[Using Threads to Run Code Simultaneously 2](#_Toc501111894)

[Creating a New Thread with spawn 4](#_Toc501111895)

[Waiting for All Threads to Finish Using join Handles 5](#_Toc501111896)

[Using move Closures with Threads 7](#_Toc501111897)

[Message Passing to Transfer Data Between Threads 11](#_Toc501111898)

[Channels and Ownership Transference 14](#_Toc501111899)

[Sending Multiple Values and Seeing the Receiver Waiting 15](#_Toc501111900)

[Creating Multiple Producers by Cloning the Transmitter 17](#_Toc501111901)

[Shared State Concurrency 19](#_Toc501111902)

[Mutexes Allow Access to Data from One Thread at a Time 19](#_Toc501111903)

[The API of Mutex<T> 20](#_Toc501111904)

[Sharing a Mutex<T> Between Multiple Threads 21](#_Toc501111905)

[Multiple Ownership with Multiple Threads 24](#_Toc501111906)

[Atomic Reference Counting with Arc<T> 26](#_Toc501111907)

[Similarities between RefCell<T>/Rc<T> and Mutex<T>/Arc<T> 28](#_Toc501111908)

[Extensible Concurrency with the Sync and Send Traits 28](#_Toc501111909)

[Allowing Transference of Ownership Between Threads with Send 28](#_Toc501111910)

[Allowing Access from Multiple Threads with Sync 29](#_Toc501111911)

[Implementing Send and Sync Manually is Unsafe 29](#_Toc501111912)

[Summary 29](#_Toc501111913)

16

Fearless Concurrency

Handling concurrent programming safely and efficiently is another of Rust’s major goals. Concurrent programming, where different parts of a program execute independently, and parallel programming, where different parts of a program execute at the same time, are becoming increasingly important as more computers take advantage of their multiple processors. Historically, programming in these contexts has been difficult and error prone: Rust hopes to change that.

Initially, the Rust team thought that ensuring memory safety and preventing concurrency problems were two separate challenges to be solved with different methods. Over time, the team discovered that the ownership and type systems are a powerful set of tools to help manage memory safety and concurrency problems! By leveraging ownership and type checking, many concurrency errors are compile time errors in Rust rather than runtime errors. Therefore, rather than you spending lots of time trying to reproduce the exact circumstances under which a runtime concurrency bug occurs, incorrect code will simply refuse to compile and present an error explaining the problem. As a result, you can fix your code while you’re working on it rather than potentially after it has been shipped to production. We’ve nicknamed this aspect of Rust fearless concurrency. Fearless concurrency allows you to write code that is free of subtle bugs and is easy to refactor without introducing new bugs.

Note For simplicity’s sake, we’ll refer to many of the problems as concurrent rather than being more precise by saying concurrent and/or parallel. If this book was specifically about concurrency and/or parallelism, we’d be more specific. For this chapter, please mentally substitute concurrent and/or parallel whenever we use concurrent.

Many languages are dogmatic about the solutions they offer for handling concurrent problems. For example, Erlang has elegant functionality for message passing concurrency but has only obscure ways to share state between threads. Supporting only a subset of possible solutions is a reasonable strategy for higher-level languages, because a higher-level language promises benefits from giving up some control to gain abstractions. However, lower-level languages are expected to provide the solution with the best performance in any given situation and have fewer abstractions over the hardware. Therefore, Rust offers a variety of tools for modeling problems in whatever way is appropriate for your situation and requirements.

Here are the topics we’ll cover in this chapter:

How to create threads to run multiple pieces of code at the same time

Message passing concurrency, where channels send messages between threads

Shared state concurrency, where multiple threads have access to some piece of data

The Sync and Send traits, which extend Rust’s concurrency guarantees to user-defined types as well as types provided by the standard library

Using Threads to Run Code Simultaneously

In most current operating systems, an executed program’s code is run in a process, and the operating system manages multiple processes at once. Within your program, you can also have independent parts that run simultaneously. The feature that runs these independent parts is called threads.

Splitting the computation in your program into multiple threads can improve performance because the program does multiple tasks at the same time, but it also adds complexity. Because threads can run simultaneously, there’s no inherent guarantee about the order in which parts of your code on different threads will run. This can lead to problems, such as:

Race conditions, where threads are accessing data or resources in an inconsistent order

Deadlocks, where two threads are waiting for each other to finish using a resource the other thread has, preventing both threads from continuing

Bugs that only happen in certain situations and are hard to reproduce and fix reliably

Rust attempts to mitigate the negative effects of using threads. Programming in a multithreaded context still takes careful thought and requires a code structure that is different from programs that run in a single thread.

Programming languages implement threads in a few different ways. Many operating systems provide an API for creating new threads. This model where a language calls the operating system APIs to create threads is sometimes called 1:1, one operating system thread per one language thread.

Many programming languages provide their own special implementation of threads. Programming language-provided threads are known as green threads, and languages that use these green threads will execute them in the context of a different number of operating system threads. For this reason, the green threaded model is called the M:N model: M green threads per N  operating system threads, where M and N are not necessarily the same number.

Each model has its own advantages and trade-offs, and the trade-off most important to Rust is runtime support. Runtime is a confusing term and can have different meanings in different contexts.

