Hands-On Concurrency with Rust

Confidently build memory-safe, parallel, and efficient software in Rust

By Brian L. Troutwine

Packt
www.packt.com

Hands-On Concurrency with Rust

Confidently build memory-safe, parallel, and efficient software in Rust

Brian L. Troutwine

BIRMINGHAM - MUMBAI

Hands-On Concurrency with Rust

Commissioning Editor: Aaron Lazar
Acquisition Editor: Chaitanya Nair
Content Development Editor: Akshada Iyer
Technical Editor: Mehul Singh
Copy Editor: Safis Editing
Project Coordinator: Prajakta Naik
Proofreader: Safis Editing
Indexer: Mariammal Chettiyar
Graphics: Jisha Chirayil
Production Coordinator: Aparna Bhagat

First published: May 2018

Production reference: 1290518

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78839-997-5

www.packtpub.com

For my parents, who taught me how to be.
For Hope, with whom life is more joyful than I ever imagined.
For Oliver, who was there at the start. He saw what there was to see.

mapt.io

Mapt is an online digital library that gives you full access to over 5,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

PacktPub.com

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the author

Brian L. Troutwine is a software engineer with an interest in low-latency and high-scale software. He has worked at Rackspace Hosting, AdRoll, and Postmates. As his first book publishes, he will be starting at Unity Technologies.

He has delivered the following talks:

- *The Charming Genius of the Apollo Guidance Computer*
- *Getting Uphill on a Candle: Crushed Spines, Detached Retinas, and One Small Step*
- *10 Billion a Day, 100 Milliseconds Per: Monitoring Real-Time Bidding at AdRoll*
- *Fetching Moths from the Works: Correctness Methods in Software*
- *Build Good Software: Of Politics and Method*

I'd like to thank Tom Santero for reading early drafts of this book. I'd like also to thank Miriam Pena, Sonny Scroggin, Irina Guberman, and Brian Mitchell for talking through what the book was and helping me find, thereby, what it could be. Special thanks to Chris Markiewicz for being an excellent roommate in high school.

What this book covers

The material that this book covers is very broad, and it attempts to go into that material at some depth. The material is written so that you can work straight through, but it's also expected that some readers will only be interested in a subset of the content.

`Chapter 1`, *Preliminaries – Machine Architecture and Getting Started with Rust*, discusses modern CPU architectures, focusing specifically on x86 and ARM. These two architectures will be the focus of the book. The reader is assumed to be familiar with Rust, but we will also discuss verifying that your installation works as expected.

`Chapter 2`, *Sequential Rust Performance and Testing*, introduces inspecting the performance of a Rust program. The details of the underlying computer hardware are especially important in this: cache interactions, memory layout, and exploiting the nature of the problem domain are key to writing fast programs. However, *fast* doesn't matter if the results are inaccurate. This chapter also focuses on testing in Rust.

`Chapter 3`, *The Rust Memory Model – Ownership, References and Manipulation*, discusses the memory model of Rust, focusing specifically on what makes Rust *memory safe*, how the language is constrained to achieve such safety and how these constraints influence the fundamental types' implementations. The reader will understand the borrow checker and its ways at the close of this chapter.

`Chapter 4`, *Sync and Send – the Foundation of Rust Concurrency*, is the first in which notions of concurrency make their appearance. The

chapter discusses the `Sync` and `Send` traits, both why they exist and their implications. The chapter closes with a concrete demonstration of a multithreaded Rust program. Not the last, either.

`Chapter 5`, *Locks – Mutex, Condvar, Barriers and RWLock*, introduces the *coarse* synchronization methods available to the Rust programmer. Each is examined in turn and demonstrated in context of an industrial Rust project, hopper. The *coarse* synchronization methods are elaborated on in more detail in a series of smaller projects and data structures.

`Chapter 6`, *Atomics – the Primitives of Synchronization*, introduces *fine* synchronization in terms of atomic primitives available on all modern CPUs. This is an exceedingly difficult topic, and a deep investigation into atomic programming and its methods is carried out. The chapter lock-free, atomic data structures, and production-grade codebases. The reader will construct many of the *coarse* synchronization mechanisms seen in `Chapter 5`, *Locks – Mutex, Condvar, Barriers and RWLock*.

`Chapter 7`, *Atomics – Safely Reclaiming Memory*, discusses at length one of the key difficulties of atomic programming in any language—safely deallocating memory. The main three methods—reference counting, hazard pointers, epoch-based reclamation—are each discussed in great detail, and production-worthy codebases are investigated. Crossbeam, especially, is discussed in great detail.

`Chapter 8`, *High-Level Parallelism – Threadpools, Parallel Iterators and Processes*, motivates and explains the implementation of thread pooling. Having this knowledge in hand, the Rayon project is investigated and subsequently used in a complex project that benefits greatly from simple data parallelism.

`Chapter 9`, *FFI and Embedding – Combining Rust and Other Languages*, extends the final project of `Chapter 8`, *High-Level*

Parallelism – Threadpools, Parallel Iterators, and Processes by embedding C code into it. The rlua project, a convenient library to extend Rust programs with lua programs, is discussed. The chapter closes by compiling Rust for embedding into C, Python, and Erlang projects.

`Chapter 10`, *Futurism – Near-Term Rust*, closes the book with a discussion of the near-term changes to the language that are apropos to parallel programmers, as well as a few miscellaneous remarks.

About the reviewer

Daniel Durante is an avid coffee drinker/roaster, motorcyclist, archer, welder, and carpenter whenever he isn't programming. From the age of 12 years, he has been involved with web and embedded programming with PHP, Node.js, Golang, Rust, and C.
He has worked on text-based browser games that have reached over million active players, created bin-packing software for CNC machines, embedded programming with cortex-m and PIC circuits, high-frequency trading applications, and helped contribute to one of the oldest ORMs of Node.js (SequelizeJS).

He also has reviewed the following books for Packt:

- *PostgreSQL Developer's Guide*
- *PostgreSQL 9.0 High*
- *Node Cookbook*

I would like to thank my parents, my brother, and friends who've all put up with my insanity sitting in front of a computer day in and day out. I would not be here today if it wasn't for their patience, guidance, and love.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit `authors.packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface

Welcome. The aim of this book is to teach beginner and moderate Rust programmers how to exploit modern parallel machines in the Rust programming language. This book will contain a variety of information relevant specifically to the Rust programming language, especially with regard to its standard library, but it will also contain information that is more generally applicable but happens to be expressed in Rust. Rust itself is *not* a terribly inventive language. Its key contribution, from where I sit, is the mainstreaming of affine types with application to memory allocation tracking. In most other respects, it is a familiar systems programming language, one that ought to feel familiar—with a bit of adjustment—to those with a background in GC-less programming languages. This is a good thing, considering our aim here is to investigate concurrency—there is a wealth of information available in the papers and books written about this subject, and we understand and apply their concepts. This book will reference a number of such works, whose contexts are C++ , ATS, ADA, and similar languages.

Who this book is for

If this is your first Rust programming book, I warmly thank you for your enthusiasm, but encourage you to seek out a suitable introduction to the programming language. This book will hit the ground running, and it will likely not be appropriate if you've got questions about the basics. The Rust community has produced *excellent* documentation, including introductory texts. The Book (`https://doc.rust-lang.org/book/first-edition/`), first edition, is how many who are already in the community learned the language. The second edition of the book, still in progress at the time of writing, looks to be an improvement over the original text, and it is also recommended. There are many other excellent introductions widely available for purchase, as well.

To get the most out of this book

This book covers a deep topic in a relatively short space. Throughout the text, the reader is expected to be comfortable with the Rust programming language, have access to a Rust compiler and a computer on which to compile, and execute Rust programs. What additional software appears in this book is covered in `Chapter 1`, *Preliminaries – Machine Architecture and Getting Started with Rust*, and it is recommended for use but not mandatory.

The basic premise of this book is as follows—parallel programming is difficult but not impossible. Parallel programming can be done, and done well, when attention is paid to the computing environment and care is put into the validation of the produced program. To that end, each chapter has been written with an eye toward imparting a solid foundation to the reader of the chapter's subject. Once a chapter is digested, the reader will, hopefully, have a solid path into the existing body of literature on that subject. In that, this is a book of beginnings, not ends.

I strongly encourage the reader to take an active role in reading this book, download the source code, investigate the projects as you read through independent of the book's take, and probe the running programs with the tools available on your operating system. Like anything else, parallel programming ability is a skill that is acquired and honed by practice.

One final note—allow the process of learning to proceed at its own pace. If one chapter's subject doesn't immediately settle in your mind, that's okay. For me, the process of writing the book was a confluence

of flashes of insight and knowledge that unfolded slowly like a flower. I imagine that the process of reading the book will proceed analogously.

Download the example code files

The code bundle for the book is hosted on GitHub at `https://github.com/PacktPublishing/Hands-On-Concurrency-with-Rust`. In case there's an update to the code, it will be updated on the existing GitHub repository.

You can also download the example code files for this book from your account at `www.packtpub.com`. If you purchased this book elsewhere, you can visit `www.packtpub.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at `www.packtpub.com`.
2. Select the SUPPORT tab.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "The seeds are for `XorShiftRng`, move to previous line `max_in_memory_bytes` and `max_disk_bytes` are for hopper."

A block of code is set as follows:

```
fn main() {
    println!("Apollo is the name of a space program but also my dog.");
}
```

Any command-line input or output is written as follows:

```
> cat hello.rs
fn main() {
 println!("Apollo is the name of a space program but also my dog.");
}
> rustc -C opt-level=2 hello.rs
> ./hello
Apollo is the name of a space program but also my dog.
```

Bold: Indicates a new term, an important word, or words that you see onscreen.

> *Warnings or important notes appear like this.*
>
> *Tips and tricks appear like this.*

Get in touch

Feedback is always welcome.

General feedback: Email `feedback@packtpub.com` and mention the book title in the subject of your message. If you have questions about any aspect of this book, please email us at `questions@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packtpub.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Reviews

Please do leave a review for the book on the site that you purchased it from. Reviews help us at Packt understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packtpub.com`.

Preliminaries – Machine Architecture and Getting Started with Rust

In this chapter, we'll work through the preliminaries of the book, making sure to cover the basics necessary to frame the remainder of the book. This book is about parallel programming in the Rust programing language. It's essential, then, to understand modern computing hardware at a high level. We'll need a sense of how a modern CPU operates, how memory buses influence the CPU's ability to perform work and what it means for a computer to be able to do multiple things at once. In addition, we'll discuss validating your Rust installation, and cover generating executables for the two CPU architectures that this book will be concerned with.

By the close of this chapter, we will have:

- Discussed a high-level model of CPU operations
- Discussed a high-level model of computer memory
- Had a preliminary discussion of the Rust memory model
- Investigated generating runnable Rust programs for x86 and ARM
- Investigated debugging these programs

Technical requirements

This chapter requires any kind of modern computer on which Rust can be installed. The exact details are covered in depth below. The interested reader might choose to invest in an ARM development board. I have used a Raspberry Pi 3 Model B while writing. The Valgrind suite of tools are used below. Many operating systems bundle Valgrind packages, but you can find further installation instructions for your system at `valgrind.org`. The `gdb` and `lldb` tools are used, often installed along with the gcc and llvm compiler toolchains.

You can find the source code for this book's projects on GitHub: `https://github.com/PacktPublishing/Hands-On-Concurrency-with-Rust`. This chapter has no relevant source code.

The machine

In this book, independent of the specific Rust techniques, we'll attempt to teach a kind of *mechanical sympathy* with the modern parallel computer. There are two model kinds of parallelism we'll touch on—concurrent memory operations and data parallelism. We'll spend most of our time in this book on concurrent memory operations, the kind of parallelism in which multiple CPUs contend to manipulate a shared, addressable memory. Data parallelism, where the CPU is able to operate with a single or multiple instructions on multiple words at a time concurrently, will be touched on, but the details are CPU specific, and the necessary intrinsics are only now becoming available in the base language as this book goes to press. Fortunately, Rust, as a systems language with modern library management, will easily allow us to pull in an appropriate library and emit the correct instructions, or we could inline the assembly ourselves.

Literature on the abstract construction of algorithms for parallel machines must choose a machine model to operate under. The parallel random access machine (PRAM) is common in the literature. In this book, we will focus on two concrete machine architectures:

- x86
- ARM

These machines were chosen because they are common and because they each have specific properties that will be important when we get to `Chapter 6`, *Atomics – the Primitives of Synchronization*. Actual machines deviate from the the PRAM model in important ways. Most

obviously, actual machines have a limited number of CPUs and a bounded amount of RAM. Memory locations are not uniformly accessible from each CPU; in fact, cache hierarchies have a significant impact on the performance of computer programs. None of this is to say that PRAM is an absurd simplification, nor is this true of any other model you'll find in literature. What should be understood is that as we work, we'll need to draw lines of abstraction out of necessity, where further detail does not improve our ability to solve problems well. We also have to understand how our abstractions, suited to our own work, relate to the abstractions of others so that we can learn and share. In this book, we will concern ourselves with empirical methods for understanding our machines, involving careful measurement, examination of assembly code, and experimentation with alternative implementations. This will be combined with an abstract model of our machines, more specific to today's machines than PRAM, but still, in the details, focusing on total cache layers, cache sizes, bus speeds, microcode versions, and so forth. The reader is encouraged to add more specificity should the need arise and should they feel so emboldened.

The CPU

The CPU is a device that interprets a stream of instructions, manipulating storage and other devices connected to it in the process. The simplest model of a CPU, and the one that is often introduced first in undergrad computer science, is that a CPU receives an instruction from some nebulous place, performs the interpretation, receives the next instruction, interprets that, and so forth. The CPU maintains an internal pace via an oscillator circuit, and all instructions take a defined number of oscillator pulses—or *clock cycles*—to execute. In some CPU models, every instruction will take the same number of clock cycles to execute, and in others the cycle count of instructions will vary. Some CPU instructions modify registers, have very low latency with specialized but exceedingly finite memory locations built into the CPU. Other CPU instructions modify the main memory, RAM. Other instructions move or copy information between registers and RAM or vice versa. RAM—whose read/write latency is much higher than that of registers but is much more plentiful—and other storage devices, such as SSDs, are connected to the CPU by specialized buses.

The exact nature of these buses, their bandwidth and transmission latency, varies between machine architectures. On some systems, every location in the RAM is addressable—meaning that it can be read or written to—in constant time from the available CPUs. In other systems, this is not the case—some RAM is CPU-local and some is CPU-remote. Some instructions control special hardware interrupts that cause memory ranges to be written to other bus-connected storage devices. Mostly, these devices are exceedingly slow, compared to the RAM, which is itself slow compared to the registers.

All of this is to explain that in the simplest model of a CPU, where instructions are executed serially, instructions may well end up stalling for many clock cycles while waiting for reads or writes to be executed. To that end, it's important to understand that almost all CPUs—and especially the CPUs we'll concern ourselves with in this book—perform out-of-order executions of their instructions. Just so long as a CPU can prove that two sequences of instructions access the memory distinctly—that is, they do not interfere with each other—then the CPU is free and will probably reorder instructions. This is one of the things that makes C's undefined behavior concerning uninitialized memory so interesting. Perhaps your program's future has already filled in the memory, or perhaps not. Out-of-order execution makes reasoning about a processor's behavior difficult, but its benefit is that CPUs can execute programs much faster by deferring a sequence of instructions that is stalled on some kind of memory access.

In the same spirit, most modern CPUs—and especially the CPUs we'll concern ourselves with in this book—perform *branch prediction*. Say that branches in our programs tend to branch the same way at execution time—for example, say we have a feature-flag test that is configured to be enabled for the lifetime of a program. CPUs that perform branch prediction will *speculatively execute* one side of a branch that tends to branch in a certain way while they wait on other stalled instruction pipelines' instructions. When the branch instruction sequence catches up to its branch test, and if the test goes the predicted way, there's already been a great deal of work done and the instruction sequence can skip well ahead. Unfortunately, if the branch was mispredicted, then all this prior work must be torn down and thrown away, and the correct branch will have to be computed, which is quite expensive. It's for this reason that you'll find that programmers who worry about the nitty-gritty performance characteristics of their programs will tend to fret about branches and try to remove them.

All of this is quite power-hungry, reordering instructions to avoid stalls or racing ahead to perform computations that may be thrown away. Power-hungry implies hot, which implies cooling, which implies more power expenditure. All of which is not necessarily great for the sustainability of technological civilization, depending on how the electricity for all this is generated and where the waste heat is dumped. To that end, many modern CPUs integrate some kind of power-scaling features. Some CPUs will lengthen the time between their clock pulses, meaning they execute fewer instructions in a certain span of time than they might otherwise have. Other CPUs race ahead as fast as they normally would and then shut themselves off for a spell, drawing minimal electricity and cooling in the meantime. The exact method by which to build and run a power-efficient CPU is well beyond the scope of this book. What's important to understand is that, as a result of all this, your program's execution speed will vary from run to run, all other things being equal, as the CPU decides whether or not it's time to save power. We'll see this in the next chapter when we manually set power-saving settings.

Memory and caches

The memory storage of a CPU, alluded to in the last section, is fairly minimal. It is limited to the handful of words in general-purpose registers, plus the special-purpose registers in some limited cases. Registers are very fast, owing to their construction and on-die location, but they are *not* suited for storage. Modern machines connect CPUs over a bus or buses to the *main memory*, a very large block of randomly addressable bytes. This random addressability is important as it means that, unlike other kinds of storage, the cost to retrieve the 0^{th} byte from RAM is not distinct from retrieving the $1{,}000{,}000{,}000^{th}$ byte. We programmers don't have to do any goofy trickery to ensure that our structures appear in the front of the RAM in order to be faster to retrieve or modify, whereas physical location in storage is a pressing concern for spinning hard disks and tape drives. Exactly how our CPUs interact with the memory varies between platforms, and the discussion that follows is heavily indebted to Mark Batty's description in his 2014 book, *The C11 and C++11 Concurrency Model*.

In a machine that exposes a sequentially consistent model of memory access, every memory load or store must be made in lockstep with one another, including in systems with multiple CPUs or multiple threads of execution per CPU. This limits important optimizations—consider the challenge of working around memory stalls in a sequentially consistent model—and so neither of the processors we'll be considering in this book offer this model. It *will* show up in literature by name because of the ease of reasoning about this model, and is worth knowing about, especially when studying atomics literature.

The x86 platform behaves as if it were sequentially consistent, excepting that every thread of execution maintains a FIFO buffer for writes prior to their being flushed to the main memory. In addition, there is a global lock for the coordination of atomic reads and writes between threads of execution. There's a lot to unpack here. Firstly, I use the words *load* and *store* interchangeably with *read* and *write*, as does most literature. There is also a variant distinction between plain load/store and *atomic* load/store. Atomic loads/stores are special in that their effects can be coordinated between threads of execution, allowing for coordination with varying degrees of guarantees. The x86 processor platform provides fence instructions that force the flush of write buffers, stalling other threads of execution attempting to access the written range of main memory until the flush is completed. This is the purpose of the global lock. Without atomicity, writes will be flushed willy-nilly. On x86 platforms, writes to an addressable location in the memory are coherent, meaning they are globally ordered, and reads will see values from that location in the order of the writes. Compared to ARM, the way this works on x86 is very simple—writes happen directly to main memory.

Let's look at an example. Taking inspiration from Batty's *excellent* dissertation, consider a setup where a parent thread sets two variables, *x* and *y*, to 0, then spawns two threads called, say, `A` and `B`. Thread `A` is responsible for setting *x* and then *y* to 1, whereas thread `B` is responsible for loading the value of `x` into a thread-local variable called `thr_x` and `y` into a thread-local variable called `thr_y`. This looks something like the following:

```
WRITE x := 0
WRITE y := 0
[A] WRITE x := 1
[A] WRITE y := 1
FLUSH A
[B] READ x
[B] WRITE thr_x := x
[B] READ y
[B] WRITE thr_y := y
```

In this specific example, `thr_x == 1` and `thr_y == 1`. Had the flushes been ordered differently by the CPU, this outcome would have been different. For instance, look at the following:

```
WRITE x := 0
WRITE y := 0
[A] WRITE x := 1
[A] WRITE y := 1
[B] READ x
[B] WRITE thr_x := x
FLUSH A
[B] READ y
[B] WRITE thr_y := y
```

The consequence of this is that `thr_x == 0` and `thr_y == 1`. Without any other coordination, the only other valid interleaving is `thr_x == 0` and `thr_y == 0`. That is, as a result of the write buffer on x86, the write of *x* in thread A can never be reordered to occur after the write of `y: thr_x == 1` and `thr_y == 0`. This kind of stinks, unless you enjoy this little program as a parlor trick. We want determinism out of our programs. To that end, x86 provides different fence and lock instructions that control how and when threads flush their local write buffers, and how and when threads may read byte ranges from the main memory. The exact interplay here is… complicated. We'll come back to it in great detail in `Chapter 3`, *The Rust Memory Model – Ownership, References, and Manipulation* and `Chapter 6`, *Atomics – the Primitives of Synchronization*. Suffice it to say that, for now, there's an `SFENCE` instruction available that forces a sequential consistency. We can employ this instruction as follows:

```
WRITE x := 0
WRITE y := 0
[A] WRITE x := 1
[A] WRITE y := 1
[A] SFENCE        [B] SFENCE
                  [B] READ x
                  [B] WRITE thr_x := x
                  [B] READ y
                  [B] WRITE thr_y := y
```

From this, we get `thr_x == 1` and `thr_y == 1`. The ARM processor is a little different—1/0 is a valid interleaving. There is no global lock to the ARM, no fences, and no write buffer. In ARM, a string of instructions—called a propagation list, for reasons that will soon be clear—is maintained in an *uncommitted* state, meaning that the thread that originated the instructions will see them as *in-flight*, but they will not have been propagated to other threads. These *uncommitted* instructions may have been performed—resulting in side-effects in the memory—or not, allowing for speculative execution and the other performance tricks discussed in the previous section. Specifically, reads may be satisfied from a thread's local propagation list, but not writes. Branch instructions cause the propagation list that led to the branch instruction to be committed, potentially out of the order from that specified by the programmer. The memory subsystem keeps track of which propagation list has been sent to which thread, meaning that it is possible for a thread's private loads and stores to be out of order and for committed loads and stores to appear out of order between threads. Coherency is maintained on ARM with more active participation from the memory subsystem.

Whereas on x86, the main memory is something that has actions done to it, on ARM, the memory subsystem responds to requests and may invalidate previous, uncommitted requests. A write request involves a read-response event, by which it is uniquely identified, and a read request must reference a write.

This sets up a *data-dependency* chain. Responses to reads may be invalidated at a later time, but not writes. Each location receives a *coherence-commitment* ordering, which records a global order of writes, built up per-thread as propagation lists are propagated to threads.

This is very complicated. The end result is that writes to a thread's view of the memory may be done out of programmer order, writes

committed to main memory may also be done out of order, and reads can be requested out of programmer order. Therefore, the following code is perfectly valid, owing to the lack of branches in our example:

```
WRITE x := 0
WRITE y := 0
[A] WRITE x := 1
[B] READ x
[A] WRITE y := 1
[B] WRITE thr_y := y
[B] WRITE thr_x := x
[B] READ y
```

ARM provides instructions for control of dependencies, called barriers. There are three types of dependency in ARM. The address dependency means a load is used to compute the address for access to the memory. The control dependency means that the program flow that leads to memory access depends on a load. We're already familiar with the data dependency.

While there is a lot of main memory available, it's not particularly fast. Well, let's clarify that. Fast here is a relative term. A photon in a vacuum will go about 30.5 centimeters in 1 nanosecond, meaning that in ideal circumstances, I, on the west coast of the United States, ought to be able to send a message to the east coast of the United States and receive a response in about 80 milliseconds. Of course, I'm ignoring request-processing time, the realities of the internet, and other factors; we're just dealing with ballpark figures, here. Consider that 80 milliseconds is 80,000,000 nanoseconds. A read access from a CPU to the main memory is around 100 nanoseconds, give or take your specific computer architecture, the details of your chips, and other factors. All this is to clarify that, when we say, not particularly fast, we're working outside normal human time scales.

It is difficult, sometimes, to keep from misjudging things as fast enough. Say, we have a 4 GHz processor. How many clock cycles do we get per nanosecond? Turns out, it's 4. Say, we have an instruction

that needs to access main memory—which, remember, takes 100 nanoseconds—and happens to be able to do its work in exactly 4 cycles, or one nanosecond. That instruction will then be stalled for 99 nanoseconds while it waits, meaning we're potentially losing out on 99 instructions that could have been executed by the CPU. The CPU will make up some of that loss with its optimization tricks. These only go so far, unless our computation is very, very lucky.

In an effort to avoid the performance impact of the main memory on computation, processor manufacturers introduced *caching* between the processor and the main memory. Many machines these days have three layers of *data cache*, as well as an *instruction cache*, called dCACHE and iCACHE, which are tools we'll be using later. You will see dCACHE often consists of three layers these days, each layer being successively larger than the last, but also slower for cost or power concerns. The lowest, smallest layer is called L1, the next L2, and so forth. CPUs read into caches from the main memory in working blocks—or simply blocks—that are the size of the cache being read into. Memory access will preferentially be done on the L1 cache, then L2, then L3, and so forth, with time penalties at each level for missing and block reads. Cache hits, however, are significantly faster than going directly to the main memory. L1 dCACHE references are 0.5 nanoseconds, or fast enough that our hypothetical 4-cycle instruction is 8 *times* slower than the memory access it requires. L2 dCACHE references are 7 nanoseconds, still a fair sight better than the main memory. Of course, the exact numbers will vary from system to system, and we'll do quite a bit in the next chapter to measure them directly. Cache coherency—maintaining a high ratio of hits to misses—is a significant component of building fast software on modern machines. We'll never get away from the CPU cache in this book.

Memory model

The details of a processor's handling of memory is both complicated and very specific to that processor. Programming languages—and Rust is no exception here—invent a *memory model* to paper over the details of all supported processors while, ideally, leaving the programmer enough freedom to exploit the specifics of each processor. Systems languages also tend to allow absolute freedom in the form of escape hatches from the language memory model, which is exactly what Rust has in terms of *unsafe*.

With regard to its memory model, Rust is very much inspired by C++. The atomic orderings exposed in Rust are those of LLVM's, which are those of C++11. This is a fine thing—any literature to do with either C++ or LLVM will be immediately applicable to Rust. Memory order is a complex topic, and it's often quite helpful to lean on material written for C++ when learning. This is *especially* important when studying up on lock-free/wait-free structures—which we'll see later in this book—as literature on those topics often deals with it in terms of C++. Literature written with C11 in mind is also suitable, if maybe a little less straightforward to translate.

Now, this is not to say that the C++ programmer will be immediately comfortable in concurrent Rust. The digs will be familiar, but not quite right. This is because Rust's memory model also includes a notion of reference consumption. In Rust, a thing in memory must be used zero or one times, but no more. This, incidentally, is an application of a version of linear-type theory called *affine typing*, if you'd like to read up more on the subject. Now, the consequence of restricting memory access in this way is that Rust is able to guarantee at compile-time safe memory access—threads cannot reference the

same location in memory at the same time without coordination; out-of-order access in the same thread are not allowed, and so forth. Rust code is *memory safe* without relying on a garbage collector. In this book's estimation, memory safety is a good win, even though the restrictions that are introduced do complicate implementing certain kinds of structures that are more straightforward to build in C++ or similar languages.

This is a topic we'll cover in much greater detail in `Chapter 3`, *The Rust Memory Model – Ownership, References and Manipulation*.

Getting set up

Now that we have a handle on the machines, this book will deal with what we need to get synchronized on our Rust compiler. There are three channels of Rust to choose from—stable, beta, and nightly. Rust operates on a six-week release cycle. Every day the nightly channel is rolled over, containing all the new patches that have landed on the master branch since the day before. The nightly channel is special, compared to beta and stable, in that it is the only version of the compiler where nightly features are able to be compiled. Rust is very serious about backward compatibility. Proposed changes to the language are baked in nightly, debated by the community, and lived with for some time before they're promoted out of nightly only status and into the language proper. The beta channel is rolled over from the current nightly channel every six weeks. The stable channel is rolled over from beta at the same time, every six weeks.

This means that a new stable version of the compiler is at most only ever six weeks old. Which channel you choose to work with is up to you and your organization. Most teams I'm aware of work with nightly and ship stable, as many important tools in the ecosystem—such as clippy and rustfmt—are only available with nightly features, but the stable channel offers, well, a stable development target. You'll find that many libraries in the crate ecosystem work to stay on stable for this reason.

Unless otherwise noted, we'll focus on the stable channel in this book. We'll need two targets installed—one for our x86 machine and the other for our ARMv7. Your operating system may package Rust for you—kudos!—but the community tends to recommend the use of `rustup`, especially when managing multiple, version-pinned targets.

If you're unfamiliar with `rustup`, you can find a persuasive explanation of and instructions for its use at `rustup.rs`. If you're installing a Rust target for use on a machine on which `rustup` is run, it will do the necessary triplet detections. Assuming that your machine has an x86 chip in it and is running Linux, then the following two commands will have equivalent results:

```
> rustup install stable

> rustup target add x86_64-unknown-linux-gnu
```

Both will instruct `rustup` to track the target `x86_64-unknown-linux-gnu` and install the stable channel version of Rust. If you're running OS X or Windows, a slightly different triplet will be installed by the first variant, but it's the chip that really matters. The second target, we'll need to be more precise with:

```
> rustup target add stable-armv7-unknown-linux-gnueabihf
```

Now you have the ability to generate x86 binaries for your development machine, for x86 (which is probably the same thing), and for an ARMv7 running Linux, the readily available RaspberryPi 3. If you intend to generate executables for the ARMv7—which is recommended, if you have or can obtain the chip—then you'll also need to install an appropriate cross-compiler to link. On a Debian-based development system, you can run the following:

```
> apt-get install gcc-arm-linux-gnueabihf
```

Instructions will vary by operating system, and, honestly, this can be the trickiest part of getting a cross-compiling project set up. Don't despair. If push comes to shove, you could always compile on host. Compilation will just be pokey. The final setup step before we get into the interesting part is to tell cargo how to link our ARMv7 target. Now, please be aware that cross-compilation is an active area of work

in the Rust community at the time of writing. The following configuration file fiddling may have changed a little between this book being published and your reading of it. Apologies. The Rust community documentation will surely help patch up some of the differences. Anyway, add the following—or similar, depending on your operating system—to `~/.cargo/config`:

```
[target.armv7-unknown-linux-gnueabihf]
linker = "arm-linux-gnueabihf-gcc"
```

If `~/.cargo/config` doesn't exist, go ahead and create it with the preceding contents.

The interesting part

Let's create a default cargo project and confirm that we can emit the appropriate assembly for our machines. Doing this will be one of the important pillars of this book. Now, choose a directory on disk to place the default project and navigate there. This example will use `~/projects`, but the exact path doesn't matter. Then, generate the default project:

```
~projects > cargo new --bin hello_world
     Created binary (application) `hello_world` project
```

Go ahead and reward yourself with some x86 assembler:

```
hello_world > RUSTFLAGS="--emit asm" cargo build --target=x86_64-
unknown-linux-gnu
   Compiling hello_world v0.1.0 (file:///home/blt/projects/hello_world)
       Finished dev [unoptimized + debuginfo] target(s) in 0.33 secs
hello_world > file target/x86_64-unknown-linux-
gnu/debug/deps/hello_world-6000abe15b385411.s
target/x86_64-unknown-linux-gnu/debug/deps/hello_world-
6000abe15b385411.s: assembler source, ASCII text
```

Please be aware that if your compilation of a Rust binary on OS X x86 will not run on Linux x86, and vice versa. This is due to the differences in the interfaces of the operating systems themselves. You're better off compiling on your x86 Linux machine or your x86 OS X machine and running the binaries there. That's the approach I take with the material presented in this book.

That said, reward yourself with some ARMv7 assembler:

```
hello_world > RUSTFLAGS="--emit asm" cargo build --target=armv7-
unknown-linux-gnueabihf
   Compiling hello_world v0.1.0 (file:///home/blt/projects/hello_world)
       Finished dev [unoptimized + debuginfo] target(s) in 0.45 secs
hello_world > file target/armv7-unknown-linux-
gnueabihf/debug/deps/hello_world-6000abe15b385411.s
target/armv7-unknown-linux-gnueabihf/debug/deps/hello_world-
6000abe15b385411.s: assembler source, ASCII text
```

Of course, if you want to build release versions, you need only to give `cargo` the `--release` flag:

```
hello_world > RUSTFLAGS="--emit asm" cargo build --target=armv7-
unknown-linux-gnueabihf --release
Compiling hello_world v0.1.0 (file:///home/blt/projects/hello_world)
Finished release [optimized] target(s) in 0.45 secs
hello_world > wc -l target/armv7-unknown-linux-gnueabihf/
debug/   release/
hello_world > wc -l target/armv7-unknown-linux-
gnueabihf/*/deps/hello_world*.s
 1448 target/armv7-unknown-linux-gnueabihf/debug/deps/hello_world-
6000abe15b385411.s
  101 target/armv7-unknown-linux-gnueabihf/release/deps/hello_world-
dd65a12bd347f015.s
 1549 total
```

It's interesting—and instructive!—to compare the differences in the generated code. Notably, the release compilation will strip debugging entirely. Speaking of which, let's talk debuggers.

Debugging Rust programs

Depending on your language background, the debugging situation in Rust may be very familiar and comfortable, or it might strike you as a touch bare bones. Rust relies on the commonly used debugging tools that other programming languages have to hand—`gdb` or `lldb`. Both will work, though historically, `lldb` has had some issues, and it's only since about mid-2016 that either tool has supported unmangled Rust. Let's try `gdb` on `hello_world` from the previous section:

```
hello_world > gdb target/x86_64-unknown-linux-gnu/debug/hello_world
GNU gdb (Debian 7.12-6) 7.12.0.20161007-git
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show
copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from target/x86_64-unknown-linux-
gnu/debug/hello_world...done.
warning: Missing auto-load script at offset 0 in section
.debug_gdb_scripts
of file /home/blt/projects/hello_world/target/x86_64-unknown-linux-
gnu/debug/hello_world.
Use `info auto-load python-scripts [REGEXP]' to list them.
(gdb) run
Starting program: /home/blt/projects/hello_world/target/x86_64-unknown-
linux-gnu/debug/hello_world
[Thread debugging using libthread_db enabled]
```

```
Using host libthread_db library "/lib/x86_64-linux-
gnu/libthread_db.so.1".
Hello, world!
[Inferior 1 (process 15973) exited normally]
```

Let's also try `lldb`:

```
hello_world > lldb --version
lldb version 3.9.1 ( revision )
hello_world > lldb target/x86_64-unknown-linux-gnu/debug/hello_world
(lldb) target create "target/x86_64-unknown-linux-
gnu/debug/hello_world"
Current executable set to 'target/x86_64-unknown-linux-
gnu/debug/hello_world' (x86_64).
(lldb) process launch
Process 16000 launched: '/home/blt/projects/hello_world/target/x86_64-
unknown-linux-gnu/debug/hello_world' (x86_64)
Hello, world!
Process 16000 exited with status = 0 (0x00000000)
```

Either debugger is viable, and you're warmly encouraged to choose the one that suits your debugging style. This book will lean toward the use of `lldb` because of vague authorial preference.

The other suite of tooling you'll commonly see in Rust development—and elsewhere in this book—is `valgrind`. Rust being memory safe, you might wonder when `valgrind` would find use. Well, whenever you use `unsafe`. The `unsafe` keyword in Rust is fairly uncommon in day-to-day code, but does appear when squeezing out extra percentage points from hot code paths now and again. Note that `unsafe` blocks will absolutely appear in this book. If we run `valgrind` on `hello_world`, we'll get no leaks, as expected:

```
hello_world > valgrind --tool=memcheck target/x86_64-unknown-linux-
gnu/debug/hello_world
==16462== Memcheck, a memory error detector
==16462== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et
al.
==16462== Using Valgrind-3.12.0.SVN and LibVEX; rerun with -h for
copyright info
```

```
==16462== Command: target/x86_64-unknown-linux-gnu/debug/hello_world
==16462==
Hello, world!
==16462==
==16462== HEAP SUMMARY:
==16462==     in use at exit: 0 bytes in 0 blocks
==16462==   total heap usage: 7 allocs, 7 frees, 2,032 bytes allocated
==16462==
==16462== All heap blocks were freed -- no leaks are possible
==16462==
==16462== For counts of detected and suppressed errors, rerun with: -v
==16462== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from
0)
```

Profiling memory use in Rust programs is an important day-to-day task when writing performance-critical projects. For this, we use Massif, the heap profiler:

```
hello_world > valgrind --tool=massif target/x86_64-unknown-linux-
gnu/debug/hello_world
==16471== Massif, a heap profiler
==16471== Copyright (C) 2003-2015, and GNU GPL'd, by Nicholas
Nethercote
==16471== Using Valgrind-3.12.0.SVN and LibVEX; rerun with -h for
copyright info
==16471== Command: target/x86_64-unknown-linux-gnu/debug/hello_world
==16471==
Hello, world!
==16471==
```

Profiling the cache is also an important routine task. For this, we use `cachegrind`, the cache and branch-prediction profiler:

```
hello_world > valgrind --tool=cachegrind target/x86_64-unknown-linux-
gnu/debug/hello_world
==16495== Cachegrind, a cache and branch-prediction profiler
==16495== Copyright (C) 2002-2015, and GNU GPL'd, by Nicholas
Nethercote et al.
==16495== Using Valgrind-3.12.0.SVN and LibVEX; rerun with -h for
copyright info
==16495== Command: target/x86_64-unknown-linux-gnu/debug/hello_world
==16495==
--16495-- warning: L3 cache found, using its data for the LL
```

```
simulation.
Hello, world!
==16495==
==16495== I   refs:      533,954
==16495== I1  misses:      2,064
==16495== LLi misses:      1,907
==16495== I1  miss rate:    0.39%
==16495== LLi miss rate:    0.36%
==16495==
==16495== D   refs:      190,313  (131,906 rd   + 58,407 wr)
==16495== D1  misses:      4,665  (  3,209 rd   +  1,456 wr)
==16495== LLd misses:      3,480  (  2,104 rd   +  1,376 wr)
==16495== D1  miss rate:     2.5% (    2.4%     +    2.5%  )
==16495== LLd miss rate:     1.8% (    1.6%     +    2.4%  )
==16495==
==16495== LL refs:         6,729  (  5,273 rd   +  1,456 wr)
==16495== LL misses:       5,387  (  4,011 rd   +  1,376 wr)
==16495== LL miss rate:      0.7% (    0.6%     +    2.4%  )
```

Each of these will be used throughout the book and on much more interesting projects than `hello_world`. But `hello_world` is the first cross-compilation achieved in this text, and that's no small thing.

Summary

In this chapter, we've discussed the preliminaries of this book, the machine architecture of modern CPUs, and the way loads and stores of memory may be interleaved for surprising results, illustrating the need for some kind of synchronization for our computations. That need for synchronization will drive much of the discussion of this book. We also discussed installing Rust, the channels of the compiler itself, and our intention to focus on the *stable* channel. We also compiled a simple program for both x86 and ARM systems. We also discussed debugging and performance analysis of our simple program, mostly as a proof-of-concept of the tools. The next chapter, `Chapter 2`, *Sequential Rust Performance and Testing*, will explore this area in much more depth.

Further reading

At the end of each chapter, we'll include a list of bibliographic materials, things that are warmly recommended for readers wishing to dive further into the topic discussed in the chapter, links to relevant Rust community discussions, or links to the documentation of important tools. Bibliographic material may appear in multiple chapters because if something's important, it bears repeating.

- *An Introduction to Parallel Algorithms*, 1992, Joseph JaJa. A fine textbook that introduces important abstract models. The book is significantly focused on abstract implementations of algorithms from an era when cache coherency and instruction pipelines were less important, so do be aware of that if you pull a copy.
- *What Every Programmer Should Know About Memory*, 2006, Ulrich Drepper. A classic, detailed description of how memory works in modern computers, despite being twelve years old at the time of writing this book.
- *Computer Architecture: A Quantitative Approach*, 2011, John Hennessy and David Patterson. A classic somewhat more geared toward computer architects than software engineers. Still, this is well worth studying in depth, even if you do skip over the circuit diagrams here and there.

- *The C11 and C++11 Concurrency Model*, 2014, Mark Batty. Batty formalizes the C++11/C11 memory model, which if you can get up to speed with his logic language, is an excellent way to learn the memory model and its consequences.
- *LLVM Atomic Instructions and Concurrency Guide*, available at `https://llvm.org/docs/Atomics.html`. Rust has specifically documented its concurrency memory model as being that of LLVM. This guide—and the documentation it links to—will be well-trod territory for any Rust programmer reading this book.
- *Cache Speculation Side-Channels*, 2018, ARM. Speculative execution of branches leads to surprising information leaks, it turns out. This paper by ARM gives a very clear discussion of speculative execution on ARM, as well as tidy examples.
- *std::memory_order*, available at `http://en.cppreference.com/w/cpp/atomic/memory_order`. While this document covers the memory order defined in C++, its examples of the consequences of the C++ memory-ordering guarantees are both straightforward and illustrative.
- *Valgrind User Manual*, available at `http://valgrind.org/docs/manual/manual.html`. We'll be making extensive use of `Valgrind`, and it's well worth it for any systems programmer to be familiar with these tools. The documentation necessarily touches on some of the same material as this book, and may help illuminate things under a different light.

- *Compiler Explorer*, available at `https://rust.godbolt.org/`. *Compiler Explorer* is not a paper so much as a well-designed web tool. It allows easy cross-compilation, and refers to simplified explanations of instructions.

Sequential Rust Performance and Testing

"Make it work, then make it beautiful, then if you really, really have to, make it fast."
- Joe Armstrong

In the previous chapter, we discussed the basics of modern computer architectures—the CPU and its function, memory hierarchies, and their interplay. We left off with a brief introduction to debugging and performance analysis of Rust programs. In this chapter, we'll continue that discussion, digging into the performance characteristics of sequential Rust programs, deferring, for now, considerations of concurrent performance. We'll also be discussing testing techniques for demonstrating the fitness for purpose of a Rust program. Why, in a book about parallel programming, would we wish to devote an entire chapter to just sequential programs? The techniques we'll discuss in this sequential setting are applicable and vital to a parallel setting. What we gain here is the meat of the concern—being fast *and* correct—without the complication that parallel programming brings, however, we'll come to that in good time. It is also important to understand that the production of fast parallel code comes part and parcel with the production of fast sequential code. That's on account of there being a cold, hard mathematical reality that we'll deal with throughout the book.

By the close of the chapter, we will:

- Have learned about Amdahl's and Gustafson's laws
- Have investigated the internals of the Rust standard library `HashMap`

- Be able to use QuickCheck to perform randomized validating of an alternative HashMap implementation
- Be able to use American Fuzzy Lop to demonstrate the lack of crashes in the same
- Have used Valgrind and Linux Perf to examine the performance of Rust software

Technical requirements

This chapter requires a working Rust installation. The details of verifying your installation are covered in `Chapter 1`, *Preliminaries – Machine Architecture and Getting Started with Rust*. The Valgrind suite of tools will be used here. Many operating systems bundle valgrind packages but you can find further installation instructions for your system at `valgrind.org`. Linux Perf is used and is bundled by many Linux distributions. Any other software required for this chapter is installed as a part of the text.

You can find the source code for this book's projects on GitHub: `https://github.com/PacktPublishing/Hands-On-Concurrency-with-Rust`. The source code for this chapter is under `Chapter02`.

