O'REILLY®

Async Rust

Unleashing the Power of Fearless Concurrency

Early Release

RAW & UNEDITED

Maxwell Flitton
& Caroline Morton

O'REILLY®

Async Rust

Unleashing the Power of Fearless Concurrency

Early Release
RAW & UNEDITED

Maxwell Flitton
& Caroline Morton

Async Rust

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.

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.

O’Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

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

- July 2024: First Edition

Revision History for the Early Release

- 2023-12-15: First Release

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

property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-098-14903-1

[LSI]

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:

```
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.

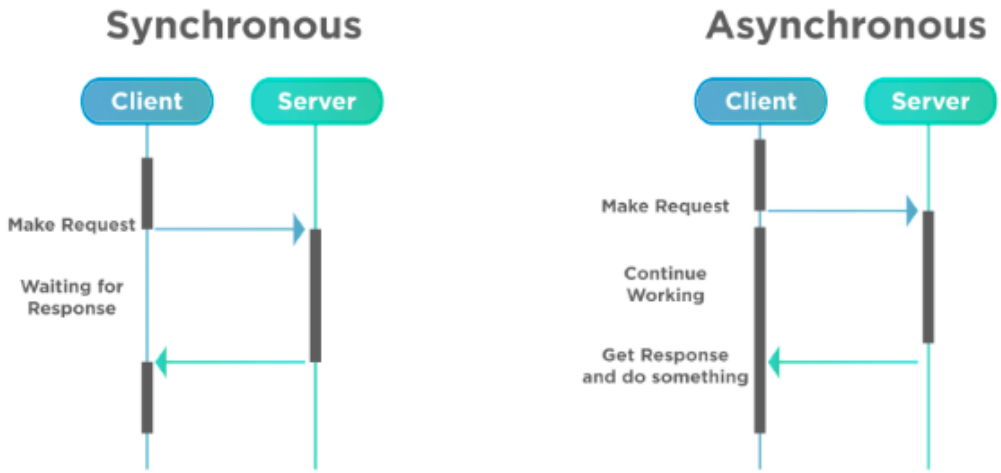

Figure 1-1. Blocking synchronous timeline compared to asynchronous timeline

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.

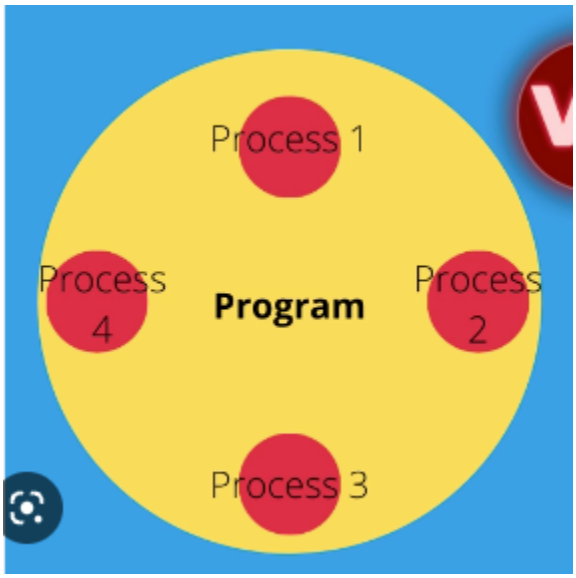

Figure 1-2. Diagram of how processes relate to the program

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: {:?}", elap
}
```

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 pr
    process2.wait().expect("Failed to wait for pr

    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::<u
        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:

```
                  38271 l                  0u  CHR   16,0      0t14667
```

```
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/k
```

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.

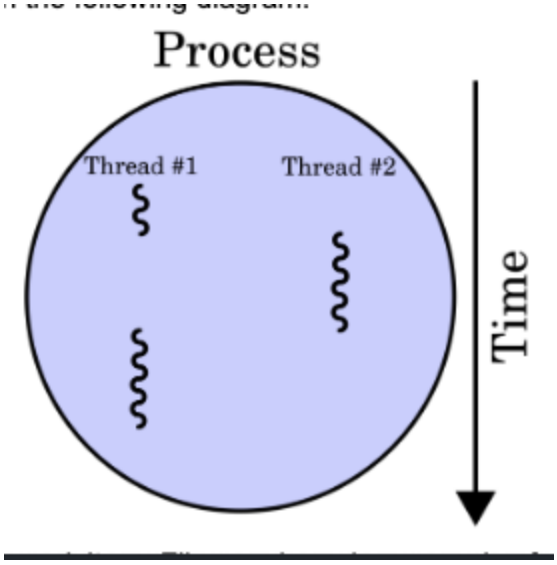

Figure 1-3. Diagram of how threads relate to a process

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.