In this context, by runtime we mean code that is included by the language in every binary. This code can be large or small depending on the language, but every non-assembly language will have some amount of runtime code. For that reason, colloquially when people say a language has “no runtime,” they often mean “small runtime.” Smaller runtimes have fewer features but have the advantage of resulting in smaller binaries, which make it easier to combine the language with other languages in more contexts. Although many languages are okay with increasing the runtime size in exchange for more features, Rust needs to have nearly no runtime and cannot compromise on being able to call into C to maintain performance.

The green threading M:N model requires a larger language runtime to manage threads. As such, the Rust standard library only provides an implementation of 1:1 threading. Because Rust is such a low-level language, there are crates that implement M:N threading if you would rather trade overhead for aspects such as more control over which threads run when and lower costs of context switching, for example.

Now that we’ve defined threads in Rust, let’s explore how to use the thread-related API provided by the standard library.

Creating a New Thread with spawn

To create a new thread, we call the thread::spawn function and pass it a closure (we talked about closures in Chapter 13) containing the code we want to run in the new thread. The example in Listing 16-1 prints some text from a main thread and other text from a new thread:

prod: confirm xref

src/main.rs

use std::thread;

fn main() {

thread::spawn(|| {

for i in 1..10 {

println!("hi number {} from the spawned thread!", i);

}

});

for i in 1..5 {

println!("hi number {} from the main thread!", i);

}

}

Listing 16-1: Creating a new thread to print one thing while the main thread prints something else

Note that with this function, the new thread will be stopped when the main thread ends, whether or not it has finished running. The output from this program might be a little different every time, but it will look similar to the following:

hi number 1 from the main thread!

hi number 1 from the spawned thread!

hi number 2 from the main thread!

hi number 2 from the spawned thread!

hi number 3 from the main thread!

hi number 3 from the spawned thread!

hi number 4 from the main thread!

hi number 4 from the spawned thread!

hi number 5 from the spawned thread!

The threads will probably take turns, but that is not guaranteed: it depends on how your operating system schedules the threads. In this run, the main thread printed first, even though the print statement from the spawned thread appears first in the code. And even though we told the spawned thread to print until i is 9, it only got to 5 before the main thread shut down.

If you run this code and only see one thread, or don’t see any overlap, try increasing the numbers in the ranges to create more opportunities for a thread to take a break and give the other thread a turn.

Waiting for All Threads to Finish Using join Handles

The code in Listing 16-1 not only stops the spawned thread prematurely most of the time due to the main thread ending, but there is no guarantee that the spawned thread will get to run at all. The reason is that there is no guarantee on the order in which threads run!

We can fix this by saving the return value of thread::spawn in a variable. The return type of thread::spawn is JoinHandle. A JoinHandle is an owned value that, when we call the join method on it, will wait for its thread to finish. Listing 16-2 shows how to use the JoinHandle of the thread we created in Listing 16-1 and call join to make sure the spawned thread finishes before the main exits:

src/main.rs

use std::thread;

fn main() {

let handle = thread::spawn(|| {

for i in 1..10 {

println!("hi number {} from the spawned thread!", i);

}

});

for i in 1..5 {

println!("hi number {} from the main thread!", i);

}

handle.join();

}

Listing 16-2: Saving a JoinHandle from thread::spawn to guarantee the thread is run to completion

Calling join on the handle blocks the thread currently running until the thread represented by the handle terminates. Blocking a thread means that thread is prevented from performing work or exiting. Because we’ve put the call to join after the main thread’s for loop, running Listing 16-2 should produce output similar to this:

hi number 1 from the main thread!

hi number 2 from the main thread!

hi number 1 from the spawned thread!

hi number 3 from the main thread!

hi number 2 from the spawned thread!

hi number 4 from the main thread!

hi number 3 from the spawned thread!

hi number 4 from the spawned thread!

hi number 5 from the spawned thread!

hi number 6 from the spawned thread!

hi number 7 from the spawned thread!

hi number 8 from the spawned thread!

hi number 9 from the spawned thread!

The two threads continue alternating, but the main thread waits because of the call to handle.join() and does not end until the spawned thread is finished.

But let’s see what happens when we instead move handle.join() before the for loop in main, like this:

src/main.rs

use std::thread;

fn main() {

let handle = thread::spawn(|| {

for i in 1..10 {

println!("hi number {} from the spawned thread!", i);

}

});

handle.join();

for i in 1..5 {

println!("hi number {} from the main thread!", i);

}

}

The main thread will wait for the spawned thread to finish and then run its for loop, so the output won’t be interleaved anymore, as shown here:

hi number 1 from the spawned thread!

hi number 2 from the spawned thread!

hi number 3 from the spawned thread!

hi number 4 from the spawned thread!

hi number 5 from the spawned thread!

hi number 6 from the spawned thread!

hi number 7 from the spawned thread!

hi number 8 from the spawned thread!

hi number 9 from the spawned thread!

hi number 1 from the main thread!

hi number 2 from the main thread!

hi number 3 from the main thread!

hi number 4 from the main thread!

Thinking about such a small detail as where to call join can affect whether or not your threads run at the same time.