Diminishing returns

The hard truth is that there's a diminishing return when applying more and more concurrent computational resources to a problem. Performing parallel computations implies some coordination overhead—spawning new threads, chunking data, and memory bus issues in the presence of barriers or fences, depending on your CPU. Parallel computing is not free. Consider this `Hello, world!` program:

```
fn main() {
    println!("GREETINGS, HUMANS");
}
```

Straightforward enough, yeah? Compile and run it 100 times:

```
hello_worlds > rustc -C opt-level=3 sequential_hello_world.rs
hello_worlds > time for i in {1..100}; do ./sequential_hello_world >
/dev/null; done

real    0m0.091s
user    0m0.004s
sys     0m0.012s
```

Now, consider basically the same program but involving the overhead of spawning a thread:

```
use std::thread;

fn main() {
    thread::spawn(|| println!("GREETINGS, HUMANS"))
        .join()
        .unwrap();
}
```

Compile and run it 100 times:

```
hello_worlds > rustc -C opt-level=3 parallel_hello_world.rs
hello_worlds > time for i in {1..100}; do ./parallel_hello_world >
/dev/null; done

real    0m0.115s
user    0m0.004s
sys     0m0.012s
```

This implies a thread-spawn time on my test system of 0.24 milliseconds. That is, of course, without any synchronization. The point is, it's not free. It's also not magic. Say you have a program that runs on a single processor in 24 hours and there are parts of the program that will have to be done sequentially and which, added together, consume 1 hour of the total runtime. The remaining 23 hours represent computations that can be run in parallel. If this computation is important and needs to be done in a hurry, the temptation is going to be to chuck in as much hardware as possible. How much of an improvement should you expect?

One well-known answer to this question is Amdahl's law. It states that the speedup of a computation is proportional to the inverse of the percentage of time taken by the sequential bit plus the percentage of the parallel time divided by the total new computation units, *1/(s + p / N)*. As *N* tends toward infinity, *1/s == 1/(1-p)*, in our example, *1/(1 - (23/24)) = 24*. That is, the maximum factor speedup you can ever hope to see is 24 times, with infinite additional capacity. Ouch.

Amdahl's law is a touch pessimistic, as noted by John Gustafson in his 1988 Reevaluating Amdahl's Law:

"At Sandia National Laboratories, we are currently engaged in research involving massively-parallel processing. There is considerable skepticism regarding the viability of massive parallelism; the skepticism centers around Amdahl's law, an argument put forth by Gene Amdahl in 1967 that even when the fraction of serial work in a given problem is small, say s, the maximum speedup obtainable from even an infinite number of parallel processors is only

1/s. We now have timing results for a 1024-processor system that demonstrate that the assumptions underlying Amdahl's 1967 argument are inappropriate for the current approach to massive ensemble parallelism."

Gustafson argues that real workloads will take the time factor of a computation as fixed and vary the input work accordingly. In some hypothetical, very important computations we'd see 24 hours as acceptable and, on increasing the total number of processors available, then rush to figure out how to add in more computation so as to get the time back up to one day. As we increase the total workload, the serial portion of the computation tends towards zero. This is, itself, maybe somewhat optimistic and is certainly not applicable to problems where there's no more datasets available. Communication overhead is not included in either analysis.

Ultimately, what has to be internalized is this—performing computations in parallel is subject to diminishing returns. Exactly how that take shape depends strongly on your problem, the machine, and so on, but it's there. What the programmer must do to bend that curve is shrink the percentage of time spent in serial computation, either by increasing the relative portion of parallel computation per Gustafson with a larger dataset or by optimizing the serial computation's runtime. The remainder of this chapter will be focused on the latter approach—improving the runtime of serial computations.

Performance

In this section, we'll focus on the serial performance of a common data structure—associative arrays. We'll apply the tools we learned about in the previous chapter to probe different implementations. We'll focus on the associative array because it is fairly well-trod territory, studied in introductory computer science courses, and is available in most higher-level languages by default, Rust being no exception save the higher-level bit. We'll look at Rust's associative array first, which is called `std::collections::HashMap`.

Standard library HashMap

Let's poke around in `HashMap`'s internals. A good starting place, I find, for inspecting unfamiliar Rust data structures is jumping into the source to the struct definition itself. Especially in the Rust codebase, there will be public `rustdoc` comments and private comments explaining implementation ambitions. The reader is warmly encouraged to inspect the `HashMap` comments for themselves. In this book, we're inspecting Rust at SHA `da569fa9ddf8369a9809184d43c600dc06bd4b4d`. The comments of `src/libstd/collections/hash/map.rs` explain that the `HashMap` is implemented with linear probing Robin Hood bucket stealing. Linear probing implies that we'll find `HashMap` implemented in terms of a cell storage —probably a contiguous memory structure for reasons we'll discuss shortly—and should be able to understand the implementation pretty directly, linear probing being a common method of implementing associative arrays. Robin Hood bucket stealing is maybe less common, but the private comments describe it as *the main performance trick in this hashmap* and then goes on to quote Pedro Celis' 1986 *Robin Hood Hashing:*

"If an insertion collides with an existing element, and that element's "probe distance" (how far away the element is from its ideal location) is higher than how far we've already probed, swap the elements."

If you pull the thesis yourself, you'll further find:

"(Robin Hood Hashing) leads to a new search algorithm which requires less than 2.6 probes on average to perform a successful search even when the table is nearly full. Unsuccessful searches require only O(ln n) probes."

Not bad. Here's `HashMap`:

```
pub struct HashMap<K, V, S = RandomState> {
    // All hashes are keyed on these values, to prevent
    // hash collision attacks.
    hash_builder: S,

    table: RawTable<K, V>,

    resize_policy: DefaultResizePolicy,
}
```

Okay, so, it interacts with `RawTable<K, V>` and we can jump to that definition:

```
pub struct RawTable<K, V> {
    capacity_mask: usize,
    size: usize,
    hashes: TaggedHashUintPtr,

    // Because K/V do not appear directly in any of the
    // types in the struct, inform rustc that in fact
    // instances of K and V are reachable from here.
    marker: marker::PhantomData<(K, V)>,
}
```

But it's maybe not the most illuminating struct definition either. Common operations on a collection are often a good place to dig in, so let's look at `HashMap::insert`:

```
    pub fn insert(&mut self, k: K, v: V) -> Option<V> {
        let hash = self.make_hash(&k);
        self.reserve(1);
        self.insert_hashed_nocheck(hash, k, v)
    }
```

And then on into `HashMap::reserve`:

```
    pub fn reserve(&mut self, additional: usize) {
        let remaining = self.capacity() - self.len(); // this can't
        overflow
        if remaining < additional {
```

```
            let min_cap =
            self.len().checked_add(additional)
                      .expect("reserve overflow");
            let raw_cap = self.resize_policy.raw_capacity(min_cap);
            self.resize(raw_cap);
        } else if self.table.tag() && remaining <= self.len() {
            // Probe sequence is too long and table is half full,
            // resize early to reduce probing length.
            let new_capacity = self.table.capacity() * 2;
            self.resize(new_capacity);
        }
    }
```

Whether `reserve` expands the underlying storage capacity because we've got near the current total capacity of that storage or to reduce probe lengths, `HashMap::resize` is called. That's:

```
    #[inline(never)]
    #[cold]
    fn resize(&mut self, new_raw_cap: usize) {
        assert!(self.table.size() <= new_raw_cap);
        assert!(new_raw_cap.is_power_of_two() || new_raw_cap == 0);

        let mut old_table = replace(&mut self.table,
        RawTable::new(new_raw_cap));
        let old_size = old_table.size();

        if old_table.size() == 0 {
            return;
        }

        let mut bucket = Bucket::head_bucket(&mut old_table);

        // This is how the buckets might be laid out in memory:
        // ($ marks an initialized bucket)
        //  ________________
        // |$$$_$$$$$$_$$$$$|
        //
        // But we've skipped the entire initial cluster of buckets
        // and will continue iteration in this order:
        //  ________________
        //     |$$$$$$_$$$$$
        //                  ^ wrap around once end is reached
        //  ________________
        //  $$$_____________|
```

```
        //    ^ exit once table.size == 0
        loop {
            bucket = match bucket.peek() {
                Full(bucket) => {
                    let h = bucket.hash();
                    let (b, k, v) = bucket.take();
                    self.insert_hashed_ordered(h, k, v);
                    if b.table().size() == 0 {
                        break;
                    }
                    b.into_bucket()
                }
                Empty(b) => b.into_bucket(),
            };
            bucket.next();
        }

        assert_eq!(self.table.size(), old_size);
    }
```

We're back at `RawTable` and have a lead on a structure called `Bucket`:

```
pub struct Bucket<K, V, M> {
    raw: RawBucket<K, V>,
    table: M,
}
```

Okay, we're going in some circles here. The bucket parameterizes the table on `M` rather than the table holding some collection of buckets. The `RawBucket` is described as an unsafe view of a `RawTable` bucket of which there are two variants—`EmptyBucket` and `FullBucket`. These variants are used to populate a `BucketState` enumeration:

```
    pub enum BucketState<K, V, M> {
        Empty(EmptyBucket<K, V, M>),
        Full(FullBucket<K, V, M>),
    }
```

Here, we can see it being used back in `HashMap::insert_hashed_ordered`:

```
fn insert_hashed_ordered(&mut self, hash: SafeHash, k: K, v: V) {
    let mut buckets = Bucket::new(&mut self.table, hash);
    let start_index = buckets.index();

    loop {
        // We don't need to compare hashes for value swap.
        // Not even DIBs for Robin Hood.
        buckets = match buckets.peek() {
            Empty(empty) => {
                empty.put(hash, k, v);
                return;
            }
            Full(b) => b.into_bucket(),
        };
        buckets.next();
        debug_assert!(buckets.index() != start_index);
    }
}
```

If we explore down into `empty.push(hash, k, v)`, we find ourselves at the following:

```
pub fn put(mut self, hash: SafeHash, key: K, value: V)
    -> FullBucket<K, V, M>
{
    unsafe {
        *self.raw.hash() = hash.inspect();
        ptr::write(self.raw.pair(), (key, value));

        self.table.borrow_table_mut().size += 1;
    }

    FullBucket {
        raw: self.raw,
        table: self.table,
    }
}
```

Tidy. `ptr::write(self.raw.pair(), (key, value))` demonstrates that we're working with a structure built out of raw memory pointer manipulation, befitting a structure that will see a lot of use in critical `paths`. `self.raw.pair()`, which returns the appropriate offset to move `(key,`

`value)` into, matching the `HashMap::insert` move semantics we're already familiar with. Take a look at the definition of `RawBucket`:

```
pub struct RawBucket<K, V> {
    hash_start: *mut HashUint,
    // We use *const to ensure covariance with respect to K and V
    pair_start: *const (K, V),
    idx: usize,
    _marker: marker::PhantomData<(K, V)>,
}
```

Here, we've got two raw pointers: `hash_start` and `pair_start`. The former is the first location in memory of our stored pairs' hashes, the latter is the first location in the memory of the pairs. What this ends up being is contiguous storage in memory of two sets of data, which is about what you'd expect. The table module documentation refers to this approach as *unzipped* arrays. `RawTable` holds the capacity and the data, kind of. In reality, the data held by `RawTable` is carefully placed in memory, as we've seen, but there's no *owner* as the Rust type system understands it. That's where `marker: marker::PhantomData<(K, V)>` comes in. `PhantomData` instructs the compiler to behave as if `RawTable<K, V>` owns pairs of `(K, V)`, even though with all the unsafe pointer manipulation we're doing that can't actually be proven by Rust. We human observers can determine by inspection that `RawTable` owns its data via `RawTable::raw_bucket_at` as it computes where in memory the data exists:

```
fn raw_bucket_at(&self, index: usize) -> RawBucket<K, V> {
    let hashes_size = self.capacity() * size_of::<HashUint>();
    let pairs_size = self.capacity() * size_of::<(K, V)>();

    let (pairs_offset, _, oflo) =
        calculate_offsets(hashes_size, pairs_size,
                          align_of::<(K, V)>());
    debug_assert!(!oflo, "capacity overflow");

    let buffer = self.hashes.ptr() as *mut u8;
    unsafe {
        RawBucket {
            hash_start: buffer as *mut HashUint,
```

```
                pair_start: buffer.offset(pairs_offset as isize)
                              as *const (K, V),
                idx: index,
                _marker: marker::PhantomData,
            }
        }
    }
```

Well, by inspection and testing, as you can see at the bottom of the module.

Naive HashMap

How else could we implement an associative array in Rust and how would it compare to standard library's `HashMap`? The standard library `HashMap` is clever, clearly, storing just enough information to reduce the total probes from ideal offsets by sharing information between modifications to the underlying table. In fact, the comments in the table module assert that the design we've just worked through is *a lot faster* than a table structured as `Vec<Option<(u64, K, V)>>`—where `u64` is the key hash, but presumably still using Robin Hood hashing and linear probing. What if we went even simpler? We'll support only two operations—`insert` and `lookup`—to keep things straightforward for ourselves. We'll also keep more or less the same type constraints as `HashMap` so we compare similar things.

Start a new Rust project called `naive_hashmap`:

```
> cargo new naive_hashmap
     Created library `naive_hashmap` project
```

Edit `naive_hashmap/Cargo.toml` to look like so:

```
[package]
name = "naive_hashmap"
version = "0.1.0"

[dependencies]
rand = "0.4.2"

[dev-dependencies]
quickcheck = "0.6"
criterion = "0.1"

[[bench]]
```

```
name = "naive"
harness = false

[[bin]]
name = "naive_interpreter"
doc = false

[[bin]]
name = "standard"
doc = false

[[bin]]
name = "naive"
doc = false

[[bench]]
name = "specialized"
harness = false

[[bin]]
name = "specialized_interpreter"
doc = false

[[bin]]
name = "specialized"
doc = false
```

Don't worry too much right now about some of the development dependencies, they'll be explained in due course. Please note that in the following discussion we will run compilation commands against the project before all the code has been discussed. If you are following along with the book's pre-written source code open already, you should have no issues. If you are writing the source out as the book goes along, you will need to comment targets out. Now, open `naive_hashmap/src/lib.rs` and add the following preamble:

```
#[cfg(test)]
extern crate quickcheck;

use std::hash::{BuildHasher, Hash, Hasher};
use std::borrow::Borrow;
use std::collections::hash_map::RandomState;
use std::{cmp, mem, ptr};
```

Most crate roots begin with a fairly long preamble, and `naive_hashmap` is no exception. Next up, our `HashMap` struct:

```
#[derive(Default)]
pub struct HashMap<K, V, S = RandomState>
where
    K: Eq,
    V: ::std::fmt::Debug,
{
    hash_builder: S,
    data: Vec<(u64, K, V)>,
}
```

How does our naive `HashMap` differ from the standard library's `HashMap` we've seen so far? The naive implementation maintains the parameterized hasher and type constraints, plus a constraint on `V` to implement the `Debug` trait for ease of fiddling. The primary difference, in terms of underlying structure, is the use of a `Vec`—as called out in the comments of the standard library `HashMap`—but without an `Option` wrapper. We're not implementing a delete operation so there's no reason to have an `Option` but, even if we were, the plan for this implementation would be to rely solely on `Vec::remove`. The fact that it is less than ideal that `Vec::remove` shifts all elements from the right of the removal index to the left should be well understood. Folks, this won't be a fast implementation. Now:

```
impl<K: Eq, V> HashMap<K, V, RandomState>
where
    K: Eq + Hash,
    V: ::std::fmt::Debug,
{
    pub fn new() -> HashMap<K, V> {
        HashMap {
            hash_builder: RandomState::new(),
            data: Vec::new(),
        }
    }
}
```

```
fn make_hash<T: ?Sized, S>(hash_builder: &S, t: &T) -> u64
where
    T: Hash,
    S: BuildHasher,
{
    let mut state = hash_builder.build_hasher();
    t.hash(&mut state);
    state.finish()
}

impl<K, V, S> HashMap<K, V, S>
where
    K: Eq + Hash,
    S: BuildHasher,
    V: ::std::fmt::Debug,
{
    pub fn with_hasher(hash_builder: S) -> HashMap<K, V, S> {
        HashMap {
            hash_builder: hash_builder,
            data: Vec::new(),
        }
    }
```

Our naive implementation is following along with the standard library here, implementing a `HashMap` that is parameterized on `RandomState`—so users don't have to think about the underlying hasher—and in which the hasher is swappable, via `HashMap::with_hasher`. The Rust team chose to implement `RandomState` in terms of a verified cryptographically secure hash algorithm, befitting a language that is intended for use on the public internet. Some users won't desire this property—opting instead for a much faster, potentially vulnerable hash—and will swap `RandomState` out for something else. Our naive `HashMap` retains this ability.

Let's examine insertion into our naive `HashMap`:

```
pub fn insert(&mut self, k: K, v: V) -> Option<V> {
    let hash = make_hash(&self.hash_builder, &k);

    let end = self.data.len();
    for idx in 0..end {
        match self.data[idx].0.cmp(&hash) {
```

```
                cmp::Ordering::Greater => {
                    self.data.insert(idx, (hash, k, v));
                    return None;
                }
                cmp::Ordering::Less => continue,
                cmp::Ordering::Equal => {
                    let old = mem::replace(&mut self.data[idx].2, v);
                    return Some(old);
                }
            }
        }
        self.data.push((hash, k, v));
        None
    }
```

Our naive implementation maintains the API of the standard library `HashMap` but that's about it. The key is hashed and then a linear search is done through the entire data store to find an index in that store where one of two conditions hold:

- Our new hash is greater than some hash in the store, in which case we can insert our key/value pair
- Our new hash is equal to some hash in the store, in which case we can replace the existing value

Key/value pairs are stored in terms of their ordered hashes. The expense of an insert includes:

- Hashing the key
- Searching for the insert index, a linear operation to the number of stored key/value pairs
- Potentially shifting key/value pairs in memory to accommodate a new key to the store

Lookup follows a similar scheme:

```
pub fn get<Q: ?Sized>(&mut self, k: &Q) -> Option<&V>
where
    K: Borrow<Q> + ::std::fmt::Debug,
    Q: Hash + Eq + ::std::fmt::Debug,
{
    let hash = make_hash(&self.hash_builder, k);

    for &(bucket_hash, _, ref v) in &self.data {
        if hash == bucket_hash {
            return Some(v);
        }
    }
    None
}
```

The key is hashed and a linear search is done for that exact hash in storage. Here, we only pay the cost for:

- Hashing the key
- Searching for the retrieval offset, if one exists

Let's take a moment and consider this before asking ourselves if our program is *fast* if it is *correct*. The usual way of demonstrating fitness for purpose of software is through unit testing, a minimal setup for which is built right into the Rust language. Unit testing is two processes wrapped into a single method. When writing unit tests, a programmer will:

- Produce *example data* that exercises some code path in the software
- Write further code to demonstrate that when the example data is applied to the system under test, a desirable

property/properties holds

This is a good testing methodology for demonstrating that the happy path of a system under test works as expected. The weak point of unit testing comes in the first process, where the programmer must think very hard and produce an example dataset that exercises the correct code paths, demonstrates the lack of edge cases, and so on. Human beings are poor at this task, owing to it being tedious, biased toward demonstrating the functioning of a thing made by one's own hands, chronic blind spots, or otherwise. What we *are* pretty good at doing is cooking up high-level properties for our systems that must always hold or hold in particular situations. Fortunately for us programmers, *computers* are exceptionally good at doing tedious, repetitive tasks and the generation of example data for tests is such a thing.

Testing with QuickCheck

With that in mind, in this book, we'll make extensive use of *property-based* testing, also called *generative* testing by some literature. Property-based testing has a slightly different, if similar, workflow to unit testing. When writing property tests, the programmer will:

- Produce a method for generating valid inputs to a system under test
- Write further code to demonstrate that for all valid inputs that a desirable property or property of the system holds

Property-based testing is exceptionally good at finding corner cases in software, wacky inputs that the programmer might not have dreamed up, or, as happens from time to time, even understand as potentially problematic. We'll use Andrew Gallant's QuickCheck, a tool that is patterned from Haskell QuickCheck, introduced by Koen Claessen and John Hughes in their 2000 *QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs*. First, a little preamble to get us into the testing module:

```
#[cfg(test)]
mod test {
    extern crate quickcheck;

    use super::*;
    use quickcheck::{Arbitrary, Gen, QuickCheck, TestResult};
```

If we were to unit test our naive `HashMap`, we'd probably write a test where a particular arbitrary key/value pair would be inserted and then

we'd assert the ability to retrieve the same value with that same key. How does this map to property-based testing? The property is that if we perform an insertion on an empty `HashMap` with a key `k` of value `v` and immediately perform a retrieval of `k`, we'll receive `v` back out. Here it is:

```
#[test]
fn get_what_you_give() {
    fn property(k: u16, v: u16) -> TestResult {
        let mut system_under_test = HashMap::new();

        assert_eq!(None, system_under_test.insert(k, v));
        assert_eq!(Some(&v), system_under_test.get(&k));

        TestResult::passed()
    }
    QuickCheck::new().quickcheck(property as fn(u16, u16) ->
                                 TestResult);
}
```

The test is named `get_what_you_give` and is annotated `#[test]` as per usual Rust tests. What's different is this test has a property `inner` function that encodes the property we elaborated on earlier and a call out to QuickCheck for actually running the property test, by default 100 times, with different inputs. Exactly how QuickCheck manages this is pretty swell. Per the Claessen and Hughes paper, QuickCheck implements the `Arbitrary` trait for many of the types available in the standard library. QuickCheck defines `Arbitrary` like so, in `quickcheck/src/arbitrary.rs`:

```
pub trait Arbitrary : Clone + Send + 'static {
    fn arbitrary<G: Gen>(g: &mut G) -> Self;

    fn shrink(&self) -> Box<Iterator<Item=Self>> {
        empty_shrinker()
    }
}
```

The `Gen` parameter is a wrapper around `rand:Rng` with some extra machinery for controlling distributions. The two functions are `arbitrary` and `shrink`. The purpose of `arbitrary` is easy to explain: it generates new, random instances of a type that is `Arbitrary`. The `shrink` function is a little more complicated to get across. Let's say we've produced a function that takes a vector of `u16` but just so happens to have a bug and will crash if a member of the vector is `0`. A QuickCheck test will likely find this but the first arbitrary vector it generates may have a thousand members and a bunch of `0` entries. This is not a very useful failure case to begin diagnosis with. After a failing case is found by QuickCheck the case is shrunk, per the definition of `Arbitrary::shrink`. Our hypothetical vector will be cut in half and if that new case is also a failure then it'll be shrunk by half again and so on until it no longer fails, at which point—hopefully—our failing case is a much more viable diagnosis tool.

The implementation of `Arbitrary` for `u16` is a touch complicated due to macro use, but if you squint some, it'll become clear:

```
macro_rules! unsigned_arbitrary {
    ($($ty:tt),*) => {
        $(
            impl Arbitrary for $ty {
                fn arbitrary<G: Gen>(g: &mut G) -> $ty {
                    #![allow(trivial_numeric_casts)]
                    let s = g.size() as $ty;
                    use std::cmp::{min, max};
                    g.gen_range(0, max(1, min(s, $ty::max_value())))
                }
                fn shrink(&self) -> Box<Iterator<Item=$ty>> {
                    unsigned_shrinker!($ty);
                    shrinker::UnsignedShrinker::new(*self)
                }
            }
        )*
    }
}

unsigned_arbitrary! {
```

```
        usize, u8, u16, u32, u64
    }
```

Arbitrary instances are generated by `unsigned_arbitrary!`, each of which in turn generates a shrinker via `unsigned_shrinker!`. This macro is too long to reprint here but the basic idea is to remove half of the unsigned integer on every shrink again, until zero is hit, at which point give up shrinking.

Fifteen years after the original QuickCheck paper, John Hughes summarized his experience in the intervening 15 years with his 2016 *Experiences with QuickCheck: Testing the Hard Stuff and Staying Sane*. Hughes noted that many property-based tests in the wild don't generate primitive types. The domain of applications in which primitive types are sufficient tends to be those rare, blessed pure functions in a code base. Instead, as many functions in a program are inherently stateful, property-based tests tend to generate arbitrary *actions* against a stateful system, *modeling* the expected behavior and validating that the system under test behaves according to its model.

How does that apply to our naive `HashMap`? The system under test is the naive HashMap, that's clear enough. Our actions are:

- `INSERT` key value
- `LOOKUP` key

Hopefully, that's straightforward. What about a model? In this particular instance, we're in luck. Our model is written for us: the standard library's `HashMap`. We need only confirm that if the same actions are applied to both our naive `HashMap` and the standard `HashMap` in the same order then we'll get the same returns from both the model and system under test. How does that look? First, we need actions:

```
#[derive(Clone, Debug)]
enum Action<T>
where
    T: Arbitrary,
{
    Insert(T, u16),
    Lookup(T),
}
```

Our `Action<T>` is parameterized on a `T: Arbitrary` and all values via the Insert are pegged as `u16`s. This is done primarily for convenience's sake. Both key and value could be arbitrary or concrete types, depending on the preference of the tester. Our `Arbitrary` definition is as follows:

```
impl<T> Arbitrary for Action<T>
where
    T: Arbitrary,
{
    fn arbitrary<G>(g: &mut G) -> Action<T>
    where
        G: Gen,
    {
        let i: usize = g.gen_range(0, 100);
        match i {
            0...50 => Action::Insert(Arbitrary::arbitrary(g),
                                     u16::arbitrary(g)),
            _ => Action::Lookup(Arbitrary::arbitrary(g)),
        }
    }
}
```

Either action is equally likely to happen and any instance of `T` or `u16` is valid for use. The validation of our naive `HashMap` against the standard library's `HashMap`:

```
#[test]
fn sut_vs_genuine_article() {
    fn property<T>(actions: Vec<Action<T>>) -> TestResult
    where
        T: Arbitrary + Eq + Hash + ::std::fmt::Debug,
```

```
        {
            let mut model = ::std::collections::HashMap::new();
            let mut system_under_test = HashMap::new();

            for action in actions.into_iter() {
                match action {
                    Action::Insert(k, v) => {
                        assert_eq!(model.insert(k.clone(), v),
                        system_under_test.insert(k, v));
                    }
                    Action::Lookup(k) => {
                        assert_eq!(model.get(&k),
                        system_under_test.get(&k));
                    }
                }
            }
            TestResult::passed()
        }
        QuickCheck::new().quickcheck(property as fn(Vec<Action<u8>>) ->
                                     TestResult);
    }
}
```

Hopefully, this looks familiar to our previous QuickCheck test. Again, we have an inner property function that does the actual work and we request that QuickCheck run this property test over arbitrary vectors of `Action<u8>`. For every action present in the vector we apply them to both the model and the system under test, validating that the results are the same. The `u8` type was intentionally chosen here compared to a large domain type such as `u64`. One of the key challenges of writing QuickCheck tests is probing for extremely unlikely events. QuickCheck is blind in the sense that it's possible for the same path to be chosen through the program for each run if that path is the most likely path to be chosen. While millions of QuickCheck tests can give high confidence in fitness for purpose the blind nature of the runs means that QuickCheck should also be paired with tools known as fuzzers. These do not check the correct function of a program against a model. Instead, their sole purpose is to validate the absence of program crashes.

Testing with American Fuzzy Lop

In this book, we'll make use of American Fuzzy Lop, a best-of-breed fuzzer commonly used in other systems languages. AFL is an external tool that takes a corpus of inputs, an executable that reads inputs from `STDIN`, and mutates the corpus with a variety of heuristics to find crashing inputs. Our aim is to seek out crash bugs in naive `HashMap`, this implies that we're going to need some kind of program to run our `HashMap` in. In fact, if you'll recall back to the project's `Cargo.toml`, we already had the infrastructure for such in place:

```
[[bin]]
name = "naive_interpreter"
doc = false
```

The source for `naive_interpreter` is a little goofy looking but otherwise uneventful:

```
extern crate naive_hashmap;

use std::io;
use std::io::prelude::*;

fn main() {
    let mut hash_map = naive_hashmap::HashMap::new();

    let n = io::stdin();
    for line in n.lock().lines() {
        if let Ok(line) = line {
            let mut cmd = line.split(" ");
            match cmd.next() {
                Some("LOOKUP") => {
                    if let Some(key) = cmd.next() {
```

```
                            let _ = hash_map.get(key);
                        } else {
                            continue;
                        }
                    }
                    Some("INSERT") => {
                        if let Some(key) = cmd.next() {
                            if let Some(val) = cmd.next() {
                                let _ = hash_map.insert(key.to_string(),
                                val.to_string());
                            }
                        } else {
                            continue;
                        }
                    }
                    _ => continue,
                }
            } else {
                break;
            }
        }
    }
```

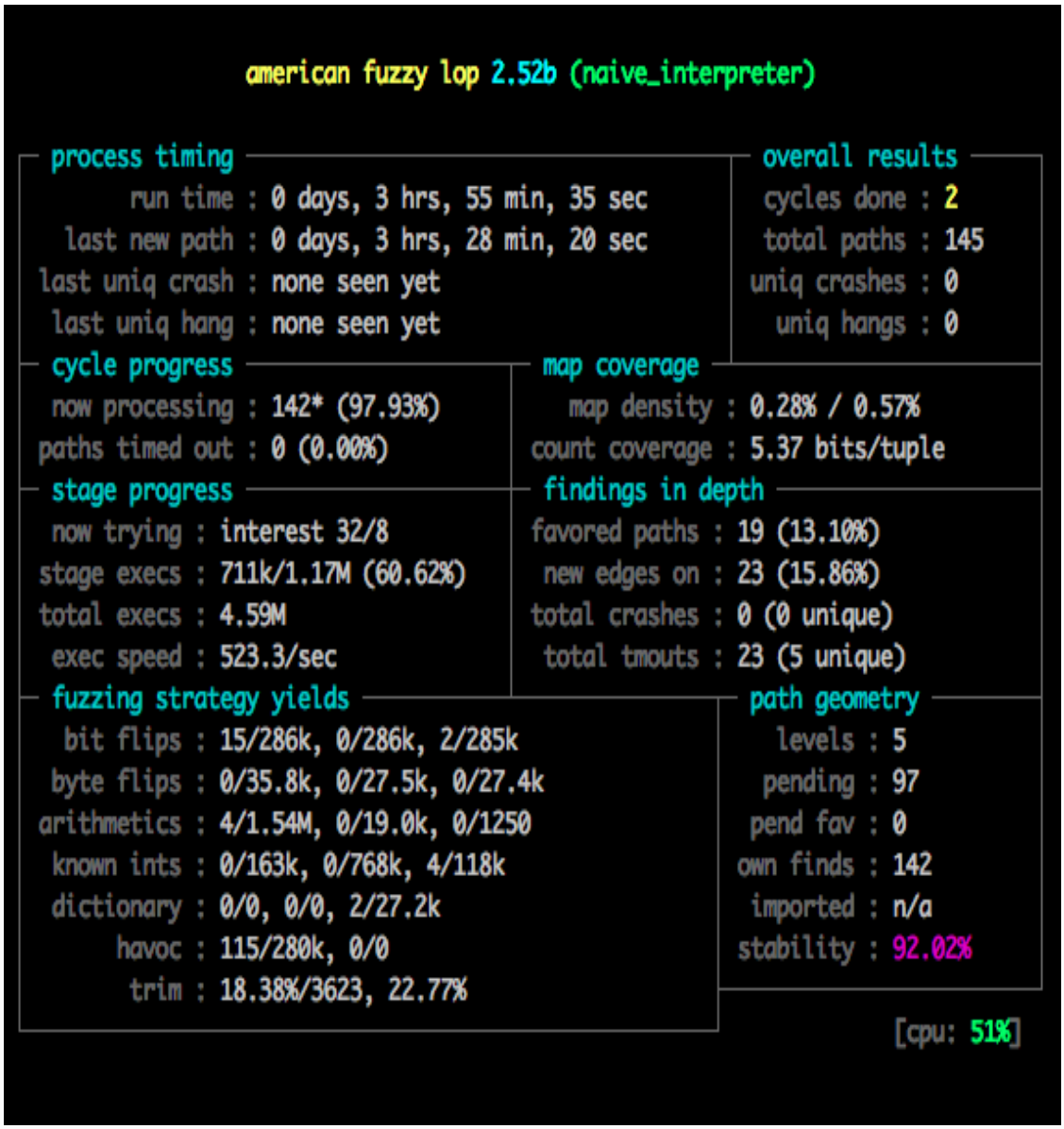
Lines are read from `stdin`, and these lines are split along spaces and interpreted as commands, either `LOOKUP` or `INSERT` and these are interpreted into actions on the naive `HashMap`. The corpus for an AFL run can live pretty much anywhere. By convention, in this book we'll store the corpus in-project in a top-level resources/directory. Here's `resources/in/mixed_gets_puts`:

```
LOOKUP 10
INSERT abs 10
LOOKUP abs
```

The larger your input corpus, the more material AFL has to mutate and the faster—maybe—you'll find crashing bugs. Even in a case such as naive `HashMap` where we can be reasonably certain that there will be no crashing bugs—owing to the lack of pointer manipulation and potentially fatal integer operations—it's worthwhile building up a good corpus to support future efforts. After building a release of the project, getting an AFL run going is a cargo command away:

```
> cargo afl build --release
> cargo afl fuzz -i resources/in/ -o resources/out/
target/release/naive_interpreter
```

This says to execute `target/release/naive_interpreter` under AFL with input corpus at `resources/in` and to output any failing cases under `resources/out`. Such crashes often make excellent unit tests. Now, the trick with fuzzing tools in general is they're not part of any kind of quick-cycle test-driven development loop. These tool runs are long and often get run on dedicated hardware overnight or over many days. Here, for example, is the AFL run I had going while writing this chapter:



There's a fair bit of information here but consider that the AFL runtime indicator is measured in days. One key advantage to the use of AFL compared to other fuzzers is the prominence of AFL in the security community. There are a good many papers describing its implementation and the interpretation of its, uh, *comprehensive* interface. You are warmly encouraged to scan the *Further reading* section of this chapter for more information.

Performance testing with Criterion

We can now be fairly confident that our naive `HashMap` has the same behavior as the standard library for the properties we've checked. But how does the runtime performance of naive `HashMap` stack up? To answer this, we'll write a benchmark, but not a benchmark using the unstable Bencher subsystem that's available in the nightly channel. Instead, we'll use Jorge Aparicio's criterion—inspired by the Haskell tool of the same name by Bryan O'Sullivan—which is available on stable and does statistically valid sampling of runs. All Rust benchmark code lives under the top-level `benches/` directory and criterion benchmarks are no different. Open `benches/naive.rs` and give it this preamble:

```
#[macro_use]
extern crate criterion;
extern crate naive_hashmap;
extern crate rand;

use criterion::{Criterion, Fun};
use rand::{Rng, SeedableRng, XorShiftRng};
```

This benchmark incorporates pseudorandomness to produce an interesting run. Much like unit tests, handcrafted datasets in benchmarks will trend towards some implicit bias of the author, harming the benchmark. Unless, of course, a handcrafted dataset is exactly what's called for. Benchmarking programs well is a non-trivial amount of labor. The `Rng` we use is `XorShift`, a pseudo-random generator known for its speed and less cryptographic security. That suits our purposes here:

```
fn insert_and_lookup_naive(mut n: u64) {
    let mut rng: XorShiftRng = SeedableRng::from_seed([1981, 1986,
                                                       2003, 2011]);
    let mut hash_map = naive_hashmap::HashMap::new();

    while n != 0 {
        let key = rng.gen::<u8>();
        if rng.gen::<bool>() {
            let value = rng.gen::<u32>();
            hash_map.insert(key, value);
        } else {
            hash_map.get(&key);
        }
        n -= 1;
    }
}
```

The first benchmark, named `insert_and_lookup_native`, performs pseudo-random insertions and lookups against the naive `HashMap`. The careful reader will note that `XorShiftRng` is given the same seed every benchmark. This is important. While we want to avoid handcrafting a benchmark dataset, we do want it to be the same every run, else the benchmark comparisons have no basis. That noted, the rest of the benchmark shouldn't be much of a surprise. Generate random actions, apply them, and so forth. As we're interested in the times for standard library `HashMap` we have a benchmark for that, too:

```
fn insert_and_lookup_standard(mut n: u64) {
    let mut rng: XorShiftRng = SeedableRng::from_seed([1981, 1986,
                                                       2003, 2011]);
    let mut hash_map = ::std::collections::HashMap::new();

    while n != 0 {
        let key = rng.gen::<u8>();
        if rng.gen::<bool>() {
            let value = rng.gen::<u32>();
            hash_map.insert(key, value);
        } else {
            hash_map.get(&key);
        }
        n -= 1;
    }
}
```

Criterion offers, as of writing this book, a method for comparing two functions *or* comparing a function run over parameterized inputs but not both. That's an issue for us here, as we'd like to compare two functions over many inputs. To that end, this benchmark relies on a small macro called `insert_lookup!`:

```
macro_rules! insert_lookup {
    ($fn:ident, $s:expr) => {
        fn $fn(c: &mut Criterion) {
            let naive = Fun::new("naive", |b, i| b.iter(||
                                    insert_and_lookup_naive(*i))
            );
            let standard = Fun::new("standard", |b, i| b.iter(||
                                       insert_and_lookup_standard(*i))
            );

            let functions = vec![naive, standard];

            c.bench_functions(&format!("HashMap/{}", $s),
                              functions, &$s);
        }
    }
}

insert_lookup!(insert_lookup_100000, 100_000);
insert_lookup!(insert_lookup_10000, 10_000);
insert_lookup!(insert_lookup_1000, 1_000);
insert_lookup!(insert_lookup_100, 100);
insert_lookup!(insert_lookup_10, 10);
insert_lookup!(insert_lookup_1, 1);
```

The meat here is we create two `Fun` for comparison called `naive` and `standard`, then use `Criterion::bench_functions` to run a comparison between the two. In the invocation of the macro, we evaluate `insert_and_lookup_*` from `1` to `100_000`, with it being the total number of insertions to be performed against the standard and naive `HashMap`s. Finally, we need the criterion group and main function in place:

```
criterion_group!{
    name = benches;
```

```
        config = Criterion::default();
        targets = insert_lookup_100000, insert_lookup_10000,
                  insert_lookup_1000, insert_lookup_100,
                  insert_lookup_10, insert_lookup_1
    }
    criterion_main!(benches);
```

Running criterion benchmarks is no different to executing Rust built-in benchmarks:

```
> cargo bench
   Compiling naive_hashmap v0.1.0
(file:///home/blt/projects/us/troutwine/concurrency_in_rust/external_pro
jects/naive_hashmap)
    Finished release [optimized] target(s) in 5.26 secs
     Running target/release/deps/naive_hashmap-fe6fcaf7cf309bb8

running 2 tests
test test::get_what_you_give ... ignored
test test::sut_vs_genuine_article ... ignored

test result: ok. 0 passed; 0 failed; 2 ignored; 0 measured; 0 filtered
out

     Running target/release/deps/naive_interpreter-26c60c76389fd26d

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
out

     Running target/release/deps/naive-96230b57fe0068b7
HashMap/100000/naive    time:   [15.807 ms 15.840 ms 15.879 ms]
Found 13 outliers among 100 measurements (13.00%)
  2 (2.00%) low mild
  1 (1.00%) high mild
  10 (10.00%) high severe
HashMap/100000/standard time:   [2.9067 ms 2.9102 ms 2.9135 ms]
Found 7 outliers among 100 measurements (7.00%)
  1 (1.00%) low severe
  5 (5.00%) low mild
  1 (1.00%) high mild

HashMap/10000/naive     time:   [1.5475 ms 1.5481 ms 1.5486 ms]
Found 9 outliers among 100 measurements (9.00%)
  1 (1.00%) low severe
```

```
  2 (2.00%) high mild
  6 (6.00%) high severe
HashMap/10000/standard  time:   [310.60 us 310.81 us 311.03 us]
Found 11 outliers among 100 measurements (11.00%)
  1 (1.00%) low severe
  2 (2.00%) low mild
  3 (3.00%) high mild
  5 (5.00%) high severe
```

And so forth. Criterion also helpfully produces gnuplot graphs of your runtimes, if gnuplot is installed on your benchmark system. This is highly recommended.

