O'REILLY®

Rust Atomics and Locks

Low-Level Concurrency in Practice

Mara Bos

O'REILLY®

Rust Atomics and Locks

The Rust programming language is extremely well suited for concurrency, and its ecosystem has many libraries that include lots of concurrent data structures, locks, and more. But implementing those structures correctly can be difficult. Even in the most well-used libraries, memory ordering bugs are not uncommon.

In this practical book, Mara Bos, team lead of the Rust library team, helps Rust programmers of all levels gain a clear understanding of low-level concurrency. You'll learn everything about atomics and memory ordering and how they're combined with basic operating system APIs to build common primitives like mutexes and condition variables. Once you're done, you'll have a firm grasp of how Rust's memory model, the processor, and the role of the operating system all fit together.

With this guide, you'll learn:

- How Rust's type system works exceptionally well for programming concurrency correctly
- All about mutexes, condition variables, atomics, and memory ordering
- What happens in practice with atomic operations on Intel and ARM processors
- How locks are implemented with support from the operating system
- How to write correct code that includes concurrency, atomics, and locks
- How to build your own locking and synchronization primitives correctly

"This book is incredible! It's exactly what I wanted *The Rustonomicon* to cover on concurrency, but far better than I dared dream. Thorough in all the right places. Mara deserves a big rest after this."

—**Aria Beingessner**
Author of *The Rustonomicon*

Mara Bos maintains the Rust standard library and builds real-time control systems in Rust. As team lead of the Rust library team, she knows all the ins and outs of the language and the standard library. In addition, she's been working with concurrent real-time systems for years as founder and CTO at Fusion Engineering. Maintaining the most used library in the Rust ecosystem and working daily on safety critical systems has given her the hands-on experience to both understand the theory and bring it into practice.

RUST / PROGRAMMING LANGUAGES

US $55.99 CAN $69.99
ISBN: 978-1-098-11944-7

9 781098 119447 55599

Twitter: @oreillymedia
linkedin.com/company/oreilly-media
youtube.com/oreillymedia

Rust Atomics and Locks

Low-Level Concurrency in Practice

Mara Bos

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Rust Atomics and Locks
by Mara Bos

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 (*https://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Suzanne McQuade
Development Editor: Shira Evans
Production Editor: Elizabeth Faerm
Copyeditor: Liz Wheeler
Proofreader: Penelope Perkins

Indexer: Ellen Troutman-Zaig
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

December 2022: First Edition

Revision History for the First Edition
2022-12-14: First Release
2023-01-27: Second Release

See *https://oreilly.com/catalog/errata.csp?isbn=9781098119447* for release details.

978-1-098-11944-7

[LSI]

To all the Rust contributors who were waiting for me to review their code while I was busy writing this book.

And to my loved ones, too, of course. ♥

In loving memory of
Amélia Ada Louise, 1994–2021

Table of Contents

Foreword xi

Preface xiii

1. Basics of Rust Concurrency 1
- Threads in Rust 2
- Scoped Threads 5
- Shared Ownership and Reference Counting 7
  - Statics 7
  - Leaking 8
  - Reference Counting 8
- Borrowing and Data Races 11
- Interior Mutability 13
  - Cell 14
  - RefCell 14
  - Mutex and RwLock 15
  - Atomics 15
  - UnsafeCell 16
- Thread Safety: Send and Sync 16
- Locking: Mutexes and RwLocks 18
  - Rust's Mutex 18
  - Lock Poisoning 21
  - Reader-Writer Lock 22
- Waiting: Parking and Condition Variables 24
  - Thread Parking 24
  - Condition Variables 26
- Summary 29

2. Atomics **31**
Atomic Load and Store Operations 32
Example: Stop Flag 32
Example: Progress Reporting 33
Example: Lazy Initialization 35
Fetch-and-Modify Operations 36
Example: Progress Reporting from Multiple Threads 38
Example: Statistics 39
Example: ID Allocation 41
Compare-and-Exchange Operations 42
Example: ID Allocation Without Overflow 44
Example: Lazy One-Time Initialization 45
Summary 47

3. Memory Ordering **49**
Reordering and Optimizations 49
The Memory Model 51
Happens-Before Relationship 51
Spawning and Joining 53
Relaxed Ordering 54
Release and Acquire Ordering 57
Example: Locking 60
Example: Lazy Initialization with Indirection 62
Consume Ordering 65
Sequentially Consistent Ordering 66
Fences 67
Common Misconceptions 71
Summary 73