Using move Closures with Threads

The move closure, which we didn’t cover in Chapter 13, is often used alongside thread::spawn because it allows us to use data from one thread in another thread.

In Chapter 13, we said that “Creating closures that capture values from their environment is mostly used in the context of starting new threads.”

prod: confirm xrefs

Now that we’re creating new threads, we’ll talk about capturing values in closures.

Notice in Listing 16-1 that the closure we pass to thread::spawn takes no arguments: we’re not using any data from the main thread in the spawned thread’s code. To do so, the spawned thread’s closure must capture the values it needs. Listing 16-3 shows an attempt to create a vector in the main thread and use it in the spawned thread. However, this won’t yet work, as you’ll see in a moment:

src/main.rs

use std::thread;

fn main() {

let v = vec![1, 2, 3];

let handle = thread::spawn(|| {

println!("Here's a vector: {:?}", v);

});

handle.join();

}

Listing 16-3: Attempting to use a vector created by the main thread in another thread

The closure uses v, so it will capture v and make it part of the closure’s environment. Because thread::spawn runs this closure in a new thread, we should be able to access v inside that new thread. But when we compile this example, we get the following error:

error[E0373]: closure may outlive the current function, but it borrows `v`,

which is owned by the current function

-->

|

6 | let handle = thread::spawn(|| {

| ^^ may outlive borrowed value `v`

7 | println!("Here's a vector: {:?}", v);

| - `v` is borrowed here

|

help: to force the closure to take ownership of `v` (and any other referenced

variables), use the `move` keyword, as shown:

| let handle = thread::spawn(move || {

Rust infers how to capture v, and because println! only needs a reference to v, the closure tries to borrow v. However, there’s a problem: Rust can’t tell how long the spawned thread will run, so it doesn’t know if the reference to v will always be valid.

Listing 16-4 provides a scenario that’s more likely to have a reference to v that won’t be valid:

src/main.rs

use std::thread;

fn main() {

let v = vec![1, 2, 3];

let handle = thread::spawn(|| {

println!("Here's a vector: {:?}", v);

});

drop(v); // oh no!

handle.join();

}

Listing 16-4: A thread with a closure that attempts to capture a reference to v from a main thread that drops v

When we run this code, there’s a possibility the spawned thread will be immediately put in the background without running at all. The spawned thread has a reference to v inside, but the main thread immediately drops v, using the drop function we discussed in Chapter 15. Then, when the spawned thread starts to execute, v is no longer valid, so a reference to it is also invalid. Oh no!

prod: confirm xref

To fix the problem in Listing 16-3, we can use the error message’s advice:

help: to force the closure to take ownership of `v` (and any other referenced

variables), use the `move` keyword, as shown:

| let handle = thread::spawn(move || {

By adding the move keyword before the closure, we force the closure to take ownership of the values it’s using rather than allowing Rust to infer that it should borrow. The modification to Listing 16-3 shown in Listing 16-5 will compile and run as we intend:

src/main.rs

use std::thread;

fn main() {

let v = vec![1, 2, 3];

let handle = thread::spawn(move || {

println!("Here's a vector: {:?}", v);

});

handle.join();

}

Listing 16-5: Using the move keyword to force a closure to take ownership of the values it uses

What would happen to the code in Listing 16-4 where the main thread called drop if we use a move closure? Would move fix that case? Unfortunately, no; we would get a different error because what Listing 16-4 is trying to do isn’t allowed for a different reason. If we add move to the closure, we would move v into the closure’s environment, and we could no longer call drop on it in the main thread. We would get this compiler error instead:

error[E0382]: use of moved value: `v`

-->

|

6 | let handle = thread::spawn(move || {

| ------- value moved (into closure) here

...

10 | drop(v); // oh no!

| ^ value used here after move

|

= note: move occurs because `v` has type `std::vec::Vec<i32>`, which does

not implement the `Copy` trait

Rust’s ownership rules have saved us again! We got an error from the code in Listing 16-3 because Rust was being conservative and only borrowing v for the thread, which meant the main thread could theoretically invalidate the spawned thread’s reference. By telling Rust to move ownership of v to the spawned thread, we’re guaranteeing Rust that the main thread won’t use v anymore. If we change Listing 16-4 in the same way, we’re then violating the ownership rules when we try to use v in the main thread. The move keyword overrides Rust’s conservative default of borrowing; it doesn’t let us violate the ownership rules.

With a basic understanding of threads and the thread API, let’s look at what we can do with threads.

Message Passing to Transfer Data Between Threads

One increasingly popular approach to ensuring safe concurrency is message passing, where threads or actors communicate by sending each other messages containing data. Here’s the idea in a slogan from the Go language documentation:

Do not communicate by sharing memory; instead, share memory by communicating.

—Effective Go at <http://golang.org/doc/effective_go.html>

One major tool Rust has for accomplishing message sending concurrency is the channel, a programming concept that Rust’s standard library provides an implementation of. You can imagine a channel in programming like a channel of water, such as a stream or a river. If you put something like a rubber duck or a boat into a stream, it will travel downstream to the end of the river.

