec: 1627237, tv_n
Data loaded from file: {"hello": [119, 111, 114,
writer instance: Instant { tv_sec: 1627237, tv_n
value: Some([119, 111, 114, 108, 100])
writer instance: Instant { tv_sec: 1627237, tv_n
writer instance: Instant { tv_sec: 1627237, tv_n
writer instance: Instant { tv_sec: 1627237, tv_n

writer instance: Instant { tv_sec: 1627237, tv_n
writer instance: Instant { tv_sec: 1627237, tv_n
writer instance: Instant { tv_sec: 1627237, tv_n
```

Here you can see that the instance before and after the reset is different, and there’s no trace of the existing writer instance after the reset. We can sleep well knowing that our reset works and there isn’t a lonely actor out there without a purpose…. In our system that is, we cannot vouch for Hollywood.

## Conclusion

In this chapter, we essentially built a system that accepts key value transactions, backs them up with a writer actor, and is monitored via a heartbeat mechanism. Even though there were a lot of moving parts, the implementation was simplified by the router pattern. The router pattern is not as efficient as directly calling an actor as the message has to go through one actor before hitting its mark. However, the router pattern is a very good starting point. You can lean on the router pattern when figuring out the actors you need to solve your problem. Once the solution has taken form, you can then move towards actors directly calling each other, as opposed to going through the router actor.

While we focused on building our entire system using actors, we must remember that they are just running on an async runtime. Because actors are isolated, and easy to test due to only communicating with messages, we can take a hybrid approach with actors. This means that we can add additional functionality to our normal async system using actors. The actor channel can be accessed anywhere. Like with the migration from the router actor to actors directly calling each other, you can slowly migrate your new async code from actors to standard async code when the overall form of the new async addition takes form. You can also use actors to break out functionality in legacy code when trying to isolate dependencies to get the legacy code into a testing harness. In general, due to their isolated nature, actors are a useful tool that you can implement in a range of different settings. Actors can also act as code limbo when you are still in your discovery phase. We’ve both reached for actors when having to come up with solutions in tight deadlines such as caching and buffering chatbot messages in a microservices cluster.

In the next chapter we continue our exploration of how to approach and structure solutions with our coverage of design patterns.

# Chapter 9. Design Patterns

Throughout the book we have covered different async concepts and how to implement async code in various different ways to solve problems. However, we know that software engineering does not exist in a vacuum. When applying your new found knowledge of async programming in the wild, we all know that you will not be able to just apply isolated async code in a perfect environment. You might be applying async code to an existing codebase that is not async. You might be interacting with a third party service like a server, where you will need to handle variances in the response to the server. In this chapter we cover a range of design patterns that help you implement async code when solving a range of problems.