Inspecting with the Valgrind Suite

Your specific benchmarking output will likely vary but it's pretty clear that the naive implementation is, well, not very fast. What tools do we have available to us to diagnose the issues with our code, guide us toward hot spots for optimization, or help convince us of the need for better algorithms? We'll now leave the realm of Rust-specific tools and dip into tools familiar to systems programmers at large. If you're not familiar with them personally that's a-okay—we'll describe their use and there are plenty of external materials available for the motivated reader. Okay, first, we're going to need some programs that exercise our naive `HashMap` and the standard `HashMap`. `naive_interpreter` would work, but it's doing a lot of extra things that'll muddy the water some. To that end, for examination purposes, we'll need two programs, one to establish a baseline and one for our implementation. Our baseline, called `bin/standard.rs`:

```
extern crate naive_hashmap;
extern crate rand;

use std::collections::HashMap;
use rand::{Rng, SeedableRng, XorShiftRng};

fn main() {
    let mut rng: XorShiftRng = SeedableRng::from_seed([1981, 1986,
                                                       2003, 2011]);
    let mut hash_map = HashMap::new();

    let mut insert_empty = 0;
    let mut insert_present = 0;
    let mut get_fail = 0;
    let mut get_success = 0;
```

```
    for _ in 0..100_000 {
        let key = rng.gen::<u16>();
        if rng.gen::<bool>() {
            let value = rng.gen::<u32>();
            if hash_map.insert(key, value).is_none() {
                insert_empty += 1;
            } else {
                insert_present += 1;
            }
        } else {
            if hash_map.get(&key).is_none() {
                get_fail += 1;
            } else {
                get_success += 1;
            }
        }
    }

    println!("INSERT");
    println!("  empty:   {}", insert_empty);
    println!("  present: {}", insert_present);
    println!("LOOKUP");
    println!("  fail:    {}", get_fail);
    println!("  success: {}", get_success);
}
```

Our test article program is exactly the same, save the preamble is a bit different:

```
extern crate naive_hashmap;
extern crate rand;

use naive_hashmap::HashMap;
use rand::{Rng, SeedableRng, XorShiftRng};

fn main() {
    let mut rng: XorShiftRng = SeedableRng::from_seed([1981, 1986,
                                                       2003, 2011]);
    let mut hash_map = HashMap::new();

    let mut insert_empty = 0;
    let mut insert_present = 0;
    let mut get_fail = 0;
    let mut get_success = 0;
```

```
        for _ in 0..100_000 {
            let key = rng.gen::<u16>();
            if rng.gen::<bool>() {
                let value = rng.gen::<u32>();
                if hash_map.insert(key, value).is_none() {
                    insert_empty += 1;
                } else {
                    insert_present += 1;
                }
            } else {
                if hash_map.get(&key).is_none() {
                    get_fail += 1;
                } else {
                    get_success += 1;
                }
            }
        }

        println!("INSERT");
        println!("  empty:   {}", insert_empty);
        println!("  present: {}", insert_present);
        println!("LOOKUP");
        println!("  fail:    {}", get_fail);
        println!("  success: {}", get_success);
    }
```

Easy enough and similar in spirit to the benchmark code explored earlier. Now, let's set our baselines. First up, memory allocations:

```
naive_hashmap > valgrind --tool=memcheck target/release/standard
==13285== Memcheck, a memory error detector
==13285== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et
al.
==13285== Using Valgrind-3.12.0.SVN and LibVEX; rerun with -h for
copyright info
==13285== Command: target/release/standard
==13285==
INSERT
  empty:   35217
  present: 15071
LOOKUP
  fail:    34551
  success: 15161
==13285==
==13285== HEAP SUMMARY:
```

```
==13285==     in use at exit: 0 bytes in 0 blocks
==13285==   total heap usage: 7 allocs, 7 frees, 2,032 bytes allocated
==13285==
==13285== All heap blocks were freed -- no leaks are possible
==13285==
==13285== For counts of detected and suppressed errors, rerun with: -v
==13285== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from
0)
```

Seven total allocations, seven total frees, and a grand total of 2,032 bytes allocated. Running memcheck against naive has the same result:

```
==13307== HEAP SUMMARY:
==13307==     in use at exit: 0 bytes in 0 blocks
==13307==   total heap usage: 7 allocs, 7 frees, 2,032 bytes allocated
```

The naive run is noticeably slower, so it's not allocating memory that kills us. As so little memory gets allocated by these programs we'll skip Valgrind massif—it's unlikely to turn up anything useful. Valgrind cachegrind should be interesting though. Here's baseline:

```
naive_hashmap > valgrind --tool=cachegrind --branch-sim=yes
target/release/standard
==13372== Cachegrind, a cache and branch-prediction profiler
==13372== Copyright (C) 2002-2015, and GNU GPL'd, by Nicholas
Nethercote et al.
==13372== Using Valgrind-3.12.0.SVN and LibVEX; rerun with -h for
copyright info
==13372== Command: target/release/standard
==13372==
--13372-- warning: L3 cache found, using its data for the LL
simulation.
INSERT
  empty:   35217
  present: 15071
LOOKUP
  fail:    34551
  success: 15161
==13372==
==13372== I   refs:      25,733,614
==13372== I1  misses:         2,454
```

```
==13372== LLi misses:         2,158
==13372== I1  miss rate:       0.01%
==13372== LLi miss rate:       0.01%
==13372==
==13372== D   refs:       5,400,581  (2,774,345 rd   + 2,626,236 wr)
==13372== D1  misses:       273,218  (  219,462 rd   +    53,756 wr)
==13372== LLd misses:        36,267  (    2,162 rd   +    34,105 wr)
==13372== D1  miss rate:        5.1% (      7.9%     +       2.0%  )
==13372== LLd miss rate:        0.7% (      0.1%     +       1.3%  )
==13372==
==13372== LL refs:          275,672  (  221,916 rd   +    53,756 wr)
==13372== LL misses:         38,425  (    4,320 rd   +    34,105 wr)
==13372== LL miss rate:         0.1% (      0.0%     +       1.3%  )
==13372==
==13372== Branches:       3,008,198  (3,006,105 cond +     2,093 ind)
==13372== Mispredicts:      315,772  (  315,198 cond +       574 ind)
==13372== Mispred rate:        10.5% (     10.5%     +      27.4%   )
```

Let's break this up some:

```
==13372== I   refs:      25,733,614
==13372== I1  misses:         2,454
==13372== LLi misses:         2,158
==13372== I1  miss rate:       0.01%
==13372== LLi miss rate:       0.01%
```

The first section details our instruction cache behavior. `I refs: 25,733,614` tells us that the standard program executed 25,733,614 instructions in all. This is often a useful comparison between closely related implementations, as we'll see here in a bit. Recall that cachegrind simulates a machine with two levels of instruction and data caching, the first level of caching being referred to as `I1` or `D1` for instruction or data caches and the last level cache being prefixed with `LL`. Here, we see the first and last level instruction caches each missed around 2,500 times during our 25 million instruction run. That squares with how tiny our program is. The second section is as follows:

```
==13372== D   refs:       5,400,581  (2,774,345 rd   + 2,626,236 wr)
==13372== D1  misses:       273,218  (  219,462 rd   +    53,756 wr)
```

```
==13372== LLd misses:        36,267  (    2,162 rd   +    34,105 wr)
==13372== D1  miss rate:        5.1% (      7.9%     +       2.0%  )
==13372== LLd miss rate:        0.7% (      0.1%     +       1.3%  )
```

This is the data cache behavior, split into the two cache layers previously discussed and further divided into read and writes. The first line tells us that 5,400,581 total reads and writes were made to the caches, 2,774,345 reads and 2,626,236 writes. These totals are then further divided by first level and last level caches. Here it turns out that the standard library `HashMap` does real well, a `D1` miss rate of 5.1% and a `LLd` miss rate of 0.7%. That last percentage is key: the higher it is, the more our program is accessing main memory. Doing so, as you'll recall from `Chapter 1`, *Preliminaries – Machine Architecture and Getting Started with Rust*, is painfully slow. The third section is as follows:

```
==13372== LL refs:          275,672  (  221,916 rd   +    53,756 wr)
==13372== LL misses:         38,425  (    4,320 rd   +    34,105 wr)
==13372== LL miss rate:         0.1% (      0.0%     +       1.3%  )
```

This focuses on the combined behavior of data and instruction cache accesses to the LL cache. Personally, I don't often find this section illuminating compared to the previously discussed sections. Your mileage may vary. The final section is as follows:

```
==13372== Branches:       3,008,198  (3,006,105 cond +     2,093 ind)
==13372== Mispredicts:      315,772  (  315,198 cond +       574 ind)
==13372== Mispred rate:        10.5% (     10.5%     +      27.4%   )
```

This details the branch misprediction behavior of our program. We're drowning out the standard library's `HashMap` by branching so extensively in the runner program but it will still serve to establish some kind of baseline. The first line informs us that 3,008,198 total branches were taken during execution. The majority—3,006,105—were conditional branches, branches that jump to a location based on some condition. This squares with the number of conditional

statements we have in the runner. The small majority of branches were indirect, meaning they jumped to offsets in memory based on the results of previous instructions. Overall, our branch misprediction rate is 10.5%.

Alright, how does the naive `HashMap` implementation stack up?

```
naive_hashmap > valgrind --tool=cachegrind --branch-sim=yes
target/release/naive
==13439== Cachegrind, a cache and branch-prediction profiler
==13439== Copyright (C) 2002-2015, and GNU GPL'd, by Nicholas
Nethercote et al.
==13439== Using Valgrind-3.12.0.SVN and LibVEX; rerun with -h for
copyright info
==13439== Command: target/release/naive
==13439==
--13439-- warning: L3 cache found, using its data for the LL
simulation.
INSERT
  empty:   35217
  present: 15071
LOOKUP
  fail:    34551
  success: 15161
==13439==
==13439== I   refs:      15,385,395,657
==13439== I1  misses:             2,376
==13439== LLi misses:             2,157
==13439== I1  miss rate:           0.00%
==13439== LLi miss rate:           0.00%
==13439==
==13439== D   refs:       1,609,901,359  (1,453,944,896 rd   +
155,956,463 wr)
==13439== D1  misses:       398,465,518  (  398,334,276 rd   +
131,242 wr)
==13439== LLd misses:            12,647  (        2,153 rd   +
10,494 wr)
==13439== D1  miss rate:           24.8% (         27.4%     +
0.1%  )
==13439== LLd miss rate:            0.0% (          0.0%     +
0.0%  )
==13439==
==13439== LL refs:          398,467,894  (  398,336,652 rd   +
131,242 wr)
```

```
==13439== LL misses:             14,804  (        4,310 rd   +
10,494 wr)
==13439== LL miss rate:             0.0% (          0.0%     +
0.0%  )
==13439==
==13439== Branches:       4,430,578,248  (4,430,540,970 cond +
37,278 ind)
==13439== Mispredicts:          193,192  (      192,618 cond +
574 ind)
==13439== Mispred rate:             0.0% (          0.0%     +
1.5%   )
```

Already there are some standouts. Firstly, naive executed 15,385,395,657 instructions compared to the 25,733,614 of baseline. The naive implementation is simply doing much, much more work than that standard library is. At this point, without looking at any further data, it's reasonable to conclude that the program under inspection is fundamentally flawed: a rethink of the algorithm is in order. No amount of micro-optimization will fix this. But, that was understood; it's why the program is called *naive* to start with. The second major area of concern is that the `D1` cache miss rate is just shy of 20% higher than baseline, not to ignore that there are simply just more reads and writes to the first level cache than at baseline. Curiously, the naive implementation suffers fewer `LLd` cache misses compared to baseline—10,494 to 34,105. No hypothesis there. Skipping on ahead to the branch misprediction section, we find that naive stays on-theme and performs drastically more branches than standard but with a lower total number of mispredictions. This squares with an algorithm dominated by linear seek and compares, as naive is.

Inspecting with Linux perf

It's worth keeping in mind that Valgrind's cachegrind is a simulation. If you have access to a Linux system you can make use of the perf tool to get real, honest numbers about your program's cache performance and more. This is highly recommended and something we'll do throughout this book. Like git, perf is many tools arranged under a banner—perf—with its own subcommands that have their own options.

It is a tool well worth reading the documentation for. Anyhow, what does the standard look like under perf?

```
naive_hashmap > perf stat --event task-clock,context-switches,page-
faults,cycles,instructions,branches,branch-misses,cache-
references,cache-misses target/release/standard > /dev/null

 Performance counter stats for 'target/release/standard':

      6.923765      task-clock (msec)         #    0.948 CPUs utilized
             0      context-switches          #    0.000 K/sec
           633      page-faults               #    0.091 M/sec
    19,703,427      cycles                    #    2.846 GHz
    26,234,708      instructions              #    1.33  insn per cycle
     2,802,334      branches                  #  404.741 M/sec
       290,475      branch-misses             #   10.37% of all branches
       635,526      cache-references          #   91.789 M/sec
     67,304      cache-misses              # 10.590 % of all cache refs

       0.007301775 seconds time elapsed
```

How does the naive implementation stand up?

```
naive_hashmap > perf stat --event task-clock,context-switches,page-
faults,cycles,instructions,branches,branch-misses,cache-
```

```
references,cache-misses target/release/naive > /dev/null

 Performance counter stats for 'target/release/naive':

    1323.724713      task-clock (msec)         #    1.000 CPUs utilized
              1      context-switches          #    0.001 K/sec
            254      page-faults               #    0.192 K/sec
  3,953,955,131      cycles                    #    2.987 GHz
15,390,499,356      instructions              #  3.89  insn per cycle
4,428,637,974      branches                  # 3345.588 M/sec
      204,132      branch-misses             #  0.00% of all branches
    455,719,875      cache-references          #  344.271 M/sec
   21,311      cache-misses                # 0.005 % of all cache refs

       1.324163884 seconds time elapsed
```

This squares pretty well with the Valgrind simulation and leads to the same conclusion: too much work is being done to insert. Too many branches, too many instructions, the well-studied reader will have seen this coming a mile off, it's just worthwhile to be able to put a thing you know to numbers.

A better naive HashMap

How can we do better than our naive implementation? Obviously, there's scope for algorithmic improvement—we could implement any kind of probing—but if we're trying to compete with standard library's HashMap it's likely that we have a specialized reason for doing so. A specialized reason implies we know something unique about our data, or are willing to make a trade-off that a general data structure cannot. Speaking broadly, the main goals one should have when building software at the limit of machine performance are as follows:

1. Improve the underlying algorithm, reducing total work done.
2. Improve cache locality of data accesses. This may mean:
    1. Keeping your working-set in L1 cache
    2. Compressing your data to fit better into cache.
3. Avoid branch mispredictions. This may mean:
    1. Shaving off branches entirely when possible
    2. Constraining the probability distribution of your branches.

How does that apply here? Well, say we knew that for our application `K == u8` but we still have an unconstrained `V`. `u8` is a key with low cardinality and we ought to be able to trade a little memory to build a

faster structure for `u8` keys. Unfortunately, Rust does not yet have type specialization in a stable channel. Type specialization is *very important* for producing high-performance software without breaking abstractions. It allows the programmer to define an abstract interface and implementation for such and then, at a later date, specialize some of the parameterized types into concrete form with a special purpose implementation. Rust RFC 1210 (`https://github.com/rust-lang/rfcs/pull/1210/files#diff-b652f1feca90247198ee29514ac22cf3`) details how specialization in Rust will work and Rust PR 31844 (`https://github.com/rust-lang/rust/issues/31844`) tracks the ongoing implementation, which is to say, all of this is only exposed in nightly. This chapter sticks to stable and so, unfortunately, we'll need to create a new HashMap rather than specializing. The reader is encouraged to try out specialization for themselves. It's quite nice.

We'll park our `HashMapU8` implementation in `naive_hashmap/src/lib.rs`. The implementation is quite small:

```
pub struct HashMapU8<V>
where
    V: ::std::fmt::Debug,
{
    data: [Option<V>; 256],
}

impl<V> HashMapU8<V>
where
    V: ::std::fmt::Debug,
{
    pub fn new() -> HashMapU8<V> {
        let data = unsafe {
            let mut data: [Option<V>; 256] = mem::uninitialized();
            for element in data.iter_mut() {
                ptr::write(element, None)
            }
            data
        };
        HashMapU8 { data: data }
    }

    pub fn insert(&mut self, k: u8, v: V) -> Option<V> {
```

```
            mem::replace(&mut self.data[(k as usize)], Some(v))
        }

        pub fn get(&mut self, k: &u8) -> Option<&V> {
            let val = unsafe { self.data.get_unchecked((*k as usize)) };
            val.as_ref()
        }
    }
```

The idea here is simple—`u8` is a type with such cardinality that we can rework every possible key into an array offset. The value for each key is an `Option<V>`, `None` if no value has ever been set for the key, and `Some` otherwise. No hashing needs to be done and, absent specialization, we drop the type requirements for that. Every `HashMapU8` will reserve `256 * ::core::mem::size_of::<Option<V>>()` bytes. Being that there's unsafe code in this implementation, it's worthwhile doing an AFL run to search for crashes. The interpreter for the specialized map is similar to the naive interpreter, except that we now take care to parse for `u8` keys:

```
    extern crate naive_hashmap;

    use std::io;
    use std::io::prelude::*;
    use std::str::FromStr;

    fn main() {
        let mut hash_map = naive_hashmap::HashMapU8::new();

        let n = io::stdin();
        for line in n.lock().lines() {
            if let Ok(line) = line {
                let mut cmd = line.split(" ");
                match cmd.next() {
                    Some("LOOKUP") => {
                        if let Some(key) = cmd.next() {
                            if let Ok(key) = u8::from_str(key) {
                                let _ = hash_map.get(&key);
                            } else {
                                continue;
                            }
                        } else {
                            continue;
```

```
                    }
                }
                Some("INSERT") => {
                    if let Some(key) = cmd.next() {
                        if let Ok(key) = u8::from_str(key) {
                            if let Some(val) = cmd.next() {
                          let _ = hash_map.insert(key,
                           val.to_string());
                            } else {
                                continue;
                            }
                        }
                    } else {
                        continue;
                    }
                }
                _ => continue,
            }
        } else {
            break;
        }
    }
}
```

I'll spare you the AFL output but, as a reminder, here's how you run the specialized interpreter through:

```
> cargo afl build --release
> cargo afl fuzz -i resources/in/ -o resources/out/
target/release/specialized_interpreter
```

Producing a criterion benchmark will be very similar to the approach taken for the naive implementation, save that we'll swap out a few names here and there. We'll skip listing the code with the hopes that you'll be able to reproduce it as desired. The results, however, are promising:

```
HashMap/100000/speciali time:   [662.01 us 665.28 us 668.44 us]
HashMap/100000/standard time:   [2.3471 ms 2.3521 ms 2.3583 ms]

HashMap/10000/specializ time:   [64.294 us 64.440 us 64.576 us]
HashMap/10000/standard  time:   [253.14 us 253.31 us 253.49 us]
```

In a like manner to `naive.rs` and `standard.rs` from previously, we've also got a `specialized.rs` runner which, to avoid duplication, we'll avoid listing here. Let's run specialized through Valgrind cachegrind:

```
naive_hashmap > valgrind --tool=cachegrind --branch-sim=yes
target/release/specialized
==24235== Cachegrind, a cache and branch-prediction profiler
==24235== Copyright (C) 2002-2015, and GNU GPL'd, by Nicholas
Nethercote et al.
==24235== Using Valgrind-3.12.0.SVN and LibVEX; rerun with -h for
copyright info
==24235== Command: target/release/specialized
==24235==
--24235-- warning: L3 cache found, using its data for the LL
simulation.
INSERT
  empty:   256
  present: 50032
LOOKUP
  fail:    199
  success: 49513
==24235==
==24235== I   refs:      5,200,051
==24235== I1  misses:        2,248
==24235== LLi misses:        2,068
==24235== I1  miss rate:      0.04%
==24235== LLi miss rate:      0.04%
==24235==
==24235== D   refs:        443,562  (233,633 rd   + 209,929 wr)
==24235== D1  misses:        4,736  (  3,249 rd   +   1,487 wr)
==24235== LLd misses:        3,512  (  2,112 rd   +   1,400 wr)
==24235== D1  miss rate:       1.1% (    1.4%     +     0.7%  )
==24235== LLd miss rate:       0.8% (    0.9%     +     0.7%  )
==24235==
==24235== LL refs:           6,984  (  5,497 rd   +   1,487 wr)
==24235== LL misses:         5,580  (  4,180 rd   +   1,400 wr)
==24235== LL miss rate:        0.1% (    0.1%     +     0.7%  )
==24235==
==24235== Branches:        393,599  (391,453 cond +   2,146 ind)
==24235== Mispredicts:      57,380  ( 56,657 cond +     723 ind)
==24235== Mispred rate:       14.6% (   14.5%     +    33.7%   )
```

Compared to the standard `valgrind` run, we're doing around 1/5 the total number of instructions and with substantially fewer `D1` and `LLd` misses. No surprise here. Our *hash* for `HashMapU8` is an exceedingly cheap pointer offset and the size of the storage is going to fit comfortably into the cache. Linux perf tells a similar story:

```
naive_hashmap > perf stat --event task-clock,context-switches,page-
faults,cycles,instructions,branches,branch-misses,cache-
references,cache-misses target/release/specialized > /dev/null

 Performance counter stats for 'target/release/specialized':

     1.433884      task-clock (msec)         #    0.788 CPUs utilized
            0      context-switches          #    0.000 K/sec
          104      page-faults               #    0.073 M/sec
    4,138,436      cycles                    #    2.886 GHz
    6,141,529      instructions              #    1.48  insn per cycle
      749,155      branches                  #  522.466 M/sec
       59,914      branch-misses             #    8.00% of all branches
       74,760      cache-references          #   52.138 M/sec

       0.001818537 seconds time elapsed
```

Phew! Let's summarize our efforts:

name	Task Clock (ms)	Instructions	Branches	Branch Misses	Cache References	Cache Misses
specialized	1.433884	6,141,529	749,155	59,914	74,760	n/a
standard	6.923765	26,234,708	2,802,334	290,475	635,526	67,304
naive	1323.7	15,390,4	4,428,63	204,	455,71	21,

	24713	99,356	7,974	132	9,875	311

Summary

What should we understand from all of this? To produce software that operates at the edge of the machine's ability, you must understand some important things. Firstly, if you aren't measuring your program, you're only guessing. Measuring runtime, as criterion does, is important but a coarse insight. *Where is my program spending its time?* is a question the Valgrind suite and perf can answer, but you've got to have benchmarks in place to contextualize your questions. Measuring and then validating behavior is also an important chunk of this work, which is why we spent so much time on QuickCheck and AFL. Secondly, have a goal in mind. In this chapter, we've made the speed of standard library `HashMap` our goal but, in an actual code base, there's always going to be places to polish and improve. What matters is knowing what needs to happen to solve the problem at hand, which will tell you where your time needs to be spent. Thirdly, understand your machine. Modern superscalar, parallel machines are odd beasts to program and, without giving a thought to their behaviors, it's going to be tough understanding why your program behaves the way it does. Finally, algorithms matter above all else. Our naive `HashMap` failed to perform well because it was a screwy idea to perform an average O(n/2) operations for every insertion, which we proved out in practice. Standard library's `HashMap` is a good, general-purpose structure based on linear probing and clearly, a lot of thought went into making it function well for a variety of cases. When your program is too slow, rather than micro-optimizing, take a step back and consider the problem space. Are there better algorithms available, is there some insight into the data that can be exploited to shift the algorithm to some other direction entirely?

That's performance work in a nutshell. Pretty satisfying, in my opinion.

Further reading

In this chapter, we covered measuring and improving the performance of a serial Rust program while demonstrating the program's fitness for purpose. This is a huge area of work and there's a deep well of literature to pull from.

- *Rust's std::collections is absolutely horrible*, available at `https://www.reddit.com/r/rust/comments/52grcl/rusts_stdcollections_is_absolutely_horrible/`. The original poster admitted the title is a bit on the click-baity side but the discussion on Reddit is well worth reading. The original author of standard library's `HashMap` weighs in on the design decisions in the implementation.

- *Robin Hood Hashing*, 1985, Pedro Celis. This thesis introduced the Robin Hood hashing strategy for constructing associative arrays and is the foundation for the implementation you'll find in Rust. The paper also goes into further search strategies that didn't find their way into Rust's implementation but should be of interest to readers with ambitions toward building hashing search structures.

- *Robin Hood hashing*, Emmanuel Goossaert, available at `http://codecapsule.com/2013/11/11/robin-hood-hashing/`. The Rust standard library HashMap makes continued reference to this blog post and its follow-on, linked in the text. The description

here is of a higher-level than that of Celis' thesis and potentially easier to understand as a result.

- *Denial of Service via Algorithmic Complexity Attacks*, 2003, Scott Crosby and Dan Wallach. This paper outlines a denial of service attack on network services by exploiting algorithmic blow-ups in their implementations. The consequence of this paper influenced Rust's decision to ship a safe-by-default HashMap.
- *QuickCheck: Lightweight Tool for Random Testing of Haskell Programs*, 2000, Koen Claessen and John Hughes. This paper introduces the QuickCheck tool for Haskell and introduces property-based testing to the world. The research here builds on previous work into randomized testing but is novel for realizing that computers had got fast enough to support type-directed generation as well as shipping with the implementation in a single page appendix. Many, many subsequent papers have built on this one to improve the probing ability of property testers.
- *An Evaluation of Random Testing*, 1984, Joe Duran and Simeon Ntafos. The 1970s and 1980s were an interesting time for software testing. Formal methods were seen as being just around the corner and the preferred testing methods relied on intimate knowledge of the program's structure. Duran and Ntafos evaluated the ideal techniques of the day against random generation of data and found that randomness compared favorably with significantly less programmer effort. This paper put random testing on the map.

- *Experiences with QuickCheck: Testing the Hard Stuff and Staying Sane*, 2016, John Hughes. This paper is a follow-on to the original QuickCheck paper by Claessen and Hughes in which Hughes describes his subsequent fifteen years of experience doing property testing. The techniques laid out in this paper are a significant evolution of those presented in the 2000 paper and well-worth studying by anyone doing property tests as a part of their work. That ought to be most people, is my take.

- *American Fuzzy Lop website*, available at `http://lcamtuf.coredump.cx/afl`. AFL is the product of a long tail of research into efficiently mutating inputs for the purpose of triggering bugs. As of writing this book, it is best of breed and has a long trophy list to show for it. The website has links to AFL's documentation and relevant research to understand its function in deeper detail.
- *Compact Data Structures: A Practical Approach*, 2016, Gonzalo Navarro. One of the major techniques of exploiting cache locality is to shrink the individual elements of a working set, implying more elements are available in the working set. Compact data structures, those that can be operated on, at, or near their information theory minimal representation, is an ongoing and exciting area. Navarro's book is excellent and well-worth studying for anyone who is interested in exploring this avenue of optimization.

- *vec_map*, various authors. `vec_map` is a Rust crate that exploits the same ideas as this chapter's `HashMapU8` but in a generic implementation, with full compatibility to the standard library HashMap. The source code is quite interesting and warmly recommended.
- *Reevaluating Amdahl's Law*, 1988, John Gustafson. This is an exceptionally short paper and clearly explains Amdahl's formulation as well as Gustafson's objection to its underlying assumptions. That the paper is describing an interpretation in which the serial portion is shrunk is clear only after a few readings, or once some kind soul explains this to you.
- *Tracking issue for specialization (RFC 1210)*, available at `https://github.com/rust-lang/rust/issues/31844`.This issue is a pretty good insight into the way the Rust community goes about stabilizing a major feature. The original RFC is from 2016. Pretty much ever since the point it was accepted that there's been a feature flag in nightly for experimentation and a debate on the consequences of making the work stable.

The Rust Memory Model – Ownership, References and Manipulation

In the previous chapter, `Chapter 2`, *Sequential Rust Performance and Testing*, we discussed factors that contribute or detract from the serial performance of a Rust program. We did not explicitly address concurrent performance for want of sufficient information about the way Rust's abstract memory model interacts with the real memory hierarchy of a machine. In this chapter, we'll discuss Rust's memory model, how to control the layout of types in memory, how types are aliased, and how Rust's memory safety works. We'll dig into the standard library to understand how this plays out in practice. This chapter will also examine common crates in the ecosystem that will be of interest to us later in this book. Please be aware that by the time you read this chapter, the `rustc` implementation will have changed, potentially making our code listings here no longer square with the naming patterns in `rustc` itself. If you wish to follow along, please check out Rust at SHA `da569fa9ddf8369a9809184d43c600dc06bd4b4d`.

By the close of this chapter, we will have:

- Investigated how Rust lays objects out in memory
- Discussed the various ways Rust points to memory and their guarantees
- Discussed how Rust allocates and deallocates memory
- Discussed how Rust denotes stack and heap allocations

- Investigated the internal implementation of `Option`, `Cell`, `CellRef` , `Rc` and `Vec`.

Technical requirements

This chapter requires a working Rust installation. The details of verifying your installation are covered in `Chapter 1`, *Preliminaries – Machine Architecture and Getting Started with Rust*. No additional software tools are required.

You can find the source code for this book's projects on GitHub: `https://github.com/PacktPublishing/Hands-On-Concurrency-with-Rust`. This chapter has its source code under `Chapter03`.

Memory layout

Rust has a handful of mechanisms to lay out compound types in memory. They are as follows:

- Arrays
- Enums
- Structs
- Tuples

Exactly how these are laid out in memory depends on the representation chosen. By default, everything in Rust is `repr(Rust)`. All `repr(Rust)` types are aligned on byte boundaries to the power of two. Every type is at least one byte in memory, then two, then four, and so forth. Primitives—`u8`, `usize`, `bool`, and `&T`—are aligned to their size. In Rust, representation structures have alignment according to the largest field. Consider the following struct:

```
struct AGC {
  elapsed_time2: u16,
  elapsed_time1: u16,
  wait_list_upper: u32,
  wait_list_lower: u16,
  digital_autopilot: u16,
  fine_scale: u16
}
```

`AGC` is aligned to `u32` with padding inserted as appropriate to match that 32-bit alignment. Rust will re-order fields to achieve maximal

packing. Enums are different, being subject to a host of optimizations, most notably null pointer optimization. See the following enumeration:

```
enum AGCInstruction {
  TC,
  TCF,
  CCS(u8),
  B(u16),
  BZF,
}
```

This will be laid out as follows:

```
struct AGCInstructionRepr {
  data: u16,
  tag: u8,
}
```

The `data` field is wide enough to accommodate the largest inner value and the `tag` allows discrimination between variants. Where this gets complicated is in the case of an enum that holds a non-nullable pointer, and other variants cannot refer to the same. `Option<&T>` means if a null pointer is discovered when dereferencing the option Rust can assume that the `None` variant was discovered. Rust will optimize away the tag for `Option<&T>`.

Rust supports other representations. `repr(C)` lays out types in memory in a manner that C would do and is often used in FFI projects, as we'll see later in this book. `repr(packed)` lays types out in memory like `repr(Rust)` except that no padding is added, and alignment occurs only to the byte. This representation is likely to cause unaligned loads and a severe effect on performance of common CPUs, certainly the two CPU architectures we concern ourselves with in this book. The remaining representations have to do with forcing the size of fieldless enumerations—that is, enumerations that have no data in their

variants—and these are useful for forcing the size of such an enum with an eye towards ABI compatability.

Rust allocations happen by default on the hardware stack, as is common for other low-level languages. Heap allocations must be performed explicitly by the programmer or be done implicitly when creating a new type that holds some sort of internal storage. There are complications here. By default, Rust types obey move semantics: the bits of the type are moved as appropriate between contexts. For example:

```
fn project_flights() -> Vec<(u16, u8)> {
    let mut t = Vec::new();
    t.push((1968, 2));
    t.push((1969, 4));
    t.push((1970, 1));
    t.push((1971, 2));
    t.push((1972, 2));
    t
}

fn main() {
    let mut total: u8 = 0;
    let flights = project_flights();
    for &(_, flights) in flights.iter() {
        total += flights;
    }
    println!("{}", total);
}
```

The `project_flights` function allocates a new `Vec<(u16, u8)>` on the heap, populates it, and then returns ownership of the heap-allocated vector to the caller. This does not mean that the bits of `t` are copied from the stack frame of `project_flights` but, instead, that the pointer `t` is returned from the `project_flights` stack to `main`. It is possible to achieve copy semantics in Rust through the use of the `Copy` trait. `Copy` types will have their bits copied in memory from one place to the other. Rust primitive types are `Copy`—copying them is as fast as moving them, especially when the type is smaller than the native pointer. It's

possible to implement `Copy` for your own type unless your type implements `Drop`, the trait that defines how a type deallocates itself. This restriction eliminates—in Rust code not using `unsafe` —the possibility of double frees. The following code block derives `Copy` for two user-defined types and is an example of a poor random generator:

```
#[derive(Clone, Copy, PartialEq, Eq)]
enum Project {
    Apollo,
    Gemini,
    Mercury,
}

#[derive(Clone, Copy)]
struct Mission {
    project: Project,
    number: u8,
    duration_days: u8,
}

fn flight() -> Mission {
    Mission {
        project: Project::Apollo,
        number: 8,
        duration_days: 6,
    }
}

fn main() {
    assert_eq!(::std::mem::size_of::<Mission>(), 3);
    let mission = flight();
    if mission.project == Project::Apollo && mission.number == 8 {
        assert_eq!(mission.duration_days, 6);
    }
}
```

Pointers to memory

Rust defines several kinds of pointers, each with a specific purpose. `&T` is a shared reference and there may be many duplicates of that shared reference. The owner of `&T` does not necessarily, own `T` and may not modify it. Shared references are immutable. Mutable references—written `&mut T`—also do not imply that the other `&mut T` owns `T` necessarily but the reference may be used to mutate `T`. There may be only one reference for any `T` with a `&mut T`. This complicates some code but means that Rust is able to prove that two variables do not overlap in memory, unlocking a variety of optimization opportunities absent from C/C++. Rust references are designed so that the compiler is able to prove the liveness of the referred to type: references cannot dangle. This is not true of Rust's raw pointer types —`*const T` and `*mut T`—which work analogously to C pointers: they are, strictly, an address in memory and no guarantees about the data at that address are made. As such, many operations on raw pointers require the `unsafe` keyword and they are almost always seen solely in the context of performance-sensitive code or FFI interfaces. Put another way, a raw pointer may be null; a reference may never be null.

The rules around references often cause difficulty in situations where an immutable borrow is accidentally made of a mutable reference. The Rust documentation uses the following small program to illustrate the difficulty:

```
fn main() {
    let mut x: u8 = 5;
    let y: &mut u8 = &mut x;

    *y += 1;
```

```
    println!("{}", x);
}
```

The `println!` macro takes its arguments by reference, implicitly here, creating a `&x`. The compiler rejects this program as `y: &mut u8` is invalid. Were this program to compile, we would be subject to a race between the update of `y` and the read of `x`, depending on the CPU and memory ordering. The exclusive nature of references could be potentially limiting when working with structures. Rust allows programs to split borrows for a structure, providing that the disjoint fields cannot be aliased.

We demonstrate this in the following brief program:

```
use std::fmt;

enum Project {
    Apollo,
    Gemini,
    Mercury,
}

impl fmt::Display for Project {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match *self {
            Project::Apollo => write!(f, "Apollo"),
            Project::Mercury => write!(f, "Mercury"),
            Project::Gemini => write!(f, "Gemini"),
        }
    }
}

struct Mission {
    project: Project,
    number: u8,
    duration_days: u8,
}

fn main() {
    let mut mission = Mission {
        project: Project::Gemini,
```

```
            number: 2,
            duration_days: 0,
        };
        let proj: &Project = &mission.project;
        let num: &mut u8 = &mut mission.number;
        let dur: &mut u8 = &mut mission.duration_days;

        *num = 12;
        *dur = 3;

        println!("{} {} flew for {} days", proj, num, dur);
    }
```

This same trick is difficult to impossible for general container types. Consider a map where two keys map to the same referenced `T`. Or, for now, a slice:

```
use std::fmt;

enum Project {
    Apollo,
    Gemini,
    Mercury,
}

impl fmt::Display for Project {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match *self {
            Project::Apollo => write!(f, "Apollo"),
            Project::Mercury => write!(f, "Mercury"),
            Project::Gemini => write!(f, "Gemini"),
        }
    }
}

struct Mission {
    project: Project,
    number: u8,
    duration_days: u8,
}

fn main() {
    let mut missions: [Mission; 2] = [
        Mission {
            project: Project::Gemini,
```

```
                number: 2,
                duration_days: 0,
            },
            Mission {
                project: Project::Gemini,
                number: 12,
                duration_days: 2,
            },
        ];

        let gemini_2 = &mut missions[0];
        let _gemini_12 = &mut missions[1];

        println!(
            "{} {} flew for {} days",
            gemini_2.project, gemini_2.number, gemini_2.duration_days
        );
    }
```

This program fails to compile with the following error:

```
> rustc borrow_split_array.rs
error[E0499]: cannot borrow `missions[..]` as mutable more than once at
a time
  --> borrow_split_array.rs:40:27
   |
39 |     let gemini_2 = &mut missions[0];
   |                         ----------- first mutable borrow occurs
here
40 |     let _gemini_12 = &mut missions[1];
   |                           ^^^^^^^^^^ second mutable borrow occurs
here
...
46 | }
   | - first borrow ends here

error: aborting due to previous error
```

We, the programmers, know that this was safe—`gemini_2` and `gemini_12` don't overlap in memory—but it's not possible for the compiler to prove this. What if we had done the following:

```
use std::fmt;

enum Project {
    Apollo,
    Gemini,
    Mercury,
}

impl fmt::Display for Project {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        match *self {
            Project::Apollo => write!(f, "Apollo"),
            Project::Mercury => write!(f, "Mercury"),
            Project::Gemini => write!(f, "Gemini"),
        }
    }
}

struct Mission {
    project: Project,
    number: u8,
    duration_days: u8,
}

fn main() {
    let gemini_2 = Mission {
        project: Project::Gemini,
        number: 2,
        duration_days: 0,
    };

    let mut missions: [&Mission; 2] = [&gemini_2, &gemini_2];

    let m0 = &mut missions[0];
    let _m1 = &mut missions[1];

    println!(
        "{} {} flew for {} days",
        m0.project, m0.number, m0.duration_days
    );
}
```

By definition, `missions[0]` and `missions[1]` overlap in memory. We, the programmers, know we're breaking the aliasing rules and the compiler, being conservative, assumes that the rules are being broken.

Allocating and deallocating memory

Deallocation happens in one of two ways, depending on whether the type is allocated on the stack or the heap. If it's on the stack, the type is deallocated when the stack frame itself ceases to exist. Each Rust stack frame comes into the world fully allocated but uninitialized and exits the world when its associated function exits. Heap allocated types are deallocated when the last valid binding moves out of scope, either through the natural flow of the program or by an explicit `std::mem::drop` being called by the programmer. The implementation of `drop` is as follows:

```
fn drop<T>(_x: T) { }
```

The value `_x` is moved into `drop` —meaning there are no other borrows of `_x`—and then immediately falls out of scope when `drop` exits. An explicit `drop` is not able to remove items from scope, however, so subtle interactions with structures where the Rust compiler is not able to prove non-overlapping aliases and the like will happen. The `drop` documentation discusses several cases and it is worth reviewing that material.

Any Rust type that can be deallocated—meaning it is not `Copy`—will implement the `Drop` trait, a trait whose sole function is `drop(&mut self)`. `Drop::drop` cannot be called explicitly and is called when the type goes out of scope or is invocable by `std::mem::drop`, as discussed previously.

So far in this chapter, we've discussed the layout of types in memory and allocation on the heap but without fully discussing how

allocation itself works in the Rust memory model. The simplest way of forcing a heap allocation is with `std::boxed::Box`. In fact, the Rust documentation for `Box`—which is also just called box—describes it as the simplest form of heap allocation in Rust. That is, a `Box<T>` allocates enough space on the heap for a type `T` and acts as the owner of that allocation. When the box is dropped, the drop of `T` occurs. Here's the definition of `Box<T>` straight from the standard library, in the file `src/liballoc/boxed.rs`:

```
pub struct Box<T: ?Sized>(Unique<T>);
```

The size of a type

There's two important things here we have not come across yet in this book—`Sized` and `Unique`. First, `Sized`, or more properly, `std::marker::Sized`. `Sized`, is a Rust `trait` that bounds a type to have a known size at compile time. Almost everything in Rust has an implicit `Sized` bound, which we can inspect. For instance:

```
use std::mem;

#[allow(dead_code)]
enum Project {
    Mercury { mission: u8 },
    Gemini { mission: u8 },
    Apollo { mission: u8 },
    Shuttle { mission: u8 },
}

fn main() {
    assert_eq!(1, mem::size_of::<u8>());
    assert_eq!(2, mem::size_of::<Project>());

    let ptr_sz = mem::size_of::<usize>();
    assert_eq!(ptr_sz, mem::size_of::<&Project>());

    let vec_sz = mem::size_of::<usize>() * 2 + ptr_sz;
    assert_eq!(vec_sz, mem::size_of::<Vec<Project>>());
}
```

`u8` is a single byte, Project is the byte to distinguish the enum variants plus the inner mission byte, pointers are the size of a machine word–a `usize-` and a `Vec<T>` is guaranteed to be a pointer and two `usize` fields no matter the size of `T`. There's something interesting going on with `Vec<T>` and we'll get into it in depth later in this chapter. Note that we said almost everything in Rust has an implicit `Sized` bound. Rust supports *dynamically sized types*, these being types that have no

known size or alignment. Rust requires known size and alignment and so all DSTs must exist behind a reference or pointer. A slice—a view into contiguous memory—is one such type. In the following program, the compiler will not be able to determine the size of the slice of values:

```
use std::time::{SystemTime, UNIX_EPOCH};

fn main() {
    let values = vec![0, 1, 2, 3, 4, 5, 7, 8, 9, 10];
    let cur: usize = SystemTime::now()
        .duration_since(UNIX_EPOCH)
        .unwrap()
        .as_secs() as usize;
    let cap: usize = cur % values.len();

    let slc: &[u8] = &values[0..cap];

    println!("{:?}", slc);
}
```

Slices are a view into a block of memory represented as a pointer and a length, as the documentation for the primitive type slice puts it. The trick is that the length is determined at runtime. The following program will compile:

```
fn main() {
    let values = vec![0, 1, 2, 3, 4, 5, 7, 8, 9, 10];
    let slc: &[u8] = &values[0..10_000];

    println!("{:?}", slc);
}
```

However, the previous code will panic at runtime:

```
> ./past_the_end thread
'main' panicked at 'index 10000 out of range for slice of length 10',
libcore/slice/mod.rs:785:5 note: Run with `RUST_BACKTRACE=1` for a
backtrace.
```

Rust allows programmers to include DSTs as the last field of a `struct`, like so:

```
struct ImportantThing {
  tag: u8,
  data: [u8],
}
```

However, this causes the struct itself to become a DST.

Static and dynamic dispatch

Trait objects likewise have no statically known size in Rust. Trait objects are the mechanism by which Rust performs dynamic dispatch. Preferentially, Rust is a static dispatch language as there are increased opportunities for inlining and optimization—the compiler simply knows more. Consider the following code:

```
use std::ops::Add;

fn sd_add<T: Add<Output=T>>(x: T, y: T) -> T {
    x + y
}

fn main() {
    assert_eq!(48, sd_add(16 as u8, 32 as u8));
    assert_eq!(48, sd_add(16 as u64, 32 as u64));
}
```

Here, we define a function, `sd_add<T: Add<Output=T>>(x: T, y: T) -> T`. Rust, like C++, will perform monomorphization at compile time, emitting two `sd_add` functions, one for `u8` and the other for `u64`. Like C++, this potentially increases the code size of a Rust program and slows compilation but at the benefit of allowing inlining at the caller site, potentially more efficient implementations owing to type specialization, and fewer branches.

When static dispatch is not desirable, the programmer can construct trait objects to perform dynamic dispatch. Trait objects do not have a known size—indeed, they can be any `T` that implements the trait—and so, like slices, must exist behind a kind of pointer. Dynamic dispatch will not see much use in this book. The reader is warmly encouraged

to consult the documentation for `std::raw::TraitObject` for full details on Rust's trait object notion.

Zero sized types

Rust also supports *zero sized types* or ZSTs; structs with no fields, the unit type `()`, or arrays with no size are all zero sized. The fact that the type has no size is a boon to optimization and dead-tree removal. Rust APIs often include return types like so: `Result<(), a_mod::ErrorKind>`. This type signals that while the function may error, its return value in the happy path is the unit type. These types are somewhat rare in practice but unsafe, and Rust must be aware of them. Many allocators return null when asked to allocate zero bytes —making the allocation of a ZST indistinguishable from the allocator being unable to find free memory—and pointer offsets from a ZST are of zero offset. Both of these considerations will be important in this chapter.

Boxed types

That's the trait `Sized`, but what does `?Sized` mean? The `?` flags the relevant trait as optional. So, a box is a type in Rust parameterized over some other type `T` which may or may not have a size. A box is a kind of a pointer to heap allocated storage. Let's look into its implementation further. What of `Unique<T>`? This type is a signal to the Rust compiler that some `*mut T` is non-null and that the unique is the sole owner of `T`, even though `T` was allocated outside the `Unique`. `Unique` is defined like so:

```
pub struct Unique<T: ?Sized> {
    pointer: NonZero<*const T>,
    // NOTE: this marker has no consequences for variance, but is
    // necessary for dropck to understand that we logically
    // own a `T`.
    _marker: PhantomData<T>,
}
```

`NonZero<T>` is a struct that the `rustc` source describes as a wrapper type for raw pointers and integers that will never be `NULL` or `0` that might allow certain optimizations. It's annotated in a special way to admit those null pointer optimizations discussed elsewhere in this chapter. Unique is also of interest for its use of `PhantomData<T>`. `PhantomData` is, in fact, a zero sized type, defined as `pub struct PhantomData<T:?Sized>;`. This type instructs the Rust compiler to consider `PhantomData<T>` as owning `T` even though, ultimately, there's nowhere for `PhantomData` to store its newfound `T`. This works well for `Unique<T>`, which must take ownership of `<T>` by maintaining a non-zero constant pointer to `T` but does not, itself, have `T` stored anywhere other than in the heap. A box is then, a unique, non-null pointer to a thing allocated somewhere in memory but not inside the storage space of the box.

The internals of box are compiler intrinsics: they sit at the interplay of the allocator and are a special consideration in Rust's borrow checker. With that in mind, we will avoid chasing down the internal details of `Box` as they will change from compiler version to compiler version and this book is explicitly not a `rustc` internals book. For our purposes, however, it is worth considering the API exposed by box. The key functions are:

- `fn from_raw(raw: *mut T) -> Box<T>`
- `fn from_unique(u: Unique<T>) -> Box<T>`
- `fn into_raw(b: Box<T>) -> *mut T`
- `fn into_unique(b: Box<T>) -> Unique<T>`
- `fn leak<'a>(b: Box<T>) -> &'a mut T`

Both `from_raw` and `from_unique` are unsafe. Conversion from a raw pointer is unsafe if a raw pointer is *boxed* more than once or if a box is made from a pointer that overlaps with another, as examples. There are other possibilities. Conversion from a `Unique<T>` is unsafe as the `T` may or may not be owned by `Unique`, resulting in a possibility of the box not being the sole owner of its memory. The `into_*` functions, however, are safe in the sense that the resulting pointers will be valid but the caller will not have full responsibility for managing the lifetime of the memory. The `Box` documentation notes that the caller can release the memory themselves or convert the pointer back into the type they came from and allow Rust to do it for them. The latter is the approach this book will take. Finally, there's `leak`. Leak is a fun one and is not available on stable channel but is worth discussing for applications that will ship to embedded targets. A common memory management strategy for embedded systems is to pre-allocate all necessary memory and only operate on that memory for the lifetime of the program. In Rust, this is trivially accomplished if you desire

uninitialized memory of a constant size: arrays and other primitive types. In the event you desire heap allocations at the start of your program, the situation is more complicated. That's where leak comes in: it causes memory to leak from a box—a heap allocation—to wherever you please. When the leaked memory is intended to live for the lifetime of the program—into the `static` lifetime—there's no issue. An example is as follows, straight from the docs for `leak`:

```
#![feature(box_leak)]

fn main() {
    let x = Box::new(41);
    let static_ref: &'static mut usize = Box::leak(x);
    *static_ref += 1;
    assert_eq!(*static_ref, 42);
}
```

Here, we see a new `usize` allocated on the heap, leaked into `static_ref`—a mutable reference of static lifetime—and then fiddled with through the remaining lifetime of the program.