4. Building Our Own Spin Lock **75**
A Minimal Implementation 75
An Unsafe Spin Lock 78
A Safe Interface Using a Lock Guard 80
Summary 83

5. Building Our Own Channels **85**
A Simple Mutex-Based Channel 85
An Unsafe One-Shot Channel 87
Safety Through Runtime Checks 90
Safety Through Types 94
Borrowing to Avoid Allocation 98

Blocking 101
Summary 104

6. Building Our Own "Arc" 105
Basic Reference Counting 105
Testing It 109
Mutation 110
Weak Pointers 111
Testing It 117
Optimizing 118
Summary 125

7. Understanding the Processor 127
Processor Instructions 128
Load and Store 132
Read-Modify-Write Operations 133
Load-Linked and Store-Conditional Instructions 137
Caching 141
Cache Coherence 142
Impact on Performance 144
Reordering 149
Memory Ordering 150
x86-64: Strongly Ordered 151
ARM64: Weakly Ordered 153
An Experiment 155
Memory Fences 158
Summary 159

8. Operating System Primitives 161
Interfacing with the Kernel 161
POSIX 163
Wrapping in Rust 164
Linux 166
Futex 167
Futex Operations 169
Priority Inheritance Futex Operations 173
macOS 174
os_unfair_lock 175
Windows 175
Heavyweight Kernel Objects 175
Lighter-Weight Objects 176

Address-Based Waiting 177
Summary 179

9. Building Our Own Locks. 181
Mutex 183
Avoiding Syscalls 186
Optimizing Further 188
Benchmarking 191
Condition Variable 193
Avoiding Syscalls 198
Avoiding Spurious Wake-ups 200
Reader-Writer Lock 203
Avoiding Busy-Looping Writers 206
Avoiding Writer Starvation 208
Summary 211

10. Ideas and Inspiration. 213
Semaphore 213
RCU 214
Lock-Free Linked List 215
Queue-Based Locks 217
Parking Lot–Based Locks 218
Sequence Lock 218
Teaching Materials 219

Index. 221

Foreword

This book provides an excellent overview of low-level concurrency in the Rust language, including threads, locks, reference counts, atomics, mailboxes/channels, and much else besides. It digs into issues with CPUs and operating systems, the latter summarizing challenges inherent in making concurrent code work correctly on Linux, macOS, and Windows. I was particularly happy to see that Mara illustrates these topics with working Rust code. It wraps up by discussing semaphores, lock-free linked lists, queued locks, sequence locks, and even RCU.

So what does this book offer someone like myself, who has been slinging C code for almost 40 years, most recently in the nether depths of the Linux kernel?

I first learned of Rust from any number of enthusiasts and Linux-related conferences. Nevertheless, I was happily minding my own business until I was called out by name in a Rust-related LWN article, "Using Rust for Kernel Development". Thus prodded, I wrote a blog series entitled "So You Want to Rust the Linux Kernel?". This blog series sparked a number of spirited discussions, a few of which are visible in the series' comments.

In one such discussion, a long-time Linux-kernel developer who has also written a lot of Rust code told me that when writing concurrent code in Rust, you should write it the way Rust wants you to. I have since learned that although this is great advice, it leaves open the question of exactly what Rust wants. This book gives excellent answers to this question, and is thus valuable both to Rust developers wishing to learn concurrency and to developers of concurrent code in other languages who would like to learn how best to do so in Rust.

I of course fall into this latter category. However, I must confess that many of the spirited discussions about Rust concurrency remind me of my parents' and grandparents' long-ago complaints about the inconvenient safety features that were being added to power tools such as saws and drills. Some of those safety features are now ubiquitous, but hammers, chisels, and chainsaws have not changed all that much. It was not at all easy to work out which mechanical safety features would stand the

test of time, so I recommend approaching software safety features with an attitude of profound humility. And please understand that I am addressing the proponents of such features as well as their detractors.

Which brings us to another group of potential readers, the Rust skeptics. While I do believe that most Rust skeptics are doing the community a valuable service by pointing out opportunities for improvement, all but the most Rust-savvy of skeptics would benefit from reading this book. If nothing else, doing so would enable them to provide sharper and better-targeted criticisms.