A channel in programming has two halves: a transmitter and a receiver. The transmitter half is the upstream location where we put rubber ducks into the river, and the receiver half is where the rubber duck ends up downstream. One part of our code calls methods on the transmitter with the data we want to send, and another part checks the receiving end for arriving messages.

Here, we’ll work up to a program that has one thread to generate values and send them down a channel, and another thread that will receive the values and print them out. We’ll be sending simple values between threads using a channel to illustrate the feature. Once you’re familiar with the technique, you could use channels to implement a chat system or a system where many threads perform parts of a calculation and send the parts to one thread that aggregates the results.

First, in Listing 16-6, we’ll create a channel but not do anything with it:

src/main.rs

use std::sync::mpsc;

fn main() {

let (tx, rx) = mpsc::channel();

}

Listing 16-6: Creating a channel and assigning the two halves to tx and rx

We create a new channel using the mpsc::channel function; mpsc stands for multiple producer, single consumer. In short, the way Rust’s standard library implements channels means a channel can have multiple sending ends that produce values but only one receiving end that consumes those values. Imagine multiple rivers and streams flowing together into one big river: everything sent down any of the streams will end up in one river at the end. We’ll start with a single producer for now, but we’ll add multiple producers when we get this example working.

The mpsc::channel function returns a tuple, the first element of which is the sending end and the second element is the receiving end. The abbreviations tx and rx are traditionally used in many fields for transmitter and receiver respectively, so we name our variables as such to indicate each end. We’re using a let statement with a pattern that destructures the tuples; we’ll discuss the use of patterns in let statements and destructuring in Chapter 18. Using a let statement this way is a convenient approach to extract the pieces of the tuple returned by mpsc::channel.

prod: confirm xref

Let’s move the transmitting end into a spawned thread and have it send one string so the spawned thread is communicating with the main thread, as shown in Listing 16-7. This is like putting a rubber duck in the river upstream or sending a chat message from one thread to another:

src/main.rs

use std::thread;

use std::sync::mpsc;

fn main() {

let (tx, rx) = mpsc::channel();

thread::spawn(move || {

let val = String::from("hi");

tx.send(val).unwrap();

});

}

Listing 16-7: Moving tx to a spawned thread and sending “hi”

Again, we’re using thread::spawn to create a new thread and then using move to move tx into the closure so the spawned thread owns tx. The spawned thread needs to own the transmitting end of the channel to be able to send messages through the channel.

The transmitting end has a send method that takes the value we want to send. The send method returns a Result<T, E> type, so if the receiving end has already been dropped and there’s nowhere to send a value, the send operation will result in an error. In this example, we’re simply calling unwrap to panic in case of an error. But in a real application, we would handle it properly: return to Chapter 9 to review strategies for proper error handling.

prod: confirm xref

In Listing 16-8, we’ll get the value from the receiving end of the channel in the main thread. This is like retrieving the rubber duck from the water at the end of the river or like getting a chat message:

src/main.rs

use std::thread;

use std::sync::mpsc;

fn main() {

let (tx, rx) = mpsc::channel();

thread::spawn(move || {

let val = String::from("hi");

tx.send(val).unwrap();

});

let received = rx.recv().unwrap();

println!("Got: {}", received);

}

Listing 16-8: Receiving the value “hi” in the main thread and printing it

The receiving end of a channel has two useful methods: recv and try\_recv. We’re using recv, short for receive, which will block the main thread’s execution and wait until a value is sent down the channel. Once a value is sent, recv will return it in a Result<T, E>. When the sending end of the channel closes, recv will return an error to signal that no more values will be coming.

The try\_recv method doesn’t block, but will instead return a Result<T, E> immediately: an Ok value holding a message if one is available and an Err value if there aren’t any messages this time. Using try\_recv is useful if this thread has other work to do while waiting for messages: we could write a loop that calls try\_recv every so often, handles a message if one is available, and otherwise does other work for a little while until checking again.

We’ve used recv in this example for simplicity; we don’t have any other work for the main thread to do other than wait for messages, so blocking the main thread is appropriate.

When we run the code in Listing 16-8, we’ll see the value printed from the main thread:

Got: hi

Perfect!

Channels and Ownership Transference

The ownership rules play a vital role in message sending because they help us write safe, concurrent code. Preventing errors in concurrent programming is the advantage we get by making the trade-off of having to think about ownership throughout our Rust programs. Let’s do an experiment to show how channels and ownership work together to prevent problems: we’ll try to use a val value in the spawned thread after we’ve sent it down the channel. Try compiling the code in Listing 16-9:

src/main.rs

use std::thread;

use std::sync::mpsc;

fn main() {

let (tx, rx) = mpsc::channel();

thread::spawn(move || {

let val = String::from("hi");

tx.send(val).unwrap();

println!("val is {}", val);

});

let received = rx.recv().unwrap();

println!("Got: {}", received);

}

Listing 16-9: Attempting to use val after we’ve sent it down the channel

Here, we try to print val after we’ve sent it down the channel via tx.send. Allowing this would be a bad idea: once the value has been sent to another thread, that thread could modify or drop it before we try to use the value again. Potentially, the other thread could cause errors or unexpected results due to inconsistent or nonexistent data. However, Rust gives us an error if we try to compile the code in Listing 16-9:

error[E0382]: use of moved value: `val`

--> src/main.rs:10:31

|

9 | tx.send(val).unwrap();

| --- value moved here

10 | println!("val is {}", val);

| ^^^ value used here after move

|

= note: move occurs because `val` has type `std::string::String`, which does

not implement the `Copy` trait

Our concurrency mistake has caused a compile time error. The send function takes ownership of its parameter, and when the value is moved, the receiver takes ownership of it. This stops us from accidentally using the value again after sending it; the ownership system checks that everything is okay.