Custom allocators

It is possible to plug in one's own allocator to Rust, as can be done in C++ or similar system-level programming languages. By default, on most targets, Rust uses jemalloc with a backup alternative to the system-provided allocator. In embedded applications, there may not *be* a system, let alone an allocator, and the interested reader is recommended to peruse RFC 1183 (`https://github.com/rust-lang/rfcs/blob/master/text/1183-swap-out-jemalloc.md`), RFC 1398 (`https://github.com/rust-lang/rfcs/pull/1398`), and related RFCs. As of writing this, an interface for plugging in custom allocators to stable Rust is under active discussion and any such capability is only available in nightly Rust. We will not make use of custom allocators in this book.

Implementations

In `Chapter 2`, *Sequential Rust Performance and Testing*, we very briefly dipped into the implementation of `std::collections::HashMap`. Let's continue with that approach of dissecting the standard library, paying special attention to the concerns of memory that pop up.

Option

Let's examine `Option<T>`. We've already discussed `Option<T>` in this chapter; that it's subject to *null pointer optimization* on account of its empty `None` variant in particular. `Option` is as simple as you might imagine, being defined in `src/libcore/option.rs`:

```
#[derive(Clone, Copy, PartialEq, PartialOrd, Eq, Ord, Debug, Hash)]
#[stable(feature = "rust1", since = "1.0.0")]
pub enum Option<T> {
    /// No value
    #[stable(feature = "rust1", since = "1.0.0")]
    None,
    /// Some value `T`
    #[stable(feature = "rust1", since = "1.0.0")]
    Some(#[stable(feature = "rust1", since = "1.0.0")] T),
}
```

As is often the case with Rust internals, there are a great deal of flags around to control when and where new features land in which channel and how documentation is generated. A slightly tidier expression of `Option<T>` is:

```
#[derive(Clone, Copy, PartialEq, PartialOrd, Eq, Ord, Debug, Hash)]
pub enum Option<T> {
    None,
    Some(T),
}
```

This is refreshingly simple, much like how you may have implemented an option type yourself up on first thinking of it. `Option` is the owner of its inner `T` and is able to pass out references to that inner data per the usual restrictions. As an example:

```
pub fn as_mut(&mut self) -> Option<&mut T> {
    match *self {
        Some(ref mut x) => Some(x),
        None => None,
    }
}
```

Rust exposes a special trait type inside each `trait: Self`. It desugars simply to the referent trait type, in this case, `Option<T>`. The `&mut self` is shorthand for `self: &mut Self`, as is `&self` for `self: &Self`. `as_mut` is then allocating a new option whose inner type is a mutable reference to the original inner type. Now, consider the humble `map`:

```
pub fn map<U, F: FnOnce(T) -> U>(self, f: F) -> Option<U> {
    match self {
        Some(x) => Some(f(x)),
        None => None,
    }
}
```

Rust allows closures as arguments to functions. In this way, Rust is similar to higher-level and, especially, functional programming languages. Unlike these higher-level programming languages, Rust has restrictions on closures in terms of the way *variable capture* occurs and with regard to call totals and mutability. Here, we see the `FnOnce` trait being used, restricting the closure being passed in as `f` to the map function as being single-use. The function traits, all defined in `std::ops`, are:

- `Fn`
- `FnMut`
- `FnOnce`

The first trait, `Fn`, is described by the Rust documentation as being a *call operator that takes an immutable receiver*. This is maybe a little obscure until we look at the definition of `Fn` in `src/libcore/ops/function.rs`:

```
pub trait Fn<Args>: FnMut<Args> {
    /// Performs the call operation.
    #[unstable(feature = "fn_traits", issue = "29625")]
    extern "rust-call" fn call(&self, args: Args) -> Self::Output;
}
```

Now it's more obscure! But, we can work that back. `Args` is distinct from `std::env::Args` but plays a similar role of being a placeholder for function arguments, save at a type-level. `: FnMut<Args>` means that the `FnMut<Args>` is a *supertrait* of `Fn<Args>`: all of the methods available to `FnMut` are available to `Fn` when used as a trait object. Recall that trait objects find use in dynamic dispatch, discussed previously. This also means that any instance of `Fn` can be used where an `FnMut` is expected, in cases of static dispatch. Of particular interest to understanding `Fn` is:

```
extern "rust-call" fn call(&self, args: Args) -> Self::Output;
```

We'll approach this in parts. Firstly, `extern "rust-call"`. Here, we are defining an inline extern block that uses the `"rust-call"` ABI. Rust supports many ABIs, three of which are cross-platform and guaranteed to be supported no matter the platform:

- `extern "Rust" fn`, implicit for all Rust functions unless otherwise specified
- `extern "C" fn`, often used in FFI and shorthanded to `extern fn`
- `extern "system" fn`, equivalent to `extern "C" fn` save for some special platforms

`"rust-call"` does not appear in that list because it is a rust-specific ABI, which also includes:

- `extern "rust-intrinsic" fn`, specific to `rustc` intrinsics
- `extern "rust-call" fn`, the ABI for all `Fn::call` functions
- `extern "platform-intrinsic" fn`, which the documentation notes as being something the programmer should never have to deal with

Here, we're signalling to the Rust compiler that the function is to be treated with a special call ABI. This particular `extern` is important when writing traits that implement the `Fn` trait, as box will when the unstable `fnbox` feature flag is enabled:

```
impl<'a, A, R> FnOnce<A> for Box<FnBox<A, Output = R> + 'a> {
    type Output = R;

    extern "rust-call" fn call_once(self, args: A) -> R {
        self.call_box(args)
    }
}
```

Secondly, `fn call(&self, args: Args)`. The implementing type is taken in as an immutable reference, in addition to the passed args; this is the immutable receiver mentioned in the documentation. The final piece here is `-> Self::Output`, the returned type after the call operator is used. This associated type defaults to `Self` but can be set by the implementer.

`FnMut` is similar to `Fn` save that it takes `&mut self` rather than `&self`, and `FnOnce` is its supertrait:

```
    pub trait FnMut<Args>: FnOnce<Args> {
        /// Performs the call operation.
        #[unstable(feature = "fn_traits", issue = "29625")]
        extern "rust-call" fn call_mut(&mut self, args: Args)
            -> Self::Output;
    }
```

Only `FnOnce` deviates in its definition:

```
pub trait FnOnce<Args> {
    /// The returned type after the call operator is used.
    #[stable(feature = "fn_once_output", since = "1.12.0")]
    type Output;

    /// Performs the call operation.
    #[unstable(feature = "fn_traits", issue = "29625")]
    extern "rust-call" fn call_once(self, args: Args) -> Self::Output;
}
```

Here, we see where `Self::Output` makes its appearance as an associated type and note that the implementations of `FnOnce` are in terms of `call_once` rather than `call`. Also, we now know that `FnOnce` is a supertrait of `FnMut`, which is a supertrait of `Fn` and it just so happens that this property is transitive: if an `FnOnce` is called for an `Fn`, it can be used. Much of the exact implementation of the function traits are kept internal to the compiler, the details of which kind of jump around some as internals change. In fact, the `"rust-call"`, `extern` means that `Fn traits` cannot be implemented outside of special, compiler-specific contexts; the exact feature flags that need to be enabled in a nightly build, their use, and upcoming changes are not documented. Happily, closures and function pointers implement function traits implicitly. For example:

```
    fn mlen(input: &str) -> usize {
        input.len()
    }

    fn main() {
        let v: Option<&str> = Some("hope");
```

```
    assert_eq!(v.map(|s| s.len()), v.map(mlen));
}
```

The compiler is good enough to figure out the details for us.

Conceptually, `Result<T>` is a type similar to `Option<T>`, save that it is able to communicate an extra piece of information in its `Err` variant. In fact, the implementation in `src/libcore/result.rs` is awfully similar to the way we'd likely write this at first though, as with `Option`:

```
pub enum Result<T, E> {
    /// Contains the success value
    #[stable(feature = "rust1", since = "1.0.0")]
    Ok(#[stable(feature = "rust1", since = "1.0.0")] T),

    /// Contains the error value
    #[stable(feature = "rust1", since = "1.0.0")]
    Err(#[stable(feature = "rust1", since = "1.0.0")] E),
}
```

Cell and RefCell

So far in this chapter, the mutable references we've been discussing have a property called, by Rust, inherited mutability. That is, values are made mutable when we inherit their ownership or the exclusive right—via a `&mut`—to mutate the value. Inherited mutability is preferred in Rust as it is statically enforcable—any defects in our program with regards to memory mutation will be caught at compilation time. Rust does provide facilities for interior mutability, that being mutability that is available for pieces of an immutable type. The documentation calls interior mutability something of a last resort but, while not common, it is not exactly rare in practice, either. The two options for interior mutability in Rust are `Cell<T>` and `RefCell<T>`.

Let's consider `Cell<T>`. How is it used? As suggested, `Cell<T>` is useful when some field or fields of an otherwise immutable structure need to be mutable and the `T` you're concerned with is `T: Copy`. Consider a graph structure where a search operation is performed. This is logically immutable—the graph structure does not need to be modified during search. But, also consider the case where we would like to record the total number of traversals along the graph's edges. Our options are to violate the logical immutability of the graph search, require storage outside of the graph, or insert an interior, mutable counter into the graph. Which choice is best will depend on the specific situation. Here is a significantly less complicated example of `Cell<T>`, compared to a graph search:

```
use std::cell::Cell;

enum Project {
    Apollo,
```

```
    Gemini,
    Mercury,
}

struct Mission {
    project: Project,
    number: u8,
    duration_days: Cell<u8>,
}

fn main() {
    let mission = Mission {
        project: Project::Mercury,
        number: 7,
        duration_days: Cell::new(255),
    };

    mission.duration_days.set(0);
    assert_eq!(0, mission.duration_days.get());
}
```

Of course, that's a bit contrived. Interior mutability shows up when the situation is non-trivial, generally speaking. Note that `Cell<u8>` had to be manipulated with `get` and `set` methods, contrary to the normal process of setting and reading directly or through a pointer. Let's dig into the implementation of `Cell<T>`, defined in `src/libcore/cell.rs`:

```
pub struct Cell<T> {
    value: UnsafeCell<T>,
}
```

As is common in Rust internals, a safe interface hides an unsafe inner core, as we saw in `Chapter 2`, *Sequential Rust Performance and Testing*, with `HashMap`. What is the definition of `UnsafeCell`?

```
pub struct UnsafeCell<T: ?Sized> {
    value: T,
}
```

Of particular note is an implementation that follows shortly afterward:

```
impl<T: ?Sized> !Sync for UnsafeCell<T> {}
```

We will discuss `Sync` in the following chapter in greater detail. Suffice it to say, for now, any type that is threadsafe must implement `Sync`—and many do automatically—and by disabling the `Sync` trait for `UnsafeCell<T>`—which is what `!Sync` means—the type is specifically being marked as not thread-safe. That is, any thread may be manipulating the data owned by `UnsafeCell` at any time. As we've previously discussed, Rust is able to make a good deal of optimizations off the back of the knowledge that `&T` is guaranteed to also not be mutable somewhere and that `&mut T` is unique. `UnsafeCell<T>` is the *only* method Rust provides to turn these compiler optimizations off; it is possible to dip into an unsafe block and transmute `&T` to `&mut T`, but this is specifically called out as undefined behavior. They key to `UnsafeCell<T>` is that it is possible for a client to retrieve multiple mutable references to the interior data, even if the `UnsafeCell<T>` is itself mutable. It is up to the caller to ensure that there is *only* one mutable reference at any time. The implementation of `UnsafeCell` —stripped of its comments for clarity's sake—is brief:

```
impl<T> UnsafeCell<T> {
    pub const fn new(value: T) -> UnsafeCell<T> {
        UnsafeCell { value: value }
    }

    pub unsafe fn into_inner(self) -> T {
        self.value
    }
}

impl<T: ?Sized> UnsafeCell<T> {
    pub fn get(&self) -> *mut T {
        &self.value as *const T as *mut T
    }
}
```

This brings us back to `Cell<T>`. We know that `Cell<T>` is built on top of an unsafe abstraction and we will have to take care not to violate the `&mut T` uniqueness constraint. How is this done? First, construction of `Cell<T>` is straightforward, as you may expect:

```
impl<T> Cell<T> {
    pub const fn new(value: T) -> Cell<T> {
        Cell {
            value: UnsafeCell::new(value),
        }
    }
```

The `const fn` is likely a surprise, on account of it being a nightly-only feature as of writing this book. In Rust, a constant function is evaluated at compile time. The benefit to the programmer of this particular definition is that the result of `Cell::new` can be assigned to a constant variable, one which will exist for the lifetime of the program. Both `as_ptr` and `get_mut` are different views of the underlying `T`, one a raw mutable pointer and the other a mutable reference:

```
pub fn as_ptr(&self) -> *mut T {
    self.value.get()
}

pub fn get_mut(&mut self) -> &mut T {
    unsafe {
        &mut *self.value.get()
    }
}

pub fn into_inner(self) -> T {
    unsafe { self.value.into_inner() }
}
```

Note that while the internals of `get_mut` are unsafe, the borrow checker is brought to bear on the problem of keeping `&mut T` unique and so `Cell::get_mut` can itself be safe. `Cell::as_ptr` is not marked as unsafe—it's safe to receive a raw pointer in Rust—but any caller will have to do deferencing of that raw pointer in an unsafe block: it's possible

that there will be more than one raw, mutable pointer floating around. Setting a new value into the cell is done in terms of replacement, discussed ahead, but with careful attention made towards forcefully dropping the `T` pulled from the cell:

```
pub fn set(&self, val: T) {
    let old = self.replace(val);
    drop(old);
}
```

`Cell::swap` and `Cell::replace` are done in terms of the lower-level memory manipulation tools from `std::ptr` and `std::mem`. `swap` is intended to replace the interior of one cell with another. Its definition is as follows:

```
pub fn swap(&self, other: &Self) {
    if ptr::eq(self, other) {
        return;
    }
    unsafe {
        ptr::swap(self.value.get(), other.value.get());
    }
}
```

`swap` is done in terms of `std::ptr::swap`, a function documented as copying the memory through the raw pointers passed to it as arguments. You'll note that `Cell::swap` is careful to avoid the `swap` if the passed `other` is equivalent to `self`. The reason for this becomes clear when we take a peek at the definition of `std::ptr::swap`, defined in `src/libcore/ptr.rs`:

```
pub unsafe fn swap<T>(x: *mut T, y: *mut T) {
    // Give ourselves some scratch space to work with
    let mut tmp: T = mem::uninitialized();

    // Perform the swap
    copy_nonoverlapping(x, &mut tmp, 1);
    copy(y, x, 1); // `x` and `y` may overlap
```

```
        copy_nonoverlapping(&tmp, y, 1);

        // y and t now point to the same thing, but we need to
    completely
        forget `tmp`
        // because it's no longer relevant.
        mem::forget(tmp);
    }
```

The exact details of `copy_nonoverlapping` and copy are unimportant here, except in noting that swapping does require allocation of uninitialized space and copying back and forth from that space. It's wise to avoid the work if you don't have to do it. `Cell::replace` works along similar lines:

```
    pub fn replace(&self, val: T) -> T {
        mem::replace(unsafe { &mut *self.value.get() }, val)
    }
}
```

`std::mem::replace` takes a `&mut T` and a `T` and then replaces the value at `&mut T` with the passed in `val`, returning the old value and dropping neither. The definition of `std::mem::replace` is in `src/libcore/mem.rs` and is as follows:

```
    pub fn replace<T>(dest: &mut T, mut src: T) -> T {
        swap(dest, &mut src);
        src
    }
```

Chasing the definition of `swap` in the same module, we find it is as follows:

```
    pub fn swap<T>(x: &mut T, y: &mut T) {
        unsafe {
            ptr::swap_nonoverlapping(x, y, 1);
        }
    }
```

`std::ptr::swap_nonoverlapping` is as follows:

```
pub unsafe fn swap_nonoverlapping<T>(x: *mut T, y: *mut T, count:
usize) {
    let x = x as *mut u8;
    let y = y as *mut u8;
    let len = mem::size_of::<T>() * count;
    swap_nonoverlapping_bytes(x, y, len)
}
```

Finally, the private `std::ptr::swap_nonoverlapping_bytes` is as follows:

```
unsafe fn swap_nonoverlapping_bytes(x: *mut u8, y: *mut u8, len:
usize) {
    // The approach here is to utilize simd to swap x & y
efficiently.
    // Testing reveals that swapping either 32 bytes or 64 bytes at
    // a time is most efficient for intel Haswell E processors.
    // LLVM is more able to optimize if we give a struct a
    // #[repr(simd)], even if we don't actually use this struct
    // directly.
    //
    // FIXME repr(simd) broken on emscripten and redox
    // It's also broken on big-endian powerpc64 and s390x.  #42778
    #[cfg_attr(not(any(target_os = "emscripten", target_os =
"redox",
                       target_endian = "big")),
               repr(simd))]
    struct Block(u64, u64, u64, u64);
    struct UnalignedBlock(u64, u64, u64, u64);

    let block_size = mem::size_of::<Block>();

    // Loop through x & y, copying them `Block` at a time
    // The optimizer should unroll the loop fully for most types
    // N.B. We can't use a for loop as the `range` impl calls
    // `mem::swap` recursively
    let mut i = 0;
    while i + block_size <= len {
        // Create some uninitialized memory as scratch space

        // Declaring `t` here avoids aligning the stack when this
loop
        // is unused
```

```
            let mut t: Block = mem::uninitialized();
            let t = &mut t as *mut _ as *mut u8;
            let x = x.offset(i as isize);
            let y = y.offset(i as isize);

            // Swap a block of bytes of x & y, using t as a temporary
            // buffer. This should be optimized into efficient SIMD
            // operations where available
            copy_nonoverlapping(x, t, block_size);
            copy_nonoverlapping(y, x, block_size);
            copy_nonoverlapping(t, y, block_size);
            i += block_size;
        }

        if i < len {
            // Swap any remaining bytes
            let mut t: UnalignedBlock = mem::uninitialized();
            let rem = len - i;

            let t = &mut t as *mut _ as *mut u8;
            let x = x.offset(i as isize);
            let y = y.offset(i as isize);

            copy_nonoverlapping(x, t, rem);
            copy_nonoverlapping(y, x, rem);
            copy_nonoverlapping(t, y, rem);
        }
    }
```

Whew! That's some rabbit hole. Ultimately, what we've discovered here is that `std::mem::replace` is defined in terms of block copies from one non-overlapping location in memory to another, a process which the Rust developers have tried to make as efficient as possible by exploiting LLVM's ability to optimize a bitwise operation on common processors in terms of SIMD instructions. Neat.

What of `RefCell<T>`? It too is a safe abstraction over `UnsafeCell<T>` except that the copy restriction of `Cell<T>` is lifted. This makes the implementation a touch more complicated, as we'll see:

```
pub struct RefCell<T: ?Sized> {
    borrow: Cell<BorrowFlag>,
```

```
        value: UnsafeCell<T>,
    }
```

Like `Cell, RefCell` has an inner unsafe value, that much is the same. What's fun here is `borrow: Cell<BorrowFlag>`. `RefCell<T>` is a client of `Cell<T>`, which makes good sense considering that the immutable `RefCell` is going to need interior mutability to track the total number of borrows of its inner data. `BorrowFlag` is defined like so:

```
    // Values [1, MAX-1] represent the number of `Ref` active
    // (will not outgrow its range since `usize` is the size
    // of the address space)
    type BorrowFlag = usize;
    const UNUSED: BorrowFlag = 0;
    const WRITING: BorrowFlag = !0;
```

The implementation of `RefCell<T>` is like to that of `Cell<T>`. `RefCell::replace` is also implemented in terms of `std::mem::replace`, `RefCell::swap` in terms of `std::mem::swap`. Where things get interesting are the functions new to `RefCell`, which are those to do with borrowing. We'll look at `try_borrow` and `try_borrow_mut` first as they're used in the implementations of the other borrowing functions. `try_borrow` is defined like so:

```
    pub fn try_borrow(&self) -> Result<Ref<T>, BorrowError> {
        match BorrowRef::new(&self.borrow) {
            Some(b) => Ok(Ref {
                value: unsafe { &*self.value.get() },
                borrow: b,
            }),
            None => Err(BorrowError { _private: () }),
        }
    }
```

With `BorrowRef` being as follows:

```
struct BorrowRef<'b> {
    borrow: &'b Cell<BorrowFlag>,
```

```
}

impl<'b> BorrowRef<'b> {
    #[inline]
    fn new(borrow: &'b Cell<BorrowFlag>) -> Option<BorrowRef<'b>> {
        match borrow.get() {
            WRITING => None,
            b => {
                borrow.set(b + 1);
                Some(BorrowRef { borrow: borrow })
            },
        }
    }
}

impl<'b> Drop for BorrowRef<'b> {
    #[inline]
    fn drop(&mut self) {
        let borrow = self.borrow.get();
        debug_assert!(borrow != WRITING && borrow != UNUSED);
        self.borrow.set(borrow - 1);
    }
}
```

`BorrowRef` is a structure that holds a reference to the borrow field of `RefCell<T>`. Creating a new `BorrowRef` depends on the value of that borrow; if the value is `WRITING` then no `BorrowRef` is created—`None` gets returned—and otherwise the total number of borrows are incremented. This achieves the mutual exclusivity of writing needed while allowing for multiple readers—it's not possible for `try_borrow` to hand out a reference when a write reference is out for the same data. Now, let's consider `try_borrow_mut`:

```
    pub fn try_borrow_mut(&self) -> Result<RefMut<T>, BorrowMutError> {
        match BorrowRefMut::new(&self.borrow) {
            Some(b) => Ok(RefMut {
                value: unsafe { &mut *self.value.get() },
                borrow: b,
            }),
            None => Err(BorrowMutError { _private: () }),
        }
    }
```

Again, we find an implementation in terms of another type, `BorrowRefMut`:

```
struct BorrowRefMut<'b> {
    borrow: &'b Cell<BorrowFlag>,
}

impl<'b> Drop for BorrowRefMut<'b> {
    #[inline]
    fn drop(&mut self) {
        let borrow = self.borrow.get();
        debug_assert!(borrow == WRITING);
        self.borrow.set(UNUSED);
    }
}

impl<'b> BorrowRefMut<'b> {
    #[inline]
    fn new(borrow: &'b Cell<BorrowFlag>) -> Option<BorrowRefMut<'b>>
{
        match borrow.get() {
            UNUSED => {
                borrow.set(WRITING);
                Some(BorrowRefMut { borrow: borrow })
            },
            _ => None,
        }
    }
}
```

The key, as with `BorrowRef`, is in `BorrowRefMut::new`. Here, we can see that if the inner borrow from `RefCell` is unused then the borrow is set to write, excluding any potential read references. Likewise, if there is a read reference in existence, the creation of a mutable reference will fail. And so, exclusive mutable references and multiple immutable references are held at runtime by abstracting over an unsafe structure that allows for the breaking of that guarantee.

Rc

Now, let's consider one last single-item container before we get into a multi-item container. We'll look into `Rc<T>`, described by the Rust documentation as being a single-threaded reference-counting pointer. Reference counting pointers are distinct from the usual Rust references in that, while they are allocated on the heap as a `Box<T>`, cloning a reference counter pointer does not cause a new heap allocation, bitwise copy. Instead, a counter inside the `Rc<T>` is incremented, somewhat analogously as to the way `RefCell<T>` works. The drop of an `Rc<T>` reduces that internal counter and when the counter's value is equal to zero, the heap allocation is released. Let's take a peek inside `src/liballoc/rc.s`:

```
pub struct Rc<T: ?Sized> {
    ptr: Shared<RcBox<T>>,
    phantom: PhantomData<T>,
}
```

We've already encountered `PhantomData<T>` in this chapter, so we know that `Rc<T>` will not directly hold the allocation of `T`, but the compiler will behave as if it does. Also, we know that `T` may or may not be sized. The pieces we need to catch up on then are `RcBox<T>` and `Shared<T>`. Let's inspect `Shared<T>` first. Its full name is `std::ptr::Shared<T>` and it's the same kind of pointer as `*mut X` except that it is non-zero. There are two variants for creating a new `Shared<T>`, `const unsafe fn new_unchecked(ptr: *mut X) -> Self` and `fn new(ptr: *mut X) -> Option<Self>`. In the first variant, the caller is responsible for ensuring that the pointer is non-null, and in the second the nulled nature of the pointer is checked, like so:

```
    pub fn new(ptr: *mut T) -> Option<Self> {
        NonZero::new(ptr as *const T).map(|nz| Shared { pointer: nz })
    }
```

We find the definition and implementation of `NonZero<T>` in `src/libcore/nonzero.rs` like so:

```
    pub struct NonZero<T: Zeroable>(T);

    impl<T: Zeroable> NonZero<T> {
        /// Creates an instance of NonZero with the provided value.
        /// You must indeed ensure that the value is actually "non-
    zero".
        #[unstable(feature = "nonzero",
                   reason = "needs an RFC to flesh out the design",
                   issue = "27730")]
        #[inline]
        pub const unsafe fn new_unchecked(inner: T) -> Self {
            NonZero(inner)
        }

        /// Creates an instance of NonZero with the provided value.
        #[inline]
        pub fn new(inner: T) -> Option<Self> {
            if inner.is_zero() {
                None
            } else {
                Some(NonZero(inner))
            }
        }

        /// Gets the inner value.
        pub fn get(self) -> T {
            self.0
        }
    }
```

`Zeroable` is an unstable trait, which is pretty straightforward:

```
pub unsafe trait Zeroable {
    fn is_zero(&self) -> bool;
}
```

Every pointer type implements an `is_null() -> bool` and this trait defers to that function, and `NonZero::new` defers to `Zeroable::is_zero`. The presence of a `Shared<T>`, then, gives the programmer the same freedom as `*mut T` but with added guarantees about the pointer's nullable situation. Jumping back up to `Rc::new`:

```
pub fn new(value: T) -> Rc<T> {
    Rc {
        // there is an implicit weak pointer owned by all the strong
        // pointers, which ensures that the weak destructor never
frees
        // the allocation while the strong destructor is running,
even
        // if the weak pointer is stored inside the strong one.
        ptr: Shared::from(Box::into_unique(box RcBox {
            strong: Cell::new(1),
            weak: Cell::new(1),
            value,
        })),
        phantom: PhantomData,
    }
}
```

`Box::into_unique` converts a `Box<T>` into a `Unique<T>`—discussed previously in this chapter—which is then converted into a `Shared<T>`. This chain preserves the non-null guarantee needed and ensures uniqueness. Now, what about strong and weak in `RcBox`? `Rc<T>` provides a method, `Self::downgrade(&self) -> Weak<T>`, that produces a non-owning pointer, a pointer which does not guarantee the liveness of the referenced data and does not extend its lifetime. This is called a *weak reference*. Dropping a `Weak<T>`, likewise, does not imply that `T` is dropped. The trick here is that a strong reference does extend the liveness of the underlying `T`—the drop of `T` is only called when the internal counter of `Rc<T>` hits zero. For the most part, things rarely require a weak reference, except when a cycle of references exist. Suppose a graph structure were to be constructed where each node holds an `Rc<T>` to its connected nodes and a cycle exists in the graph. Calling drop on the current node will recursively call drop on the connected nodes, which will recurse again onto the current node and

so forth. Were the graph to store a vector of all nodes and have each node store weak references to connections, then a drop of the vector would cause a drop of all nodes, cycles or not. We can see how this works in practice by inspecting the `Drop` implementation of `Rc<T>`:

```
   unsafe impl<#[may_dangle] T: ?Sized> Drop for Rc<T> {
       fn drop(&mut self) {
           unsafe {
               let ptr = self.ptr.as_ptr();

               self.dec_strong();
               if self.strong() == 0 {
                   // destroy the contained object
                   ptr::drop_in_place(self.ptr.as_mut());

                   // remove the implicit "strong weak" pointer
                   // now that we've destroyed the contents.
                   self.dec_weak();

                   if self.weak() == 0 {
                       Heap.dealloc(ptr as *mut u8,
                       Layout::for_value(&*ptr));
                   }
               }
           }
       }
   }
```

This is trimmed some for clarity but the notion is as we've described —when the total number of strong references is zero, a full deallocation occurs. The referenced `Heap` and `Layout` are compiler internals and won't be discussed further here, but the interested reader is warmly encouraged to go spelunking on their own. Recall that in `Rc<T>::new`, both strong and weak counters started at `1`. To avoid invalidating the weak pointers, the actual `T` is only deallocated if there are no strong or weak pointers available. Let's have a look at `Drop for Weak<T>`, again trimmed some for clarity:

```
impl<T: ?Sized> Drop for Weak<T> {
    fn drop(&mut self) {
```

```
        unsafe {
            let ptr = self.ptr.as_ptr();

            self.dec_weak();
            // the weak count starts at 1, and will only go to
            // zero if all the strong pointers have disappeared.
            if self.weak() == 0 {
                Heap.dealloc(ptr as *mut u8, Layout::for_value(&*ptr));
            }
        }
    }
}
```

As expected, the `T` can only be deallocated when the weak pointer total falls to zero, which is only possible if there are no strong pointers left. That's `Rc<T>`—a handful of important traits, a specialized box, and a few compiler internals.

Vec

As our last subject in this chapter, let's consider `Vec<T>`. The Rust vector is a growable array, that is, an area of contiguous, homogeneous storage that can be grown through reallocation as the need arises, or it cane be shrunk. The advantage compared to array is in not needing to know ahead of time exactly what size of storage you need, plus all the benefits of a slicable structure and additional functions in the type API. `Vec<T>` is an extremely common Rust structure, so much so that its actual name is `std::vec::Vec<T>` but Rust imports the type by default. Rust programs are full of vectors and it stands to reason we'd do well to understand how it interacts with the memory it holds.

`Vec<T>` is defined in `src/liballoc/vec.rs` and is defined as follows:

```
pub struct Vec<T> {
    buf: RawVec<T>,
    len: usize,
}
```

The Rust documentation declares that a `Vec<T>` will be laid out as a pointer, a capacity of `usize`, and a length of `usize`. The order of these fields is completely unspecified. As we've already discussed, Rust will reorder fields as it sees fit. Moreover, the pointer is guaranteed to be non-null, allowing for a null-pointer optimization. Previously, we saw the usual trick of Rust: define a higher-level structure in terms of a lower-level—or raw – structure, or even in terms of a fully unsafe structure. We see the length `usize` already, called `len`. This means there's a distinct difference between capacity and length, the distinction of which we'll come back to as we dig into `RawVec<T>`. Let's take a peek at `RawVec<T>`, defined in `src/liballoc/raw_vec.rs`:

```
pub struct RawVec<T, A: Alloc = Heap> {
    ptr: Unique<T>,
    cap: usize,
    a: A,
}
```

What we have is a pointer to a `Unique<T>`, a `*mut T` with added guarantees that the `RawVec<T>` is the only possessor of the allocation and that the pointer is not null. The `cap` is the capacity of the raw vector. `a: A` is the allocator, `Heap` by default. Rust does allow for allocators to be swapped, so long as the implementation obeys—as of writing this book—the `std::heap::Alloc` trait. Swapping allocators is an unstable feature of Rust, available only in the nightly channel, but one that is stable enough to see common use in the embedded Rust community's libraries. In this book, we won't use anything other than the default allocator, but the reader is warmly encouraged to explore the topic in more detail. Allocator aside, there's the pointers, length, and capacity that the Rust documentation promised. Let's pop back to `Vec<T>` and take a look at `new`:

```
pub fn new() -> Vec<T> {
    Vec {
        buf: RawVec::new(),
        len: 0,
    }
}
```

Now, the new of raw `vec`:

```
pub fn new() -> Self {
    Self::new_in(Heap)
}
```

This defers creation to `new_in`, a function on the same `trait`:

```
pub fn new_in(a: A) -> Self {
    // !0 is usize::MAX. This branch should be stripped
```

```
        // at compile time.
        let cap = if mem::size_of::<T>() == 0 { !0 } else { 0 };

        // Unique::empty() doubles as "unallocated" and "zero-sized
        // allocation"
        RawVec {
            ptr: Unique::empty(),
            cap,
            a,
        }
    }
```

The `cap` computation is interesting. It's possible to store a `T` in `Vec<T>`—and by extension `RawVec<T>`—that has zero size. The implementers knew this and came up with a fun solution: if the size of the type is zero, set the capacity to `usize::MAX`, else 0. It's not possible to cram `usize::MAX` elements into a vector since we'd run out of memory one allocation prior to hitting the cap and it's now possible to discriminate the case of zero-sized types without having to introduce an enumeration or a flag variable. Tidy trick. If we bounce back to vector and inspect `with_capacity`, we'll find that defers to `RawVec::with_capacity`, which defers to `RawVec::allocate_in`:

```
    fn allocate_in(cap: usize, zeroed: bool, mut a: A) -> Self {
        unsafe {
            let elem_size = mem::size_of::<T>();

            let alloc_size =
            cap.checked_mul(elem_size).expect("capacity overflow");
            alloc_guard(alloc_size);

            // handles ZSTs and `cap = 0` alike
            let ptr = if alloc_size == 0 {
                mem::align_of::<T>() as *mut u8
            } else {
                let align = mem::align_of::<T>();
                let result = if zeroed {
                    a.alloc_zeroed(Layout::from_size_align(
                        alloc_size, align).unwrap())
                } else {
                    a.alloc(Layout::from_size_align(
                        alloc_size, align).unwrap())
                };
```

```
                match result {
                    Ok(ptr) => ptr,
                    Err(err) => a.oom(err),
                }
            };

            RawVec {
                ptr: Unique::new_unchecked(ptr as *mut _),
                cap,
                a,
            }
        }
    }
```

There's a lot going on here, but it's important, so let's break it into small pieces. Firstly, see the following:

```
            let elem_size = mem::size_of::<T>();

            let alloc_size =
            cap.checked_mul(elem_size).expect("capacity overflow");
            alloc_guard(alloc_size);
```

At the head of the function, the size of the element is computed and the total allocation requested by the caller is confirmed to be no more than the available system memory. Note that `checked_mul` ensures we don't overflow usize and accidentally allocate too little memory. Finally, a function called `alloc_guard` is called. That is as follows:

```
fn alloc_guard(alloc_size: usize) {
    if mem::size_of::<usize>() < 8 {
        assert!(
            alloc_size <= ::core::isize::MAX as usize,
            "capacity overflow"
        );
    }
}
```

This is a guarantee check. Remember that usize and isize are the signed and unsigned size of the system pointer. To understand this

guard, we must understand the answer to two questions:

1. On a given machine, how much memory can I allocate?
2. On a given machine, how much memory can I address?

It's possible to allocate usize bytes from the operating system but here, Rust is checking that the allocation size is less than the maximum isize. The Rust reference (`https://doc.rust-lang.org/stable/reference/types.html#machine-dependent-integer-types`) explains why:

"The isize type is a signed integer type with the same number of bits as the platform's pointer type. The theoretical upper bound on object and array size is the maximum isize value. This ensures that isize can be used to calculate differences between pointers into an object or array and can address every byte within an object along with one byte past the end."

Combined with the capacity check from `checked_mul`, we know that the allocation is properly sized and that it's addressable along the whole of itself:

```
            let ptr = if alloc_size == 0 {
                mem::align_of::<T>() as *mut u8
            } else {
```

In the event that the desired capacity is zero *or* the size of `T` is zero, the implementation coerces the minimum alignment of `T` into a pointer to bytes, the `*mut u8`. This pointer is, well, it points nowhere useful but the implementation has avoided an allocation when there is nothing that could be allocated, whether there will never be anything to allocate because of zero-sized types or not. This is good, and all the implementation will have to do is be aware that when the capacity is zero, or if the type size is zero, the pointer cannot be dereferenced. Right:

```
            } else {
                let align = mem::align_of::<T>();
```

```
                let result = if zeroed {
                    a.alloc_zeroed(Layout::from_size_align(alloc_size,
                        align).unwrap())
                } else {
                    a.alloc(Layout::from_size_align(alloc_size,
                        align).unwrap())
                };
                match result {
                    Ok(ptr) => ptr,
                    Err(err) => a.oom(err),
                }
            };
```

This branch is hit when there's memory to be allocated. Notably, two sub-possibilities, controlled by the `zeroed` argument: either memory is zeroed or it is left uninitialized. `Vec<T>` does not expose this option to the end user but we know from inspection that memory starts off uninitialized, an optimization for non-trivial allocations:

```
            RawVec {
                ptr: Unique::new_unchecked(ptr as *mut _),
                cap,
                a,
            }
        }
    }
```

The lack of an initialization flag is kind of tricky in some cases. Consider `std::io::Read::read_exact`. This function takes a `&mut [u8]` and it's common enough to create this slice from a specially created vec. This code will *not* read 1024 bytes:

```
use std::io;
use std::io::prelude::*;
use std::fs::File;

let mut f = File::open("foo.txt")?;
let mut buffer = Vec:with_capacity(1024);

f.read_exact(&mut buffer)?;
```

Why? The slice that we pass in is actually of zero length! Rust dereferences are allowed on a type by two traits: `std::ops::Deref` and `std::ops::DerefMut`, depending on your desire for an immutable or mutable slide. The `Deref` trait is defined as follows:

```
pub trait Deref {
    type Target: ?Sized;
    fn deref(&self) -> &Self::Target;
}
```

And the mutable version analogously. When we slice our vector as in the preceding code block, we're calling this `DerefMut::deref`:

```
impl<T> ops::DerefMut for Vec<T> {
    fn deref_mut(&mut self) -> &mut [T] {
        unsafe {
            let ptr = self.buf.ptr();
            assume(!ptr.is_null());
            slice::from_raw_parts_mut(ptr, self.len)
        }
    }
}
```

This is the very important part: `slice::from_raw_parts_mut(ptr, self.len)`. Vector slices are built by length, not capacity. The capacity of a vector serves to distinguish how much memory has been allocated versus how much memory has been initialized, either to zeros or some other inserted values. This is an important difference. It's possible to initialize memory ourselves:

```
let mut buffer: Vec<u8> = Vec::with_capacity(1024);
for _ in 0..1024 {
  buffer.push(0);
}
assert_eq!(1024, buffer.len());
```

Or to rely on the `Vec` API to convert from a fixed-size array:

```
    let buffer = [0; 1024];
    let buffer = &mut buffer.to_vec();
    assert_eq!(1024, buffer.len());
    }
```

Either will work. Which you choose depends on if you know the size of the buffer ahead of time or not.

Befitting such an important type, there are a good many API functions on `Vec<T>`. In the remainder of this chapter, we'll occupy ourselves with two mutations: `push` and `insert`. The `push` function is a constant operation, modulo any reallocations necessary to cope with the case of our capacity limit being reached. Here's the `push` function:

```
    pub fn push(&mut self, value: T) {
        // This will panic or abort if we would allocate > isize::MAX
bytes
        // or if the length increment would overflow for zero-sized
types.
        if self.len == self.buf.cap() {
            self.buf.double();
        }
        unsafe {
            let end = self.as_mut_ptr().offset(self.len as isize);
            ptr::write(end, value);
            self.len += 1;
        }
    }
```

If you'll recall, `self.buf` is the underlying `RawVec<T>`. The documentation for `Vec<T>` notes that when reallocation is required, the underlying memory will be doubled, which, it turns out, is handled by `RawVec<T>::double`. That function is fairly long and, as you might suspect, is a bunch of arithmetic to compute the new, doubled size matched with a realloc when there's an existing allocation, else when there's a new allocation. That is worth listing:

```
            None => {
                // skip to 4 because tiny Vec's are dumb;
                // but not if that would cause overflow
                let new_cap = if elem_size > (!0) / 8 {
                                  1
                              } else {
                                  4
                              };
                match self.a.alloc_array::<T>(new_cap) {
                    Ok(ptr) => (new_cap, ptr),
                    Err(e) => self.a.oom(e),
                }
            }
```

`Alloc::alloc_array` allocates space for contiguous memory, which is suitable to hold `new_cap` number of elements the size of `T`, returning the pointer to the first address of this newly allocated space. Here then, is the contiguous memory promised in the documentation of `Vec<T>`! Back in `Vec<T>`, now that the capacity is twice that of what it was, at least, the value `T` can be inserted:

```
        unsafe {
            let end = self.as_mut_ptr().offset(self.len as isize);
            ptr::write(end, value);
            self.len += 1;
        }
```

Rust pointers are handy in that their offset function takes into account the size of `T`; there's no need to do additional multiplication as the caller. The implementation is determining the first of the unused `T` sized spaces in the contiguous allocation—denoted end—and then writes the moved `T` onto that address. The fact that the space is unused is important, `std::ptr::write` does not deallocate any memory that may have originally existed at the written-to pointer. If you `ptr::write` over the top of a live reference, wacky things will happen.

Finally, `insert`. The `insert` function of `Vec<T>` allows for the insertion of `T` at any valid index in the vector. If the insertion is to the end of the vector, the method is functionally equivalent to pushing, though

mechanically different, as we'll see shortly. If, however, insertion occurs somewhere inside of the vector, all elements to the right of the insertion index are shifted over once, a non-trivial operation depending on the size of the allocation. Here's the full listing of `insert`:

```
    pub fn insert(&mut self, index: usize, element: T) {
        let len = self.len();
        assert!(index <= len);

        // space for the new element
        if len == self.buf.cap() {
            self.buf.double();
        }

        unsafe {
            // infallible
            // The spot to put the new value
            {
                let p = self.as_mut_ptr().offset(index as isize);
                // Shift everything over to make space. (Duplicating
the
                // `index`th element into two consecutive places.)
                ptr::copy(p, p.offset(1), len - index);
                // Write it in, overwriting the first copy of the
`index`th
                // element.
                ptr::write(p, element);
            }
            self.set_len(len + 1);
        }
    }
```

The index checking and potential buffering is straightforward at this point in the chapter. In the unsafe block, `p` is the proper offset for insertion. The two important lines are:

```
        ptr::copy(p, p.offset(1), len - index);
        ptr::write(p, element);
```

When a value is inserted into a vector, no matter the location, all the memory starting from `p` to the end of the list is copied over the top of the memory, starting at `p+1`. Once this is done, the inserted value is written over the top of `p`. This is a non-trivial operation and can become incredibly slow, especially if the allocation to be shifted is fairly large. Performance-focused Rust will use `Vec::insert` sparingly, if at all. `Vec<T>` will almost surely make an appearance.

Summary

In this chapter, we covered the layout of Rust objects in memory, the way references work across both safe and unsafe Rust, and have addressed the various allocation strategies of a Rust program. We did a deep-dive on types in the Rust standard library to make these concepts concrete and it is hoped that the reader will now feel comfortable further exploring the compiler and will do so with confidence.

Further reading

The memory model of a programming language is a broad topic. Rust's, as of writing this book, must be understood from the inspection of the Rust documentation, the rustc source code, and research into LLVM. That is, Rust's memory model is not formally documented, though there are rumblings in the community of providing it. Independent of that, it is also important for the working programmer to understand the underlying machine. There's a staggering amount of material to be covered.