Then there are those dyed-in-the-wool non-Rust developers who would prefer to implement Rust's concurrency-related safety mechanisms in their own favorite language. This book will give them a deeper understanding of the Rust mechanisms that they would like to replicate, or, better yet, improve upon.

Finally, any number of Linux-kernel developers are noting the progress that Rust is making toward being included in the Linux kernel; for example, see Jonathan Corbet's article, "Next Steps for Rust in the Kernel". As of October 2022, this is still an experiment, but one that is being taken increasingly seriously. In fact, seriously enough that Linus Torvalds has accepted the first bits of Rust-language support into version 6.1 of the Linux kernel.

Whether you are reading this book to expand your Rust repertoire to include concurrency, to expand your concurrency repertoire to include Rust, to improve your existing non-Rust environment, or just to look at concurrency from a different viewpoint, I wish you the very best on your journey!

— Paul E. McKenney
Meta Platforms Kernel Team
Meta
October 2022

Preface

Rust has played, and keeps playing, a significant role in making systems programming more accessible. However, low-level concurrency topics such as atomics and memory ordering are still often thought of as somewhat mystical subjects that are best left to a very small group of experts.

While working on Rust-based real-time control systems and the Rust standard library over the past few years, I found that many of the available resources on atomics and related topics only cover a small part of the information I was looking for. Many resources focus entirely on C and C++, which can make it hard to form the connection with Rust's concept of (memory and thread) safety and type system. The resources that cover the details of the abstract theory, like C++'s memory model, often only vaguely explain how it relates to actual hardware, if at all. There are many resources that cover every detail of the actual hardware, such as processor instructions and cache coherency, but forming a holistic understanding often requires collecting bits and pieces of information from many different places.

This book is an attempt to put relevant information in one place, connecting it all together, providing everything you need to build your own correct, safe, and ergonomic concurrency primitives, while understanding enough about the underlying hardware and the role of the operating system to be able to make design decisions and basic optimization trade-offs.

Who This Book Is For

The primary audience for this book is Rust developers who want to learn more about low-level concurrency. Additionally, this book can also be suitable for those who are not very familiar with Rust yet, but would like to know what low-level concurrency looks like from a Rust perspective.

It is assumed you know the basics of Rust, have a recent Rust compiler installed, and know how to compile and run Rust code using `cargo`. Rust concepts that are important for concurrency are briefly explained when relevant, so no prior knowledge about Rust concurrency is necessary.

Overview of the Chapters

This book consists of ten chapters. Here's what to expect from each chapter, and what to look forward to:

Chapter 1 — Basics of Rust Concurrency
: This chapter introduces all the tools and concepts we need for basic concurrency in Rust, such as threads, mutexes, thread safety, shared and exclusive references, interior mutability, and so on, which are foundational to the rest of the book.

  For experienced Rust programmers who are familiar with these concepts, this chapter can serve as a quick refresher. For those who know these concepts from other languages but aren't very familiar with Rust yet, this chapter will quickly fill you in on any Rust-specific knowledge you might need for the rest of the book.

Chapter 2 — Atomics
: In the second chapter we'll learn about Rust's atomic types and all their operations. We start with simple load and store operations, and build our way up to more advanced *compare-and-exchange loops*, exploring each new concept with several real-world use cases as usable examples.

  While *memory ordering* is relevant for every atomic operation, that topic is left for the next chapter. This chapter only covers situations where *relaxed* memory ordering suffices, which is the case more often than one might expect.

Chapter 3 — Memory Ordering
: After learning about the various atomic operations and how to use them, the third chapter introduces the most complicated topic of the book: memory ordering.

  We'll explore how the memory model works, what *happens-before relationships* are and how to create them, what all the different memory orderings mean, and why *sequentially consistent ordering* might not be the answer to everything.

Chapter 4 — Building Our Own Spin Lock
: After learning the theory, we put it to practice in the next three chapters by building our own versions of several common concurrency primitives. The first of these chapters is a short one, in which we implement a *spin lock*.

  We'll start with a very minimal version to put *release and acquire memory ordering* to practice, and then we'll explore Rust's concept of *safety* to turn it into an ergonomic and hard-to-misuse Rust data type.