By the end of this chapter you will be able to implement async code in an existing codebase that previously did not support async programming. You will also be able to implement the waterfall design pattern to enable the building of pathways with reusable async components. Instead of altering the code of our async tasks to add features, you will be able to implement the decorator pattern so you can easily slot in extra functionality such as logging by just adding a compilation flag when running or building your program. Finally, you will be able to get the entire async system to adapt to errors by implementing the retry and circuit breaker patterns.

First of all, we need to be able to implement our async code in our system before implementing design patterns. So we should start with building an isolated module.

## Building an Isolated Module

Let us imagine that we have a Rust codebase that does not have any async code, but we would like to integrate some async Rust into our existing codebase. Instead of rewriting the entire codebase to incorporate async Rust, it is advised to keep the blast radius of the interactions small. Massive rewrites rarely keep to deadlines, and as the rewrite is delayed, more features get added to the existing codebase threatening the completion of the rewrite. Considering the danger of a large rewrite, it is best to start small. We can do this by writing our async code in its own module, and then offering synchronous entry points into that module. The synchronous entry points enable our async module to be implemented anywhere into the existing codebase. The synchronous entry points also enable other developers to use our async module without having to read up on async programming. This eases the integration, and other developers can get to grips with async programming in their own time. But how can we offer the benefits of async programming with synchronous entry points? figure 9-1 depicts a high level flow of how we can offer async benefits to the non-async codebase.

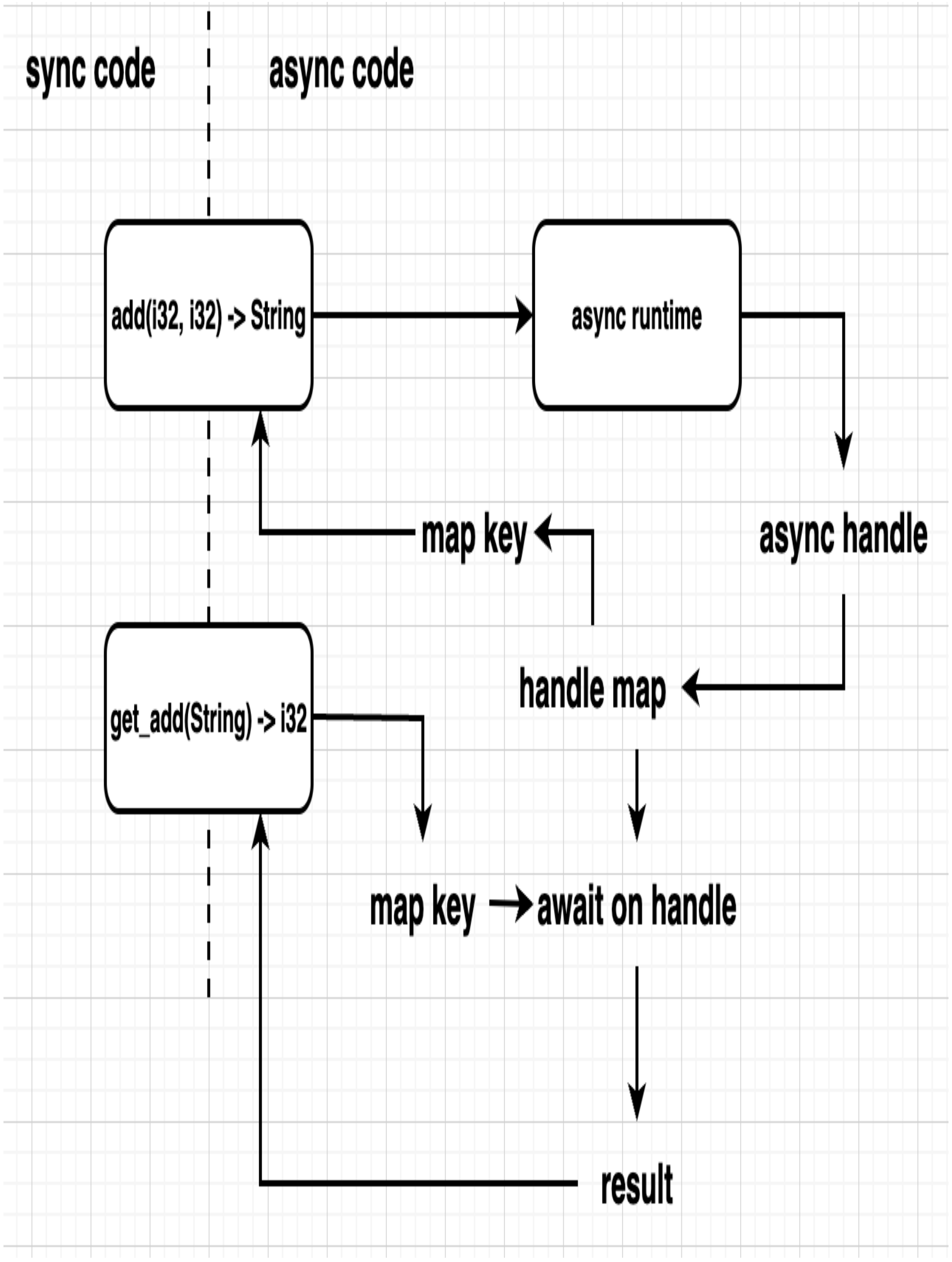


Figure 9-1. Overview of our isolated async module

What figure 9-1 lays out is that we send an async task to the runtime, put the handle in a map for those async tasks, and return a key that corresponds to the handle in the map. The developer using the module essentially calls a normal blocking function, and receives a unique ID back. The task is being progressed in the async runtime, and the developer can write some more synchronous code. When the developer needs the result, they pass the unique id through the `get_add` function which will block the synchronous code until the result is yielded. The developer is treating the unique ID like an async handle, but does not have to interact with any async code directly. Before we can implement this approach, we will need the following dependencies:

```
tokio = { version = "1.33.0", features = ["full"]
uuid = { version = "1.5.0", features = ["v4"] }
once_cell = "1.18.0"
```

With these dependencies, we can create our `async_mod.rs` file next to our `main.rs`. Our `async_mod.rs` file is going to house our async module code. Inside our `async_mod.rs` file, we are going to need the imports below:

```
use once_cell::sync::Lazy;
use tokio::runtime::{Runtime, Builder};
use tokio::task::JoinHandle;
use std::collections::HashMap;
```

```rust
use std::sync::{Arc, Mutex};

pub type AddFutMap = Lazy<Arc<Mutex<HashMap<Strin
```

When it comes to our runtime, we are going to use the following:

```rust
static TOKIO_RUNTIME: Lazy<Runtime> = Lazy::new(
    Builder::new_multi_thread()
        .enable_all()
        .build()
        .expect("Failed to create Tokio runtime")
});
```

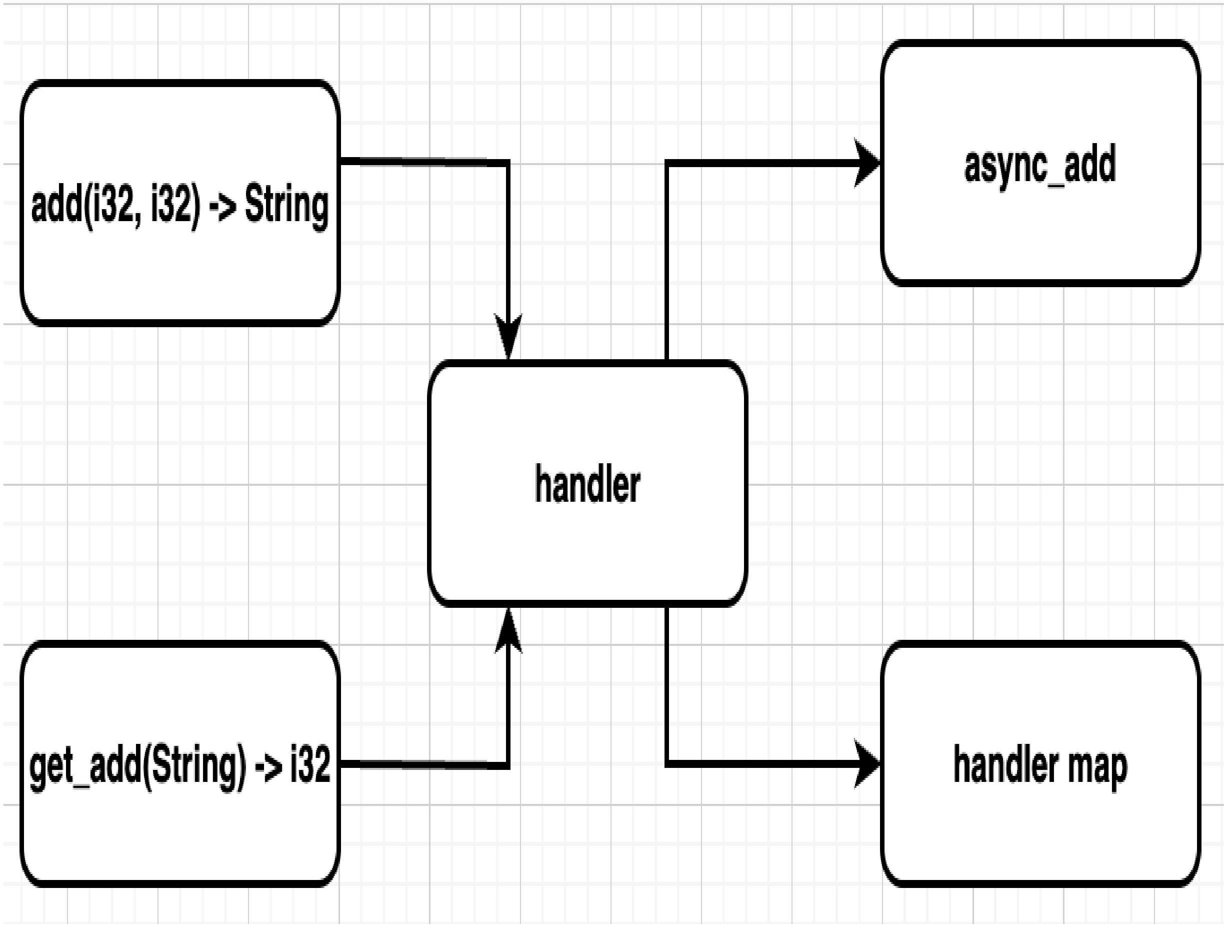
We then define our trivial async add function with a sleep to represent some async task with the code below:

```rust
async fn async_add(a: i32, b: i32) -> i32 {
    println!("starting async_add");
    tokio::time::sleep(tokio::time::Duration::fro

    println!("finished async_add");
    a + b
}
```

This is going to be our core async task that we are going to expose to the async runtime but not outside the module which is why the runtime and the `async_add` function are not public.

Now that we have defined our async runtime, and async add task, we can build our handler. As seen in figure 9-2, our handler is essentially a router for our entry points to interact with the runtime and map.



Our handler just needs to be a function that either accepts the numbers to be added, or the unique ID to get the result with the following outline:

```
fn add_handler(a: Option<i32>, b: Option<i32>, i
    -> Result<(Option<i32>, Option<String>), Str
    static MAP: AddFutMap = Lazy::new(|| Arc::new
                                                  Mutex::new
                                             ));
    match (a, b, id) {
        (Some(a), Some(b), None) => {
            . . .
        },
        (None, None, Some(id)) => {
            . . .
        },
        _ => Err(
            "either a or b need to be provided o
            handle_id".to_string()
        )

    }
}
```

We can see that our future map is lazily evaluated, like in chapter three when we defined the queue in the spawn_task function as a lazy evaluation. If we call the handler function and update the `MAP`, the next time we will call the handler, we will have the updated `MAP` within the handler function. Even though we are only going to be calling the handler function from the main thread in synchronous code, we cannot guarantee that another developer will spin up a thread and call this function. If you are 100%

certain that the handler will only be called in the main thread, then you can get rid of the `Arc` and `Mutex`, make the `MAP` mutable, and access the `MAP` in the rest of the function with unsafe code. However, as you have probably guessed, it is unsafe. You could also use `thread_local` to get rid of the `Arc` and `Mutex`. This can be safe as long as the developer gets the result in the same thread that the task was spawned. A developer does not need access to the entire map of the program. The developer only needs access to the map that holds their async handle for their task.

In our first match branch of our handler function, we are providing the numbers to be added, so we spawn a task, tether this task to a unique id in our MAP, and return the unique ID with the code below:

```
let handle = TOKIO_RUNTIME.spawn(async_add(a, b)
let id = uuid::Uuid::new_v4().to_string();
MAP.lock().unwrap().insert(id.clone(), handle);
Ok((None, Some(id)))
```

We can now define our branch that handles the unique id for getting the result of the task. Here, we get the task handle from the `MAP`, pass the handle into the async runtime to block the current thread until result has been yielded, and return the result with the following code:

```
let handle = match MAP.lock().unwrap().remove(&i
    Some(handle) => handle,
    None => return Err("No handle found".to_strin
};
let result: i32 = match TOKIO_RUNTIME.block_on(as
    handle.await
}){
    Ok(result) => result,
    Err(e) => return Err(e.to_string())
};
Ok((Some(result), None))
```

Our handler now works. However, we must note that our handler is not public. This is because the interface is not ergonomic. A developer using our module could pass in the wrong combination of inputs. We can start with our first public interface with the code below:

```
pub fn send_add(a: i32, b: i32) -> Result<String,
    match add_handler(Some(a), Some(b), None) {
        Ok((None, Some(id))) => Ok(id),
        Ok(_) => Err(
            "Something went wrong, please contact
        ),
        Err(e) => Err(e)
    }
}
```

Here we can see that we give the developer no option but to provide two integers which are passed into our handler. We then return the id. However, if we return any other variant that is not an error, something is seriously wrong with our implementation. To save the developer using our module time trying to debug what they did wrong, we just tell them to contact us as it is our issue to solve.

The get result interface is similar to our send interface just inverted, taking the following form:

```
pub fn get_add(id: String) -> Result<i32, String
    match add_handler(None, None, Some(id)) {
        Ok((Some(result), None)) => Ok(result),
        Ok(_) => Err(
            "Something went wrong, please contact
        ),
        Err(e) => Err(e)
    }
}
```

Now our async module is complete, we can use it in our main.rs with the code below:

```
mod async_mod;

fn main() {
    println!("Hello, world!");
    let id = async_mod::send_add(1, 2).unwrap();
    println!("id: {}", id);
    std::thread::sleep(std::time::Duration::from_
    println!("main sleep done");
    let result = async_mod::get_add(id).unwrap()
    println!("result: {}", result);
}
```

Running the code above will give us an output similar to the following printout:

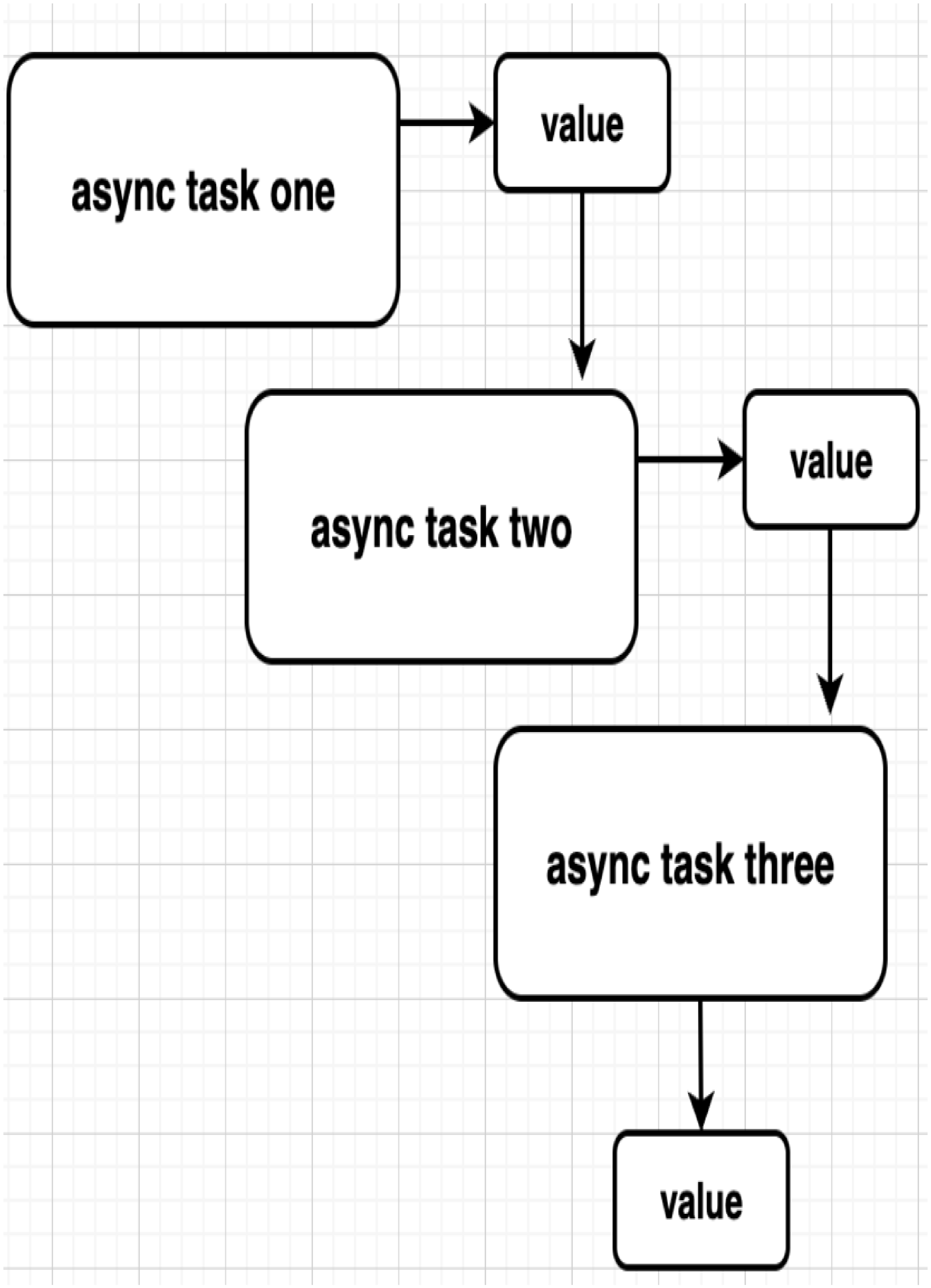
```
Hello, world!
starting async_add
id: e2a2f3e1-2a77-432c-b0b8-923483ae637f
finished async_add
main sleep done
result: 3
```

Your id will be different but the order should be the same. Here, we can see that our async task is being processed as our main thread continues, and we can get the result. We can see how isolated our async code is. We now have the freedom to experiment. For instance, you will be able to experiment with different runtimes and runtime configurations. Recall in chapter seven, customizing Tokio, we could switch over to localsets and start using local thread states to cache recently calculated values if the computational needs increase for our calculations. However, our interface is completely decoupled from async primitives so other developers using our module will not notice the difference, and thus their implementations of our interface will not break.

Now that we have covered how to implement an async module with a minimal footprint on the rest of the codebase, we can now implement other design patterns that we can implement on our codebases. We can start with the waterfall design pattern.

## Waterfall Design pattern

The waterfall design pattern is essentially a chain of async tasks that feed values directly into each other as laid out in figure 9-3.


async task one
value
async task two
value
async task three
value

Implementing a basic waterfall design pattern is straightforward. With rust, we can exploit the error handling system for safe and concise code. We can demonstrate this with the following three async tasks:

```
type WaterFallResult = Result<String, Box<dyn st

async fn task1() -> WaterFallResult {
    Ok("Task 1 completed".to_string())
}
async fn task2(input: String) -> WaterFallResult
    Ok(format!("{} then Task 2 completed", input
}
async fn task3(input: String) -> WaterFallResult
    Ok(format!("{} and finally Task 3 completed",
}
```

Because they all return the same error type, they all lock into each other with the `?` operator as seen below:

```
#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error
    let output1 = task1().await?;
    let output2 = task2(output1).await?;
    let result = task3(output2).await?;
    println!("{}", result);
    Ok(())
```

```
}
```

The waterfall approach is simple, and predictable. It also enables us to reuse our async tasks for building blocks. For instance, our three async tasks could be accepting i32 data types. We could add logic around these async tasks as seen with the following code:

```
#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error
    let output1 = task1().await?;
    let output2: i32;
    if output1 > 10 {
        output2 = task2(output1).await?;
    } else {
        output2 = task3(output1).await?;
    }
    println!("{}", output2);
    Ok(())
}
```

Considering how we can use logic to direct the flow of the waterfall, we can see where the waterfall implementation might be useful for building pathways that differ slightly but use the same core components. We can also easily slot metrics into these workflows between the components as and when we need. While inserting metrics/logging between the components can be useful, we can also use the decorator pattern to add functionality to our tasks.

## The Decorator Pattern

The decorator pattern is essentially a wrapper around some functionality that either adds to the functionality, or executes some logic before or after the main execution of logic. Classic examples for decorators are fixtures. This is where a unit test sets up the state of some data storage before the test, and then destroys the state after the test. The setup and destroying of state between tests ensures that tests are atomic, and a failed test will not alter the outcome of other tests. This state management can be wrapped around code that we are testing. Logging is also a classic use as we can easily switch off the logging without having to change our core logic. Decorators are also used for session management.

Before we look at implementing the decorator pattern in an async context, we should understand how to implement a basic decorator for a struct. Our decorator will merely add to a string. Our functionality that we are going to decorate will merely yield a string with the following code:

```
trait Greeting {
    fn greet(&self) -> String;
}
```

We then define a struct that implements our trait with the code below:

```rust
struct HelloWorld;
impl Greeting for HelloWorld {
    fn greet(&self) -> String {
        "Hello, World!".to_string()
    }
}
```

We can define a decorator struct that implements our trait, and it contains an inner component which similarly embodies our trait, as demonstrated by the following code:

```rust
struct ExcitedGreeting<T: Greeting> {
    inner: T,
}


impl<T: Greeting> Greeting for ExcitedGreeting<T
    fn greet(&self) -> String {
        let mut greeting = self.inner.greet();
        greeting.push_str(" I'm so excited to be
        greeting
    }
}
```

Here, we can see that we are calling the trait from the inner struct, and adding to the string, returning the altered string. We can test our decorator pattern easily with the code below:

```
fn main() {
    let raw_one = HelloWorld;
    let raw_two = HelloWorld;
    let decorated = ExcitedGreeting { inner: raw_
    println!("{}", raw_one.greet());
    println!("{}", decorated.greet());
}
```

We can see that we can easily wrap functionality around our struct. Because we are also implementing the same trait for the wrapper, we can also pass our wrapped struct into functions that expect structs that have implemented our trait. Therefore we can see that there is no need to change any code in our codebase if we are expecting traits as opposed to structs.

We can even make our implementation of the decorator pattern dependant on the compilation features. For example, we can add a feature in our Cargo.toml with the following code:

```
[features]
logging_decorator = []
```

We can then rewrite our main function to compile with the decorated logic or not depending on the feature flags with the code below:

```
fn main() {
    #[cfg(feature = "logging_decorator")]
    let hello = ExcitedGreeting { inner: HelloWo

    #[cfg(not(feature = "logging_decorator"))]
    let hello = HelloWorld;

    println!("{}", hello.greet());
}
```

To run our decorator we would need to call the following terminal command:

```
cargo run --features "logging_decorator"
```

We can set this feature to default if needed and we can also add extra dependencies to the feature if the feature relies on any dependencies.

Now that we understand the basics of a decorator, we can implement the same functionality in a future. Instead of a struct, we just have an inner future. Before we build our future, we need the imports below:

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

For this decorator, we are going to implement a logging trait, and our example is going to call a log function before we poll the inner future. Our logging trait takes the following form:

```
trait Logging {
    fn log(&self);
}
```

We then define our logging struct that contains an inner future with the code below:

```
struct LoggingFuture<F: Future + Logging> {
    inner: F,
}

impl<F: Future + Logging> Future for LoggingFutu
    type Output = F::Output;

    fn poll(self: Pin<&mut Self>, cx: &mut Conte
        -> Poll<Self::Output> {
        let inner = unsafe { self.map_unchecked_
        inner.log();
```

```
            inner.poll(cx)
        }
    }
```

While we are using unsafe code in our poll, our code is safe. We have to use the unsafe block because the rust compiler cannot check projections of the pin. We are not moving the value out of the pin.

We now need to implement the Logging trait for any type that also implements the Future with the following code:

```
impl<F: Future> Logging for F {
    fn log(&self) {
        println!("Polling the future!");
    }
}
```

This means that whatever future is held by our decorator, we can call the log function. We could get creative with combining other traits so that our futures being passed into the decorator can yield specific values about the future, but for this example, we are merely demonstrating how to implement an async decorator. We can now define a simple future, wrap it, and call it with the code below:

```
async fn my_async_function() -> String {
    "Result of async computation".to_string()
```

```
}

#[tokio::main]
async fn main() {
    let logged_future = LoggingFuture { inner: my
    let result = logged_future.await;
    println!("{}", result);
}
```

If we run our code, we will get the following printout:

```
Polling the future!
Result of async computation
```

Here, we can see that our logging decorator works. We can use the same compilation feature approaches for our decorator.

As decorators are designed to be slotted in with minimal friction and have the same type signature, decorators should not affect the logic of the program too much. If we want to alter the flow of our program based on certain conditions, we can consider using a state machine pattern.

## The State Machine Pattern

State machines hold a particular state, and logic around how that state is changed. Other processes can then reference that state to inform how they act. A simple real world example of a state machine is a set of traffic lights. Depending on the country traffic lights can change, but they all have at least two states, red, and green. Depending on the system there can be a range of inputs and hard coded logic that changes the state of each traffic light through time. What is important to note is that drivers directly observe the state of the traffic lights and act accordingly. We can have as many or as little drivers as we want, but the contract stays the same. The lights focus on maintaining the state and changing it depending on inputs, and the drivers merely observe and react to that state. With this analogy, it is not surprising that state machines can be used for scheduling tasks and managing job queues, networking, workflows and pipelines, and controlling machinery/systems that have distinct states that respond to a combination of async inputs and timed events.

For our example, we can build a basic switch state that is either on or off. Enums are great for managing states as we have the match pattern, and enum variants can also house data. Our simple state takes the following form:

```
enum State {
    On,
    Off,
}
```

We now define the event state that our state machine consumes to change the state with the code below:

```
enum Event {
    SwitchOn,
    SwitchOff,
}
```

We now have events and the state. The interface between the events and state can be defined with the following code:

```
impl State {
    async fn transition(self, event: Event) -> Se
        match (&self, event) {
            (State::On, Event::SwitchOff) => {
                println!("Transitioning to the Of
                State::Off
            },
            (State::Off, Event::SwitchOn) => {
                println!("Transitioning to the On
                State::On
            },
            _ => {
                println!(
                    "No transition possible,
                    staying in the current state'
                );
                self
            },
        }
    }
}
```

Here, we can see that if the state of the switch is on, the event of switching off our switch would turn the state to off and vice versa. We can test our state machine with the following code:

```
#[tokio::main]
async fn main() {
    let mut state = State::On;

    state = state.transition(Event::SwitchOff).aw
    state = state.transition(Event::SwitchOn).awa
    state = state.transition(Event::SwitchOn).awa

    match state {
        State::On => println!("State machine is i
        _ => println!("State machine is not in th
    }
}
```

Running this code will give us the printout below:

```
Transitioning to the Off state
Transitioning to the On state
No transition possible, staying in the current st
State machine is in the On state
```

In our example, async code was not essential, but this is because our example was simple. We could utilize async code such as accessing a state through a Mutex, or listening to events through an async channel. Like our traffic light example, our state machine decouples the logic behind the state with the async tasks being processed in the runtime. For instance, our state machine would be a struct with a count and enum of on or off. Other tasks when starting could send an event to our state machine via a channel to increase the count. When the count is over a certain threshold, the state machine could switch the state to off. If new tasks are required to check the state machine and have the result to be on before starting, we have implemented a simple signal system that throttles the progression of new async tasks if the task count is too high. However, we could replace this switch with a counter with an AtomicBool and AtomicUsize if we wanted. But our state machine example sets us up to implement more complex logic if needed.

Our state machine can also poll different futures depending on the state of our state machine. The code below is an example of how we can poll different futures based off the state of our switch:

```
struct StateFuture<F: Future, X: Future> {
    pub state: State,
    pub on_future: F,
    pub off_future: X,
}
```

Now that state machine has the state and the two futures to poll, we can implement the polling logic with the following code:

```
impl<F: Future, X: Future> Future for StateFuture
    type Output = State;

    fn poll(mut self: Pin<&mut Self>, cx: &mut Co
        -> Poll<Self::Output> {
        match self.state {
            State::On => {
                let inner = unsafe {
                    self.map_unchecked_mut(|s| &r
                };
                let _ = inner.poll(cx);
                cx.waker().wake_by_ref();
                Poll::Pending
            },
            State::Off => {
                let inner = unsafe {
                    self.map_unchecked_mut(|s| &
                };
                let _ = inner.poll(cx);
                cx.waker().wake_by_ref();
                Poll::Pending
            },
        }


    }
}
```

In this example, the future will constantly poll in the background. This enables our state machine to switch its continuous operations based on the state. Adding extra functionality such as listening to events through a channel to potentially change the state before polling the futures can easily be done.

Going back to our example of our state machine throttling the progression of new tasks if the count is too high, how should the async tasks checking the state machine handle the off state? This is where the retry pattern comes in.

## The Retry Pattern

We might be in a situation where our async tasks are blocked when trying to access something. This could be our state machine saying that there are too many tasks, or a server could be overloaded. Considering this, we do not want our async task to give up, so retry might yield the result we want. However, we also do not want to hammer our target relentlessly. If a server, `Mutex`, or database is overloaded, the last thing we want to do is flood the overloaded target with back to back requests. The retry pattern allows the async task to retry the request. However, within each retry, there is a delay, and this delay doubles every attempt. This backing off will allow our target to get a drop in the frequency of requests to catch up on tasks that the target is processing.

To explore the retry pattern, we initially define a get_data function that will always return an error to test out retry pattern with the following code:

```
async fn get_data() -> Result<String, Box<dyn std
    Err("Error".into())
}
```

We then define an async task that implements the retry function with the code below:

```
async fn do_something() -> Result<(), Box<dyn std
    let mut miliseconds = 1000;
    let total_count = 5;
```

```
        let mut count = 0;

        let result: String;
        loop {
            match get_data().await {
                Ok(data) => {
                    result = data;
                    break;
                },
                Err(err) => {
                    println!("Error: {}", err);
                    count += 1;
                    if count == total_count {
                        return Err(err);
                    }
                }
            }
            tokio::time::sleep(
                tokio::time::Duration::from_millis(mi
            ).await;
            miliseconds *= 2;
        }
        Ok(())
    }
```

If we run our retry pattern with the following code:

```
#[tokio::main]
async fn main() {
    let outcome = do_something().await;
    println!("Outcome: {:?}", outcome);
}
```

We get the following printout:

```
Error: Error
Error: Error
Error: Error
Error: Error
Error: Error
Outcome: Err("Error")
```

We can see that our retry works. Retry patterns are more of a utility than a design choice for an entire application. Sprinkling the retry pattern throughout the application when an async task needs to access a target will give your system more flexibility if the system handles spikes in traffic due to reducing pressure on the targets. However, what if we keep getting errors? Surely, if a threshold is passed, it doesn't make sense to keep spawning tasks. For instance, if a server has completely crashed, there has to be a state where we no longer waste CPU resources sending further requests. This is where the circuit breaker pattern helps us.

## The Circuit Breaker Pattern

A circuit breaker pattern essentially stops tasks from being spawned if the number of errors exceeds the threshold. Instead of defining our own state machine that is either on or off, we can replicate the same effect with two simple atomic values that are defined in the code below:

```
use std::sync::atomic::{AtomicBool, AtomicUsize,
use std::future::Future;
use tokio::task::JoinHandle;

static OPEN: AtomicBool = AtomicBool::new(false)
static COUNT : AtomicUsize = AtomicUsize::new(0)
```

The premise is fairly simple. If `OPEN` is true, we state that the circuit is open and we can no longer spawn new tasks. If there is an error, we increase the `COUNT` by one and set `OPEN` to true if the `COUNT` exceeds the threshold. We also need to write our own spawn task function that checks the `OPEN` before spawning a task. Our spawn task function take the following form:

```
fn spawn_task<F, T>(future: F) -> Result<JoinHan
where
    F: Future<Output = T> + Send + 'static,
    T: Send + 'static,
{
    let open = OPEN.load(Ordering::SeqCst);
    if open == false {
        return Ok(tokio::task::spawn(future))
    }
    Err("Circuit Open".to_string())
}
```

We can now define two simple async tasks. One task to throw an error, and another to merely just pass with the code below:

```
async fn error_task() {
    println!("error task running");
    let count = COUNT.fetch_add(1, Ordering::Seq
    if count == 2 {
        println!("opening circuit");
        OPEN.store(true, Ordering::SeqCst);
    }
}
async fn passing_task() {
    println!("passing task running");
}
```

With these tasks we can be deterministic on when our system is going to break. We can test that our system breaks when it reaches three errors with the following code:

```rust
#[tokio::main]
async fn main() -> Result<(), String> {
    let _ = spawn_task(passing_task())?.await;
    let _ = spawn_task(error_task())?.await;
    let _ = spawn_task(error_task())?.await;
    let _ = spawn_task(error_task())?.await;
    let _ = spawn_task(passing_task())?.await;
    Ok(())
}
```

This will give us the following printout:

```
passing task running
error task running
error task running
error task running
opening circuit
Error: "Circuit Open"
```

Here we can see that we can no longer spawn tasks once the threshold is reached. We can get creative with what we do when the threshold is reached. Maybe we keep track of all tasks and only block certain types of tasks if their own individual thresholds were broken. We could stop the program all together with a graceful shutdown and trigger some alert system so developers and IT are informed of the shutdown. We could also take time snapshots and close the circuit after a certain amount of time has passed. These variances all depend on the problem you are solving and what solution is needed. And with this circuit breaker pattern, we have covered enough design patterns to aid your implementation of async code into a codebase.

## Conclusion

In this chapter we covered a range of design patterns to enable you to implement the async code that you learned throughout the book. It’s key to think about the codebase as a whole. If you are integrating into an existing codebase with no async code then the isolated module is the obvious first step. All of the design patterns in this chapter were chosen with simplistic code examples. Small simplistic steps are the best approach for implementing the async code. It makes testing easier, and enables you to rollback if the recent implementation is no longer needed, or is breaking something else in the code. While it can be tempting to preemptively apply design patterns, over engineering seems to be the number one criticism of design patterns in general. Write your code as you would, and consider implementing a design pattern when it presents itself. Setting out to force a design pattern increases the risk of your implementation resulting in over-engineering. Knowing your design patterns is crucial to knowing when and where to implement design patterns.

In the next chapter, we will be covering async approaches to networking by building our own async TCP server just using the standard library and no external dependencies.

# About the Authors

**Maxwell Flitton** is a software engineer who works for the open source financial loss modeling foundation OasisLMF. In 2011, Maxwell achieved his Bachelor of Science degree in nursing from the University of Lincoln, UK and a degree in physics from the Open University with a postgraduate diploma in physics and engineering in medicine from UCL in London while working as a nurse at Charing Cross A&E. He’s worked on numerous projects such as medical simulation software for the German government and supervising computational medicine students at Imperial College London. He also has experience in financial tech and Monolith AI. While building the medical simulation software, Maxwell and Caroline had to build Rust async systems in the Kubernetes cluster to solve real-time event solutions and caching mechanisms. Maxwell has written the Packt textbooks *Rust Web Programming* and *Speed up Your Python with Rust*.

**Caroline Morton** studied Medicine and International Health at the University of Birmingham before moving to London to work as a doctor. She completed an Epidemiology MSc at the London School of Hygiene and Tropical Medicine. She later set up the first course in the UK training doctors and medical students to learn programming (Coding for Medicine) which later developed into a 10 week module and wrote a textbook covering the same topic called *Computational Medicine* (Elsevier, 2018). In 2019, she moved to University of Oxford to work as an Epidemiologist and

Software Developer and was key in developing OpenSAFELY, a trusted research environment that processed COVID-19 data during the pandemic. This resulted in over 40 peer reviewed papers in journals such as *Nature*, the *Lancet*, and the *BMJ*. Together with Maxwell, she has developed cutting-edge techniques in Rust to solve problems in developing a Virtual Emergency Room app for training new doctors. She therefore has real-world experience in writing and deploying async Rust in production.