These notes are a small start, focusing especially on the Rust documentation that relates most to this chapter:

- *High Performance Code 201: Hybrid Data Structures*, Chandler Carruth, available at `https://www.youtube.com/watch?v=vElZc6zSIXM&index=5&list=PLKW_WLANyJtqQ6IWm3BjzHZvrFxFSooey`. This, in point of fact, is a talk from CppCon 2016. Carruth is an engaging speaker and is a member of the LLVM team focused on compiler performance. This talk is especially interesting from the point of view of building information-dense data structures that interact well with CPU caches. While the talk is in C++, the techniques apply directly to Rust.
- *Cache-oblivious Algorithms*, Matteo Frigo, Charles Leiserson, Harald Prokop, and Sridhar Ramachandran. This paper introduces the concept of building data structures that

are cache oblivious, or, native to machines with memory hierarchies and interact with them well, in addition to a machine model to analyze such data structures.

- *Cache-Oblivious Algorithms and Data Structures*, Erik Demaine. This paper is a classic in the cache-oblivious space, building on the work presented in the last by Frigo et al and summarizing existing work. This is a highly recommended read, especially in conjunction with the previous paper. It is well worth scanning the bibliography as well.

- *The Stack and the Heap*, available at `https://doc.rust-lang.org/book/first-edition/the-stack-and-the-heap.html`. This chapter from the first edition of the Rust book explains the difference between the hardware stack and heap, allocations to each, and the implications for Rust. This chapter has gone into further detail in some areas, but the Rust book's chapter is warmly recommended for anyone needing a refresher or a more gentle climb.

- *Splitting Borrows*, available at `https://doc.rust-lang.org/beta/nomicon/borrow-splitting.html`. The Nomicon is a Rust book intended to teach low-level Rust programming, not unlike this book. While it is a work-in-progress, the information in it is invaluable. *Splitting Borrows* explains the reasoning behind a common issue with new Rust developers: performing multiple mutable borrows out of a vector or array. The fact that this works with *structs* is often a source of great confusion and anguish.

- *Rust Reference*, available at `https://doc.rust-lang.org/reference/`. Like any established programming language, the Rust Reference is invaluable for understanding the subtle details of the language itself, which have been hashed out in mailing lists and over chat for years. The reference in its current form can be a touch hard to search—it used to be one long page—but it's hoped the situation will be improved upon by the time our book here has gone to press.
- *Closures: Anonymous Functions that can Capture their Environment*, available at `http://doc.rust-lang.org/1.23.0/book/second-edition/ch13-01-closures.html`. Rust closures have some subtle implications to them that can be hard to internalize for new Rust developers. This chapter in the second edition of the Rust Book is excellent in this regard.
- *External blocks*, available at `https://doc.rust-lang.org/reference/items/external-blocks.html`. External blocks are relatively rare in this book—and, perhaps, in most of the Rust code you're likely to see—but there's a fair few of them available. It is well worth having a passing knowledge of this document's existence.
- *Hacker's Delight*, Henry Warren Jr.This book is a classic. Many of the tricks present in the book are now available as simple instructions on some chips, such as x86, but you'll see the occasional delight here or there in the `rustc` source code, the `swap_nonoverlapping_bytes` trick especially.

Sync and Send – the Foundation of Rust Concurrency

Rust aims to be a programming language in which fearless concurrency is possible. What does this mean? How does it work? In `Chapter 2`, *Sequential Rust Performance Testing*, we discussed the performance of sequential Rust programs, intentionally setting aside discussion of concurrent programs. In `Chapter 3`, *The Rust Memory Model – Ownership, References and Manipulation*, we saw an overview of the way Rust handles memory, especially with regard to composing high-performance structures. In this chapter, we'll expand on what we've learned previously and, at long last, dig in to Rust's concurrency story.

By the end of this chapter, we will have:

- Discussed the `Sync` and `Send` traits
- Inspected parallel races in a ring data structure with Helgrind
- Resolved this race with a mutex
- Investigated the use of the standard library MPSC
- Built a non-trivial data multiplexing project

Technical requirements

This chapter requires a working Rust installation. The details of verifying your installation are covered in `Chapter 1`, *Preliminaries – Machine Architecture and Getting Started with Rust*. The Valgrind suite of tools are used below. Many operating systems bundle valgrind packages but you can find further installation instructions for your system at `valgrind.org`. Linux Perf is used and is bundled by many Linux distributions. Any other software required for this chapter is installed as a part of the text.

You can find the source code for this book's projects on GitHub: `https://github.com/PacktPublishing/Hands-On-Concurrency-with-Rust/`. This chapter has its source code under `Chapter04`.

Sync and Send

There are two key traits that the parallel Rust programmer must understand—`Send` and `Sync`. The `Send` trait is applied to types that can be transferred across thread boundaries. That is, any type `T: Send` is safe to move from one thread to another. `Rc<T>: !Send` means that it is explicitly marked as being unsafe to move between threads. Why? Consider if it were marked `Send`; what would happen? We know from the previous chapter that `Rc<T>` is a boxed `T` with two counters in place for weak and strong references to `T`. These counters are the trick. Suppose we spread an `Rc<T>` across two threads—call them `A` and `B`—each of which created references, dropped them, and the like. What would happen if both `A` and `B` dropped the last reference to `Rc<T>` at the same time? We have a race between either thread to see which can deallocate `T` first and which will suffer a double-free for its troubles. Worse, suppose the taking of the strong reference counter in `Rc<T>` were spread across three pseudo-instructions:

```
LOAD counter            (tmp)
ADD counter 1 INTO tmp  (tmp = tmp+1)
STORE tmp INTO counter  (counter = tmp)
```

Likewise, suppose the dropping of a strong reference counter were spread across three pseudo-instructions:

```
LOAD counter            (tmp)
SUB counter 1 INTO tmp  (tmp = tmp-1)
STORE tmp INTO counter  (counter = tmp)
```

In a single-threaded context, this works well, but consider this result in a multi-threaded context. Let `counter` be equal to 10 for all threads at the beginning of the following thought experiment:

```
[A]LOAD counter                         (tmp_a == 10)
[B]LOAD counter                         (tmp_b == 10)
[B]SUB counter 1 INTO tmp               (tmp_b = tmp_b-1)
[A]ADD counter 1 INTO tmp               (tmp_a = tmp_a + 1)
[A]STORE tmp INTO counter               (counter = tmp_a)    == 11
[A]ADD counter 1 INTO tmp               (tmp_a = tmp_a + 1)
[A]STORE tmp INTO counter               (counter = tmp_a)    == 12
[A]ADD counter 1 INTO tmp               (tmp_a = tmp_a + 1)
[A]STORE tmp INTO counter               (counter = tmp_a)    == 13
[B]STORE tmp INTO counter               (counter == tmp_b)   == 9
```

By the end, we've lost three references to `Rc<T>`, meaning while `T` is not lost in memory, it is entirely possible that when we drop `T`, references will remain to its no longer valid memory out in the wild, the results of which are undefined but not likely to be great.

The `Sync` trait is derived from `Send` and has to do with references: `T: Sync` if `&T: Send`. That is, a `T` is `Sync` only if sharing a `&T` which acts as if that `T` were sent into the thread. We know from the previous code that `Rc<T>: !Send` and so we also know that `Rc<T>: !Sync`. Rust types inherit their constituent parts' `Sync` and `Send` status. By convention, any type which is `Sync` + `Send` is called thread-safe. Any type we implement on top of `Rc<T>` will not be thread-safe. Now, for the most part, `Sync` and `Send` are automatically derived traits. `UnsafeCell` , discussed in `Chapter 3`, *The Rust Memory Model – Ownership, References and Manipulation*, is not thread-safe. Neither are raw pointers, to go with their lack of other safety guarantees. As you poke around Rust code bases, you'll find traits that would otherwise have been derived thread-safe but are marked as not.

Racing threads

You can also mark types as thread-safe, effectively promising the compiler that you've arranged all the data races in such a way as to make them safe. This is relatively rare in practice but it's worth seeing what it takes to make a type thread-safe in Rust before we start building on top of safe primitives. First, let's take a look at code that's intentionally sideways:

```
use std::{mem, thread};
use std::ops::{Deref, DerefMut};

unsafe impl Send for Ring {}
unsafe impl Sync for Ring {}

struct InnerRing {
    capacity: isize,
    size: usize,
    data: *mut Option<u32>,
}

#[derive(Clone)]
struct Ring {
    inner: *mut InnerRing,
}
```

What we have here is a ring, or a circular buffer, of `u32`. `InnerRing` holds a raw mutable pointer and is not automatically thread-safe as a result. But, we've promised to Rust that we know what we're doing and implement `Send` and `Sync` for `Ring`. Why not on `InnerRing`? When we manipulate an object in memory from multiple threads, the location of that object has to be fixed. `InnerRing`—and the data it contains—have to occupy a stable place in memory. `Ring` can and will be bounced around, at the very least from a creating thread to a worker. Now, what's that data there in `InnerRing`? It's a pointer to the 0th offset

of a contiguous block of memory that will be the store of our circular buffer. At the time of writing this book, Rust has no stable allocator interface and, so, to get a contiguous allocation we have to do it in a roundabout fashion—strip a `Vec<u32>` down to its pointer:

```
impl Ring {
    fn with_capacity(capacity: usize) -> Ring {
        let mut data: Vec<Option<u32>> =
Vec::with_capacity(capacity);
        for _ in 0..capacity {
            data.push(None);
        }
        let raw_data = (&mut data).as_mut_ptr();
        mem::forget(data);
        let inner_ring = Box::new(InnerRing {
            capacity: capacity as isize,
            size: 0,
            data: raw_data,
        });

        Ring {
            inner: Box::into_raw(inner_ring),
        }
    }
}
```

`Ring::with_capacity` functions much the same as other types' `with_capacity` from the Rust ecosystem: sufficient space is allocated to fit capacity items. In our case, we piggyback off `Vec::with_capacity`, being sure to allocate enough room for capacity `Option<u32>` instances, initializing to None along the full length of the memory block. If you'll recall from `chapter 3`, *The Rust Memory Model – Ownership, References and Manipulation*, this is done as `Vec` is lazy about allocating and we require the allocation. `Vec::as_mut_ptr` returns a raw pointer to a slice but does not consume the original object, a problem for `Ring`. When data falls out of scope, the allocated block must survive. The standard library's `mem::forget` is ideal for this very use case. The allocation now being safe, an `InnerRing` is boxed to store it. The box is then consumed by `Box::into_raw` and passed into a `Ring`. Ta-da!

Interacting with a type that has an inner raw pointer can be verbose, scattering unsafe blocks around to little good effect. To that end, `Ring` gets a `Deref` and `DerefMut` implementation, both of which tidy up the interaction with `Ring`:

```
impl Deref for Ring {
    type Target = InnerRing;

    fn deref(&self) -> &InnerRing {
        unsafe { &*self.inner }
    }
}

impl DerefMut for Ring {
    fn deref_mut(&mut self) -> &mut InnerRing {
        unsafe { &mut *self.inner }
    }
}
```

Now that we have `Ring` defined, we can get into the meat of the program. We'll define two operations that will run concurrently with one another—writer and reader. The idea here is that writer will race around the ring writing, increasing `u32` values into the ring whenever there's capacity to do so. (At type boundary the `u32` will wrap.) The reader will race around behind the writer reading the values written, checking that each read value is up one from the previous read, with the caveat of wrapping. Here's the writer:

```
fn writer(mut ring: Ring) -> () {
    let mut offset: isize = 0;
    let mut cur: u32 = 0;
    loop {
        unsafe {
            if (*ring).size != ((*ring).capacity as usize) {
                *(*ring).data.offset(offset) = Some(cur);
                (*ring).size += 1;
                cur = cur.wrapping_add(1);
                offset += 1;
                offset %= (*ring).capacity;
            } else {
                thread::yield_now();
```

```
            }
        }
    }
}
```

Now, to be crystal clear, there's a *lot* that's wrong here. The ambition is to only write when the size of the ring buffer has not reached its capacity—meaning there's free space available. The actual writing is:

```
            *(*ring).data.offset(offset) = Some(cur);
            (*ring).size += 1;
```

That is, we dereference the `ring (*ring)` and get the pointer to the `Option<u32>` sized block at `(*ring).data.offset(offset)`, which we then dereference and move `Some(cur)` onto the top of whatever was previously there. It is entirely possible that because of races on the size of the `Ring` that we'll overwrite an unread `u32`. The remainder of the write block sets up our next `cur` and our next offset, adding one and modulating around if need be:

```
        } else {
            thread::yield_now();
        }
```

`thread::yield_now` is new. The writer is a fast spin-loop—it checks a single condition and loops back around again for another try. This is very CPU and power inefficient. `thread::yield_now` hints to the operating system that this thread had no work to do and should be deprioritized in favor of other threads. The effect is OS and running environment-dependent but it's still a good idea to yield if you have to spin-loop:

```
fn reader(mut ring: Ring) -> () {
    let mut offset: isize = 0;
    let mut cur: u32 = 0;
    while cur < 1_000 {
        unsafe {
```

```
                if let Some(num) = mem::replace(
                    &mut *(*ring).data.offset(offset),
                    None)
                {
                    assert_eq!(num, cur);
                    (*ring).size -= 1;
                    cur = cur.wrapping_add(1);
                    offset += 1;
                    offset %= (*ring).capacity;
                } else {
                    thread::yield_now();
                }
            }
        }
    }
```

The reader is similar to the writer, with the major difference being that it's not an infinite loop. Reads are done with `mem::replace`, swapping the block at the reader offset with `None`. When we hit bingo and score a `Some`, the memory of that `u32` is now owned by the reader—a drop will be called when it goes out of scope. This is important. The writer is responsible for losing memory inside of a raw pointer and the reader is responsible for finding it. In this way, we are careful not to leak memory. Or, well, we would if there wasn't a race on the size of the `Ring`.

Finally, we have the `main` function:

```
fn main() {
    let capacity = 10;
    let ring = Ring::with_capacity(capacity);

    let reader_ring = ring.clone();
    let reader_jh = thread::spawn(move || {
        reader(reader_ring);
    });
    let _writer_jh = thread::spawn(move || {
        writer(ring);
    });

    reader_jh.join().unwrap();
}
```

There are two new things going on here. The first is our use of `thread::spawn` to start a `reader` and a `writer`. The `move || {}` construct is called a *move closure*. That is, every variable reference inside the closure from the outer scope is moved into the closure's scope. It's for this reason that we clone ring to `reader_ring`. Otherwise, there'd be no `ring` for the writer to work with. The second new thing is the `JoinHandle` that `thread::spawn` returns. Rust threads are not a drastic departure from the common POSIX or Windows threads. Rust threads receive their own stack and are independently runnable by the operating system.

Every Rust thread has a return value, though here ours is `()`. We get at that return value by *joining* on the thread's `JoinHandler`, pausing execution of our thread until the thread wraps up successfully or crashes. Our main thread assumes its child threads will return successfully, hence the `join().unwrap()`.

What happens when we run our program? Well, failure, which is what we were expecting:

```
> rustc -C opt-level=3 data_race00.rs && ./data_race00
thread '<unnamed>' panicked at 'assertion failed: `(left == right)`
  left: `31`,
   right: `21`', data_race00.rs:90:17
   note: Run with `RUST_BACKTRACE=1` for a backtrace.
   thread 'main' panicked at 'called `Result::unwrap()` on an `Err`
value: Any', libcore/result.rs:916:5
```

The flaw of the Ring

Let's explore what's going wrong here. Our ring is laid out in memory as a contiguous block and we have a few control variables hung off the side. Here's a diagram of the system before any reads or writes happen:

```
size: 0
capacity: 5

 rdr
 |
|0: None |1: None |2: None |3: None |4: None |
 |
 wrt
```

Here's a diagram of the system just after the writer has written its first value:

```
size: 0
capacity: 5

 rdr
 |
|0: Some(0) |1: None |2: None |3: None |4: None |
             |
             wrt
```

To do this, the writer has performed a load of both size and capacity, performed a comparison between then, written to its offset inside the block, and incremented size. None of these operations are guaranteed to be ordered as they are in the program, either due to speculative execution or compiler reordering. As we've seen in the previous run example, the writer has stomped its own writes and raced well ahead.

How? Consider what happens when the execution of the reader and writer are interleaved in this setup:

```
size: 5
capacity: 5

 rdr
 |
|0: Some(5) |1: Some(6) |2: Some(7) |3: Some(8) |4: Some(9) |
 |
 wrt
```

The writer thread has looped through the ring twice and is poised at the start of the ring to write `10`. The reader has been through the ring once and is expecting to see `5`. Consider what happens if the reader's decrement of size makes it into main memory before the `mem::replace` happens. Imagine if, then, the writer is woken up just as its size and capacity is checked. Imagine if, in addition to that, the writer writes its new `cur` to the main memory before the reader wakes back up. You'll get this situation:

```
size: 5
capacity: 5

 rdr
 |
|0: Some(10) |1: Some(6) |2: Some(7) |3: Some(8) |4: Some(9) |
              |
              wrt
```

That's what we see in practice. Sometimes. Here's the trick with concurrent programming at the metal of the machine: you're dealing with probabilities. It's entirely possible for our program to run successfully or, worse, run successfully on one CPU architecture and not on another. Randomized testing and introspection of programs like this are *vital*. In fact, back in `Chapter 1`, *Preliminaries – Machine Architecture and Getting Started with Rust*, we discussed a `valgrind` suite tool called `helgrind`, but didn't have a real use for it then. We do

now. What does `helgrind` have to say about our intentionally racey program?

We'll run `helgrind` like so:

```
> valgrind --tool=helgrind --history-level=full --log-
file="results.txt" ./data_race00
```

This will print the full history of the program but place the results in a file on disk for easier inspection. Here's some of the output:

```
==19795== ------------------------------------------------------------
---
==19795==
==19795== Possible data race during write of size 4 at 0x621A050 by
thread #3
==19795== Locks held: none
==19795==    at 0x10F8E2: data_race00::writer::h4524c5c44483b66e (in
/home/blt/projects/us/troutwine/concurrency_in_rust/external_projects/d
ata_races/data_race00)
==19795==    by 0x10FE75:
_ZN3std10sys_common9backtrace28__rust_begin_short_backtrace17ha5ca0c098
55dd05fE.llvm.5C463C64 (in
/home/blt/projects/us/troutwine/concurrency_in_rust/external_projects/d
ata_races/data_race00)
==19795==    by 0x12668E: __rust_maybe_catch_panic (lib.rs:102)
==19795==    by 0x110A42:
_$LT$F$u20$as$u20$alloc..boxed..FnBox$LT$A$GT$$GT$::call_box::h6b5d5a2a
83058684 (in
/home/blt/projects/us/troutwine/concurrency_in_rust/external_projects/d
ata_races/data_race00)
==19795==    by 0x1191A7: call_once<(),()> (boxed.rs:798)
==19795==    by 0x1191A7:
std::sys_common::thread::start_thread::hdc3a308e21d56a9c (thread.rs:24)
==19795==    by 0x112408:
std::sys::unix::thread::Thread::new::thread_start::h555aed63620dece9
(thread.rs:90)
==19795==    by 0x4C32D06: mythread_wrapper (hg_intercepts.c:389)
==19795==    by 0x5251493: start_thread (pthread_create.c:333)
==19795==    by 0x5766AFE: clone (clone.S:97)
==19795==
==19795== This conflicts with a previous write of size 8 by thread #2
==19795== Locks held: none
```

```
==19795==    at 0x10F968: data_race00::reader::h87e804792f6b43da (in
/home/blt/projects/us/troutwine/concurrency_in_rust/external_projects/d
ata_races/data_race00)
==19795==    by 0x12668E: __rust_maybe_catch_panic (lib.rs:102)
==19795==    by 0x110892:
_$LT$F$u20$as$u20$alloc..boxed..FnBox$LT$A$GT$$GT$::call_box::h4b5b7e5f
469a419a (in
/home/blt/projects/us/troutwine/concurrency_in_rust/external_projects/d
ata_races/data_race00)
==19795==    by 0x1191A7: call_once<(),()> (boxed.rs:798)
==19795==    by 0x1191A7:
std::sys_common::thread::start_thread::hdc3a308e21d56a9c (thread.rs:24)
==19795==    by 0x112408:
std::sys::unix::thread::Thread::new::thread_start::h555aed63620dece9
(thread.rs:90)
==19795==    by 0x4C32D06: mythread_wrapper (hg_intercepts.c:389)
==19795==    by 0x5251493: start_thread (pthread_create.c:333)
==19795==    by 0x5766AFE: clone (clone.S:97)
==19795==  Address 0x621a050 is in a rw- anonymous segment
==19795==
==19795== ----------------------------------------------------------------
---
```

The output is less clear than it might be, but helgrind is warning us about the data stomping that we're already aware of. The reader is encouraged to run helgrind for themselves and inspect the whole history.

Let's improve this situation. Clearly, we have a problem with racing reads and writes, but we also have a problem in terms of the behavior of the writer. It stomps its own writes and is entirely unaware of it. With an adjustment to the writer, we can stop absent-mindedly stomping on writes as follows:

```
fn writer(mut ring: Ring) -> () {
    let mut offset: isize = 0;
    let mut cur: u32 = 0;
    loop {
        unsafe {
            if (*ring).size != ((*ring).capacity as usize) {
                assert!(mem::replace(&mut *(*ring).data.offset(offset),
                Some(cur)).is_none());
```

```
                (*ring).size += 1;
                cur = cur.wrapping_add(1);
                offset += 1;
                offset %= (*ring).capacity;
            } else {
                thread::yield_now();
            }
        }
    }
}
```

Afterwards, we run the program a few times and find the following:

```
> ./data_race01
thread '<unnamed>thread '' panicked at '<unnamed>assertion failed:
mem::replace(&mut *(*ring).data.offset(offset), Some(cur)).is_none()'
panicked at '', assertion failed: `(left == right)`
  left: `20`,
   right: `10`data_race01.rs', :data_race01.rs65::8517:
   17note: Run with `RUST_BACKTRACE=1` for a backtrace.

thread 'main' panicked at 'called `Result::unwrap()` on an `Err` value:
Any', libcore/result.rs:945:5
```

That's fun! Both threads have crashed for the same reason. The writer has inappropriately stomped a write and the reader has read it.

Getting back to safety

Like we discussed previously, the preceding examples were intentionally low-level and unsafe. How can we build something similar with the bits and pieces that Rust provides us with? Here's one approach:

```
use std::{mem, thread};
use std::sync::{Arc, Mutex};

struct Ring {
    size: usize,
    data: Vec<Option<u32>>,
}

impl Ring {
    fn with_capacity(capacity: usize) -> Ring {
        let mut data: Vec<Option<u32>> =
Vec::with_capacity(capacity);
        for _ in 0..capacity {
            data.push(None);
        }
        Ring {
            size: 0,
            data: data,
        }
    }

    fn capacity(&self) -> usize {
        self.data.capacity()
    }

    fn is_full(&self) -> bool {
        self.size == self.data.capacity()
    }

    fn emplace(&mut self, offset: usize, val: u32) -> Option<u32> {
        self.size += 1;
        let res = mem::replace(&mut self.data[offset], Some(val));
```

```
            res
        }

        fn displace(&mut self, offset: usize) -> Option<u32> {
            let res = mem::replace(&mut self.data[offset], None);
            if res.is_some() {
                self.size -= 1;
            }
            res
        }
    }

    fn writer(ring_lk: Arc<Mutex<Ring>>) -> () {
        let mut offset: usize = 0;
        let mut cur: u32 = 0;
        loop {
            let mut ring = ring_lk.lock().unwrap();
            if !ring.is_full() {
                assert!(ring.emplace(offset, cur).is_none());
                cur = cur.wrapping_add(1);
                offset += 1;
                offset %= ring.capacity();
            } else {
                thread::yield_now();
            }
        }
    }

    fn reader(read_limit: usize, ring_lk: Arc<Mutex<Ring>>) -> () {
        let mut offset: usize = 0;
        let mut cur: u32 = 0;
        while (cur as usize) < read_limit {
            let mut ring = ring_lk.lock().unwrap();
            if let Some(num) = ring.displace(offset) {
                assert_eq!(num, cur);
                cur = cur.wrapping_add(1);
                offset += 1;
                offset %= ring.capacity();
            } else {
                drop(ring);
                thread::yield_now();
            }
        }
    }

    fn main() {
        let capacity = 10;
        let read_limit = 1_000_000;
```

```
        let ring = Arc::new(Mutex::new(Ring::with_capacity(capacity)));

        let reader_ring = Arc::clone(&ring);
        let reader_jh = thread::spawn(move || {
            reader(read_limit, reader_ring);
        });
        let _writer_jh = thread::spawn(move || {
            writer(ring);
        });

        reader_jh.join().unwrap();
    }
```

This is awfully similar to the previous, racey programs. We have Ring, which holds a size and is a set of operations around a `Vec<Option<u32>>`. This time, the `vec` is not exploded into a raw pointer and the implementation of Ring is fleshed out some more. Actually, it was possible to provide more of an abstract implementation in our previous examples—as we have seen by poking around inside Rust itself—but indirection and unsafety make for a rough combination. It's sometimes the case that indirect, unsafe code is harder to recognize as flawed than direct, unsafe code. Anyhow, you'll note that `Ring: !Send` in this implementation. Instead, the writer and reader threads operate on `Arc<Mutex<Ring>>`.

Safety by exclusion

Let's talk about `Mutex`. Mutexes provide MUTual EXclusion among threads. Rust's `Mutex` works just as you'd expect coming from other programming languages. Any thread that calls a lock on a `Mutex` will acquire the lock or block until such a time as the thread that holds the mutex unlocks it. The type for lock is `lock(&self) -> LockResult<MutexGuard<T>>`. There's a neat trick happening here. `LockResult<Guard>` is `Result<Guard>, PoisonError<Guard>>`. That is, the thread that acquires a mutex lock actually gets a result, which containers the `MutexGuard<T>` on success or a *poisoned* notification. Rust poisons its mutexes when the holder of the mutex crashes, a tactic that helps prevent the continued operation of a multi-threaded propagation that has crashed in only one thread. For this very reason, you'll find that many Rust programs do not inspect the return of a lock call and immediately unwrap it. The `MutexGuard<T>`, when dropped, unlocks the mutex. Once the mutex guard is unlocked, it's no longer possible to access the data ensconced inside and, so, there's no way for one thread to interact poorly with another. A `Mutex` is both `Sync` and `Send`, which makes good sense. Why then do we wrap our `Mutex<Ring>` in an `Arc`? What even is an `Arc<T>`?

The `Arc<T>` is an atomic reference counting pointer. As discussed previously, `Rc<T>` is not thread-safe because if it were marked as such there'd be a race on the inner strong/weak counters, much like in our intentionally racey programs. `Arc<T>` is built on top of atomic integers, which we'll go into more detail on in the next chapter. Suffice it to say for now, the `Arc<T>` is able to act as a reference counting pointer without introducing data races between threads. `Arc<T>` can be used everywhere `Rc<T>` can, except those atomic integers are not free. If you can use `Rc<T>`, do so. Anyway, more on this next chapter.

Why `Arc<Mutex<Ring>>`? Well, `Mutex<Ring>` can be moved but it cannot be cloned. Let's take a look inside `Mutex`:

```
pub struct Mutex<T: ?Sized> {
    // Note that this mutex is in a *box*, not inlined into
    // the struct itself. Once a native mutex has been used
    // once, its address can never change (it can't be
    // moved). This mutex type can be safely moved at any time,
    // so to ensure that the native mutex is used correctly we
    // box the inner mutex to give it a constant address.
    inner: Box<sys::Mutex>,
    poison: poison::Flag,
    data: UnsafeCell<T>,
}
```

We can see that `T` may or may not be sized and that `T` is stored in an `UnsafeCell<T>` next to something called `sys::Mutex`. Each platform that Rust runs on will provide its own form of mutex, being that these are often tied into the operating system environment. If you take a look at the rustc code, you'll find that `sys::Mutex` is a wrapper around system-dependent mutex implementations. The `Unix` implementation is in `src/libstd/sys/unix/mutex.rs` and is an `UnsafeCell` around `pthread_mutex_t`, as you might expect:

```
pub struct Mutex { inner: UnsafeCell<libc::pthread_mutex_t> }
```

It's not strictly necessary for `Mutex` to be implemented on top of system-dependent foundations, as we'll see in the chapter on atomic primitives when we build our own locks. Generally speaking though, it's a good call to use what's available and well-tested unless there's a good reason not to (like pedagogy).

Now, what would happen if we were to clone `Mutex<T>`? We would need a new allocation of the system mutex, for one, and possibly a new `UnsafeCell`, if `T` itself were even clonable. The new system mutex is the real problem—threads have to synchronize on the same structure in memory. Tossing `Mutex` inside of an `Arc` solves that

problem. Cloning the `Arc`, like cloning an `Rc`, creates new strong references to the `Mutex`. These references are immutable, though. How does that work out? For one, the mutex inside the `Arc` is never changed. In the abstract model that Rust provides, the mutex itself has no internal state and there's nothing really being mutated by locking threads. Of course, that's not actually true, by inspection. The Rust `Mutex` only behaves that way because of the interior `UnsafeCell` surrounding system-dependent structures. Rust `Mutex` makes use of the interior mutability that `UnsafeCell` allows. The mutex itself stays immutable while the interior `T` is mutably referenced through the `MutexGuard`. This is safe in Rust's memory model as there's only one mutable reference to `T` at any given time on account of mutual exclusion:

```
fn main() {
    let capacity = 10;
    let read_limit = 1_000_000;
    let ring = Arc::new(Mutex::new(Ring::with_capacity(capacity)));

    let reader_ring = Arc::clone(&ring);
    let reader_jh = thread::spawn(move || {
        reader(read_limit, reader_ring);
    });
    let _writer_jh = thread::spawn(move || {
        writer(ring);
    });

    reader_jh.join().unwrap();
}
```

We wrap our `Ring`, which is not thread-safe in the least, in an `Arc`, `Mutex` layer, clone this to the reader, and move it into the writer. Here, this is a matter of style, but it's important to realize that if a main thread creates and clones an `Arc` into child threads then the contents of `Arc` are still alive, at least as long as the main thread is. For instance, if a file handler were held in a main-thread `Arc`, cloned to temporary start-up workers, and then not dropped, the file handler itself would never be closed. This may or may not be what your program intends, of course. The reader and the writer each take a lock on the mutex—`Arc`

has convenient `Deref`/`DerefMut` implementations—and then perform their action. Running this new program through `helgrind` gives a clean run. The reader is encouraged to confirm this on their own system.

Moreover, the program itself runs to completion successfully after repeated runs. The unfortunate thing is that, theoretically, it's not especially efficient. Locking a mutex is not free, and while our thread's operations are short—a handful of arithmetic in addition to one memory swap—while one thread holds the mutex the other waits dumbly. Linux perf proves this somewhat:

```
> perf stat --event task-clock,context-switches,page-
faults,cycles,instructions,branches,branch-misses,cache-
references,cache-misses ./data_race02

 Performance counter stats for './data_race02':

        988.944526      task-clock (msec)         #    1.943 CPUs
utilized
             8,005      context-switches          #    0.008 M/sec
               137      page-faults               #    0.139 K/sec
     2,836,237,759      cycles                    #    2.868 GHz
     1,074,887,696      instructions              #    0.38  insn per
cycle
       198,103,111      branches                  #  200.318 M/sec
         1,557,868      branch-misses             #    0.79% of all
branches
        63,377,456      cache-references          #   64.086 M/sec
             3,326      cache-misses              #    0.005 % of all
cache refs

       0.508990976 seconds time elapsed
```

Only 1.3 CPUs are used during the execution. For a program as simple as this, we're likely better off with the simplest approach. Also, because of cache write effects in the presence of a memory barrier—of which a mutex assuredly is one—it *may* be cheaper to prefer mutual exclusion instead of fine-grained locking strategies. Ultimately, it comes down to finding the trade-off in development time, need for machine efficiency, and defining what machine

efficiency is for the CPU in use. We'll see some of this in later chapters.

Using MPSC

We've come at Ring from an odd point of view, looking to build a malfunctioning program from the start only to improve it later. In a real program where purposes and functions gradually drift from their origins, it's worth occasionally asking what a piece of software actually *does* as an abstract concept. So, what is it that Ring actually *does*? For one, it's bounded, we know that. It has a fixed capacity and the reader/writer pair are careful to stay within that capacity. Secondly, it's a structure that's intended to be operated on by two or more threads with two roles: reading and writing. Ring is a means of passing `u32` between threads, being read in the order that they were written.

Happily, as long as we're willing to accept only a single reader thread, Rust ships with something in the standard library to cover this common need—`std::sync::mpsc`. The Multi Producer Single Consumer queue is one where the writer end—called `Sender<T>`—may be cloned and moved into multiple threads. The reader thread—called `Receiver<T>`—can only be moved. There are two variants of sender available—`Sender<T>` and `SyncSender<T>`. The first represents an unbounded MPSC, meaning the channel will allocate as much space as needed to hold the `T`s sent into it. The second represents a bounded MPSC, meaning that the internal storage of the channel has a fixed upper capacity. While the Rust documentation describes `Sender<T>` and `SyncSender<T>` as being asynchronous and synchronous respectively this is not, strictly, true. `SyncSender<T>::send` will block if there is no space available in the channel but there is also a `SyncSender<T>::try_send` which is of type `try_send(&self, t: T) -> Result<(), TrySendError<T>>`. It's possible to use Rust's MPSC in bounded memory, keeping in mind that the

caller will have to have a strategy for what to do with inputs that are rejected for want of space to place them in.

What does our `u32` passing program look like using Rust's MPSC? Like so:

```
use std::thread;
use std::sync::mpsc;

fn writer(chan: mpsc::SyncSender<u32>) -> () {
    let mut cur: u32 = 0;
    while let Ok(()) = chan.send(cur) {
        cur = cur.wrapping_add(1);
    }
}

fn reader(read_limit: usize, chan: mpsc::Receiver<u32>) -> () {
    let mut cur: u32 = 0;
    while (cur as usize) < read_limit {
        let num = chan.recv().unwrap();
        assert_eq!(num, cur);
        cur = cur.wrapping_add(1);
    }
}

fn main() {
    let capacity = 10;
    let read_limit = 1_000_000;
    let (snd, rcv) = mpsc::sync_channel(capacity);

    let reader_jh = thread::spawn(move || {
        reader(read_limit, rcv);
    });
    let _writer_jh = thread::spawn(move || {
        writer(snd);
    });

    reader_jh.join().unwrap();
}
```

This is significantly shorter than any of our previous programs as well as being not at all prone to arithmetic bugs. The send type of `SyncSender` is `send(&self, t: T) -> Result<(), SendError<T>>`, meaning that

`SendError` has to be cared for to avoid a program crash. `SendError` is only returned when the remote side of an MPSC channel is disconnected, as will happen in this program when the reader hits its `read_limit`. The performance characteristics of this odd program are not quite as quick as for the last `Ring` program:

```
> perf stat --event task-clock,context-switches,page-
faults,cycles,instructions,branches,branch-misses,cache-
references,cache-misses ./data_race03

 Performance counter stats for './data_race03':

        760.250406      task-clock (msec)   #    0.765 CPUs utilized
           200,011      context-switches    #    0.263 M/sec
               135      page-faults         #    0.178 K/sec
     1,740,006,788      cycles              #    2.289 GHz
     1,533,396,017      instructions        #    0.88  insn per cycle
       327,682,344      branches            #  431.019 M/sec
     741,095      branch-misses             #    0.23% of all branches
        71,426,781      cache-references    #   93.952 M/sec
          4,082      cache-misses          #  0.006 % of all cache refs

       0.993142979 seconds time elapsed
```

But it's well within the margin of error, especially considering the ease of programming. One thing worth keeping in mind is that messages in MPSC flow in only one direction: the channel is not bi-directional. In this way, MPSC is not suitable for request/response patterns. It's not unheard of to layer two or more MPSCs together to achieve this, with the understanding that a single consumer on either side is sometimes not suitable.

A telemetry server

Let's build a descriptive statistics server. These pop up often in one way or another inside organizations: a thing is needed that consumes events, does some kind of descriptive statistic computation over those things, and then multiplexes the descriptions out to other systems. The very reason my work project, postmates/cernan (`https://crates.io/crates/cernan`), exists is to service this need at scale on resource constrained devices without tying operations staff into any kind of pipeline. What we'll build here now is a kind of mini-cernan, something whose flow is as follows:

```
                              _--> high_filter --> cma_egress
                             /
 telemetry -> ingest_point ---
   (udp)                     \_--> low_filter --> ckms_egress
```

The idea is to take `telemetry` from a simple UDP protocol, receive it as quickly as possible to avoid the OS dumping packets, pass these points through a high and low filter, and then hand the filtered points off to two different statistical `egress` points. The `egress` associated with the high filter computes a continuous moving average, while the low filter associated `egress` computes a quantile summary using an approximation algorithm from the postmates/quantiles (`https://crates.io/crates/quantiles`) library.

Let's dig in. First, let's look at `Cargo.toml` for the project:

```
[package]
name = "telem"
version = "0.1.0"
```

```
[dependencies]
quantiles = "0.7"
seahash = "3.0"

[[bin]]
name = "telem"
doc = false
```

Short and to the point. We draw in quantiles, as mentioned previously, as well as seahasher. Seahasher is a particularly fast—but not cryptographically safe—hasher that we'll substitute into `HashMap`. More on that shortly. Our executable is broken out into `src/bin/telem.rs` since this project is a split library/binary setup:

```
extern crate telem;

use std::{thread, time};
use std::sync::mpsc;
use telem::IngestPoint;
use telem::egress::{CKMSEgress, CMAEgress, Egress};
use telem::event::Event;
use telem::filter::{Filter, HighFilter, LowFilter};

fn main() {
    let limit = 100;
    let (lp_ic_snd, lp_ic_rcv) = mpsc::channel::<Event>();
    let (hp_ic_snd, hp_ic_rcv) = mpsc::channel::<Event>();
    let (ckms_snd, ckms_rcv) = mpsc::channel::<Event>();
    let (cma_snd, cma_rcv) = mpsc::channel::<Event>();

    let filter_sends = vec![lp_ic_snd, hp_ic_snd];
    let ingest_filter_sends = filter_sends.clone();
    let _ingest_jh = thread::spawn(move || {
        IngestPoint::init("127.0.0.1".to_string(), 1990,
        ingest_filter_sends).run();
    });
    let _low_jh = thread::spawn(move || {
        let mut low_filter = LowFilter::new(limit);
        low_filter.run(lp_ic_rcv, vec![ckms_snd]);
    });
    let _high_jh = thread::spawn(move || {
        let mut high_filter = HighFilter::new(limit);
        high_filter.run(hp_ic_rcv, vec![cma_snd]);
    });
```

```
        let _ckms_egress_jh = thread::spawn(move || {
            CKMSEgress::new(0.01).run(ckms_rcv);
        });
        let _cma_egress_jh = thread::spawn(move || {
            CMAEgress::new().run(cma_rcv);
        });

        let one_second = time::Duration::from_millis(1_000);
        loop {
            for snd in &filter_sends {
                snd.send(Event::Flush).unwrap();
            }
            thread::sleep(one_second);
        }
    }
```

There's a fair bit going on here. The first eight lines are imports of the library bits and pieces we need. The body of main is dominated by setting up our worker threads and feeding the appropriate channels into them. Note that some threads take multiple sender sides of a channel:

```
    let filter_sends = vec![lp_ic_snd, hp_ic_snd];
    let ingest_filter_sends = filter_sends.clone();
    let _ingest_jh = thread::spawn(move || {
        IngestPoint::init("127.0.0.1".to_string(), 1990,
        ingest_filter_sends).run();
    });
```

That's how we do fanout using Rust MPSC. Let's take a look at `IngestPoint`, in fact. It's defined in `src/ingest_point.rs`:

```
use event;
use std::{net, thread};
use std::net::ToSocketAddrs;
use std::str;
use std::str::FromStr;
use std::sync::mpsc;
use util;

pub struct IngestPoint {
    host: String,
```

```
        port: u16,
        chans: Vec<mpsc::Sender<event::Event>>,
    }
```

An `IngestPoint` is a host—either an IP address or a DNS hostname, per `ToSocketAddrs`—a port and a vector of `mpsc::Sender<event::Event>`. The inner type is something we've defined:

```
    #[derive(Clone)]
    pub enum Event {
        Telemetry(Telemetry),
        Flush,
    }

    #[derive(Debug, Clone)]
    pub struct Telemetry {
        pub name: String,
        pub value: u32,
    }
```

`telem` has two kinds of *events* that flow through it—`Telemetry`, which comes from `IngresPoint` and `Flush`, which comes from the main thread. `Flush` acts like a clock-tick for the system, allowing the individual subsystems of the project to keep track of time without making reference to the wall-clock. It's not uncommon in embedded programs to define time in terms of some well-known pulse and, when possible, I've tried to keep to that in parallel programming as well. If nothing else, it helps with testing to have time as an externally pushed attribute of the system. Anyhow, back to `IngestPoint`:

```
impl IngestPoint {
    pub fn init(
        host: String,
        port: u16,
        chans: Vec<mpsc::Sender<event::Event>>,
    ) -> IngestPoint {
        IngestPoint {
            chans: chans,
            host: host,
            port: port,
```

```
        }
    }

    pub fn run(&mut self) {
        let mut joins = Vec::new();

        let addrs = (self.host.as_str(), self.port).to_socket_addrs();
        if let Ok(ips) = addrs {
            let ips: Vec<_> = ips.collect();
            for addr in ips {
                let listener =
                    net::UdpSocket::bind(addr)
                        .expect("Unable to bind to UDP socket");
                let chans = self.chans.clone();
                joins.push(thread::spawn(move || handle_udp(chans,
                                                    &listener)));
            }
        }

        for jh in joins {
            jh.join().expect("Uh oh, child thread panicked!");
        }
    }
}
```

The first bit of this, `init`, is just setup. The run function calls `to_socket_addrs` on our host/port pair and retrieves all the associated IP addresses. Each of these addresses get a `UdpSocket` bound to them and an OS thread to listen for datagrams from that socket. This is wasteful in terms of thread overhead and later in this book we'll discuss evented-IO alternatives. Cernan, discussed previously, being a production system makes use of Mio in its *Sources*. The key function here is `handle_udp`, the function that gets passed to the new listener threads. It is as follows:

```
fn handle_udp(mut chans: Vec<mpsc::Sender<event::Event>>,
              socket: &net::UdpSocket) {
    let mut buf = vec![0; 16_250];
    loop {
        let (len, _) = match socket.recv_from(&mut buf) {
            Ok(r) => r,
            Err(e) => {
                panic!(
```

```
                    format!("Could not read UDP socket with \
                            error {:?}", e)),
            }
        };
        if let Some(telem) =
        parse_packet(str::from_utf8(&buf[..len]).unwrap()) {
            util::send(&mut chans, event::Event::Telemetry(telem));
        }
    }
}
```

The function is a simple infinite loop that pulls datagrams off the socket into a 16 KB buffer—comfortably larger than most datagrams—and then calls `parse_packet` on the result. If the datagram was a valid example of our as yet unspecified protocol, then we call `util::send` to send the `Event::Telemetry` out over the `Sender<event::Event>` in `chans`. `util::send` is little more than a for loop:

```
pub fn send(chans: &[mpsc::Sender<event::Event>], event:
event::Event) {
    if chans.is_empty() {
        return;
    }

    for chan in chans.iter() {
        chan.send(event.clone()).unwrap();
    }
}
```

The ingest payload is nothing special: a name of non-whitespace characters followed by one or more whitespace characters followed by a `u32`, all string encoded and utf8 valid:

```
fn parse_packet(buf: &str) -> Option<event::Telemetry> {
    let mut iter = buf.split_whitespace();
    if let Some(name) = iter.next() {
        if let Some(val) = iter.next() {
            match u32::from_str(val) {
                Ok(int) => {
                    return Some(event::Telemetry {
                        name: name.to_string(),
```

```
                            value: int,
                        })
                    }
                    Err(_) => return None,
                };
            }
        }
        None
    }
```

Popping out to the filters, both `HighFilter` and `LowFilter` are done in terms of a common `Filter` trait, defined in `src/filter/mod.rs`:

```
use event;
use std::sync::mpsc;
use util;

mod high_filter;
mod low_filter;

pub use self::high_filter::*;
pub use self::low_filter::*;

pub trait Filter {
    fn process(
        &mut self,
        event: event::Telemetry,
        res: &mut Vec<event::Telemetry>,
    ) -> ();

    fn run(
        &mut self,
        recv: mpsc::Receiver<event::Event>,
        chans: Vec<mpsc::Sender<event::Event>>,
    ) {
        let mut telems = Vec::with_capacity(64);
        for event in recv.into_iter() {
            match event {
                event::Event::Flush => util::send(&chans,
                 event::Event::Flush),
                event::Event::Telemetry(telem) => {
                    self.process(telem, &mut telems);
                    for telem in telems.drain(..) {
                        util::send(&chans,
                        event::Event::Telemetry(telem))
```

```
                    }
                }
            }
        }
    }
}
```

Any implementing filter is responsible for providing their own process. It's this function that the default run calls when an `Event` is pulled from the `Receiver<Event>` and found to be a `Telemetry`. Though neither high nor low filters make use of it, the process function is able to inject new `Telemetry` into the stream if it's programmed to do so by pushing more onto the passed `telems` vector. That's how cernan's *programmable filter* is able to allow end users to create `telemetry` from Lua scripts. Also, why pass `telems` rather than have the process return a vector of `Telemetry`? It avoids continual small allocations. Depending on the system, allocations will not necessarily be uncoordinated between threads—meaning high-load situations can suffer from mysterious pauses—and so it's good style to avoid them where possible if the code isn't twisted into some weird version of itself by taking such care.