Chapter 5 — Building Our Own Channels
: In Chapter 5, we'll implement from scratch a handful of variations of a *one-shot channel*, a primitive that can be used to send data from one thread to another.

Starting with a very minimal but entirely `unsafe` version, we'll work our way through several ways to design a safe interface, while considering design decisions and their consequences.

Chapter 6 — Building Our Own "Arc"

For the sixth chapter, we'll take on a more challenging memory ordering puzzle. We're going to implement our own version of atomic reference counting from scratch.

After adding support for *weak pointers* and optimizing it for performance, our final version will be practically identical to Rust's standard `std::sync::Arc` type.

Chapter 7 — Understanding the Processor

The seventh chapter is a deep dive into all the low-level details. We'll explore what happens at the processor level, what the *assembly instructions* behind the atomic operations look like on the two most popular processor architectures, what caching is and how it affects the performance of our code, and we'll find out what remains of the memory model at the hardware level.

Chapter 8 — Operating System Primitives

In Chapter 8 we acknowledge that there are things we can't do without the help of the operating system's *kernel* and learn what functionality is available on Linux, macOS, and Windows.

We'll discuss the concurrency primitives that are available through *pthreads* on POSIX systems, find out what we can do with the Windows API, and learn what the Linux *futex syscall* does.

Chapter 9 — Building Our Own Locks

Using what we've learned in the previous chapters, in Chapter 9 we're going to build several implementations of a *mutex*, *condition variable*, and *reader-writer lock* from scratch.

For each of these, we'll start with a minimal but complete version, which we'll then attempt to optimize in various ways. Using some simple benchmark tests, we'll find out that our attempts at optimization don't always increase performance, while we discuss various design trade-offs.

Chapter 10 — Ideas and Inspiration

The final chapter makes sure you don't fall into a void after finishing the book, but are instead left with ideas and inspiration for things to build and explore with your new knowledge and skills, perhaps kicking off an exciting journey further into the depths of low-level concurrency.

Code Examples

All code in this book is written for and tested using Rust 1.66.0, which was released on December 15, 2022. Earlier versions do not include all features used in this book. Later versions, however, should work just fine.

For brevity, the code examples do not include `use` statements, except for the first time a new item from the standard library is introduced. As a convenience, the following prelude can be used to import everything necessary to compile any of the code examples in this book:

```
#[allow(unused)]
use std::{
    cell::{Cell, RefCell, UnsafeCell},
    collections::VecDeque,
    marker::PhantomData,
    mem::{ManuallyDrop, MaybeUninit},
    ops::{Deref, DerefMut},
    ptr::NonNull,
    rc::Rc,
    sync::{*, atomic::{*, Ordering::*}},
    thread::{self, Thread},
};
```

Supplemental material, including complete versions of all code examples, is available at *https://marabos.nl/atomics/*.

You may use all example code offered with this book for any purpose.

Attribution is appreciated, but not required. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Rust Atomics and Locks* by Mara Bos (O'Reilly). Copyright 2023 Mara Bos, 978-1-098-11944-7."

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Used for new terms, URLs, and emphasis.

`Constant width`
: Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, data types, statements, and keywords.

This element signifies a tip or suggestion.

This element signifies a general note.

This element indicates a warning or caution.

Contact Information

O'Reilly has a web page for this book, where errata, examples, and any additional information are listed. It is available at *https://oreil.ly/rust-atomics-and-locks*.

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book. If you wish to reuse content from this book, and you feel your reuse falls outside fair use or the permission given in this Preface, feel free to contact O'Reilly at *permissions@oreilly.com*.

For news and information about O'Reilly, visit *https://oreilly.com*.

Follow O'Reilly on Twitter: *https://twitter.com/oreillymedia*.

Follow the author on Twitter: *https://twitter.com/m_ou_se*.

Acknowledgments

I'd like to thank everyone who had a part in the creation this book. Many people provided support and useful input, which has been incredibly helpful. In particular, I'd like to thank Amanieu d'Antras, Aria Beingessner, Paul McKenney, Carol Nichols, and Miguel Raz Guzmán Macedo for their invaluable and thoughtful feedback on the early drafts. I'd also like to thank everyone at O'Reilly, and in particular my editors, Shira Evans and Zan McQuade, for their inexhaustible enthusiasm and support.