Sending Multiple Values and Seeing the Receiver Waiting

The code in Listing 16-8 compiled and ran, but it didn’t clearly show us that two separate threads were talking to each other over the channel. In Listing 16-10 we’ve made some modifications that will prove this code is running concurrently: the spawned thread will now send multiple messages and pause for a second between each message:

src/main.rs

use std::thread;

use std::sync::mpsc;

use std::time::Duration;

fn main() {

let (tx, rx) = mpsc::channel();

thread::spawn(move || {

let vals = vec![

String::from("hi"),

String::from("from"),

String::from("the"),

String::from("thread"),

];

for val in vals {

tx.send(val).unwrap();

thread::sleep(Duration::from\_secs(1));

}

});

for received in rx {

println!("Got: {}", received);

}

}

Listing 16-10: Sending multiple messages and pausing between each one

This time, the spawned thread has a vector of strings that we want to send to the main thread. We iterate over them, sending each individually, and pause between each by calling the thread::sleep function with a Duration value of one second.

In the main thread, we’re not calling the recv function explicitly anymore: instead, we’re treating rx as an iterator. For each value received, we’re printing it. When the channel is closed, iteration will end.

When running the code in Listing 16-10, you should see the following output with a one second pause in between each line:

Got: hi

Got: from

Got: the

Got: thread

Because we don’t have any code that pauses or delays in the for loop in the main thread, we can tell that the main thread is waiting to receive values from the spawned thread.

Creating Multiple Producers by Cloning the Transmitter

Earlier we mentioned that mpsc was the acronym for multiple producer, single consumer. Let’s put  mpsc to use and expand the code in Listing 16-10 to create multiple threads that all send values to the same receiver. We can do so by cloning the transmitting half of the channel, as shown in Listing 16-11:

src/main.rs

// --snip--

let (tx, rx) = mpsc::channel();

let tx1 = mpsc::Sender::clone(&tx);

thread::spawn(move || {

let vals = vec![

String::from("hi"),

String::from("from"),

String::from("the"),

String::from("thread"),

];

for val in vals {

tx1.send(val).unwrap();

thread::sleep(Duration::from\_secs(1));

}

});

thread::spawn(move || {

let vals = vec![

String::from("more"),

String::from("messages"),

String::from("for"),

String::from("you"),

];

for val in vals {

tx.send(val).unwrap();

thread::sleep(Duration::from\_secs(1));

}

});

// --snip--

Listing 16-11: Sending multiple messages and pausing between each one

This time, before we create the first spawned thread, we call clone on the sending end of the channel. This will give us a new sending handle we can pass to the first spawned thread. We pass the original sending end of the channel to a second spawned thread. This gives us two threads, each sending different messages to the receiving end of the channel.

When you run the code, you’ll probably see output like this:

Got: hi

Got: more

Got: from

Got: messages

Got: for

Got: the

Got: thread

Got: you

You might see the values in a another order; it depends on your system. This is what makes concurrency interesting as well as difficult. If you experiment with thread::sleep, giving it various values in the different threads, each run will be more non-deterministic and create different output each time.

Now that we’ve looked at how channels work, let’s look at a different method of concurrency.

Shared State Concurrency

Message passing is a fine way of handling concurrency, but it’s not the only one. Consider the second part of the slogan from the Go language documentation again: “share memory by communicating.”

What would communicate by sharing memory look like? In addition, why would message passing enthusiasts not use it and do the opposite instead?

In a way, channels in any programming language are similar to single ownership, because once you transfer a value down a channel, you should no longer use that value. Shared memory concurrency is like multiple ownership: multiple threads can access the same memory location at the same time. As you saw in Chapter 15 where smart pointers made multiple ownership possible, multiple ownership can add additional complexity because these different owners need managing.

Rust’s type system and ownership rules greatly assist in getting this management correct. For an example, let’s look at mutexes, one of the more common concurrency primitives for shared memory.

Mutexes Allow Access to Data from One Thread at a Time

A mutex is an abbreviation for “mutual exclusion,” as in, it only allows one thread to access some data at any given time. To access the data in a mutex, a thread must first signal that it wants access by asking to acquire the mutex’s lock. The lock is a data structure that is part of the mutex that keeps track of who currently has exclusive access to the data. Therefore, we describe the mutex as guarding the data it holds via the locking system.

Mutexes have a reputation for being difficult to use because you have to remember two rules:

You must attempt to acquire the lock before using the data.

When you’re done with the data that the mutex guards, you must unlock the data so other threads can acquire the lock.