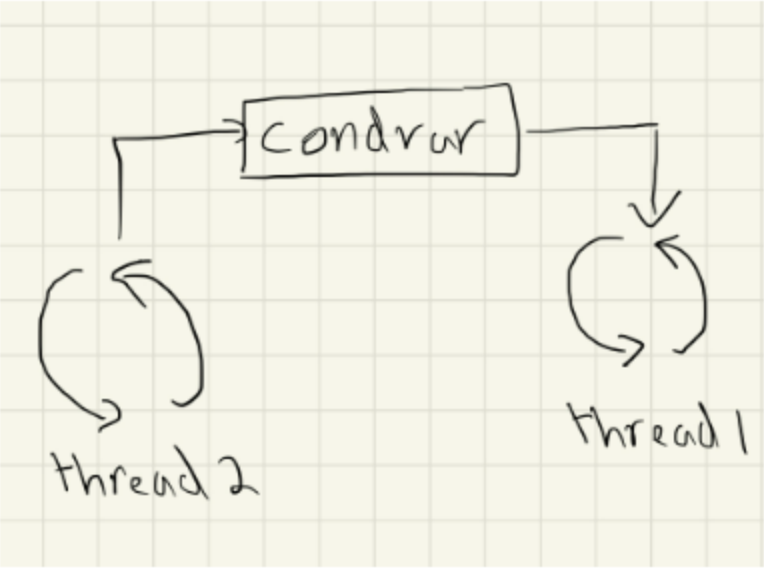

Figure 1-4. Condvar alerting another thread of a change

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");
```

```
        println!("Shop 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.

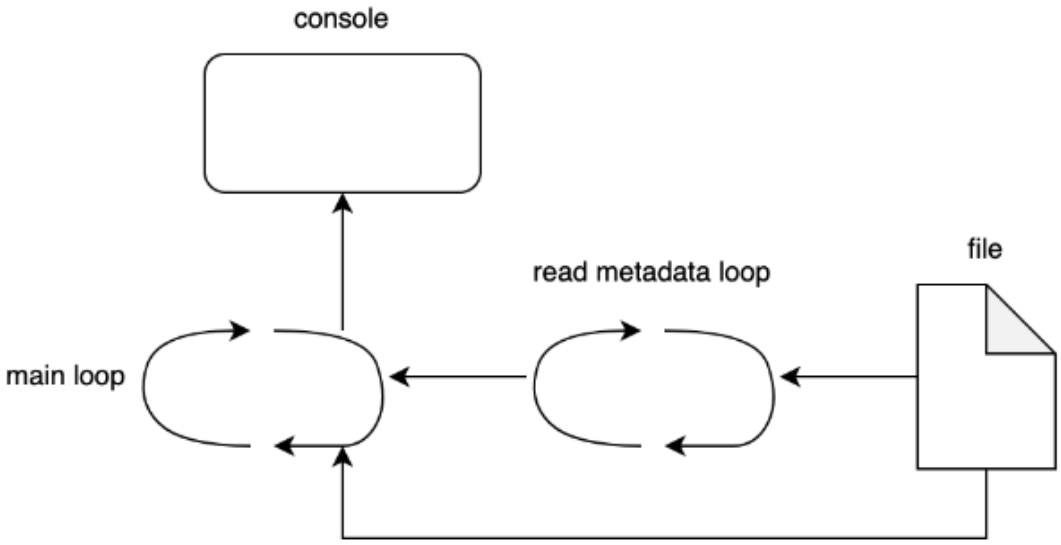

Figure 1-5. Overview of a system keeping track of changes in a file

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:

```
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().un

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

```
            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:

1. We get the path to the file that we are checking.
2. set the last modified time to none as we have not checked the file yet.
3. We then have an infinite loop.
4. In that loop we extract the time of last modified.
5. 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.
6. 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
            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.o
    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/
```

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

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 2nd 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.

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**
    a. Boil kettle
    b. Pour milk
    c. Put in instant coffee
    d. Pour boiled water
2. **Make toast**
    a. Put bread in toaster
    b. 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**
    a. Pour milk
    b. Put in instant coffee
2. **Make coffee**
    a. Boil kettle
    b. Pour boiled water
3. **Make toast**
    a. Put bread in toaster
    b. 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:

```
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:

```
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_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
```

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

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::task
        counter_one.await
    });
    let handle_two: JoinHandle<u32> = tokio::task
```

```
        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.

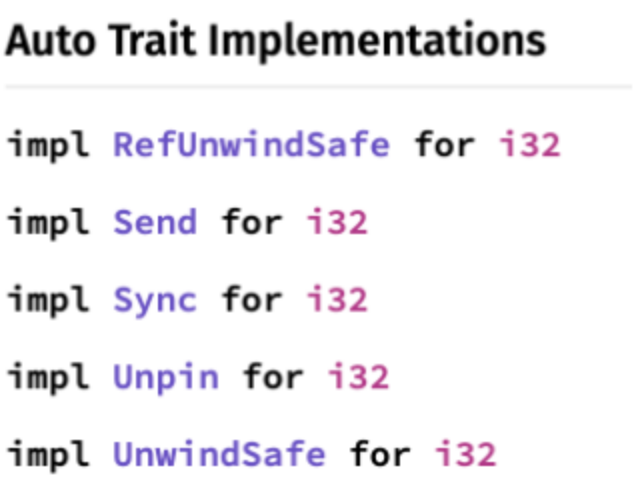

Figure 2-1. Example of Auto Trait Implementations in documention showning thread safety of a struct or primative

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.

```
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"`.

```
#[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;

            match self.counter_type {
```

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

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(SharedD
    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::task
        counter_one.await
    });
    let handle_two: JoinHandle<u32> = tokio::task
        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: CounterTyp
    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::sp
        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) -> Asynch
    match OpenOptions::new().append(true).open(fi
        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
```

```
                    return Poll::Pending
                }
            };
            let lined_entry = format!("{}\n", self.er
            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_strin
```

Or we can directly await on it like below:

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

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.