CHAPTER 1
Basics of Rust Concurrency

Long before multi-core processors were commonplace, operating systems allowed for a single computer to run many programs concurrently. This is achieved by rapidly switching between processes, allowing each to repeatedly make a little bit of progress, one by one. Nowadays, virtually all our computers and even our phones and watches have processors with multiple cores, which can truly execute multiple processes in parallel.

Operating systems isolate processes from each other as much as possible, allowing a program to do its thing while completely unaware of what any other processes are doing. For example, a process cannot normally access the memory of another process, or communicate with it in any way, without asking the operating system's kernel first.

However, a program can spawn extra *threads of execution*, as part of the same *process*. Threads within the same process are not isolated from each other. Threads share memory and can interact with each other through that memory.

This chapter will explain how threads are spawned in Rust, and all the basic concepts around them, such as how to safely share data between multiple threads. The concepts explained in this chapter are foundational to the rest of the book.

If you're already familiar with these parts of Rust, feel free to skip ahead. However, before you continue to the next chapters, make sure you have a good understanding of threads, interior mutability, `Send` and `Sync`, and know what a mutex, a condition variable, and thread parking are.

Threads in Rust

Every program starts with exactly one thread: the main thread. This thread will execute your `main` function and can be used to spawn more threads if necessary.

In Rust, new threads are spawned using the `std::thread::spawn` function from the standard library. It takes a single argument: the function the new thread will execute. The thread stops once this function returns.

Let's take a look at an example:

```
use std::thread;

fn main() {
    thread::spawn(f);
    thread::spawn(f);

    println!("Hello from the main thread.");
}

fn f() {
    println!("Hello from another thread!");

    let id = thread::current().id();
    println!("This is my thread id: {id:?}");
}
```

We spawn two threads that will both execute `f` as their main function. Both of these threads will print a message and show their *thread id*, while the main thread will also print its own message.

Thread ID

The Rust standard library assigns every thread a unique identifier. This identifier is accessible through `Thread::id()` and is of the type `ThreadId`. There's not much you can do with a `ThreadId` other than copying it around and checking for equality. There is no guarantee that these IDs will be assigned consecutively, only that they will be different for each thread.

If you run our example program above several times, you might notice the output varies between runs. This is the output I got on my machine during one particular run:

```
Hello from the main thread.
Hello from another thread!
This is my thread id:
```

Surprisingly, part of the output seems to be missing.

What happened here is that the main thread finished executing the `main` function before the newly spawned threads finished executing their functions.

Returning from `main` will exit the entire program, even if other threads are still running.

In this particular example, one of the newly spawned threads had just enough time to get to halfway through the second message, before the program was shut down by the main thread.

If we want to make sure the threads are finished before we return from `main`, we can wait for them by *joining* them. To do so, we have to use the `JoinHandle` returned by the `spawn` function:

```
fn main() {
    let t1 = thread::spawn(f);
    let t2 = thread::spawn(f);

    println!("Hello from the main thread.");

    t1.join().unwrap();
    t2.join().unwrap();
}
```

The `.join()` method waits until the thread has finished executing and returns a `std::thread::Result`. If the thread did not successfully finish its function because it panicked, this will contain the panic message. We could attempt to handle that situation, or just call `.unwrap()` to panic when joining a panicked thread.

Running this version of our program will no longer result in truncated output:

```
Hello from the main thread.
Hello from another thread!
This is my thread id: ThreadId(3)
Hello from another thread!
This is my thread id: ThreadId(2)
```

The only thing that still changes between runs is the order in which the messages are printed:

```
Hello from the main thread.
Hello from another thread!
Hello from another thread!
This is my thread id: ThreadId(2)
This is my thread id: ThreadId(3)
```

Output Locking

The `println` macro uses `std::io::Stdout::lock()` to make sure its output does not get interrupted. A `println!()` expression will wait until any concurrently running one is finished before writing any output. If this was not the case, we could've gotten more interleaved output such as:

```
Hello fromHello from another thread!
 another This is my threthreadHello fromthread id: ThreadId!
( the main thread.
2)This is my thread
id: ThreadId(3)
```

Rather than passing the name of a function to `std::thread::spawn`, as in our example above, it's far more common to pass it a *closure*. This allows us to capture values to move into the new thread:

```
let numbers = vec![1, 2, 3];

thread::spawn(move || {
    for n in &numbers {
        println!("{n}");
    }
}).join().unwrap();
```

Here, ownership of `numbers` is transferred to the newly spawned thread, since we used a `move` closure. If we had not used the `move` keyword, the closure would have captured `numbers` by reference. This would have resulted in a compiler error, since the new thread might outlive that variable.