Both low and high filters are basically the same. The low filter passes a point through itself if the point is less than or equal to a pre-defined limit, where the high filter is greater than or equal to it. Here's `LowFilter`, defined in `src/filter/low_filter.rs`:

```
use event;
use filter::Filter;

pub struct LowFilter {
    limit: u32,
}

impl LowFilter {
    pub fn new(limit: u32) -> Self {
        LowFilter { limit: limit }
    }
}
```

```
impl Filter for LowFilter {
    fn process(
        &mut self,
        event: event::Telemetry,
        res: &mut Vec<event::Telemetry>,
    ) -> () {
        if event.value <= self.limit {
            res.push(event);
        }
    }
}
```

`Egress` of telemetry is defined similarly to the way filter is done, split out into a sub-module and a common trait. The trait is present in `src/egress/mod.rs`:

```
use event;
use std::sync::mpsc;

mod cma_egress;
mod ckms_egress;

pub use self::ckms_egress::*;
pub use self::cma_egress::*;

pub trait Egress {
    fn deliver(&mut self, event: event::Telemetry) -> ();

    fn report(&mut self) -> ();

    fn run(&mut self, recv: mpsc::Receiver<event::Event>) {
        for event in recv.into_iter() {
            match event {
                event::Event::Telemetry(telem) =>
self.deliver(telem),
                event::Event::Flush => self.report(),
            }
        }
    }
}
```

The deliver function is intended to give the `egress` its `Telemetry` for storage. The report is intended to force the implementors of `Egress` to

issue their summarized telemetry to the outside world. Both of our `Egress` implementors—`CKMSEgress` and `CMAEgress`—merely print their information but you can well imagine an `Egress` that emits its information out over some network protocol to a remote system. This is, in fact, exactly what cernan's `Sinks` do, across many protocols and transports. Let's look at a single egress, as they're both very similar. `CKMSEgress` is defined in `src/egress/ckms_egress.rs`:

```
use egress::Egress;
use event;
use quantiles;
use util;

pub struct CKMSEgress {
    error: f64,
    data: util::HashMap<String, quantiles::ckms::CKMS<u32>>,
    new_data_since_last_report: bool,
}

impl Egress for CKMSEgress {
    fn deliver(&mut self, event: event::Telemetry) -> () {
        self.new_data_since_last_report = true;
        let val = event.value;
        let ckms = self.data
            .entry(event.name)
            .or_insert(quantiles::ckms::CKMS::new(self.error));
        ckms.insert(val);
    }

    fn report(&mut self) -> () {
        if self.new_data_since_last_report {
            for (k, v) in &self.data {
                for q in &[0.0, 0.25, 0.5, 0.75, 0.9, 0.99] {
                    println!("[CKMS] {} {}:{}", k, q,
                             v.query(*q).unwrap().1);
                }
            }
            self.new_data_since_last_report = false;
        }
    }
}

impl CKMSEgress {
    pub fn new(error: f64) -> Self {
```

```
        CKMSEgress {
            error: error,
            data: Default::default(),
            new_data_since_last_report: false,
        }
    }
}
```

Note that `data: util::HashMap<String, quantiles::ckms::CKMS<u32>>`. This `util::HashMap` is a type alias for `std::collections::HashMap<K, V, hash::BuildHasherDefault<SeaHasher>>`, as mentioned previously. The cryptographic security of hashing here is less important than the speed of hashing, which is why we go with `SeaHasher`. There are a great many alternative hashers available in crates and it's a fancy trick to be able to swap them out for your use case. `quantiles::ckms::CKMS` is an approximate data structure, defined in *Effective Computation of Biased Quantiles Over Data Streams* by Cormode et al. Many summary systems run in limited space but are willing to tolerate errors. The CKMS data structure allows for point shedding while keeping guaranteed error bounds on the quantile approximations. The discussion of the data structure is outside the domain of this book but the implementation is interesting and the paper is remarkably well-written. Anyhow, that's what the error setting is all about. If you flip back to the main function, note that we hard-code the error as being 0.01, or, any quantile summary is guaranteed to be off true within 0.01.

That, honestly, is pretty much it. We've just stepped through the majority of a non-trivial Rust program built around the MPSC abstraction provided in the standard library. Let's fiddle with it some. In one shell, start `telem`:

```
> cargo run --release
   Compiling quantiles v0.7.0
   Compiling seahash v3.0.5
   Compiling telem v0.1.0
(file:///Users/blt/projects/us/troutwine/concurrency_in_rust/external_p
rojects/telem)
```

```
    Finished release [optimized] target(s) in 7.16 secs
     Running `target/release/telem`
```

In another shell, start sending UDP packets. On macOS, you can use `nc` like so:

```
> echo "a 10" | nc -c -u 127.0.0.1 1990
> echo "a 55" | nc -c -u 127.0.0.1 1990
```

The call is similar on Linux; you just have to be careful not to wait for a response is all. In the original shell, you should see an output like this after a second:

```
[CKMS] a 0:10
[CKMS] a 0.25:10
[CKMS] a 0.5:10
[CKMS] a 0.75:10
[CKMS] a 0.9:10
[CKMS] a 0.99:10
[CKMS] a 0:10
[CKMS] a 0.25:10
[CKMS] a 0.5:10
[CKMS] a 0.75:55
[CKMS] a 0.9:55
[CKMS] a 0.99:55
```

The points, being below the limit, have gone through the low filter and into the CKMS `egress`. Back to our other shell:

```
> echo "b 1000" | nc -c -u 127.0.0.1 1990
> echo "b 2000" | nc -c -u 127.0.0.1 1990
> echo "b 3000" | nc -c -u 127.0.0.1 1990
```

In the telem shell:

```
[CMA] b 1000
[CMA] b 1500
[CMA] b 2000
```

Bang, just as expected. The points, being above the limit, have gone through the high filter and into the CMA `egress`. So long as no points are coming in, the `telem` should be drawing almost no CPU, waking up each of the threads once every second or so for the Flush pulse. Memory consumption will also be very low, being primarily represented by the large input buffer in `IngestPoint`.

There are problems with this program. In many places, we explicitly panic or unwrap on potential problem points, rather than deal with the issues. While it's reasonable to unwrap in a production program, you should be *very* sure that the error you're blowing your program up for cannot be otherwise dealt with. Most concerning is that the program has no concept of *back-pressure*. If `IngestPoint` were able to produce points faster than the filters or the `egress`es, they could absorb the channels between threads that would keep allocating space. A slight rewrite of the program to use `mpsc::SyncSender` would be suitable—back-pressure is applied as soon as a channel is filled to capacity—but only if dropping points is acceptable. Given that the ingest protocol is in terms of UDP it almost surely is, but the reader can imagine a scenario where it would not be. Rust's standard library struggles in areas where alternative forms of back-pressure are needed but these are, admittedly, esoteric. If you're interested in reading through a production program that works along the lines of `telem` but has none of the defects identified here, I warmly recommend the cernan codebase.

All in all, Rust's MPSC is a very useful tool when constructing parallel systems. In this chapter, we built our own buffered channel, Ring, but that isn't common at all. You'd need a fairly specialized use case to consider not using the standard library's MPSC for intra-thread channel-based communication. We'll examine one such use case in the next chapter after we cover more of Rust's basic concurrency primitives.

Summary

In this chapter, we discussed the foundations of Rust concurrency—`Sync` and `Send`. Furthermore, we started on what makes a primitive thread-safe in Rust and how to build concurrent structures with those primitives. We reasoned through an improperly synchronized program, showing how knowledge of the Rust memory model, augmented by tools such as `helgrind`, allow us to determine what's gone sideways in our programs. This is, perhaps unsurprisingly to the reader, a painstaking process that is, like as not, prone to error. In `Chapter 5`, *Locks – Mutex, Condvar, Barriers and RWLock*, we'll discuss the higher-level coarse synchronization primitives that Rust exposes to the programmer. In `Chapter 6`, *Atomics – the Primitives of Synchronization*, we'll discuss the fine synchronization primitives that modern machines expose.

Further reading

Safe concurrent programming is, unsurprisingly, a very broad topic and the recommendations for further reading here reflect that. The careful reader will note that these references span time and approach, reflecting the broad changes in machines and languages over time.

- *The Art of Multiprocessor Programming*, Maurice Herlihy and Nir Shavit. This book is an excellent introduction to multiprocessor algorithms. Application to systems languages is made a touch difficult by the fact that the authors assume a Java environment—garbage collection is a huge win for implementing reclamation in that, well, you don't have to do it.
- *C++ Concurrency in Action: Practical Multithreading*, Anthony Williams. This book is an excellent pair to TAoMP, being focused on implementation of similar structures in C++. While there is a translation step needed between C++ and Rust, it's not so great a jump as from Java to Rust.
- *LVars: Lattice-based Data Structures for Deterministic Parallelism*, Lindsey Kuper and Ryan Newton. This book presents a certain kind of parallel construction, one influenced by current machines. It is possible, however, that our current models are overly complicated and will someday be seen as archaic, even without having to sacrifice raw

performance as we've done in the move to VM-based languages. This paper presents a construction alternative, influenced by the work done on distributed algorithms in recent years. It may or may not be the future but the reader is encouraged to keep a look out.

- *ffwd: delegation is (much) faster than you think*, Sepideh Roghanchi, Jakob Eriksson, and Nilanjana Basu. Modern machines are odd beasts. Conceptually, the fastest data structure is one that minimizes the wait time of working threads, speeding through instructions to work completion. This is… not entirely the case, as this paper demonstrates. The authors, by carefully maintaining cache locality, are able to outpace more complicated structures that might, theoretically, be much faster due to more aggressive sharing between threads with fine-grained locks.
- *Flat Combining and the Synchronization-Parallelism Tradeoff*, Danny Hendler, Itai Incze, and Nir Shavit. Along the same lines as ffwd, the authors present a flat combining approach to constructing concurrent structures, which relies on coarse exclusive locking between threads with a periodic combination of logs of operations. The interaction with cache makes flat combining *faster* than more complicated lock-free/wait-free alternatives, which do not interact with the cache as gracefully.
- *New Rustacean, e022: Send and Sync*, available at `http://www.newrustacean.com/show_notes/e022/struct.Script.html`. The New Rustacean is an excellent podcast for Rust developers of all

levels. This episode dovetails nicely with the material discussed in the current chapter. Warmly recommended.

- *Effective Computation of Biased Quantiles Over Data Streams*, Graham Cormode, Flip Korn, S. Muthukrishnan, and Divesh Srivastava. This paper underpins the CKMS structure used in telem. It's instructive to examine the difference between the implementation outlined in the paper —based on linked-lists—and the implementation found in the library, a variant of a skip-list. This difference is wholly due to cache locality concerns.

- *The Cernan Project*, various developers, available at `https://github.com/postmates/cernan` under the MIT license. Cernan is an event multiplexing server, the production version of the toy telem discussed in this chapter. As of writing this book, it is 17,000 lines of code contributed by 14 people. Careful attention has been taken to maintain low resource consumption and high levels of performance. I am the primary author of this project and the techniques discussed in this book are applied in cernan.

Locks – Mutex, Condvar, Barriers and RWLock

In this chapter, we're going to do a deep-dive on hopper, the grown-up version of Ring from `Chapter 4`, *Sync and Send – the Foundation of Rust Concurrency*. Hopper's approach to back-pressure—the weakness we identified in telem—is to block when filled to capacity, as `SyncSender` does. Hopper's special trick is that it pages out to disk. The hopper user defines how many bytes of in-memory space hopper is allowed to consume, like `SyncSender`, except in terms of bytes rather than total elements of `T`. Furthermore, the user is able to configure the number of on-disk bytes that are consumed when hopper's in-memory capacity is filled and it has to page out to disk. The other properties of MSPC are held, in-order delivery, retention of data once stored, and so on.

Before we can dig through hopper, however, we need to introduce more of Rust's concurrency primitives. We'll work on some puzzles from *The Little Book of Semaphores* to explain them, which will get a touch hairy in some places on account of how Rust does not have a semaphore available.

By the close of this chapter, we will have:

- Discussed the purpose and use of Mutex, Condvar, Barriers, and RWLock
- Investigated a disk-backed specialization of the standard library's MPSC called hopper

- Seen how to apply QuickCheck, AFL, and comprehensive benchmarks in a production setting

Technical requirements

This chapter requires a working Rust installation. The details of verifying your installation are covered in `Chapter 1`, *Preliminaries – Machine Architecture and Getting Started with Rust*. No additional software tools are required.

You can find the source code for this book's projects on GitHub at: `https://github.com/PacktPublishing/Rust-Concurrency/`. This chapter has its source code under `Chapter05`.

Read many, write exclusive locks – RwLock

Consider a situation where you have a resource that must be manipulated only a single thread at a time, but is safe to be queried by many—that is, you have many readers and only one writer. While we could protect this resource with a `Mutex`, the trouble is that the mutex makes no distinction between its lockers; every thread will be forced to wait, no matter what their intentions. `RwLock<T>` is an alternative to the mutex concept, allowing for two kinds of locks—read and write. Analogously to Rust's references, there can only be one write lock taken at a time but multiple reader locks, exclusive of a write lock. Let's look at an example:

```
use std::thread;
use std::sync::{Arc, RwLock};

fn main() {
    let resource: Arc<RwLock<u16>> = Arc::new(RwLock::new(0));

    let total_readers = 5;

    let mut reader_jhs = Vec::with_capacity(total_readers);
    for _ in 0..total_readers {
        let resource = Arc::clone(&resource);
        reader_jhs.push(thread::spawn(move || {
            let mut total_lock_success = 0;
            let mut total_lock_failure = 0;
            let mut total_zeros = 0;
            while total_zeros < 100 {
                match resource.try_read() {
                    Ok(guard) => {
                        total_lock_success += 1;
                        if *guard == 0 {
                            total_zeros += 1;
                        }
```

```
                    }
                    Err(_) => {
                        total_lock_failure += 1;
                    }
                }
            }
            (total_lock_failure, total_lock_success)
        }));
    }

    {
        let mut loops = 0;
        while loops < 100 {
            let mut guard = resource.write().unwrap();
            *guard = guard.wrapping_add(1);
            if *guard == 0 {
                loops += 1;
            }
        }
    }

    for jh in reader_jhs {
        println!("{:?}", jh.join().unwrap());
    }
}
```

The idea here is that we'll have one writer thread spinning and incrementing, in a wrapping fashion, a shared resource—a `u16`. Once the `u16` has been wrapped 100 times, the writer thread will exit. Meanwhile, a `total_readers` number of read threads will attempt to take a read lock on the shared resource—a `u16` `until` it hits zero `100` times. We're gambling here, essentially, on thread ordering. Quite often, the program will exit with this result:

```
(0, 100)
(0, 100)
(0, 100)
(0, 100)
(0, 100)
```

This means that each reader thread never failed to get its read lock—there were no write locks present. That is, the reader threads were

scheduled before the writer. Our main function only joins on reader handlers and so the writer is left writing as we exit. Sometimes, we'll hit just the right scheduling order, and get the following result:

```
(0, 100)
(126143752, 2630308)
(0, 100)
(0, 100)
(126463166, 2736405)
```

In this particular instance, the second and final reader threads were scheduled just after the writer and managed to catch a time when the guard was not zero. Recall that the first element of the pair is the total number of times the reader thread was not able to get a read lock and was forced to retry. The second is the number of times that the lock was acquired. In total, the writer thread did `(2^18 * 100) ~= 2^24` writes, whereas the second reader thread did `log_2 2630308 ~= 2^21` reads. That's a lot of lost writes, which, maybe, is okay. Of more concern, that's a lot of useless loops, approximately `2^26`. Ocean levels are rising and we're here burning up electricity like nobody had to die for it.

How do we avoid all this wasted effort? Well, like most things, it depends on what we're trying to do. If we need every reader to get every write, then an MPSC is a reasonable choice. It would look like this:

```
use std::thread;
use std::sync::mpsc;

fn main() {
    let total_readers = 5;
    let mut sends = Vec::with_capacity(total_readers);

    let mut reader_jhs = Vec::with_capacity(total_readers);
    for _ in 0..total_readers {
        let (snd, rcv) = mpsc::sync_channel(64);
        sends.push(snd);
        reader_jhs.push(thread::spawn(move || {
            let mut total_zeros = 0;
```

```
                let mut seen_values = 0;
                for v in rcv {
                    seen_values += 1;
                    if v == 0 {
                        total_zeros += 1;
                    }
                    if total_zeros >= 100 {
                        break;
                    }
                }
                seen_values
            }));
        }

        {
            let mut loops = 0;
            let mut cur: u16 = 0;
            while loops < 100 {
                cur = cur.wrapping_add(1);
                for snd in &sends {
                    snd.send(cur).expect("failed to send");
                }
                if cur == 0 {
                    loops += 1;
                }
            }
        }

        for jh in reader_jhs {
            println!("{:?}", jh.join().unwrap());
        }
    }
```

It will run—for a while—and print out the following:

```
6553600
6553600
6553600
6553600
6553600
```

But what if every reader does not need to see every write, meaning that it's acceptable for a reader to miss writes so long as it does not miss all of the writes? We have options. Let's look at one.

Blocking until conditions change – condvar

One option is a condvar, or CONDition VARiable. Condvars are a nifty way to block a thread, pending a change in some Boolean condition. One difficulty is that condvars are associated exclusively with mutexes, but in this example, we don't mind all that much.

The way a condvar works is that, after taking a lock on a mutex, you pass the `MutexGuard` into `Condvar::wait`, which blocks the thread. Other threads may go through this same process, blocking on the same condition. Some other thread will take the same exclusive lock and eventually call either `notify_one` or `notify_all` on the condvar. The first wakes up a single thread, the second wakes up *all* threads. Condvars are subject to spurious wakeup, meaning the thread may leave its block without a notification being sent to it. For this reason condvars check their conditions in a loop. But, once the condvar wakes, you *are* guaranteed to hold the mutex, which prevents deadlocks on spurious wakeup.

Let's adapt our example to use a condvar. There's actually a fair bit going on in this example, so we'll break it down into pieces:

```
use std::thread;
use std::sync::{Arc, Condvar, Mutex};

fn main() {
    let total_readers = 5;
    let mutcond: Arc<(Mutex<(bool, u16)>, Condvar)> =
        Arc::new((Mutex::new((false, 0)), Condvar::new()));
```

Our preamble is similar to the previous examples, but the setup is already quite strange. We're synchronizing threads on `mutcond`, which is an `Arc<(Mutex<(bool, u16)>, Condvar)>`. Rust's condvar is a touch awkward. It's undefined behavior to associate a condvar with more than one mutex, but there's really nothing in the type of `Condvar` that makes that an invariant. We just have to remember to keep them associated. To that end, it's not uncommon to see a `Mutex` and `Condvar` paired up in a tuple, as here. Now, why `Mutex<(bool, u16)>`? The second element of the tuple is our *resource*, which is common to other examples. The first element is a Boolean flag, which we use as a signal to mean that there are writes available. Here are our reader threads:

```
let mut reader_jhs = Vec::with_capacity(total_readers);
for _ in 0..total_readers {
    let mutcond = Arc::clone(&mutcond);
    reader_jhs.push(thread::spawn(move || {
        let mut total_zeros = 0;
        let mut total_wakes = 0;
        let &(ref mtx, ref cnd) = &*mutcond;

        while total_zeros < 100 {
            let mut guard = mtx.lock().unwrap();
            while !guard.0 {
                guard = cnd.wait(guard).unwrap();
            }
            guard.0 = false;

            total_wakes += 1;
            if guard.1 == 0 {
                total_zeros += 1;
            }
        }
        total_wakes
    }));
}
```

Until `total_zeros` hits 100, the reader thread locks the mutex, checks the guard inside the mutex for write availability, and, if there are no writes, does a wait on the condvar, which gives up the lock. The reader thread is then blocked until a `notify_all` is called—as we'll see

shortly. Every reader thread then races to be the first to reacquire the lock. The lucky winner notes that there are no more writes to be read and then does the normal flow we've seen in previous examples. It bears repeating that every thread that wakes up from a condition wait is racing to be the first to reacquire the mutex. Our reader is uncooperative in that it immediately prevents the chance of any other reader threads finding a resource available. However, they will still wake up spuriously and be forced to wait again. Maybe. The reader threads are also competing with the writer thread to acquire the lock. Let's look at the writer thread:

```
    let _ = thread::spawn(move || {
        let &(ref mtx, ref cnd) = &*mutcond;
        loop {
            let mut guard = mtx.lock().unwrap();
            guard.1 = guard.1.wrapping_add(1);
            guard.0 = true;
            cnd.notify_all();
        }
    });
```

The writer thread is an infinite loop, which we orphan in an unjoined thread. Now, it's entirely possible that the writer thread will acquire the lock, bump the resource, notify waiting reader threads, give up the lock, and then immediately re-acquire the lock to begin the while process before any reader threads can get scheduled in. This means it's entirely possible that the resource being zero will happen several times before a reader thread is lucky enough to notice. Let's close out this program:

```
    for jh in reader_jhs {
        println!("{:?}", jh.join().unwrap());
    }
}
```

Ideally, what we'd like is some manner of bi-directionality—we'd like the writer to signal that there are reads and the reader to signal that

there is capacity. This is suspiciously like how Ring in the previous chapter worked through its size variable, when we were careful to not race on that variable, that is. We might, for instance, layer another condition variable into the mix, this one for the writer, but that's not what we have here and the program suffers for it. Here's one run:

```
7243473
6890156
6018468
6775609
6192116
```

Phew! That's significantly more loops than previous examples. None of this is to say that condition variables are hard to use—they're not—it's just that they need to be used in conjunction with other primitives. We'll see an excellent example of this later in the chapter.

Blocking until the gang's all here - barrier

A barrier is a synchronization device that blocks threads until such time that a predefined number of threads have waited on the same barrier. When a barrier's waiting threads wake up, one is declared leader—discoverable by inspecting the `BarrierWaitResult`—but this confers no scheduling advantage. A barrier becomes useful when you wish to delay threads behind an unsafe initialization of some resource—say a C library's internals that have no thread-safety at startup, or have a need to force participating threads to start a critical section at roughly the same time. The latter is the broader category, in your author's experience. When programming with atomic variables, you'll run into situations where a barrier will be useful. Also, consider for a second writings multi-threaded code for low-power devices. There are two strategies possible these days for power management: scaling the CPU to meet requirements, adjusting the runtime of your program live, or burning through a section of your program as quickly as possible and then shutting down the chip. In the latter approach, a barrier is just the primitive you need.

More mutexes, condvars, and friends in action

Admittedly, the examples in the preceding sections got a little convoluted. There's a reason to that madness, I promise, but before we move on I'd like to show you some working examples from the charming *The Little Book of Semaphores*. This book, in case you skipped previous bibliographic notes, is a collection of concurrency puzzles suitable for self-learning, on account of the puzzles being amusing and coming with good hints. As the title implies, the book does make use of the semaphore primitive, which Rust does not have. Though, as mentioned in the previous chapter, we will build a semaphore in the next chapter.

The rocket preparation problem

This puzzle does not actually appear in *The Little Book of Semaphores* but it is based on one of the puzzles there - the cigarette smoker's problem from section 4.5. Personally, I think cigarettes are gross so we're going to reword things a touch. The idea is the same.

We have four threads in total. One thread, the `producer`, randomly publishes one of `fuel`, `oxidizer`, or `astronauts`. The remaining three threads are the consumers, or *rockets*, which must take their resources in the order listed previously. If a rocket doesn't get its resources in that order, it's not safe to prepare the rocket, and if it doesn't have all three, the rocket can't lift-off. Moreover, once all the rockets are prepped, we want to start a 10 second count-down, only after which may the rockets lift-off.

The preamble to our solution is a little longer than usual, in the interest of keeping the solution as a standalone:

```
use std::{thread, time};
use std::sync::{Arc, Barrier, Condvar, Mutex};

// NOTE if this were a crate, rather than a stand-alone
// program, we could and _should_ use the XorShiftRng
// from the 'rand' crate.
pub struct XorShiftRng {
    state: u32,
}

impl XorShiftRng {
    fn with_seed(seed: u32) -> XorShiftRng {
        XorShiftRng { state: seed }
    }
```

```
        fn next_u32(&mut self) -> u32 {
            self.state ^= self.state << 13;
            self.state ^= self.state >> 17;
            self.state ^= self.state << 5;
            return self.state;
        }
    }
```

We don't really need excellent randomness for this solution—the OS scheduler injects enough already—but just something small-ish. `XorShift` fits the bill. Now, for our resources:

```
    struct Resources {
        lock: Mutex<(bool, bool, bool)>,
        fuel: Condvar,
        oxidizer: Condvar,
        astronauts: Condvar,
    }
```

The struct is protected by a single `Mutex<(bool, bool, bool)>`, the Boolean being a flag to indicate that there's a resource available. We hold the first flag to mean `fuel`, the second `oxidizer`, and the third `astronauts`. The remainder of the struct are condvars to match each of these resource concerns. The `producer` is a straightforward infinite loop:

```
fn producer(resources: Arc<Resources>) {
    let mut rng = XorShiftRng::with_seed(2005);
    loop {
        let mut guard = resources.lock.lock().unwrap();
        (*guard).0 = false;
        (*guard).1 = false;
        (*guard).2 = false;
        match rng.next_u32() % 3 {
            0 => {
                (*guard).0 = true;
                resources.fuel.notify_all()
            }
            1 => {
                (*guard).1 = true;
```

```
                resources.oxidizer.notify_all()
            }
            2 => {
                (*guard).2 = true;
                resources.astronauts.notify_all()
            }
```

```
            _ => unreachable!(),
        }
    }
}
```

On each iteration, the producer chooses a new resource—`rng.next_u32() % 3`—and sets the Boolean flag for that resource before notifying all threads waiting on the `fuel` condvar. Meanwhile, the compiler and CPU are free to re-order instructions and the memory `notify_all` acts like a causality gate; everything before in the code is before in causality, and likewise, afterwards. If the resource bool flip was after the notification, the wake-up would be spurious from the point of view of the waiting threads and lost from the point of view of the producer. The `rocket` is straightforward:

```
   fn rocket(name: String, resources: Arc<Resources>,
             all_go: Arc<Barrier>, lift_off: Arc<Barrier>) {
       {
           let mut guard = resources.lock.lock().unwrap();
           while !(*guard).0 {
               guard = resources.fuel.wait(guard).unwrap();
           }
           (*guard).0 = false;
           println!("{:<6} ACQUIRE FUEL", name);
       }
       {
           let mut guard = resources.lock.lock().unwrap();
           while !(*guard).1 {
               guard = resources.oxidizer.wait(guard).unwrap();
           }
           (*guard).1 = false;
           println!("{:<6} ACQUIRE OXIDIZER", name);
       }
       {
           let mut guard = resources.lock.lock().unwrap();
```

```
            while !(*guard).2 {
                guard = resources.astronauts.wait(guard).unwrap();
            }
            (*guard).2 = false;
            println!("{:<6} ACQUIRE ASTRONAUTS", name);
        }

        all_go.wait();
        lift_off.wait();
        println!("{:<6} LIFT OFF", name);
    }
```

Each thread, regarding the resource requirements, waits on the condvar for the producer to make it available. A race occurs to re-acquire the mutex, as discussed, and only a single thread gets the resource. Finally, once all the resources are acquired, the `all_go` barrier is hit to delay any threads ahead of the count-down. Here we need the `main` function:

```
fn main() {
    let all_go = Arc::new(Barrier::new(4));
    let lift_off = Arc::new(Barrier::new(4));
    let resources = Arc::new(Resources {
        lock: Mutex::new((false, false, false)),
        fuel: Condvar::new(),
        oxidizer: Condvar::new(),
        astronauts: Condvar::new(),
    });

    let mut rockets = Vec::new();
    for name in &["KSC", "VAB", "WSMR"] {
        let all_go = Arc::clone(&all_go);
        let lift_off = Arc::clone(&lift_off);
        let resources = Arc::clone(&resources);
        rockets.push(thread::spawn(move || {
            rocket(name.to_string(), resources,
                   all_go, lift_off)
        }));
    }

    thread::spawn(move || {
        producer(resources);
    });
```

```
        all_go.wait();
        let one_second = time::Duration::from_millis(1_000);
        println!("T-11");
        for i in 0..10 {
            println!("{:>4}", 10 - i);
            thread::sleep(one_second);
        }
        lift_off.wait();

        for jh in rockets {
            jh.join().unwrap();
        }
    }
```

Note that, roughly, the first half of the function is made up of the barriers and resources or rocket threads. `all_go.wait()` is where it gets interesting. This main thread has spawned all the children and is now blocked on the all-go signal from the rocket threads, meaning they've collected their resources and are also blocked on the same barrier. That done, the count-down happens, to add a little panache to the solution; meanwhile, the rocket threads have started to wait on the `lift_off` barrier. It is, incidentally, worth noting that the producer is still producing, drawing CPU and power. Once the count-down is complete, the rocket threads are released, the main thread joins on them to allow them to print their goodbyes, and the program is finished. Outputs will vary, but here's one representative example:

```
KSC    ACQUIRE FUEL
WSMR   ACQUIRE FUEL
WSMR   ACQUIRE OXIDIZER
VAB    ACQUIRE FUEL
WSMR   ACQUIRE ASTRONAUTS
KSC    ACQUIRE OXIDIZER
KSC    ACQUIRE ASTRONAUTS
VAB    ACQUIRE OXIDIZER
VAB    ACQUIRE ASTRONAUTS
T-11
  10
   9
   8
   7
   6
```

```
   5
   4
   3
   2
   1
VAB    LIFT OFF
WSMR   LIFT OFF
KSC    LIFT OFF
```

The rope bridge problem

This puzzle appears in *The Little Book of Semaphores* as the *Baboon crossing problem*, section 6.3. Downey notes that it is adapted from Tanenbaum's *Operating Systems: Design and Implementation*, so you're getting it third-hand here. The problem description is thus:

"There is a deep canyon somewhere in Kruger National Park, South Africa, and a single rope that spans the canyon. Baboons can cross the canyon by swinging hand-over-hand on the rope, but if two baboons going in opposite directions meet in the middle, they will fight and drop to their deaths. Furthermore, the rope is only strong enough to hold 5 baboons. If there are more baboons on the rope at the same time, it will break.Assuming that we can teach the baboons to use semaphores, we would like to design a synchronization scheme with the following properties:

- *Once a baboon has begun to cross, it is guaranteed to get to the other side without running into a baboon going the other way.*
- *There are never more than 5 baboons on the rope."*

Our solution does not use semaphores. Instead, we lean on the type system to provide guarantees, leaning on it to ensure that no left-traveler will ever meet a right-traveler. Let's dig in:

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

#[derive(Debug)]
enum Bridge {
    Empty,
    Left(u8),
    Right(u8),
}
```

We see here the usual preamble, and then a type, `Bridge`. `Bridge` is the model of the `rope` bridge of the problem statement and is either empty,

has left-traveling baboons on it, or has right-traveling baboons on it; there's no reason to fiddle with flags and infer state when we can just encode it into a type. In fact, leaning on the type system, our synchronization is very simple:

```
fn main() {
    let rope = Arc::new(Mutex::new(Bridge::Empty));
```

Just a single mutex. We represent either side of the bridge as a thread, each side trying to get its own baboons across but co-operatively allowing baboons to reach its side. Here's the left side of the bridge:

```
    let lhs_rope = Arc::clone(&rope);
    let lhs = thread::spawn(move || {
        let rope = lhs_rope;
        loop {
            let mut guard = rope.lock().unwrap();
            match *guard {
                Bridge::Empty => {
                    *guard = Bridge::Right(1);
                }
                Bridge::Right(i) => {
                    if i < 5 {
                        *guard = Bridge::Right(i + 1);
                    }
                }
                Bridge::Left(0) => {
                    *guard = Bridge::Empty;
                }
                Bridge::Left(i) => {
                    *guard = Bridge::Left(i - 1);
                }
            }
        }
    });
```

When the left side of the bridge finds the bridge itself empty, one right-traveling baboon is sent on its way. Further, when the left side finds that there are already right-traveling baboons on the bridge and the rope's capacity of five has not been reached, another baboon is

sent on its way. Left-traveling baboons are received from the rope, decrementing the total baboons on the rope. The special case here is the clause for `Bridge::Left(0)`. While there are no baboons on the bridge still, technically, if the right side of the bridge were to be scheduled before the left side, it would send a baboon on its way, as we'll see shortly. We could make the removal of a baboon more aggressive and set the bridge to `Bridge::Empty` as soon as there are no travelers, of course. Here's the right side of the bridge, which is similar to the left:

```
    let rhs = thread::spawn(move || loop {
        let mut guard = rope.lock().unwrap();
        match *guard {
            Bridge::Empty => {
                *guard = Bridge::Left(1);
            }
            Bridge::Left(i) => {
                if i < 5 {
                    *guard = Bridge::Left(i + 1);
                }
            }
            Bridge::Right(0) => {
                *guard = Bridge::Empty;
            }
            Bridge::Right(i) => {
                *guard = Bridge::Right(i - 1);
            }
        }
    });

    rhs.join().unwrap();
    lhs.join().unwrap();
}
```

Now, is this fair? It depends on your system mutex. Some systems provide mutexes that are fair in that, if one thread has acquired the lock repeatedly and starved other threads attempting to lock the same primitive, the greedy thread will be de-prioritized under the others. Such fairness may or may not incur additional overheads when locking, depending on the implementation. Whether you want fairness, in fact, depends strongly on the problem you're trying to

solve. If we were only interested in shuffling baboons across the rope bridge as quickly as possible—maximizing throughput—then it doesn't truly matter which direction they're coming from. The original problem statement makes no mention of fairness, which it kind of shuffles around by allowing the stream of baboons to be infinite. Consider what would happen if the baboons were finite on either side of the bridge and we wanted to reduce the time it takes for any individual baboon to cross to the other side, to minimize latency. Our present solution, adapted to cope with finite streams, is pretty poor, then, in that regard. The situation could be improved by occasional yielding, layering in more aggressive rope packing, intentional back off, or a host of other strategies. Some platforms allow you to dynamically shift the priority of threads with regard to locks, but Rust does not offer that in the standard library.

Or, you know, we could use two bounded MPSC channels. That's an option. It all depends on what you're trying to get done.

Ultimately, the tools in safe Rust are very useful for performing computations on existing data structures without having to dip into any funny business. If that's your game, you're very unlikely to need to head much past `Mutex` and `Condvar`, and possibly into `RwLock` and `Barrier`. But, if you're building structures that are made of pointers, you'll have to dip into some of the funny business we saw in Ring, with all the dangers that brings.

Hopper—an MPSC specialization

As mentioned at the tail end of the last chapter, you'd need a fairly specialized use-case to consider not using stdlib's MPSC. In the rest of this chapter, we'll discuss such a use-case and the implementation of a library meant to fill it.

The problem

Recall back to the last chapter, where the role-threads in telem communicated with one another over MPSC channels. Recall also that telem was a quick version of the cernan (`https://crates.io/crates/cernan`) project, which fulfills basically the same role but over many more ingress protocols, egress protocols, and with the sharp edges worn down. One of the key design goals of cernan is that if it receives your data, it will deliver it downstream at least once. This implies that, for supporting ingress protocols, cernan must know, along the full length of the configured routing topology, that there is sufficient space to accept a new event, whether it's a piece of telemetry, a raw byte buffer, or a log line. Now, that's possible by using `SyncSender`, as we've seen. The trick comes in combination with a second design goal of cernan—very low, configurable resource consumption. Your author has seen cernan deployed to high-availability clusters with a single machine dedicated to running a cernan, as well as cernan running shotgun with an application server or as a part of a daemonset in a k8s cluster. In the first case, cernan can be configured to consume all of the machine's resources. In the later two, some thought has to be taken to giving cernan just enough resources to do its duties, relative to the expected input of the telemetered systems.

On modern systems, there's often an abundance of disk space and limited—relatively speaking—RAM. The use of `SyncSender` would require either a relatively low number of ingestible events or a high possible consumption of memory by cernan. These cases usually hit when an egress protocol is failing, possibly because the far-system is down. If cernan were to exacerbate a partial system failure by crowding out an application server because of a fault in a loosely-coupled system, well, that'd be pretty crummy.

For many users, it's also not acceptable to go totally blind during such a partial outage. Tricky constraints for `SyncSender`. Tricky enough, in fact, that we decided to write our own MPSC, called hopper (`https://crates.io/crates/hopper`). Hopper allows endusers to configure the total in-memory consumption of the queue, in bytes. That's not far off `SyncSender`, with a bit of calculation. What's special about hopper is that it can page to disk.

Hopper shines where memory constraints are very tight but you do not want to shed events. Or, you want to push shedding events off as far as possible. Similarly to the in-memory queue, hopper allows endusers to configure the maximum number of bytes to be consumed ondisk. A single hopper queue will hold `in_memory_bytes + on_disk_bytes` maximally before it's forced to shed events; we'll see the exact mechanism here directly. All of this is programmed with an eye toward maximal throughput and blocking threads that have no work, to save CPU time.

Hopper in use

Before we dig into the implementation of hopper, it'll be instructive to see it in practice. Let's adapt the last iteration of our ring program series. For reference, here's what that looked like:

```
use std::thread;
use std::sync::mpsc;

fn writer(chan: mpsc::SyncSender<u32>) -> () {
    let mut cur: u32 = 0;
    while let Ok(()) = chan.send(cur) {
        cur = cur.wrapping_add(1);
    }
}

fn reader(read_limit: usize, chan: mpsc::Receiver<u32>) -> () {
    let mut cur: u32 = 0;
    while (cur as usize) < read_limit {
        let num = chan.recv().unwrap();
        assert_eq!(num, cur);
        cur = cur.wrapping_add(1);
    }
}

fn main() {
    let capacity = 10;
    let read_limit = 1_000_000;
    let (snd, rcv) = mpsc::sync_channel(capacity);

    let reader_jh = thread::spawn(move || {
        reader(read_limit, rcv);
    });
    let _writer_jh = thread::spawn(move || {
        writer(snd);
    });

    reader_jh.join().unwrap();
}
```

The hopper version is a touch different. This is the preamble:

```
extern crate hopper;
extern crate tempdir;

use std::{mem, thread};
```

No surprise there. The function that will serve as the `writer` thread is:

```
fn writer(mut chan: hopper::Sender<u32>) -> () {
    let mut cur: u32 = 0;
    while let Ok(()) = chan.send(cur) {
        cur = cur.wrapping_add(1);
    }
}
```

Other than the switch from `mpsc::SyncSender<u32>` to `hopper::Sender<u32>`—all hopper senders are impliclty bounded—the only other difference is that `chan` must be mutable. Now, for the `reader` thread:

```
fn reader(read_limit: usize,
          mut chan: hopper::Receiver<u32>) -> () {
    let mut cur: u32 = 0;
    let mut iter = chan.iter();
    while (cur as usize) < read_limit {
        let num = iter.next().unwrap();
        assert_eq!(num, cur);
        cur = cur.wrapping_add(1);
    }
}
```

Here there's a little more to do. In hopper, the receiver is intended to be used as an iterator, which *is* a little awkward here as we want to limit the total number of received values. The iterator will block on calls to `next`, never returning a `None`. However, the `send` of the sender in hopper is very different to that of MPSC's—`hopper::Sender<T>::send(&mut self, event: T) -> Result<(), (T, Error)>`. Ignore `Error` for a second; why return a tuple in the error condition

that contains a `T`? To be clear, it's the same `T` that is passed in. When the caller sends a `T` into hopper, its ownership is passed into hopper as well, which is a problem in the case of an error. The caller might well want that `T` back to avoid its loss. Hopper wants to avoid doing a clone of every `T` that comes through and, so, hopper smuggles the ownership of the `T` back to the caller. What of `Error`? It's a simple enumeration:

```
   pub enum Error {
       NoSuchDirectory,
       IoError(Error),
       NoFlush,
       Full,
   }
```

The important one is `Error::Full`. This is the condition that occurs when both the in-memory and disk-backed buffers are full at the time of sending. This error is recoverable, but in a way that only the caller can determine. Now, finally, the `main` function of our hopper example:

```
fn main() {
    let read_limit = 1_000_000;
    let in_memory_capacity = mem::size_of::<u32>() * 10;
    let on_disk_capacity = mem::size_of::<u32>() * 100_000;

    let dir = tempdir::TempDir::new("queue_root").unwrap();
    let (snd, rcv) = hopper::channel_with_explicit_capacity::<u32>(
        "example",
        dir.path(),
        in_memory_capacity,
        on_disk_capacity,
        1,
    ).unwrap();

    let reader_jh = thread::spawn(move || {
        reader(read_limit, rcv);
    });
    let _writer_jh = thread::spawn(move || {
```

```
        writer(snd);
    });
```

```
    reader_jh.join().unwrap();
}
```

The `read_limit` is in place as before, but the big difference is the creation of the channel. First, hopper has to have some place to put its disk storage. Here we're deferring to some temporary directory—`let dir = tempdir::TempDir::new("queue_root").unwrap();`—to avoid cleaning up after running the example. In real use, the disk location is chosen carefully. `hopper::channel_with_explicit_capacity` creates the same sender as `hopper::channel` except that all the configuration knobs are open to the caller. The first argument is the *name* of the channel. This value is important as it will be used to create a directory under `dir` for disk storage. It is important for the channel name to be unique. `in_memory_capacity` is in bytes, as well as `on_disk_capacity`, which is why we have the use of our old friend `mem::size_of`. Now, what's that last configuration option there, set to `1`? That's the maximum number of *queue files*.

Hopper's disk storage is broken into multiple *queue files*, each of `on_disk_capacity` size. Senders carefully coordinate their writes to avoid over-filling the queue files, and the receiver is responsible for destroying them once it's sure there are no more writes coming—we'll talk about the signalling mechanism later in this chapter. The use of queue files allows hopper to potentially reclaim disk space that it may not otherwise have been able to, were one large file to be used in a circular fashion. This does incur some complexity in the sender and receiver code, as we'll see, but is worth it to provide a less resource-intensive library.

A conceptual view of hopper

Before we dig into the implementation, let's walk through how the previous example works at a conceptual level. We have enough in-memory space for 10 u32s. If the writer thread is much faster than the reader thread—or gets more scheduled time—we could end up with a situation like this:

```
write: 99

in-memory-buffer: [90, 91, 92, 93, 94, 95, 96, 97, 98]
disk-buffer: [87, 88, 89]
```

That is, a write of `99` is entering the system when the in-memory buffer is totally full and there are three queued writes in the diskbuffer. While it is possible for the state of the world to shift between the time a write enters the system for queuing and between the time it is queued, let's assume that no receivers pull items between queuing. The result will then be a system that looks like this:

```
in-memory-buffer: [90, 91, 92, 93, 94, 95, 96, 97, 98]
disk-buffer: [87, 88, 89, 99]
```

The receivers, together, must read only three elements from the diskbuffer, then all the in-memory elements, and then a single element from the diskbuffer, to maintain the queue's ordering. This is further complicated considering that a write may be split by one or more reads from the receivers. We saw something analogous in the previous chapter with regard to guarding writes by doing loads and checks - the conditions that satisfy a check may change between the load, check, and operation. There's a further complication as well; the

unified disk buffer displayed previously does not actually exist. Instead, there are potentially many individual queue files.

Let's say that hopper has been configured to allow for 10 `u32` in-memory, as mentioned, and 10 on-disk but split across five possible queue files. Our revised after-write system is as follows:

```
in-memory-buffer: [90, 91, 92, 93, 94, 95, 96, 97, 98]
disk-buffer: [ [87, 88], [89, 99] ]
```

The senders are responsible for creating queue files and filling them to their max. The `Receiver` is responsible for reading from the queue file and deleting the said file when it's exhausted. The mechanism for determining that a queue file is exhausted is simple; when the `Sender` exits a queue file, it moves on to create a new queue file, and marks the old path as read-only in the filesystem. When the `Receiver` attempts to read bytes from disk and finds there are none, it checks the write status of the file. If the file is still read-write, more bytes will come eventually. If the file is read-only, the file is exhausted. There's a little trick to it yet, and further unexplained cooperation between `Sender` and `Receiver`, but that should be enough abstract detail to let us dig in effectively.

The deque

Roughly speaking, hopper is a concurrent deque with two cooperating finite state machines layered on top. We'll start in with the deque, defined in `src/deque.rs`.

> *The discussion of hopper that follows lists almost all of its source code. We'll call out the few instances where the reader will need to refer to the listing in the book's source repository.*

To be totally clear, a deque is a data structure that allows for queuing and dequeuing at either end of the queue. Rust's `stdlib` has `VecDeque<T>`, which is very useful. Hopper is unable to use it, however, as one of its design goals is to allow for parallel sending and receiving against the hopper queue and `VecDeque` is not thread-safe. Also, while there are concurrent deque implementations in the crate ecosystem, the hopper deque is closely tied to the finite state machines it supports and to hopper's internal ownership model. That is, you probably can't use hopper's deque in your own project without some changes. Anyhow, here's the preamble:

```
use std::sync::{Condvar, Mutex, MutexGuard};
use std::sync::atomic::{AtomicUsize, Ordering};
use std::{mem, sync};

unsafe impl<T, S> Send for Queue<T, S> {}
unsafe impl<T, S> Sync for Queue<T, S> {}
```

The only unfamiliar piece here are the things imported from `std::sync::atomic`. We'll be covering atomics in more detail in the next chapter, but we're going to breeze over them at a high-level as needed in the explanation to follow. Note as well the unsafe declarations of

`send` and `sync` for some as yet unknown type `Queue<T, S>`. We're about to go off-road:

```
pub struct Queue<T, S> {
    inner: sync::Arc<InnerQueue<T, S>>,
}
```

The `Queue<T, S>` definition is similar to what we saw in previous chapters: a simple structure wrapping an inner structure, here called `InnerQueue<T, S>`. The `InnerQueue` is wrapped in an `Arc`, meaning there's only one allocated `InnerQueue` on the heap. As you might expect, the clone of `Queue` is a copy of the `Arc` into a new struct:

```
impl<T, S> Clone for Queue<T, S> {
    fn clone(&self) -> Queue<T, S> {
        Queue {
            inner: sync::Arc::clone(&self.inner),
        }
    }
}
```

It's important that every thread that interacts with `Queue` sees the same `InnerQueue`. Otherwise, the threads are dealing with distinct areas of memory and have no relationship with one another. Let's look at `InnerQueue`:

```
struct InnerQueue<T, S> {
    capacity: usize,
    data: *mut Option<T>,
    size: AtomicUsize,
    back_lock: Mutex<BackGuardInner<S>>,
    front_lock: Mutex<FrontGuardInner>,
    not_empty: Condvar,
}
```

Okay, this is much more akin to the internals we saw in Rust itself, and there's a lot going on. The field `capacity` is the maximum number of elements that the `InnerQueue` will hold in data. Like the first `Ring` in

the previous chapter, `InnerQueue` uses a contiguous block of memory to store its `T` elements, exploding a `Vec` to get that contiguous block. Also, like the first `Ring`, we store `Option<T>` elements in the contiguous block of memory. Technically, we could deal with a contiguous block of raw pointers or copy memory blocks in and out. But the use of `Option<T>` simplifies the code, both for insertion and removal, at the cost of a single byte of memory per element. The added complication just isn't worth it for the performance goals hopper is trying to hit.

The `size` field is an atomic `usize`. Atomics will be covered in more detail in the next chapter, but the behavior of `size` is going to be important. For now, think of it as a very small piece of synchronization between threads; a little hook that will allow us to order memory accesses that happens also to act like a `usize`. The condvar `not_empty` is used to signal to any potential readers waiting for new elements to pop that there are, in fact, elements to pop. The use of condvar greatly reduces the CPU load of hopper without sacrificing latency to busy loops with sleeps. Now, `back_lock` and `front_lock`. What's going on here? Either side of the deque is protected by a mutex, meaning there can be only one enqueuer and one dequeuer at a time, but these can be running in parallel to each other. Here are the definitions of the two inner values of the mutexes:

```
#[derive(Debug, Clone, Copy)]
pub struct FrontGuardInner {
    offset: isize,
}

#[derive(Debug)]
pub struct BackGuardInner<S> {
    offset: isize,
    pub inner: S,
}
```

`FrontGuardInner` is the easier to explain of the two. The only field is `offset`, which defines the offset from the first pointer of `InnerGuard`'s data of the thread manipulating the front of the queue. This

contiguous store is also used in a ring buffer fashion. In `BackGuardInner`, we see the same offset, but an additional `inner`, `S`. What is this? As we'll see, the threads manipulating the back of the buffer need extra coordination between them. Exactly what that is, the queue does not care. Therefore, we make it a type parameter and allow the caller to sort everything out, being careful to pass the data around as needed. In this fashion, the queue smuggles state through itself but does not have to inspect it.

Let's start on the implementation of `InnerQueue`:

```
impl<T, S> InnerQueue<T, S>
where
    S: ::std::default::Default,
{
    pub fn with_capacity(capacity: usize) -> InnerQueue<T, S> {
        assert!(capacity > 0);
        let mut data: Vec<Option<T>> = Vec::with_capacity(capacity);
        for _ in 0..capacity {
            data.push(None);
        }
        let raw_data = (&mut data).as_mut_ptr();
        mem::forget(data);
        InnerQueue {
            capacity: capacity,
            data: raw_data,
            size: AtomicUsize::new(0),
            back_lock: Mutex::new(BackGuardInner {
                offset: 0,
                inner: S::default(),
            }),
            front_lock: Mutex::new(FrontGuardInner { offset: 0 }),
            not_empty: Condvar::new(),
        }
    }
```

The type lacks a `new() -> InnerQueue`, as there was no call for it to be made. Instead, there's only `with_capacity` and that's quite similar to what we saw of `Ring`'s `with_capacity`—a vector is allocated, and exploded into a raw pointer, and the original reference is forgotten before the pointer is loaded into a newly minted struct.

The type `S` has to implement a default for initialization that is sufficient, as the caller's smuggled state will always be the same value, which is more than adequately definable as a default. If this deque were intended for general use, we'd probably need to offer a `with_capacity` that also took an `S` directly. Now, a few further functions in the implementation that we'll just breeze right past:

```
pub fn capacity(&self) -> usize {
    self.capacity
}

pub fn size(&self) -> usize {
    self.size.load(Ordering::Relaxed)
}

pub fn lock_back(&self) -> MutexGuard<BackGuardInner<S>> {
    self.back_lock.lock().expect("back lock poisoned")
}

pub fn lock_front(&self) -> MutexGuard<FrontGuardInner> {
    self.front_lock.lock().expect("front lock poisoned")
}
```

The next function, `push_back`, is very important and subtle:

```
pub unsafe fn push_back(
    &self,
    elem: T,
    guard: &mut MutexGuard<BackGuardInner<S>>,
) -> Result<bool, Error<T>> {
    let mut must_wake_dequeuers = false;
    if self.size.load(Ordering::Acquire) == self.capacity {
        return Err(Error::Full(elem));
    } else {
        assert!((*self.data.offset((*guard).offset)).is_none());
        *self.data.offset((*guard).offset) = Some(elem);
        (*guard).offset += 1;
        (*guard).offset %= self.capacity as isize;
        if self.size.fetch_add(1, Ordering::Release) == 0 {
            must_wake_dequeuers = true;
        }
    }
```

```
        Ok(must_wake_dequeuers)
    }
```

`InnerQueue::push_back` is responsible for placing a `T` onto the current back of the ring buffer, or failing capacity to signal that the buffer is full. When we discussed Ring, we noted that the `size == capacity` check was a race. Not so in `InnerQueue`, thanks to the atomic nature of the size. `self.size.load(Ordering::Acquire)` performs a memory load of the `self.size` but does so with certainty that it's the only thread with `self.size` as a manipulable value. Subsequent memory operations in the thread will be ordered after `Acquire`, at least until a store of `Ordering::Release` happens. A store of that nature does happen just a handful of lines down—`self.size.fetch_add(1, Ordering::Release)`. Between these two points, we see the element `T` loaded into the buffer—with a prior check to ensure that we're not stomping a `Some` value—and a wrapping bump of the `BackGuardInner`'s offset. Just like in the last chapter. Where this implementation differs is the return of `Ok(must_wake_dequeuers)`. Because the inner `S` is being guarded, it's not possible for the queue to know if there will be any further work that needs to be done before the mutex can be given up. As a result, the queue cannot itself signal that there's a value to read, even though it's already been written to memory by the time the function returns. The caller has to run the notification. That's a sharp edge. If the caller forgets to notify a thread blocked on the condvar, the blocked thread will stay that way forever.

The `InnerQueue::push_front` is a little simpler and not a radical departure from `push_back`:

```
pub unsafe fn pop_front(&self) -> T {
    let mut guard = self.front_lock.lock()
                        .expect("front lock poisoned");
    while self.size.load(Ordering::Acquire) == 0 {
        guard = self.not_empty
            .wait(guard)
            .expect("oops could not wait pop_front");
    }
```

```
        let elem: Option<T> = mem::replace(&mut
        *self.data.offset((*guard).offset), None);
        assert!(elem.is_some());
        *self.data.offset((*guard).offset) = None;
        (*guard).offset += 1;
        (*guard).offset %= self.capacity as isize;
        self.size.fetch_sub(1, Ordering::Release);
        elem.unwrap()
    }
}
```

The thread popping front, because it does not have to coordinate, is able to take the front lock itself, as there's no state that needs to be threaded through. After receiving the lock, the thread then enters a condition check loop to guard against spurious wake-ups on `not_empty`, replacing the item at offset with `None` when the thread is able to wake up. The usual offset maintenance occurs.

The implementation of `Queue<T, S>` is pretty minimal in comparison to the inner structure. Here's `push_back`:

```
   pub fn push_back(
       &self,
       elem: T,
       mut guard: &mut MutexGuard<BackGuardInner<S>>,
   ) -> Result<bool, Error<T>> {
       unsafe { (*self.inner).push_back(elem, &mut guard) }
   }
```

The only function that's substantially new to `Queue` is `notify_not_empty(&self, _guard: &MutexGuard<FrontGuardInner>) -> ()`. The caller is responsible for calling this whenever `push_back` signals that the dequeuer must be notified and, while the guard is not used, one rough edge of the library is smoothed down by requiring that it be passed in, proving that it's held.

That's the deque at the core of hopper. This structure was very difficult to get right. Any slight re-ordering of the load and store on the atomic size with respect to other memory operations will

introduce bugs, such as parallel access to the same memory of a region without coordination. These bugs are *very* subtle and don't manifest immediately. Liberal use of helgrind plus quickcheck testing across x86 and ARM processors—more on that later—will help drive up confidence in the implementation. Test runs of hours were not uncommon to find bugs that were not deterministic but could be reasoned about, given enough time and repeat examples. Building concurrent data structures out of very primitive pieces is *hard*.

The Receiver

Now that we've covered the deque, let's jump on to the `Receiver`, defined in `src/receiver.rs`. As mentioned previously, the receiver is responsible for either pulling a value out of memory or from disk in the style of an iterator. The `Receiver` declares the usual machinery for transforming itself into an iterator, and we won't cover that in this book, mostly just because it's a touch on the tedious side. That said, there are two important functions to cover in the `Receiver`, and neither of them are in the public interface. The first is `Receiver::next_value`, called by the iterator version of the receiver. This function is defined as follows:

```
fn next_value(&mut self) -> Option<T> {
    loop {
        if self.disk_writes_to_read == 0 {
            match self.mem_buffer.pop_front() {
                private::Placement::Memory(ev) => {
                    return Some(ev);
                }
                private::Placement::Disk(sz) => {
                    self.disk_writes_to_read = sz;
                    continue;
                }
            }
        } else {
            match self.read_disk_value() {
                Ok(ev) => return Some(ev),
                Err(_) => return None,
            }
        }
    }
}
```

A hopper defined to move `T` values does not define a deque—as discussed in the previous section—over `T`s. Instead, the deque

actually holds `Placement<T>`. Placement, defined in `src/private.rs`, is a small enumeration:

```
pub enum Placement<T> {
    Memory(T),
    Disk(usize),
}
```

This is the trick that makes hopper work. The primary challenge with a concurrent data structure is providing sufficient synchronization between threads that your results can remain coherent despite the chaotic nature of scheduling, but not require so much synchronization that you're underwater compared to a sequential solution.

That does happen.

Now, recall that maintaining order is a key design need for any channel-style queue. The Rust standard library MPSC achieves this with atomic flags, aided by the fact that, ultimately, there's only one place for inserted elements to be stored and one place for them to be removed from. Not so in hopper. But, that one-stop-shop is a very useful, low synchronization approach. That's where `Placement` comes in. When the `Memory` variant is hit, the `T` is present already and the receiver simply returns it. When `Disk(usize)` is returned, that sends a signal to the receiver to flip itself into *disk mode*. In disk mode, when `self.disk_writes_to_read` is not zero, the receiver preferentially reads a value from disk. Only when there are no more disk values to be read does the receiver attempt to read from memory again. This mode-flipping approach maintains ordering but also has the added benefit of requiring no synchronization when in disk mode, saving critical time when reading from a slow disk.

The second important function to examine is `read_disk_value`, referenced in `next_value`. It's long and mostly book-keeping, but I did want to call out the first part of that function here:

```
fn read_disk_value(&mut self) -> Result<T, super::Error> {
    loop {
        match self.fp.read_u32::<BigEndian>() {
            Ok(payload_size_in_bytes) => {
                let mut payload_buf = vec![0; payload_size_in_bytes
                 as usize];
                match self.fp.read_exact(&mut payload_buf[..]) {
                    Ok(()) => {
                        let mut dec =
                        DeflateDecoder::new(&payload_buf[..]);
                        match deserialize_from(&mut dec) {
                            Ok(event) => {
                                self.disk_writes_to_read -= 1;
                                return Ok(event);
                            }
                            Err(e) => panic!("Failed decoding.
                            Skipping {:?}", e),
                        }
                    }
```

This small chunk of code uses two very useful libraries. Hopper stores its disk-slop elements to disk in bincode format. Bincode (`https://crates.io/crates/bincode`) was invented for the servo project and is a serialization library intended for IPC - more or less the exact use hopper has for it. The advantage of bincode is that it's fast to serialize and deserialize but with the disadvantage of not being a standard and not having a guaranteed binary format from version to version. The second library to be called out is almost invisible in this example; byteorder. You can see it here: `self.fp.read_u32::<BigEndian>`. Byteorder extends `std::io::Read` to allow for the deserialization of primitive types from byte-buffers. It is possible to do this yourself by hand but it's error-prone and tedious to repeat. Use byteorder. So, what we're seeing here is hopper reading a 32-bit length big-ending length prefix from `self.fp`—a `std::io::BufReader` pointed to the current on-disk queue file—and using the said prefix to read exactly that many bytes from disk, before passing those on into the deserializer. That's it. All hopper on-disk slop elements are a 32 bit length prefix chased by that many bytes.

The Sender

Now that we've covered the `Receiver`, all that remains is the `Sender`. It's defined in `src/sender.rs`. The most important function in the `Sender` is `send`. The `Sender` follows the same disk/memory mode idea that `Receiver` uses but is more complicated in its operation. Let's dig in:

```
        let mut back_guard = self.mem_buffer.lock_back();
        if (*back_guard).inner.total_disk_writes == 0 {
            // in-memory mode
            let placed_event = private::Placement::Memory(event);
            match self.mem_buffer.push_back(placed_event, &mut
            back_guard) {
                Ok(must_wake_receiver) => {
                    if must_wake_receiver {
                        let front_guard = self.mem_buffer.lock_front();
                        self.mem_buffer.notify_not_empty(&front_guard);
                        drop(front_guard);
                    }
                }
                Err(deque::Error::Full(placed_event)) => {
                    self.write_to_disk(placed_event.extract().unwrap(),
                    &mut back_guard)?;
                    (*back_guard).inner.total_disk_writes += 1;
                }
            }
```

Recall that the deque allows the holder of the back guard to smuggle state for coordination through it. We're seeing that pay off here. The `Sender`'s internal state is called `SenderSync` and is defined as follows:

```
pub struct SenderSync {
    pub sender_fp: Option<BufWriter<fs::File>>,
    pub bytes_written: usize,
    pub sender_seq_num: usize,
    pub total_disk_writes: usize,
```

```
        pub path: PathBuf, // active fp filename
    }
```

Every sender thread has to be able to write to the current disk queue file, which `sender_fp` points to. Likewise, `bytes_written` tracks how many bytes have been, well, written to disk. The `Sender` must keep track of this value in order to correctly roll queue files over when they grow too large. `sender_seq_num` defines the name of the current writable queue file, being as they are named sequentially from zero on up. The key field for us is `total_disk_writes`. Notice that a memory write—`self.mem_buffer.push_back(placed_event, &mut back_guard)`—might fail with a `Full` error. In that case, `self.write_to_disk` is called to write the `T` to disk, increasing the total number of disk writes. This write mode was prefixed with a check into the cross-thread `SenderSync` to determine if there were outstanding disk writes. Remember, at this point, the `Receiver` has no way to determine that there has been an additional write go to disk; the sole communication channel with the `Receiver` is through the in-memory deque. To that end, the next `Sender` thread will flip into a different write mode:

```
        } else {
            // disk mode
            self.write_to_disk(event, &mut back_guard)?;
            (*back_guard).inner.total_disk_writes += 1;
            assert!((*back_guard).inner.sender_fp.is_some());
            if let Some(ref mut fp) = (*back_guard).inner.sender_fp {
                fp.flush().expect("unable to flush");
            } else {
                unreachable!()
            }
            if let Ok(must_wake_receiver) = self.mem_buffer.push_back(
                private::Placement::Disk(
                    (*back_guard).inner.total_disk_writes
                ),
                &mut back_guard,
            ) {
                (*back_guard).inner.total_disk_writes = 0;
                if must_wake_receiver {
                    let front_guard = self.mem_buffer.lock_front();
                    self.mem_buffer.notify_not_empty(&front_guard);
                    drop(front_guard);
```

```
                }
            }
        }
```

Once there's a single write to disk, the `Sender` flips into disk preferential write mode. At the start of this branch the `T` goes to disk, is flushed. Then, the `Sender` attempts to push a `Placement::Disk` onto the in-memory deque, which may fail if the `Receiver` is slow or unlucky in its scheduling assignment. Should it succeed, however, the `total_disk_writes` is set to zero—there are no longer any outstanding disk writes—and the `Receiver` is woken if need be to read its new events. The next time a `Sender` thread rolls through it may or may not have space in the in-memory deque to perform a memory placement but that's the next thread's concern.

That's the heart of `Sender`. While there is another large function in-module, `write_to_disk`, we won't list it here. The implementation is primarily book-keeping inside a mutex, a topic that has been covered in detail in this and the previous chapter, plus filesystem manipulation. That said, the curious reader is warmly encouraged to read through the code.

Testing concurrent data structures

Hopper is a subtle beast and proves to be quite tricky to get correct, owing to the difficulty of manipulating the same memory across multiple threads. Slight synchronization bugs were common in the writing of the implementation, bugs that highlighted fundamental mistakes but were rare to trigger and even harder to reproduce. Traditional unit testing is not sufficient here. There are no clean units to be found, being that the computer's non-deterministic behavior is fundamental to the end result of the program's run. With that in mind, there are three key testing strategies used on hopper: randomized testing over random inputs searching for logic bugs (QuickCheck), randomized testing over random inputs searching for crashes (fuzz testing), and multiple-million runs of the same program searching for consistency failures. In the rest of the chapter, we'll discuss each of these in turn.

QuickCheck and loops

Previous chapters discussed QuickCheck at length and we'll not duplicate that here. Instead, let's dig into a test from `hopper`, defined in `src/lib.rs`:

```
#[test]
fn multi_thread_concurrent_snd_and_rcv_round_trip() {
    fn inner(
        total_senders: usize,
        in_memory_bytes: usize,
        disk_bytes: usize,
        max_disk_files: usize,
        vals: Vec<u64>,
    ) -> TestResult {
        let sz = mem::size_of::<u64>();
        if total_senders == 0 ||
           total_senders > 10 ||
           vals.len() == 0 ||
           (vals.len() < total_senders) ||
           (in_memory_bytes / sz) == 0 ||
           (disk_bytes / sz) == 0
        {
            return TestResult::discard();
        }
        TestResult::from_bool(
          multi_thread_concurrent_snd_and_rcv_round_trip_exp(
            total_senders,
            in_memory_bytes,
            disk_bytes,
            max_disk_files,
            vals,
        ))
    }
    QuickCheck::new()
        .quickcheck(inner as fn(usize, usize, usize, usize,
         Vec<u64>) -> TestResult);
}
```

This test sets up a multiple sender, single receiver round trip environment, being careful to reject inputs that allow less space in the in-memory buffer than 64 bits, no senders, or the like. It defers to another function, `multi_thread_concurrent_snd_and_rcv_round_trip_exp`, to actually run the test.

This setup is awkward, admittedly, but has the benefit of allowing `multi_thread_concurrent_snd_and_rcv_round_trip_exp` to be run over explicit inputs. That is, when a bug is found you can easily re-play that test by creating a manual—or *explicit*, in `hopper` testing terms—test. The inner test function is complicated and we'll consider it in parts:

```
fn multi_thread_concurrent_snd_and_rcv_round_trip_exp(
    total_senders: usize,
    in_memory_bytes: usize,
    disk_bytes: usize,
    max_disk_files: usize,
    vals: Vec<u64>,
) -> bool {
    if let Ok(dir) = tempdir::TempDir::new("hopper") {
        if let Ok((snd, mut rcv)) =
          channel_with_explicit_capacity(
            "tst",
            dir.path(),
            in_memory_bytes,
            disk_bytes,
            max_disk_files,
        ) {
```

Much like our example usage of hopper at the start of this section, the inner test function uses tempdir (`https://crates.io/crates/tempdir`) to create a temporary path for passing into `channel_with_explicit_capacity`. Except, we're careful not to unwrap here. Because hopper makes use of the filesystem and because Rust QuickCheck is aggressively multi-threaded, it's possible that any individual test run will hit a temporary case of file-handler exhaustion. This throws QuickCheck off, with the test failure being totally unrelated to the inputs of this particular execution.

The next piece is as follows:

```
                let chunk_size = vals.len() / total_senders;

                let mut snd_jh = Vec::new();
                let snd_vals = vals.clone();
                for chunk in snd_vals.chunks(chunk_size) {
                    let mut thr_snd = snd.clone();
                    let chunk = chunk.to_vec();
                    snd_jh.push(thread::spawn(move || {
                        let mut queued = Vec::new();
                        for mut ev in chunk {
                            loop {
                                match thr_snd.send(ev) {
                                    Err(res) => {
                                        ev = res.0;
                                    }
                                    Ok(()) => {
                                        break;
                                    }
                                }
                            }
                            queued.push(ev);
                        }
                        let mut attempts = 0;
                        loop {
                            if thr_snd.flush().is_ok() {
                                break;
                            }
                            thread::sleep(
                              ::std::time::Duration::from_millis(100)
                            );
                            attempts += 1;
                            assert!(attempts < 10);
                        }
                        queued
                    }))
                }
```

Here we have the creation of the sender threads, each of which grabs an equal sized chunk of `vals`. Now, because of indeterminacy in thread scheduling, it's not possible for us to model the order in which elements of `vals` will be pushed into hopper. All we can do is confirm

that there are no lost elements after transmission through hopper. They may, in fact, be garbled in terms of order:

```
            let expected_total_vals = vals.len();
            let rcv_jh = thread::spawn(move || {
                let mut collected = Vec::new();
                let mut rcv_iter = rcv.iter();
                while collected.len() < expected_total_vals {
                    let mut attempts = 0;
                    loop {
                        match rcv_iter.next() {
                            None => {
                                attempts += 1;
                                assert!(attempts < 10_000);
                            }
                            Some(res) => {
                                collected.push(res);
                                break;
                            }
                        }
                    }
                }
                collected
            });

            let mut snd_vals: Vec<u64> = Vec::new();
            for jh in snd_jh {
                snd_vals.append(&mut jh.join().expect("snd join
                failed"));
            }
            let mut rcv_vals = rcv_jh.join().expect("rcv join
            failed");

            rcv_vals.sort();
            snd_vals.sort();

            assert_eq!(rcv_vals, snd_vals);
```

Another test with a single sender, `single_sender_single_rcv_round_trip`, is able to check for correct ordering as well as no data loss:

```
#[test]
fn single_sender_single_rcv_round_trip() {
```

```
    // Similar to the multi sender test except now with a single
    // sender we can guarantee order.
    fn inner(
        in_memory_bytes: usize,
        disk_bytes: usize,
        max_disk_files: usize,
        total_vals: usize,
    ) -> TestResult {
        let sz = mem::size_of::<u64>();
        if total_vals == 0 || (in_memory_bytes / sz) == 0 ||
        (disk_bytes / sz) == 0 {
            return TestResult::discard();
        }
        TestResult::from_bool(
          single_sender_single_rcv_round_trip_exp(
            in_memory_bytes,
            disk_bytes,
            max_disk_files,
            total_vals,
        ))
    }
    QuickCheck::new().quickcheck(inner as fn(usize, usize,
                                 usize, usize) -> TestResult);
}
```

Like it's multi-cousin, this QuickCheck test uses an inner function to perform the actual test:

```
fn single_sender_single_rcv_round_trip_exp(
    in_memory_bytes: usize,
    disk_bytes: usize,
    max_disk_files: usize,
    total_vals: usize,
) -> bool {
    if let Ok(dir) = tempdir::TempDir::new("hopper") {
        if let Ok((mut snd, mut rcv)) =
        channel_with_explicit_capacity(
            "tst",
            dir.path(),
            in_memory_bytes,
            disk_bytes,
            max_disk_files,
        ) {
            let builder = thread::Builder::new();
            if let Ok(snd_jh) = builder.spawn(move || {
```

```
                    for i in 0..total_vals {
                        loop {
                            if snd.send(i).is_ok() {
                                break;
                            }
                        }
                    }
                    let mut attempts = 0;
                    loop {
                        if snd.flush().is_ok() {
                            break;
                        }
                        thread::sleep(
                          ::std::time::Duration::from_millis(100)
                        );
                        attempts += 1;
                        assert!(attempts < 10);
                    }
                }) {
                    let builder = thread::Builder::new();
                    if let Ok(rcv_jh) = builder.spawn(move || {
                        let mut rcv_iter = rcv.iter();
                        let mut cur = 0;
                        while cur != total_vals {
                            let mut attempts = 0;
                            loop {
                                if let Some(rcvd) = rcv_iter.next() {
                                    debug_assert_eq!(
                                        cur, rcvd,
                                    "FAILED TO GET ALL IN ORDER: {:?}",
                                        rcvd,
                                    );
                                    cur += 1;
                                    break;
                                } else {
                                    attempts += 1;
                                    assert!(attempts < 10_000);
                                }
                            }
                        }
                    }) {
                        snd_jh.join().expect("snd join failed");
                        rcv_jh.join().expect("rcv join failed");
                    }
                }
            }
        }
```

```
        true
    }
```

This, too, should appear familiar, except that the Receiver thread is now able to check order. Where previously a `Vec<u64>` was fed into the function, we now stream `0, 1, 2, .. total_vals` through the hopper queue, asserting that order and there are no gaps on the other side. Single runs on a single input will fail to trigger low-probability race issues reliably, but that's not the goal. We're searching for logic goofs. For instance, an earlier version of this library would happily allow an in-memory maximum amount of bytes less than the total bytes of an element `T`. Another could fit multiple instances of `T` into the buffer but if the `total_vals` were odd *and* the in-memory size were small enough to require disk-paging then the last element of the stream would never be kicked out. In fact, that's still an issue. It's a consequence of the lazy flip to disk mode in the sender; without another element to potentially trigger a disk placement to the in-memory buffer, the write will be flushed to disk but the receiver will never be aware of it. To that end, the sender does expose a `flush` function, which you see in use in the tests. In practice, in cernan, flushing is unnecessary. But, it's a corner of the design that the authors did not expect and may well have had a hard time noticing had this gone out into the wild.

The inner test of our single-sender variant is also used for the repeat-loop variant of hopper testing:

```
#[test]
fn explicit_single_sender_single_rcv_round_trip() {
    let mut loops = 0;
    loop {
     assert!(single_sender_single_rcv_round_trip_exp(8, 8, 5, 10));
        loops += 1;
        if loops > 2_500 {
            break;
        }
        thread::sleep(::std::time::Duration::from_millis(1));
    }
}
```

Notice that here the inner loop is only for 2,500 iterations. This is done in deference to the needs of the CI servers which, don't care to run high CPU load code for hours at a time. In development, that 2,500 will be adjusted up. But the core idea is apparent; check that a stream of ordered inputs returns through the hopper queue in order and intact over and over and over again. QuickCheck searches the dark corners of the state space and more traditional manual testing hammers the same spot to dig in to computer indeterminism.

Searching for crashes with AFL

The repeat-loop variant of testing in the previous section is an interesting one. It's not, like QuickCheck, a test wholly focused on finding logic bugs. We know that the test will work for most iterations. What's being sought out there is a failure to properly control the indeterminism of the underlying machine. In some sense that variant of test is using a logical check to search for crashes. It is a kind of fuzz test.

Hopper interacts with the filesystem, spawns threads, and manipulates bytes up from files. Each of these activities is subject to failure for want of resources, offsetting errors, or fundamental misunderstandings of the medium. An early version of hopper, for instance, assumed that small atomic writes to the disk would not interleave, which is true for XFS but not true for most other filesystems. Or, another version of hopper always created threads in-test by use of the more familiar `thread::spawn`. It turns out, however, that this function will panic if no thread can be created, which is why the tests use the `thread::Builder` pattern instead, allowing for recovery.

Crashes are important to figure out in their own right. To this end, hopper has a fuzzing setup, based on AFL. To avoid producing an unnecessary binary on users' systems, the hopper project does not host its own fuzzing code as of this writing. Instead, it lives in `blt/hopper-fuzz` (`https://github.com/blt/hopper-fuzz`). This is, admittedly, awkward, but fuzz testing, being uncommon, often is. Fuzz rounds easily run for multiples of days and do not fit well into modern CI systems. AFL itself is not a tool that admits easy automation, either, compounding the problem. Fuzz runs tend to be done in small

batches on users' private machines, at least for open-source projects with small communities. Inside hopper-fuzz, there's a single program for fuzzing, the preamble of which is:

```
extern crate hopper;
extern crate rand;
#[macro_use]
extern crate serde_derive;
extern crate bincode;
extern crate tempdir;

use bincode::{deserialize_from, Bounded};
use self::hopper::channel_with_explicit_capacity;
use std::{io, process};
use rand::{Rng, SeedableRng, XorShiftRng};
```

The fairly straightforward, based on what we've seen so far. Getting input into a fuzz program is sometimes non-trivial, especially when the input is not much more than a set of inputs rather than, say in the case of a parser, an actual unit of work for the program. Your author's approach is to rely on serde to deserialize the byte buffer that AFL will fling into the program, being aware that most payloads will fail to decode but not minding all that much. To that end, the top of your author's fuzz programs usually have an input struct filled with control data, and this program is no different:

```
#[derive(Debug, PartialEq, Deserialize)]
struct Input { // 22 bytes
    seed_a: u32, // 4
    seed_b: u32, // 4
    seed_c: u32, // 4
    seed_d: u32, // 4
    max_in_memory_bytes: u32, // 4
    max_disk_bytes: u16, // 2
}
```

The seeds are for `XorShiftRng`, whereas `max_in_memory_bytes` and `max_disk_bytes` are for hopper. A careful tally of the bytesize of input is made to avoid fuzz testing the bincode deserializer's ability to reject

abnormally large inputs. While AFL is not blind—it has instrumented the branches of the program, after all—it is also not very smart. It's entirely possible that what you, the programmer, intends to fuzz is not what gets fuzzed to start. It's not unheard of to shake out bugs in any additional libraries brought into the fuzz project. It's to keep the libraries drawn in to a minimum and their use minimal.

The `main` function of the fuzz program starts off with a setup similar to other hopper tests:

```
fn main() {
    let stdin = io::stdin();
    let mut stdin = stdin.lock();
    if let Ok(input) = deserialize_from(&mut stdin, Bounded(22)) {
        let mut input: Input = input;
        let mut rng: XorShiftRng =
            SeedableRng::from_seed([input.seed_a, input.seed_b,
                                    input.seed_c, input.seed_d]);

        input.max_in_memory_bytes =
          if input.max_in_memory_bytes > 2 << 25 {
            2 << 25
        } else {
            input.max_in_memory_bytes
        };

        // We need to be absolutely sure we don't run into another
        // running process. Which, this isn't a guarantee but
        // it's _pretty_ unlikely to hit jackpot.
        let prefix = format!(
            "hopper-{}-{}-{}",
            rng.gen::<u64>(),
            rng.gen::<u64>(),
            rng.gen::<u64>()
        );
```

The only oddity is the production of prefix. The entropy in the `tempdir` name is relatively low, compared to the many, many millions of tests that AFL will run. We want to be especially sure that no two hopper runs are given the same data directory, as that is undefined behavior:

```
        if let Ok(dir) = tempdir::TempDir::new(&prefix) {
            if let Ok((mut snd, mut rcv)) =
            channel_with_explicit_capacity(
                "afl",
                dir.path(),
                input.max_in_memory_bytes as usize,
                input.max_disk_bytes as usize,
            ) {
```

The meat of the AFL test is surprising:

```
                let mut writes = 0;
                let mut rcv_iter = rcv.iter();
                loop {
                    match rng.gen_range(0, 102) {
                        0...50 => {
                            if writes != 0 {
                                let _ = rcv_iter.next();
                            }
                        }
                        50...100 => {
                            snd.send(rng.gen::<u64>());
                            writes += 1;
                        }
                        _ => {
                            process::exit(0);
                        }
                    }
                }
            }
        }
    }
}
```

This program pulled in a random number generator to switch off between performing a read and performing a write into the hopper channel. Why no threads? Well, it turns out, AFL struggles to accommodate threading in its model. AFL has a stability notion, which is its ability to re-run a program and achieve the same results in terms of instruction execution and the like. This is not going to fly with a multi-threaded use of hopper. Still, despite missing out on

probing the potential races between sender and receiver threads, this fuzz test found:

- File-descriptor exhaustion crashes
- Incorrect offset computations in deque
- Incorrect offset computations at the Sender/Receiver level
- Deserialization of elements into a non-cleared buffer, resulting in phantom elements
- Arithmetic overflow/underflow crashes
- A failure to allocate enough space for serialization.

Issues with the interior deque that happen only in a multi-threaded context will be missed, of course, but the preceding list is nothing to sneeze at.

Benchmarking

Okay, by now we can be reasonably certain that hopper is fit for purpose, at least in terms of not crashing and producing the correct results. But, it also needs to be *fast*. To that end, hopper ships with the criterion (`https://crates.io/crates/criterion`) benchmarks. As the time of writing, criterion is a rapidly evolving library that performs statistical analysis on bench run results that Rust's built-in, nightly-only benchmarking library does not. Also, criterion is available for use on the stable channel. The target to match is standard library's MPSC, and that sets the baseline for hopper. To that end, the benchmark suite performs a comparison, living in `benches/stdlib_comparison.rs` in the hopper repository.

The preamble is typical:

```
#[macro_use]
extern crate criterion;
extern crate hopper;
extern crate tempdir;

use std::{mem, thread};
use criterion::{Bencher, Criterion};
use hopper::channel_with_explicit_capacity;
use std::sync::mpsc::channel;
```

Note that we've pulled in both MPSC and hopper. The function for MPSC that we'll be benching is:

```
fn mpsc_tst(input: MpscInput) -> () {
    let (tx, rx) = channel();

    let chunk_size = input.ith / input.total_senders;
```

```
        let mut snd_jh = Vec::new();
        for _ in 0..input.total_senders {
            let tx = tx.clone();
            let builder = thread::Builder::new();
            if let Ok(handler) = builder.spawn(move || {
                for i in 0..chunk_size {
                    tx.send(i).unwrap();
                }
            }) {
                snd_jh.push(handler);
            }
        }

        let total_senders = snd_jh.len();
        let builder = thread::Builder::new();
        match builder.spawn(move || {
            let mut collected = 0;
            while collected < (chunk_size * total_senders) {
                let _ = rx.recv().unwrap();
                collected += 1;
            }
        }) {
            Ok(rcv_jh) => {
                for jh in snd_jh {
                    jh.join().expect("snd join failed");
                }
                rcv_jh.join().expect("rcv join failed");
            }
            _ => {
                return;
            }
        }
    }
```

Some sender threads get made, and a receiver exists and pulls the values from MPSC as rapidly as possible. This is not a logic check in any sense and the collected materials are immediately discarded. Like with the fuzz testing, the input to the function is structured data. `MpscInput` is defined as follows:

```
    #[derive(Debug, Clone, Copy)]
    struct MpscInput {
        total_senders: usize,
```

```
    ith: usize,
}
```

The hopper version of this function is a little longer, as there are more error states to cope with, but it's nothing we haven't seen before:

```
fn hopper_tst(input: HopperInput) -> () {
    let sz = mem::size_of::<u64>();
    let in_memory_bytes = sz * input.in_memory_total;
    let max_disk_bytes = sz * input.max_disk_total;
    if let Ok(dir) = tempdir::TempDir::new("hopper") {
        if let Ok((snd, mut rcv)) = channel_with_explicit_capacity(
            "tst",
            dir.path(),
            in_memory_bytes,
            max_disk_bytes,
            usize::max_value(),
        ) {
            let chunk_size = input.ith / input.total_senders;

            let mut snd_jh = Vec::new();
            for _ in 0..input.total_senders {
                let mut thr_snd = snd.clone();
                let builder = thread::Builder::new();
                if let Ok(handler) = builder.spawn(move || {
                    for i in 0..chunk_size {
                        let _ = thr_snd.send(i);
                    }
                }) {
                    snd_jh.push(handler);
                }
            }

            let total_senders = snd_jh.len();
            let builder = thread::Builder::new();
            match builder.spawn(move || {
                let mut collected = 0;
                let mut rcv_iter = rcv.iter();
                while collected < (chunk_size * total_senders) {
                    if rcv_iter.next().is_some() {
                        collected += 1;
                    }
                }
            }) {
                Ok(rcv_jh) => {
```

```
                    for jh in snd_jh {
                        jh.join().expect("snd join failed");
                    }
                    rcv_jh.join().expect("rcv join failed");
                }
                _ => {
                    return;
                }
            }
        }
    }
}
```

The same is true of `HopperInput`:

```
#[derive(Debug, Clone, Copy)]
struct HopperInput {
    in_memory_total: usize,
    max_disk_total: usize,
    total_senders: usize,
    ith: usize,
}
```

Criterion has many options for running benchmarks, but we've chosen here to run over inputs. Here's the setup for MPSC:

```
fn mpsc_benchmark(c: &mut Criterion) {
    c.bench_function_over_inputs(
        "mpsc_tst",
        |b: &mut Bencher, input: &MpscInput| b.iter(||
         mpsc_tst(*input)),
        vec![
            MpscInput {
                total_senders: 2 << 1,
                ith: 2 << 12,
            },
        ],
    );
}
```

To explain, we've got a function, `mpsc_benchmark`, that takes the mutable criterion structure, which is opaque to use but in which criterion will

store run data. This structure exposes `bench_function_over_inputs`, which consumes a closure that we can thread our `mpsc_test` through. The sole input is listed in a vector. The following is a setup that does the same thing, but for hopper:

```
fn hopper_benchmark(c: &mut Criterion) {
    c.bench_function_over_inputs(
        "hopper_tst",
        |b: &mut Bencher, input: &HopperInput| b.iter(||
         hopper_tst(*input)),
        vec![
            // all in-memory
            HopperInput {
                in_memory_total: 2 << 11,
                max_disk_total: 2 << 14,
                total_senders: 2 << 1,
                ith: 2 << 11,
            },
            // swap to disk
            HopperInput {
                in_memory_total: 2 << 11,
                max_disk_total: 2 << 14,
                total_senders: 2 << 1,
                ith: 2 << 12,
            },
        ],
    );
}
```

Notice now that we have two inputs, one guaranteed to be all in-memory and the other guaranteed to require disk paging. The disk paging input is sized appropriately to match the MSPC run. There'd be no harm in doing an in-memory comparison for both hopper and MPSC, but your author has a preference for pessimistic benchmarks, being an optimistic sort. The final bits needed by criterion are more or less stable across all the benchmarks we'll see in the rest of this book:

```
criterion_group!{
    name = benches;
    config = Criterion::default().without_plots();
```