For a real-world metaphor of a mutex, imagine a panel discussion at a conference with only one microphone. Before a panelist can speak, they have to ask or signal that they want to use the microphone. When they get the microphone, they can talk for as long as they want to and then hand the microphone to the next panelist who requests to speak. If a panelist forgets to hand the microphone off when they’re finished with it, no one else is able to speak. If management of the shared microphone goes wrong, the panel wouldn’t work as planned!

Management of mutexes can be incredibly tricky to get right, which is why so many people are enthusiastic about channels. However, thanks to Rust’s type system and ownership rules, we can’t get locking and unlocking wrong.

The API of Mutex<T>

As an example of how to use a mutex, let’s start simply by using a mutex in a single-threaded context, as shown in Listing 16-12:

src/main.rs

use std::sync::Mutex;

fn main() {

let m = Mutex::new(5);

{

let mut num = m.lock().unwrap();

\*num = 6;

}

println!("m = {:?}", m);

}

Listing 16-12: Exploring the API of Mutex<T> in a single-threaded context for simplicity

As with many types, we create a Mutex<T> using the associated function new. To access the data inside the mutex, we use the lock method to acquire the lock. This call will block the current thread so it can’t do any work until it’s our turn to have the lock.

The call to lock would fail if another thread holding the lock panicked. In that case, no one would ever be able to get the lock, so we’ve chosen to unwrap and have this thread panic if we’re in that situation.

After we’ve acquired the lock, we can treat the return value, named num in this case, as a mutable reference to the data inside. The type system ensures that we acquire a lock before using this value: Mutex<i32> is not an i32, so we must acquire the lock to be able to use the i32 value. We can’t forget; the type system won’t let us do it otherwise.

As you might suspect, Mutex<T> is a smart pointer. More accurately, the call to lock returns a smart pointer called MutexGuard. This smart pointer implements Deref to point at our inner data; the smart pointer also has a Drop implementation that releases the lock automatically when MutexGuard goes out of scope, which happens at the end of the inner scope in Listing 16-12. As a result, we don’t risk forgetting to release the lock and blocking it from use by other threads because it happens automatically.

After dropping the lock, we can print the mutex value and see that we were able to change the inner i32 to 6.

Sharing a Mutex<T> Between Multiple Threads

Now, let’s try to share a value between multiple threads using Mutex<T>. We’ll spin up 10 threads and have them each increment a counter value by 1, so the counter goes from 0 to 10. Note that the next few examples will have compiler errors, and we’ll use those errors to learn more about using Mutex<T> and how Rust helps us use it correctly. Listing 16-13 has our starting example:

src/main.rs

use std::sync::Mutex;

use std::thread;

fn main() {

let counter = Mutex::new(0);

let mut handles = vec![];

for \_ in 0..10 {

let handle = thread::spawn(move || {

let mut num = counter.lock().unwrap();

\*num += 1;

});

handles.push(handle);

}

for handle in handles {

handle.join().unwrap();

}

println!("Result: {}", \*counter.lock().unwrap());

}

Listing 16-13: Ten threads each increment a counter guarded by a Mutex<T>.

We’re creating a counter variable to hold an i32 inside a Mutex<T>, as we did in Listing 16-12. Next, we’re creating 10 threads by mapping over a range of numbers. We use thread::spawn and give all the threads the same closure, one that moves the counter into the thread, acquires a lock on the Mutex<T> by calling the lock method, and then adds 1 to the value in the mutex. When a thread finishes running its closure, num will go out of scope and release the lock so another thread can acquire it.

In the main thread, we collect all the join handles, as we did in Listing 16-2, and then call join on each to make sure all the threads finish. At that point, the main thread will acquire the lock and print the result of this program.

We hinted that this example won’t compile, now let’s find out why!

error[E0382]: capture of moved value: `counter`

-->

|

9 | let handle = thread::spawn(move || {

| ------- value moved (into closure) here

10 | let mut num = counter.lock().unwrap();

| ^^^^^^^ value captured here after move

|

= note: move occurs because `counter` has type `std::sync::Mutex<i32>`,

which does not implement the `Copy` trait

error[E0382]: use of moved value: `counter`

-->

|

9 | let handle = thread::spawn(move || {

| ------- value moved (into closure) here

...

21 | println!("Result: {}", \*counter.lock().unwrap());

| ^^^^^^^ value used here after move

|

= note: move occurs because `counter` has type `std::sync::Mutex<i32>`,

which does not implement the `Copy` trait

error: aborting due to 2 previous errors

The error message states that the counter value is moved into the closure and then is captured when we call lock. That description sounds like what we wanted, but it’s not allowed!

Let’s figure this out by simplifying the program. Instead of making 10 threads in a for loop, let’s just make two threads without a loop and see what happens. Replace the first for loop in Listing 16-13 with this code instead:

let handle = thread::spawn(move || {

let mut num = counter.lock().unwrap();

\*num += 1;

});

handles.push(handle);

let handle2 = thread::spawn(move || {

let mut num2 = counter.lock().unwrap();

\*num2 += 1;

});

handles.push(handle2);

We make two threads and change the variable names used with the second thread to handle2 and num2. When we run the code this time, compiling gives us the following:

error[E0382]: capture of moved value: `counter`

-->

|

8 | let handle = thread::spawn(move || {

| ------- value moved (into closure) here

...