Since a thread might run until the very end of the program's execution, the `spawn` function has a `'static` lifetime bound on its argument type. In other words, it only accepts functions that may be kept around forever. A closure capturing a local variable by reference may not be kept around forever, since that reference would become invalid the moment the local variable ceases to exist.

Getting a value back out of the thread is done by returning it from the closure. This return value can be obtained from the `Result` returned by the `join` method:

```
let numbers = Vec::from_iter(0..=1000);

let t = thread::spawn(move || {
    let len = numbers.len();
    let sum = numbers.iter().sum::<usize>();
    sum / len  ❶
});

let average = t.join().unwrap(); ❷
```

```
    println!("average: {average}");
```

Here, the value returned by the thread's closure (❶) is sent back to the main thread through the `join` method (❷).

If `numbers` had been empty, the thread would've panicked while trying to divide by zero (❶), and `join` would've returned that panic message instead, causing the main thread to panic too because of `unwrap` (❷).

Thread Builder

The `std::thread::spawn` function is actually just a convenient shorthand for `std::thread::Builder::new().spawn().unwrap()`.

A `std::thread::Builder` allows you to set some settings for the new thread before spawning it. You can use it to configure the stack size for the new thread and to give the new thread a name. The name of a thread is available through `std::thread::current().name()`, will be used in panic messages, and will be visible in monitoring and debugging tools on most platforms.

Additionally, `Builder`'s `spawn` function returns an `std::io::Result`, allowing you to handle situations where spawning a new thread fails. This might happen if the operating system runs out of memory, or if resource limits have been applied to your program. The `std::thread::spawn` function simply panics if it is unable to spawn a new thread.

Scoped Threads

If we know for sure that a spawned thread will definitely not outlive a certain scope, that thread could safely borrow things that do not live forever, such as local variables, as long as they outlive that scope.

The Rust standard library provides the `std::thread::scope` function to spawn such *scoped threads*. It allows us to spawn threads that cannot outlive the scope of the closure we pass to that function, making it possible to safely borrow local variables.

How it works is best shown with an example:

```
let numbers = vec![1, 2, 3];

thread::scope(|s| { ❶
    s.spawn(|| { ❷
        println!("length: {}", numbers.len());
    });
    s.spawn(|| { ❷
        for n in &numbers {
```

```
            println!("{n}");
        }
    });
}); ❸
```

❶ We call the `std::thread::scope` function with a closure. Our closure is directly executed and gets an argument, `s`, representing the scope.

❷ We use `s` to spawn threads. The closures can borrow local variables like `numbers`.

❸ When the scope ends, all threads that haven't been joined yet are automatically joined.

This pattern guarantees that none of the threads spawned in the scope can outlive the scope. Because of that, this scoped `spawn` method does not have a `'static` bound on its argument type, allowing us to reference anything as long as it outlives the scope, such as `numbers`.

In the example above, both of the new threads are concurrently accessing `numbers`. This is fine, because neither of them (nor the main thread) modifies it. If we were to change the first thread to modify `numbers`, as shown below, the compiler wouldn't allow us to spawn another thread that also uses `numbers`:

```
let mut numbers = vec![1, 2, 3];

thread::scope(|s| {
    s.spawn(|| {
        numbers.push(1);
    });
    s.spawn(|| {
        numbers.push(2); // Error!
    });
});
```

The exact error message depends on the version of the Rust compiler, since it's often improved to produce better diagnostics, but attempting to compile the code above will result in something like this:

```
error[E0499]: cannot borrow `numbers` as mutable more than once at a time
 --> example.rs:7:13
  |
4 |     s.spawn(|| {
  |             -- first mutable borrow occurs here
5 |         numbers.push(1);
  |         ------- first borrow occurs due to use of `numbers` in closure
  |
7 |     s.spawn(|| {
  |             ^^ second mutable borrow occurs here
8 |         numbers.push(2);
  |         ------- second borrow occurs due to use of `numbers` in closure
```