16 | let mut num2 = counter.lock().unwrap();

| ^^^^^^^ value captured here after move

|

= note: move occurs because `counter` has type `std::sync::Mutex<i32>`,

which does not implement the `Copy` trait

error[E0382]: use of moved value: `counter`

-->

|

8 | let handle = thread::spawn(move || {

| ------- value moved (into closure) here

...

26 | println!("Result: {}", \*counter.lock().unwrap());

| ^^^^^^^ value used here after move

|

= note: move occurs because `counter` has type `std::sync::Mutex<i32>`,

which does not implement the `Copy` trait

error: aborting due to 2 previous errors

Aha! The first error message indicates that counter is moved into the closure for the thread associated with handle. That move is preventing us from capturing counter when we try to call lock on it and store the result in num2 in the second thread! So Rust is telling us that we can’t move ownership of counter into multiple threads. This was hard to see earlier because our threads were in a loop, and Rust can’t point to different threads in different iterations of the loop. Let’s fix this with a multiple-ownership method we discussed in Chapter 15.

prod: confirm xref

Multiple Ownership with Multiple Threads

In Chapter 15, we gave a value multiple owners by using the smart pointer Rc<T> to create a reference-counted value. Let’s do the same here and see what happens. We’ll wrap the Mutex<T> in Rc<T> in Listing 16-14 and clone the Rc<T> before moving ownership to the thread. Now that we’ve seen the errors, we’ll also switch back to using the for loop, and we’ll keep the move keyword with the closure:

prod: confirm xref

src/main.rs

use std::rc::Rc;

use std::sync::Mutex;

use std::thread;

fn main() {

let counter = Rc::new(Mutex::new(0));

let mut handles = vec![];

for \_ in 0..10 {

let counter = Rc::clone(&counter);

let handle = thread::spawn(move || {

let mut num = counter.lock().unwrap();

\*num += 1;

});

handles.push(handle);

}

for handle in handles {

handle.join().unwrap();

}

println!("Result: {}", \*counter.lock().unwrap());

}

Listing 16-14: Attempting to use Rc<T> to allow multiple threads to own the Mutex<T>

Once again, we compile and get. . .different errors! The compiler is teaching us a lot.

error[E0277]: the trait bound `std::rc::Rc<std::sync::Mutex<i32>>:

std::marker::Send` is not satisfied

-->

|

11 | let handle = thread::spawn(move || {

| ^^^^^^^^^^^^^ the trait `std::marker::Send` is not

implemented for `std::rc::Rc<std::sync::Mutex<i32>>`

|

= note: `std::rc::Rc<std::sync::Mutex<i32>>` cannot be sent between threads

safely

= note: required because it appears within the type

`[closure@src/main.rs:11:36: 15:10

counter:std::rc::Rc<std::sync::Mutex<i32>>]`

= note: required by `std::thread::spawn`

Wow, that error message is very wordy! Here are some important parts to focus on: the first note says Rc<Mutex<i32>> cannot be sent between threads safely. The reason for this is in the error message, which, once distilled, says the trait bound Send is not satisfied. We’ll talk about Send in the next section: it’s one of the traits that ensures the types we use with threads are meant for use in concurrent situations.

Unfortunately, Rc<T> is not safe to share across threads. When Rc<T> manages the reference count, it adds to the count for each call to clone and subtracts from the count when each clone is dropped. But it doesn’t use any concurrency primitives to make sure that changes to the count can’t be interrupted by another thread. This could lead to wrong counts—subtle bugs that could in turn lead to memory leaks or a value being dropped before we’re done with it. What we need is a type exactly like Rc<T> but one that makes changes to the reference count in a thread-safe way.

Atomic Reference Counting with Arc<T>

Fortunately, Arc<T> is a type like Rc<T> that is safe to use in concurrent situations. The ‘a’ stands for atomic, meaning it’s an atomically reference counted type. Atomics are an additional kind of concurrency primitive that we won’t cover in detail here: see the standard library documentation for std::sync::atomic for more details. At this point, you just need to know that atomics work like primitive types but are safe to share across threads.

You might then wonder why all primitive types aren’t atomic and why standard library types aren’t implemented to use Arc<T> by default. The reason is that thread safety comes with a performance penalty that you only want to pay when you really need to. If you’re just performing operations on values within a single thread, your code can run faster if it doesn’t have to enforce the guarantees atomics provide.

Let’s return to our example: Arc<T> and Rc<T> have the same API, so we fix our program by changing the use line and the call to new. The code in Listing 16-15 will finally compile and run:

src/main.rs

use std::sync::{Mutex, Arc};

use std::thread;

fn main() {

let counter = Arc::new(Mutex::new(0));

let mut handles = vec![];

for \_ in 0..10 {

let counter = Arc::clone(&counter);

let handle = thread::spawn(move || {

let mut num = counter.lock().unwrap();

\*num += 1;

});

handles.push(handle);

}

for handle in handles {

handle.join().unwrap();

}

println!("Result: {}", \*counter.lock().unwrap());

}

Listing 16-15: Using an Arc<T> to wrap the Mutex<T> to be able to share ownership across multiple threads

This code will print the following:

Result: 10

We did it! We counted from 0 to 10, which may not seem very impressive, but it did teach us a lot about Mutex<T> and thread safety. You could also use this structure to do more complicated operations than just incrementing a counter. These methods allow us to divide calculations into independent parts, which we could split across threads; then we could use a Mutex<T> to have each thread update the final result with its part.

Similarities Between RefCell<T>/Rc<T> and Mutex<T>/Arc<T>

You might have noticed that counter is immutable, but we could get a mutable reference to the value inside it; this means Mutex<T> provides interior mutability, like the Cell family does. In the same way we used RefCell<T> in Chapter 15 to allow us to mutate contents inside an Rc<T>, we use Mutex<T> to mutate contents inside an Arc<T>.

prod: confirm xref

Another detail to note is that Rust can’t protect us from all kinds of logic errors when we use Mutex<T>. Recall in Chapter 15 that using Rc<T> came with the risk of creating reference cycles, where two Rc<T> values refer to each other, causing memory leaks. Similarly, Mutex<T> comes with the risk of creating deadlocks. These occur when an operation needs to lock two resources and two threads have each acquired one of the locks, causing them to wait for each other forever. If you’re interested in deadlocks, try creating a Rust program that has a deadlock; then research deadlock mitigation strategies for mutexes in any language and have a go at implementing them in Rust. The standard library API documentation for Mutex<T> and MutexGuard  offers useful information.

prod: confirm xref

We’ll round out this chapter by talking about the Send and Sync traits, and how we can use them with custom types.

Extensible Concurrency with the Sync and Send Traits

Interestingly, the Rust language knows very little about concurrency. Almost every concurrency feature we’ve talked about so far in this chapter has been part of the standard library, not the language. Our concurrency options are not limited to the language or the standard library; we can write our own concurrency options or use those written by others.

However, two concurrency concepts are embedded in the language: the std::marker traits Sync and Send.

Allowing Transference of Ownership Between Threads with Send

The Send marker trait indicates that ownership of the type implementing Send can be transferred between threads. Almost every Rust type is Send, but there are some exceptions, including Rc<T>: this cannot be Send because if we cloned an Rc<T> value and tried to transfer ownership of the clone to another thread, both threads might update the reference count at the same time. For this reason, Rc<T> is implemented for use in single-threaded situations where you don’t want to pay the thread-safe performance penalty.

Therefore, Rust’s type system and trait bounds ensure that we can never accidentally send an Rc<T> value across threads unsafely. When we tried to do this in Listing 16-14, we got the error the trait Send is not implemented for Rc<Mutex<i32>>. When we switched to Arc<T>, which is Send, the code compiled.

Any type composed entirely of Send types is automatically marked as Send as well. Almost all primitive types are Send, aside from raw pointers, which we’ll discuss in Chapter 19.

prod: confirm xref

Allowing Access from Multiple Threads with Sync

The Sync marker trait indicates that it is safe for the type implementing Sync to be referenced from multiple threads. In other words, any type T is Sync if &T (a reference to T) is Send, meaning the reference can be sent safely to another thread. Similar to Send, primitive types are Sync and types composed entirely of types that are Sync are also Sync.

The smart pointer Rc<T> is also not Sync for the same reasons that it’s not Send. The RefCell<T> type (which we talked about in Chapter 15) and the family of related Cell<T> types are not Sync. The implementation of borrow checking that RefCell<T> does at runtime is not thread safe. The smart pointer Mutex<T> is Sync and can be used to share access with multiple threads, as you saw in the previous section.

prod: confirm xref

Implementing Send and Sync Manually Is Unsafe

Because types that are made up of Send and Sync traits are automatically also Send and Sync, we don’t have to implement those traits manually. As marker traits, they don’t even have any methods to implement. They’re just useful for enforcing concurrency related invariants.

Manually implementing these traits involves implementing unsafe Rust code. We’ll talk about using unsafe Rust code in Chapter 19; for now, the important information is that building new concurrent types not made up of Send and Sync parts requires careful thought to uphold the safety guarantees. The Nomicon at <https://doc.rust-lang.org/stable/nomicon/> has more information about these guarantees and how to uphold them.

prod: confirm xref

Summary

This isn’t the last you’ll see of concurrency in this book: the project in Chapter 20 will use the concepts examined in this chapter in a more realistic situation than the smaller examples discussed here.

confirm xref

As mentioned earlier, because very little of how Rust handles concurrency is part of the language, many concurrency solutions are implemented as crates. These evolve more quickly than the standard library, so be sure to search online for the current, state-of-the-art crates to use in multithreaded situations.

Rust provides channels for message passing and smart pointer types, such as Mutex<T> and Arc<T>, that are safe to use in concurrent contexts. The type system and the borrow checker ensure that the code using these solutions won’t end up with data races or invalid references. Once we get our code to compile, we can rest assured that it will happily run on multiple threads without the kinds of hard-to-track-down bugs common in other languages. Concurrent programming is no longer a concept to be afraid of: go forth and make your programs concurrent, fearlessly!

Next, we’ll talk about idiomatic ways to model problems and structure solutions as your Rust programs get bigger. In addition, we’ll discuss how Rust’s idioms relate to those you might be familiar with from object oriented programming.