```
    targets = hopper_benchmark, mpsc_benchmark
}
criterion_main!(benches);
```

I encourage you to run the benchmarks yourself. We see times for hopper that are approximately three times faster for the systems we intended hopper for. That's more than fast enough.

Summary

In this chapter, we covered the remainder of the essential non-atomic Rust synchronization primitives, doing a deep-dive on the postmates/hopper libraries to explore their use in a production code base. After having digested `Chapter 4`, *Sync and Send – the Foundation of Rust Concurrency*, and this chapter, the reader should be in a fine position to build lock-based, concurrent data structures in Rust. For readers that need even more performance, we'll explore the topic of atomic programming in `Chapter 6`, *Atomics – the Primitives of Synchronization*, and in `Chapter 7`, *Atomics – Safely Reclaiming Memory*.

If you thought lock-based concurrency was hard, wait until you see atomics.

Further reading

Building concurrent data structure is a broad field of wide concern. These notes cover much the same space as the notes from `Chapter 4`, *Sync and Send – the Foundation of Rust Concurrency*. Please do refer back to those notes.

- *The Little Book of Semaphores*, Allen Downey, available at `http://greenteapress.com/wp/semaphores/`.This is a charming book of concurrency puzzles, suitable for undergraduates but challenging enough in Rust for the absence of semaphores. We'll revisit this book in the next chapter when we build concurrency primitives out of atomics.
- *The Computability of Relaxed Data Structures: Queues and Stacks as Examples*, Nir Shavit and Gadi Taubenfeld. This chapter discussed the implementation of a concurrent queue based on the presentation of *The Art of Multiprocessor Programming* and the author's knowledge of Erlang's process queue. Queues are a common concurrent data structure and there are a great many possible approaches. This paper discusses an interesting notion. Namely, if we relax the ordering constraint of the queue, can we squeeze out more performance from a modern machine?
- *Is Parallel Programming Hard, and, If So, What Can You Do About It?*, Paul McKenney. This book covers roughly the

same material as `Chapter 4`, *Sync and Send – the Foundation of Rust Concurrency,* and `Chapter 5`, *Locks – Mutex, Condvar, Barriers, and RWLock,* but in significantly more detail. I highly encourage readers to look into getting a copy and reading it, especially the eleventh chapter on validating implementations.

Atomics – the Primitives of Synchronization

In `Chapter 4`, *Sync and Send – the Foundation of Rust Concurrency*, and `Chapter 5`, *Locks – Mutex, Condvar, Barriers and RWLock*, we discussed the fundamentals of lock-based concurrency in Rust. However, there are some cases where lock-based programming is not suitable, such as when extreme performance is a concern or threads may *never* block. In such domains, the programmer must rely on the atomic synchronization primitives of modern CPUs.

In this chapter, we'll be discussing the atomic primitives available to the Rust programmer. Programming with atomics is a complex topic and an area of active research. An entire book could be dedicated to the topic of atomic Rust programming. As such, we'll be shifting our tactics slightly for this chapter, focusing on more of a tutorial style than previous chapters where we've done deep dives on existing software. Everything presented in the prior chapters will come to bear here. By the end of this chapter, you ought to have a working understanding of atomics, being able to digest existing literature with more ease and validate your implementations.

By the end of this chapter, we will have:

- Discussed the concept of linearizability
- Discussed the various atomic memory orderings, along with their meanings and implications
- Built a mutex from atomic primitives

- Built a queue from atomic primitives
- Built a semaphore
- Made clear the difficulties of memory reclamation in an atomic context

Technical requirements

This chapter requires a working Rust installation. The details of verifying your installation are covered in `Chapter 1`, *Preliminaries – Machine Architecture and Getting Started with Rust*. No additional software tools are required.

You can find the source code for this book's projects on GitHub: `https://github.com/PacktPublishing/Hands-On-Concurrency-with-Rust`. This chapter has its source code under `Chapter06`.

Linearizability

Up till this point in the book, we've avoided dipping into the formal terms for specifying concurrent systems, because while they are *very* useful for discussing ideas and reasoning, they can be difficult to learn absent some context. Now that we have context, it's time.

How do we decide whether a concurrent algorithm is correct? How, if we're game for the algorithm, do we analyze an implementation and reason about it's correctness? To this point, we've used techniques to demonstrate fitness-for-purpose of an implementation through randomized and repeat testing as well as simulation, in the case of helgrind. We will continue to do so. In fact, if that's all we did, demonstrating fitness-for-purpose of implementations, then we'd be in pretty good condition. The working programmer will find themselves inventing more often than not, taking an algorithm and adapting it—as was seen in the previous chapter's discussion of hopper—to fit some novel domain. It's easy enough to do this and not notice.

What we're searching for, when we sit down to reason about the systems we're constructing, is a unifying concept that'll separate the workable ideas from the goofs. That concept for us, here, in the design and construction of concurrent data structures, is linearizability. What we're looking to build are objects that, paraphrasing Nancy Lynch, make it seem like operations on them occur one at a time, in some sequential order, consistent with the order of operations and responses. Lynch's *Distributed Algorithms* is concerned with the behavior of distributed systems, but I've always thought that her explanation of what she refers to as atomic objects is brilliantly concise.

The notion of linearizability was introduced in *Axioms for Concurrent Objects* by Herlihy and Wing in 1986. Their definition is a bit more involved, done up in terms of histories—*a finite sequence of operation invocation and response events*—and specifications, which are, to be quick about it, a set of axiomatic pre and post conditions that hold on, the object in terms of the operations defined on it.

Let the result of an operation be called `res(op)` and the application or invocation of an operation be called `inv(op)`, and distinguish operations by some extra identifier. `op_1` is distinct from `op_2` but the identifier has no bearing on their ordering; it's just a name. A history `H` is a partial order over operations so that `op_0 < op_1` if `res(op_0)` happens before `inv(op_1)` in `H`. Operations that have no ordering according to `<` are concurrent in the history. A history `H` is sequential if all of its operations are ordered with regard to `<`. Now, the history `H` is linearizable if it can be adjusted by adding zero or more operations into the history to make some other history `H'` so that:

1. `H'` is composed only of invocations and responses and are equivalent to some legal, sequential history `S`
2. The ordering operation of `H'` is an inclusive subset of the ordering of `S`.

Phew! An object is linearizable if you can record the order of the operations done to it, fiddle with them some, and find an equivalent sequential application of operations on the same object. We've actually done linearizabilty analysis in previous chapters, I just didn't call it out as such. Let's keep on with Herlihy and Wing and take a look at their examples of linearizable vs. unlinearizable histories. Here's a history on a familiar queue:

```
A: Enq(x)       0
B: Enq(y)       1
B: Ok(())       2
A: Ok(())       3
B: Deq()        4
B: Ok(x)        5
A: Deq()        6
A: Ok(y)        7
A: Enq(z)       8
```

We have two threads, `A` and `B`, performing the operations `enq(val: T) -> Result((), Error)` and `deq() -> Result(T, Error)`, in pseudo-Rust types. This history is in fact linearizable and is equivalent to:

```
A: Enq(x)       0
A: Ok(())       3
B: Enq(y)       1
B: Ok(())       2
B: Deq()        4
B: Ok(x)        5
A: Deq()        6
A: Ok(y)        7
A: Enq(z)       8
A: Ok(())
```

Notice that the last operation does not exist in the original history. It's possible for a history to be linearizable, as Herlihy and Wing note it, even if an operation *takes effect* before its response. For example:

```
A: Enq(x)       0
B: Deq()        1
C: Ok(x)        2
```

This is equivalent to the sequential history:

```
A: Enq(x)       0
A: Ok(())
B: Deq()        1
B: Ok(x)        2
```

This history is not linearizable:

```
A: Enq(x)       0
A: Ok(())       1
B: Enq(y)       2
B: Ok(())       3
A: Deq()        4
A: Ok(y)        5
```

The culprit in this case is the violation of the queue's ordering. In sequential history, the dequeue would receive *x* and not *y*.

That's it. When we talk about a structure being *linearizable*, what we're really asking is, can any list of valid operations against the structure be shuffled around or added to in such a way that the list is indistinguishable from a sequential history? Each of the synchronization tools we looked at in the last chapter were used to force linearizability by carefully sequencing the ordering of operations across threads. At a different level, these tools also manipulated the ordering of memory loads and stores. Controlling this ordering directly, while also controlling the order of operations on structures, will consume the remainder of this chapter.

Memory ordering – happens-before and synchronizes-with

Each CPU architecture treats memory ordering—the dependency relationships between loads and stores—differently. We discussed this in detail in `Chapter 1`, *Preliminaries – Machine Architecture and Getting Started with Rust*. Suffice it to say here in summary, x86 is a *strongly-ordered* architecture; stores by some thread will be seen by all other threads in the order they were performed. ARM, meanwhile, is a weakly-ordered architecture with data-dependency; loads and stores may be re-ordered in any fashion excepting those that would violate the behavior of a single, isolated thread, and, if a load depends on the results of a previous load, you are guaranteed that the previous load will occur rather than be cached. Rust exposes its own model of memory ordering to the programmer, abstracting away these details. Our programs must, then, be correct according to Rust's model, and we must trust `rustc` to interpret this correctly for our target CPU. If we mess up in Rust's model, we might see `rustc` come and re-order instructions to break linearizability, even if the guarantees of our target CPU would otherwise have helped us skate by.

We discussed Rust's memory model in detail in `Chapter 3`, *The Rust Memory Model – Ownership, References and Manipulation*, noting that its low-level details are inherited near whole-cloth from LLVM (and thus C/C++). We did, however, defer a discussion of dependency relationships in the model until this chapter. The topic is tricky enough in its own right, as you'll see directly. We will not go into the full details here, owing to the major difficulty in doing so; see LLVM's *Memory Model for Concurrent Operations* (`http://llvm.org/do`

cs/LangRef.html#memory-model-for-concurrent-operations) and its link tree for the full, challenging read. We will instead note that, outside of writing optimizers or especially challenging low-level code for which you do need to study up on the LLVM memory model reference, it's sufficient to understand that when we talk about ordering in Rust we're talking about a particular kind of causality—*happens-before*/*synchronizes-with*. So long as we structure our operations according to this causality and can demonstrate linearizability, we'll be in good shape.

What is the happens-before/synchronizes-with causality? Each of these refers to a particular kind of ordering relationship. Say we have three operations A, B, and C. We say that A *happens-before* B if:

- A and B are performed on the same thread and A is executed and then B is executed according to program order
- A happens-before C and C happens-before B
- A *synchronizes-with* B

Note that the first point refers to single-thread program order. Rust makes no guarantees about multi-threaded program order but does assert that single-threaded code will run in its apparent textual order, even if, in actuality, both the optimized Rust is re-ordered and the underlying hardware is re-ordering as well. We can say that A *synchronizes-with* B if all of the following are true:

- B is a load from some atomic variable `var` with `Acquire` or `SeqCst` ordering
- A is a store to the same atomic variable `var` with `Release` or `SeqCst` ordering

- `B` reads the value from `var`

The last point exists to ensure that compilers or hardware won't optimize away the load in the first point. But, did some additional information get stuck in there? Yep! More on that in a minute.

How does linearizability link up to the causal ordering we've just described? It's important to understand this. A structure in Rust can obey the causal ordering but still not linearize under examination. Take, for instance, the discussion of Ring in previous chapters, where, even though the causal ordering was protected by a mutex, writes were stomped due to an offset bug. That implementation was not linearizable but it was causally ordered. In that sense, then, getting the causality of your program is a necessary but not sufficient condition for writing a fit-for-purpose concurrent data structure. Linearizability is an analysis done in terms of the operations on a structure, while happens-before/synchronizes-with is an ordering that happens inside of and between operations.

Now, we mentioned `Acquire`, `SeqCst`, and `Release`; what were they? Rust's memory ordering model allows for different atomic orderings to be applied to loads and stores. These orderings control the underlying CPU and the `rustc` optimizer's take on which instructions to shuffle around and when to stall pipelines waiting for results from a load. The orderings are defined in an enumeration called `std::sync::atomic::Ordering`. We'll go through them one by one.

Ordering::Relaxed

This ordering didn't show up in the definition of happens-before/synchronizes-with. There's a very good reason for that. `Relaxed` ordering implies no guarantees. Loads and stores are free to be re-ordered around a load or store with relaxed ordering. That is, loads and stores can migrate *up* or *down* in program order from the point of view of a `Relaxed` operation.

The compiler and the hardware are free to do whatever they please in the presence of `Relaxed` ordering. There's quite a lot of utility in this, as we'll see in the upcoming detailed examples. For now, consider counters in a concurrent program used for self-telemetry, or data structures that have been designed to be eventually consistent. No need for strict ordering there.

Before we move on, it's worth being very clear about what we mean by the ordering of loads and stores. Loads and stores on *what*? Well, atomics. In Rust, these are exposed in `std::sync::atomic` and there are, as of writing this book, four stable atomic types available:

- `AtomicUsize`
- `AtomicIsize`
- `AtomicPtr`
- `AtomicBool`

There's an ongoing discussion centered around using Rust issue `#32976` to support smaller atomic types—you'll note, of the stable four, three

are machine word sized and `AtomicBool` is currently also one of them—but progress there looks to have stalled out, for want of a champion. So, for now, these are the types. Of course, there's more loads and stores than just atomics—manipulation of raw pointers, to name a familiar example. These are data accesses and have no ordering guarantees beyond those that are built into Rust to begin with. Two threads may load and store to the same raw pointer at the exact same moment and there's nothing that can be done about it without prior coordination.

Ordering::Acquire

`Acquire` is an ordering that deals with loads. Recall that we previously said that if a thread, `B`, loads var with `Acquire` ordering and thread `A` then stores with `Release` ordering, then `A` synchronizes-with `B`, which means that `A` happens-before `B`. Recall that back in the previous chapter we ran into `Acquire` in the context of hopper:

```
pub unsafe fn push_back(
    &self,
    elem: T,
    guard: &mut MutexGuard<BackGuardInner<S>>,
) -> Result<bool, Error<T>> {
    let mut must_wake_dequeuers = false;
    if self.size.load(Ordering::Acquire) == self.capacity {
        return Err(Error::Full(elem));
    } else {
        assert!((*self.data.offset((*guard).offset)).is_none());
        *self.data.offset((*guard).offset) = Some(elem);
        (*guard).offset += 1;
        (*guard).offset %= self.capacity as isize;
        if self.size.fetch_add(1, Ordering::Release) == 0 {
            must_wake_dequeuers = true;
        }
    }
    Ok(must_wake_dequeuers)
}
```

The size seen there is an `AtomicUsize`, which the current thread is using to determine if there's space left for it to emplace a new element. Later, once the element has been emplaced, the thread increases the size with `Release` ordering. That… seems preposterous. It's clearly counter-intuitive to program order for the increase of size to happen-before the capacity check. And, that's true. It's worth keeping in mind that there's another thread in the game:

```
pub unsafe fn pop_front(&self) -> T {
    let mut guard = self.front_lock.lock()
                        .expect("front lock poisoned");
    while self.size.load(Ordering::Acquire) == 0 {
        guard = self.not_empty
            .wait(guard)
            .expect("oops could not wait pop_front");
    }
    let elem: Option<T> = mem::replace(&mut
    *self.data.offset((*guard).offset), None);
    assert!(elem.is_some());
    *self.data.offset((*guard).offset) = None;
    (*guard).offset += 1;
    (*guard).offset %= self.capacity as isize;
    self.size.fetch_sub(1, Ordering::Release);
    elem.unwrap()
}
```

Consider what would happen if there were only two threads, `A` and `B`, and `A` only performed `push_back` and `B` only performed `pop_front`. The `mutexes` function to provide isolation for threads of the same kind—those that push or pop—and so are meaningless in this hypothesis and can be ignored. All that matters are the atomics. If the size is zero and `B` is scheduled first, it will empty the condvar loop on an Acquire-load of size. When `A` is scheduled and finds that there is spare capacity for its element, the element will be enqueued and, once done, the size will be `Release`-stored, meaning that some lucky loop of `B` will happen-after a successful push-back from `A`.

The Rust documentation for `Acquire` says:

"When coupled with a load, all subsequent loads will see data written before a store with Release ordering on the same value in other threads."

The LLVM documentation says:

"Acquire provides a barrier of the sort necessary to acquire a lock to access other memory with normal loads and stores."

How should we take this? An `Acquire` load keeps loads and stores those that come after it from migrating up, with regard to program order. Loads and stores prior to the `Acquire` might migrate down, with regard to program order, though. Think of an `Acquire` as akin to the starting line of a lock, but one that is porous on the top side.

Ordering::Release

If `Acquire` is the starting line of a lock, then `Release` is the finish line. In fact, the LLVM documentation says:

"Release is similar to Acquire, but with a barrier of the sort necessary to release a lock."

It's may be a little more subtle than `lock` might suggest, though. Here's what the Rust documentation has to say:

"When coupled with a store, all previous writes become visible to the other threads that perform a load with Acquire ordering on the same value."

Where `Acquire` stopped loads and stores after itself from migrating upward, `Release` stops loads and stores prior to itself from migrating downward, with regard to program order. Like `Acquire`, `Release` does not stop loads and stores prior to itself from migrating up. A `Release` store is akin to the finish line of a lock, but one that is porous on the bottom side.

Incidentally, there's an interesting complication here that lock intuition is not helpful with. Question: can two `Acquire` loads or two `Release` stores happen simultaneously? The answer is, well, it depends on quite a lot, but it is possible. In the preceding hopper example, the spinning `pop_front` thread does not block the `push_back` thread from performing its check of size, either by permanently holding the size until a `Release` came along to free the `pop_front` thread, even temporarily, until the while loop can be checked. Remember, the causality model of the language says nothing about the ordering of two `Acquire` loads or `Release` stores. It's undefined what happens and is probably going to strongly depend on your hardware. All we do know is that the `pop_front` thread will not partially see the stores done to

memory by `push_back`; it'll be all-or-none. All of the loads and stores after an `Acquire` and before a `Release` come as a unit.

Ordering::AcqRel

This variant is a combination of `Acquire` and `Release`. When a load is performed with `AcqRel`, then `Acquire` is used, and stored to `Release`. `AcqRel` is not just a convenience in that it combines the ordering behaviors of both `Acquire` and `Release`—the ordering is porous on neither side. That is, loads and stores after an `AcqRel` cannot move up, as with `Acquire`, and loads and stores prior to an `AcqRel` cannot move down, as with `Release`. Nifty trick.

Before moving on to the next ordering variation, it's worth pointing out that so far we've only seen examples of a thread performing an `Acquire`/`Release` in a pair, in cooperation with another thread. It doesn't have to be this way. One thread can always perform `Acquire` and another can always perform `Release`. The causality definition is specifically in terms of threads that perform one but not both, except in cases when `A` is equal to `B`, of course. Let's break down an example:

```
A: store X                  1
A: store[Release] Y         2
B: load[Acquire] Y          3
B: load X                   4
B: store[Release] Z         5
C: load[Acquire] Z          6
C: load X                   7
```

Here, we have three threads, `A`, `B`, and `C`, performing a mix of plain loads and stores with atomic loads and stores. `store[Release]` is an atomic store where store is not. As in the section on linearizability, we've numbered each one of the operations, but be aware that this does not represent a timeline so much as the numbers are convenient names. What can we say of the causality of this example? We know:

- `1 happens-before 2`
- `2 synchronizes-with 3`
- `3 happens-before 4`
- `3 happens-before 5`
- `5 synchronizes-with 6`
- `6 happens-before 7`

We'll see more analysis of this sort later in the chapter.

Ordering::SeqCst

Finally, and on the opposite side of the spectrum from `Relaxed`, there is `SeqCst`, which stands for SEQuentially ConsiSTent. `SeqCst` is like `AcqRel` but with the added bonus that there's a total order across all threads between sequentially consistent operations, hence the name. `SeqCst` is a pretty serious synchronization primitive and one that you should use only at great need, as forcing a total order between all threads is not cheap; every atomic operation is going to require a synchronization cycle, for whatever that means for your CPU. A mutex, for instance, is an acquire-release structure—more on that shortly—but there are legitimate cases for a super-mutex like `SeqCst`. For one, implementing older papers. In the early days of atomic programming, the exact consequence of more relaxed—but not `Relaxed`—memory orders were not fully understood. You'll see literature that'll make use of sequentially consistent memory and think nothing of it. At the very least, follow along and then relax as needed. This happens more than you might think, at least to your author. Your mileage may vary. There are cases where it seems like we're stuck on `SeqCst`. Consider a multiple-producer, multiple-consumer setup like this one that has been adapted from CppReference (see the *Further reading* section shown in the final section):

```
#[macro_use]
extern crate lazy_static;

use std::thread;
use std::sync::Arc;
use std::sync::atomic::{AtomicBool, AtomicUsize, Ordering};

lazy_static! {
static ref X: Arc<AtomicBool> = Arc::new(AtomicBool::new(false));
```

```
static ref Y: Arc<AtomicBool> = Arc::new(AtomicBool::new(false));
static ref Z: Arc<AtomicUsize> = Arc::new(AtomicUsize::new(0));
}

fn write_x() {
    X.store(true, Ordering::SeqCst);
}

fn write_y() {
    Y.store(true, Ordering::SeqCst);
}

fn read_x_then_y() {
    while !X.load(Ordering::SeqCst) {}
    if Y.load(Ordering::SeqCst) {
        Z.fetch_add(1, Ordering::Relaxed);
    }
}

fn read_y_then_x() {
    while !Y.load(Ordering::SeqCst) {}
    if X.load(Ordering::SeqCst) {
        Z.fetch_add(1, Ordering::Relaxed);
    }
}

fn main() {
    let mut jhs = Vec::new();
    jhs.push(thread::spawn(write_x)); // a
    jhs.push(thread::spawn(write_y)); // b
    jhs.push(thread::spawn(read_x_then_y)); // c
    jhs.push(thread::spawn(read_y_then_x)); // d
    for jh in jhs {
        jh.join().unwrap();
    }
    assert!(Z.load(Ordering::Relaxed) != 0);
}
```

Say we weren't enforcing `SeqCst` between the threads in our example. What then? The thread `c` loops until the thread `a` flips Boolean `X`, loads `Y` and, if true, increments `Z`. Likewise for thread `d`, except `b` flips the Boolean `Y` and the increment is guarded on `X`. For this program to not crash after its threads have hung up, the threads have to see the flips of `X` and `Y` happen in the same order. Note that the order itself does not matter, only so much as it is the same as seen from every thread.

Otherwise it's possible that the stores and loads will interleave in such a way as to avoid the increments. The reader is warmly encouraged to fiddle with the ordering on their own.

Building synchronization

Now that we have a good grounding on the atomic primitives available to us in the Rust standard library, and have, moreover, a solid theoretical background, it's time for us to build on these foundations. In the past few chapters, we've been teasing our intention to build up mutexes and semaphores from primitives and, well, now the time has come.

Mutexes

Now that we understand linearizability and memory orderings, let's ask ourselves a question. What *exactly* is a mutex? We know its properties as an atomic object:

- Mutex supports two operations, *lock* and *unlock*.
- A mutex is either *locked* or *unlocked*.
- The operation *lock* will move a mutex into a *locked* state if and only if the mutex is *unlocked*. The thread that completes *lock* is said to hold the lock.
- The operation *unlock* will move a mutex into *unlocked* state if an only if the mutex is previously *locked* and the caller of *unlock* is the holder of the lock.
- All loads and stores which occur after and before a *lock* in program order must not be moved prior to or after the *lock* operation.
- All loads and stores that occur before an *unlock* in program order must not be moved to after the *unlock*.

The second to last point is subtle. Consider a soft-mutex whose lock only keeps loads and stores after migrating up. This means that load/stores could migrate into the soft-mutex, which is all well and good unless your migration is the lock of another soft-mutex. Consider the following:

```
lock(m0)
unlock(m0)
lock(m1)
unlock(m1)
```

This could be re-arranged to the following:

```
lock(m0)
lock(m1)
unlock(m0)
unlock(m1)
```

Here, the `unlock(m0)` has migrated down in program order into the mutual exclusion zone of `m1`, deadlocking the program. That is a problem.

You might not be surprised to learn that there's decades of research, at this point, into the question of mutual exclusion. How should a mutex be made? How does fairness factor into the implementation? The latter is a significantly broad topic, avoiding thread *starvation*, and is one we'll more or less dodge here. Much of the historical research is written with regard to machines that have no concept of concurrency control. Lamport's Bakery algorithm—see the *Further reading* section– provides mutual exclusion that's absent in any hardware support but assumes reads and writes are sequentially consistent, an assumption that does not hold on modern memory hierarchies. In no small sense do we live in a very happy time for concurrent programming: our machines expose synchronization primitives directly, greatly simplifying the production of fast and correct synchronization mechanisms.

Compare and set mutex

First, let's look at a very simple mutex implementation, an atomic swap mutex that spins around in a loop until the correct conditions arrive. That is, let's build a *spin-lock*. This mutex is so named because every thread that blocks on `lock` is burning up CPU, spinning on the condition that holds it from acquiring the lock. Anyhow, you'll see. We'll lay down our `Cargo.toml` first:

```
[package]
name = "synchro"
version = "0.1.0"
authors = ["Brian L. Troutwine <brian@troutwine.us>"]

[[bin]]
name = "swap_mutex"
doc = false

[[bin]]
name = "status_demo"
doc = false

[[bin]]
name = "mutex_status_demo"
doc = false

[[bin]]
name = "spin_mutex_status_demo"
doc = false

[[bin]]
name = "queue_spin"
doc = false

[[bin]]
name = "crossbeam_queue_spin"
doc = false
```

```
[dependencies]
crossbeam = { git = "https://github.com/crossbeam-rs/crossbeam.git",
rev = "89bd6857cd701bff54f7a8bf47ccaa38d5022bfb" }

[dev-dependencies]
quickcheck = "0.6"
```

Now, we need `src/lib.rs`:

```
    extern crate crossbeam;

    mod queue;
    mod swap_mutex;
    mod semaphore;

    pub use semaphore::*;
    pub use swap_mutex::*;
    pub use queue::*;
```

There's a fair bit here we're not going to use straight away, but we'll come to it shortly. Now, let's dive into `src/swap_mutex.rs`. First, our preamble:

```
extern crate crossbeam;

mod queue;
mod swap_mutex;
mod semaphore;
```

Very little surprise here by this point. We see the usual imports, plus a few new ones—`AtomicBool` and `Ordering`, as discussed previously. We implement `Send` and `Sync` ourselves—discussed in the previous chapter—because while *we* can prove that our `SwapMutex<T>` is `threadsafe`, Rust cannot. The `SwapMutex<T>` is small:

```
pub struct SwapMutex<T> {
    locked: AtomicBool,
    data: *mut T,
}
```

The field `locked` is an `AtomicBool`, which we'll be using to provide isolation between threads. The idea is simple enough—if a thread shows up and finds that `locked` is false then the thread may acquire the lock, else it has to spin. It's also worth noting that our implementation lives a little on the wild side by keeping a raw pointer to `T`. The Rust standard library `std::sync::Mutex` keeps its interior data, as of writing this book, in an `UnsafeCell`, which we don't have access to in stable. In this implementation we're just going to be storing the pointer and not doing any manipulation through it. Still, we've got to be careful about dropping the `SwapMutex`. Creating a `SwapMutex` happens like you might expect:

```
impl<T> SwapMutex<T> {
    pub fn new(t: T) -> Self {
        let boxed_data = Box::new(t);
        SwapMutex {
            locked: AtomicBool::new(false),
            data: Box::into_raw(boxed_data),
        }
    }
```

The `T` is moved into `SwapMutex` and we box it to then get its raw pointer. That's necessary to avoid a stack value being pushed into `SwapMutex` and promptly disappearing when the stack frame changes, all of which was discussed in great detail in `Chapter 2`, *Sequential Rust Performance and Testing*. A mutex always starts unlocked, else there'd be no way to ever acquire it. Now, let's look at locking the mutex:

```
pub fn lock(&self) -> SwapMutexGuard<T> {
    while self.locked.swap(true, Ordering::AcqRel) {
        thread::yield_now();
    }
    SwapMutexGuard::new(self)
}
```

The API is a little different compared to the standard library `Mutex`. For one, `SwapMutex` does not track poisoning, and, as a result, if lock is

invoked it's guaranteed to return with a guard, called `SwapMutexGuard`. `SwapMutex` *is* scope-based, however; just like standard library `MutexGuard` there's no need—nor ability—to ever call an explicit unlock. There is an unlock though, used by the drop of `SwapMutexGuard`:

```
fn unlock(&self) -> () {
    assert!(self.locked.load(Ordering::Relaxed) == true);
    self.locked.store(false, Ordering::Release);
}
```

In this simple mutex, all that needs to be done to unlock is set the `locked` field to false, but not before confirming that, in fact, the calling thread has a right to unlock the mutex.

Now, can we convince ourselves that this mutex is correct? Let's go through our criteria:

"Mutex supports two operations, lock and unlock."

Check. Next:

"A mutex is either locked or unlocked."

Check, by reason that this is flagged by a Boolean. Finally:

"The operation lock will move a mutex into a locked state if and only if the mutex is unlocked. The thread that completes the lock is said to hold the lock."

Let's talk through what `AtomicBool::swap` does. The full type is `swap(&self, val: bool, order: Ordering) -> bool` and the function does what you might expect by the name; it atomically swaps the bool at self with the bool `val` atomically, according to the passed ordering, and returns the previous value. Here, then, each thread is competing to write true into the locked flag and only one thread at a time will see false returned as the previous value, owing to the atomicity of swapping. The thread that has written false is now the holder of the lock, returning `SwapMutexGuard`. Locking a `SwapMutex` can only be done to an unlocked `SwapMutex` and it is done exclusive of other threads.

"The operation unlock will move a mutex into an unlocked state if and only if the mutex is previously locked and the caller of unlock is the holder of the lock."

Let's consider unlock. Recall first that it's only possible to call from `SwapMutexGuard`, so the assertion that the caller has the right to unlock the `SwapMutex` is a check against a malfunctioning lock: only one thread at a time can hold `SwapMutex`, and as a result there will only ever be one `SwapMutexGuard` in memory. Because of the nature of `Release` ordering, we're guaranteed that the `Relaxed` load of `locked` will occur before the store, so it's guaranteed that when the store does happen the value of `locked` will be true. Only the holder of the lock can unlock it, and this property is satisfied as well.

"All loads and stores that occur after and before a lock in program order must not be moved prior to or after the lock."

This follows directly from the behavior of `AcqRel` ordering.

"All loads and stores that occur before an unlock in program order must not be moved to after the unlock."

This follows from the behavior of `Release` ordering.

Bang, we have ourselves a mutex. It's not an especially power-friendly mutex, but it does have the essential properties. Here's the rest of the thing, for completeness's sake:

```
impl<T> Drop for SwapMutex<T> {
    fn drop(&mut self) {
        let data = unsafe { Box::from_raw(self.data) };
        drop(data);
    }
}

pub struct SwapMutexGuard<'a, T: 'a> {
    __lock: &'a SwapMutex<T>,
}

impl<'a, T> SwapMutexGuard<'a, T> {
    fn new(lock: &'a SwapMutex<T>) -> SwapMutexGuard<'a, T> {
        SwapMutexGuard { __lock: lock }
```

```
        }
    }

    impl<'a, T> Deref for SwapMutexGuard<'a, T> {
        type Target = T;

        fn deref(&self) -> &T {
            unsafe { &*self.__lock.data }
        }
    }

    impl<'a, T> DerefMut for SwapMutexGuard<'a, T> {
        fn deref_mut(&mut self) -> &mut T {
            unsafe { &mut *self.__lock.data }
        }
    }

    impl<'a, T> Drop for SwapMutexGuard<'a, T> {
        #[inline]
        fn drop(&mut self) {
            self.__lock.unlock();
        }
    }
```

Now, let's build something with `SwapMutex`. In `src/bin/swap_mutex.rs`, we'll replicate the bridge problem from the last chapter, but on top of `SwapMutex`, plus some fancy add-ons now that we know some clever things to do with atomics. Here's the preamble:

```
    extern crate synchro;

    use std::sync::atomic::{AtomicUsize, Ordering};
    use std::sync::Arc;
    use synchro::SwapMutex;
    use std::{thread, time};

    #[derive(Debug)]
    enum Bridge {
        Empty,
        Left(u8),
        Right(u8),
    }
```

Familiar enough. We pull in our `SwapMutex`, and define `Bridge`. Fairly unsurprising material. To continue:

```
static LHS_TRANSFERS: AtomicUsize = AtomicUsize::new(0);
static RHS_TRANSFERS: AtomicUsize = AtomicUsize::new(0);
```

This is new, though. Statics in Rust are compile-time globals. We've seen their lifetime static floating around from time to time but have never remarked on it. A static will be part of the binary itself and any initialization code needed for the static will have to be evaluated at compile-time as well as being threadsafe. We're creating two static `AtomicUsizes` here at the top of our program. Why? One of the major issues with the previous rope bridge implementation was its silence. You could watch it in a debugger, sure, but that's slow and won't impress your friends. Now that we have atomics at hand, what we're going to do is get each side of the bridge to tally up how many baboons cross. `LHS_TRANSFERS` are the number of baboons that go from right to left, `RHS_TRANSFERS` is the other direction. Our left and right sides of the bridge are broken out into functions this time around, too:

```
fn lhs(rope: Arc<SwapMutex<Bridge>>) -> () {
    loop {
        let mut guard = rope.lock();
        match *guard {
            Bridge::Empty => {
                *guard = Bridge::Right(1);
            }
            Bridge::Right(i) => {
                if i < 5 {
                    *guard = Bridge::Right(i + 1);
                }
            }
            Bridge::Left(0) => {
                *guard = Bridge::Empty;
            }
            Bridge::Left(i) => {
                LHS_TRANSFERS.fetch_add(1, Ordering::Relaxed);
                *guard = Bridge::Left(i - 1);
            }
        }
```

```
    }
}

fn rhs(rope: Arc<SwapMutex<Bridge>>) -> () {
    loop {
        let mut guard = rope.lock();
        match *guard {
            Bridge::Empty => {
                *guard = Bridge::Left(1);
            }
            Bridge::Left(i) => {
                if i < 5 {
                    *guard = Bridge::Left(i + 1);
                }
            }
            Bridge::Right(0) => {
                *guard = Bridge::Empty;
            }
            Bridge::Right(i) => {
                RHS_TRANSFERS.fetch_add(1, Ordering::Relaxed);
                *guard = Bridge::Right(i - 1);
            }
        }
    }
}
```

This ought to be familiar enough from the previous chapter, but do note that the transfer counters are now being updated. We've used `Relaxed` ordering—the increments are guaranteed to land but it's not especially necessary for them to occur strictly before or after the modification to the guard. Finally, the `main` function:

```
fn main() {
    let mtx: Arc<SwapMutex<Bridge>> =
Arc::new(SwapMutex::new(Bridge::Empty));

    let lhs_mtx = Arc::clone(&mtx);
    let _lhs = thread::spawn(move || lhs(lhs_mtx));
    let _rhs = thread::spawn(move || rhs(mtx));

    let one_second = time::Duration::from_millis(1_000);
    loop {
        thread::sleep(one_second);
        println!(
```

```
                "Transfers per second:\n    LHS: {}\n    RHS: {}",
                LHS_TRANSFERS.swap(0, Ordering::Relaxed),
                RHS_TRANSFERS.swap(0, Ordering::Relaxed)
            );
        }
    }
```

We can see here that the mutex is set up, the threads have been spawned, and that the main thread spends its time in an infinite loop printing out information on transfer rates. This is done by sleeping the thread in one-second intervals, swapping the transfer counters with zero and printing the previous value. The output looks something like this:

```
> cargo build
> ./target/debug/swap_mutex
Transfers per second:
    LHS: 787790
    RHS: 719371
Transfers per second:
    LHS: 833537
    RHS: 770782
Transfers per second:
    LHS: 848662
    RHS: 776678
Transfers per second:
    LHS: 783769
    RHS: 726334
Transfers per second:
    LHS: 828969
    RHS: 761439
```

The exact numbers will vary depending on your system. Also note that the numbers are not very close in some instances, either. The `SwapMutex` is not *fair*.

An incorrect atomic queue

Before we build anything else, we're going to need a key data structure—a unbounded queue. In `Chapter 5`, *Locks – Mutex, Condvar, Barriers, and RWLock*, we discussed a bounded deque protected at either end by mutexes. We're in the business, now, of building synchronization and can't make use of the mutex approach. Our ambition is going to be to produce an unbounded first-in-first-out data structure that has no locks, never leaves an enqueuer or dequeuer deadlocked, and is linearizable to a sequential queue. It turns out there's a pretty straightforward data structure that achieves this aim; the Michael and Scott Queue, introduced in their 1995 paper *Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms*. The reader is encouraged to breeze through that paper before continuing on with our discussion here, but it's not strictly necessary.

A word of warning. Our implementation will be *wrong*. The careful reader will note that we're following the paper closely. There are two, and maybe more, issues with the implementation we'll present, one major and unresolvable and the other addressable with some care. Both are fairly subtle. Let's dig in.

Queue is defined in `src/queue.rs`, the preamble of which is:

```
use std::ptr::null_mut;
use std::sync::atomic::{AtomicPtr, Ordering};

unsafe impl<T: Send> Send for Queue<T> {}
unsafe impl<T: Send> Sync for Queue<T> {}
```

Right off you can see from `null_mut` that we're going to be dealing with raw pointers. The `AtomicPtr` is new, though we did mention it in passing earlier in the chapter. An atomic pointer adapts a raw pointer —`*mut T`—to be suitable for atomic operations. There's no runtime overhead associated with AtomicPtr; the Rust documentation notes that the type has the same in-memory representation as a `*mut T`. Modern machines expose instructions, giving the programmer the ability to fiddle with memory atomically. Capabilities vary by processor, as you'd expect. LLVM and therefore Rust expose these atomic memory fiddling capabilities through `AtomicPtr`, allowing the range of pointer fiddling in the sequential language but atomically. What this means, in practice, is that we can start setting up happens-before/synchronizes-with causality relationships for pointer manipulation, which is essential for building data structures.

Here's the next part:

```
struct Node<T> {
    value: *const T,
    next: AtomicPtr<Node<T>>,
}

impl<T> Default for Node<T> {
    fn default() -> Self {
        Node {
            value: null_mut(),
            next: AtomicPtr::default(),
        }
    }
}

impl<T> Node<T> {
    fn new(val: T) -> Self {
        Node {
            value: Box::into_raw(Box::new(val)),
            next: AtomicPtr::default(),
        }
    }
}
```

The interior of deque from the previous chapter was a single, contiguous block. That's not the approach we're taking here. Instead, every element inserted into the queue will get a `Node` and that `Node` will point to the next `Node`, which may or may not exist yet. It's a linked list. The contiguous block approach is a bit harder to pull off in an atomic context—though it is entirely possible and there are discussions in the *Further reading* section papers—and would come down to a linked list of contiguous blocks. It's more trouble than it's worth for our purposes here:

```
struct InnerQueue<T> {
    head: AtomicPtr<Node<T>>,
    tail: AtomicPtr<Node<T>>,
}
```

One key thing to note is that `Node` holds a pointer to a heap allocated `T`, not the `T` directly. In the preceding code, we have the `InnerQueue<T>` of `Queue<T>`, pulling the usual inner/outer structure detailed elsewhere in this book and in `rustc`. Why is it important to note that `Node` doesn't hold its `T` directly? The value inside of the head of the `Queue<T>` is never inspected. The head of the `Queue` is a sentinel. When the `InnerQueue` is created, we'll see the following:

```
   impl<T> InnerQueue<T> {
       pub fn new() -> Self {
           let node = Box::into_raw(Box::new(Node::default()));
           InnerQueue {
               head: AtomicPtr::new(node),
               tail: AtomicPtr::new(node),
           }
       }
```

Both the `head` and `tail` of the `InnerQueue` point to the same nonsense-valued `Node`, as expected. The value at the outset is, in fact, null. Atomic data structures have issues with memory reclamation in that coordinating drops is problematic and must be done only once. It's possible to alleviate this issue somewhat by relying on Rust's type

system, but it's still a non-trivial project and is an active area of research, generally. Here, we note that we're careful to hand out the ownership of the element only once. Being a raw pointer, it can be given away more than once at a time, but that path leads to double-frees. `InnerQueue` converts `*const T` into `T`—an unsafe operation—and just never dereferences the `*const T` again, allowing the caller to do the drop in its own sweet time:

```
pub unsafe fn enq(&mut self, val: T) -> () {
    let node = Box::new(Node::new(val));
    let node: *mut Node<T> = Box::into_raw(node);

    loop {
        let tail: *mut Node<T> = self.tail.load(Ordering::Acquire);
        let next: *mut Node<T> =
        (*tail).next.load(Ordering::Relaxed);
        if tail == self.tail.load(Ordering::Relaxed) {
            if next.is_null() {
                if (*tail).next.compare_and_swap(next, node,
                 Ordering::Relaxed) == next {
                    self.tail.compare_and_swap(tail, node,
                     Ordering::Release);
                    return;
                }
            }
        } else {
            self.tail.compare_and_swap(tail, next,
            Ordering::Release);
        }
    }
}
```

This is the `enq` operation, marked unsafe because of the raw pointer manipulation going on. That's an important point to consider—`AtomicPtr` is necessarily going to be done with raw pointers. There's a lot going on here, so let's break it up into smaller chunks:

```
pub unsafe fn enq(&mut self, val: T) -> () {
    let node = Box::new(Node::new(val));
    let node: *mut Node<T> = Box::into_raw(node);
```

Here, we're constructing the `Node` for `val`. Notice we're using the same boxing, the `into_raw` approach used so often in previous chapters. This node doesn't have a place in the queue yet and the calling thread does not hold an exclusive lock over the queue. Insertion will have to take place in the midst of other insertions:

```
        loop {
            let tail: *mut Node<T> = self.tail.load(Ordering::Acquire);
            let next: *mut Node<T> =
            (*tail).next.load(Ordering::Relaxed);
```

With that in mind, it's entirely possible that an insertion attempt can fail. The enqueing of an element in a queue takes place at the `tail`, the pointer to which we load up and call last. The next node after `tail` is called `next`. In a sequential queue, we'd be guaranteed that the next of `tail` is null, but that's not so here. Consider that between the load of the `tail` pointer and the load of the `next` pointer an `enq` run by another thread may have already completed.

Enqueing is, then, an operation that might take several attempts before we hit just the right conditions for it to succeed. Those conditions are last still being the `tail` of the structure and next being `null`:

```
            if tail == self.tail.load(Ordering::Relaxed) {
                if next.is_null() {
```

Note that the first load of `tail` is `Acquire` and each of the possible stores of it, in either branch, are `Release`. This satisfies our `Acquire`/`Release` needs, with regard to locking primitives. All other stores and loads here are conspicuously `Relaxed`. How can we be sure we're not accidentally stomping writes or, since this is a linked list, cutting them loose in memory? That's where the `AtomicPtr` comes in:

```
                    if (*tail).next.compare_and_swap(next, node,
                     Ordering::Relaxed) == next {
                        self.tail.compare_and_swap(tail, node,
                         Ordering::Release);
                        return;
                    }
```

It *is* entirely possible that by the time we've detected the proper conditions for enqueing another thread will have been scheduled in, have detected the proper conditions for enqueing, and have been enqueued. We attempt to slot our new node in with `(*last).next.compare_and_swap(next, node, Ordering::Relaxed)`, that is, we compare the current next of last and if and only if that succeeds—that's the `== next` bit—do we attempt to set `tail` to the node pointer, again with a compare and swap. If both of those succeed then the new element has been fully enqueued. It's possible that the swap of `tail` will fail, however, in which case the linked list is correctly set up but the `tail` pointer is off. Both the `enq` and `deq` operations must be aware they could stumble into a situation where the `tail` pointer needs to be adjusted. That is in fact how the `enq` function finishes off:

```
                }
            } else {
                self.tail.compare_and_swap(tail, next,
                 Ordering::Release);
            }
        }
    }
```

On an x86, all of these `Relaxed` operations are more strict but on ARMv8 there will be all sorts of reordering. It's very important, and difficult, to establish a causal relationship between all modifications. If, for example, we swapped the `tail` pointer and then the `next` of the `tail` pointer, we'd open ourselves up to breaking the linked list, or making whole isolated chains depending on the threads' view of memory. The `deq` operation is similar:

```
    pub unsafe fn deq(&mut self) -> Option<T> {
        let mut head: *mut Node<T>;
        let value: T;
        loop {
            head = self.head.load(Ordering::Acquire);
            let tail: *mut Node<T> = self.tail.load(Ordering::Relaxed);
            let next: *mut Node<T> =
            (*head).next.load(Ordering::Relaxed);
            if head == self.head.load(Ordering::Relaxed) {
                if head == tail {
                    if next.is_null() {
                        return None;
                    }
                    self.tail.compare_and_swap(tail, next,
                     Ordering::Relaxed);
                } else {
                    let val: *mut T = (*next).value as *mut T;
                    if self.head.compare_and_swap(head, next,
                    Ordering::Release) == head {
                        value = *Box::from_raw(val);
                        break;
                    }
                }
            }
        }
        let head: Node<T> = *Box::from_raw(head);
        drop(head);
        Some(value)
    }
}
```

The function is a loop, like `enq`, in which we search for the correct conditions and circumstance to dequeue an element. The first outer if clause checks that head hasn't shifted on us, while the inner first branch is to do with a queue that has no elements, where first and last are pointers to the same storage. Note here that if next is not null we try and patch up a partially completed linked list of nodes before looping back around again for another pass at dequeing.

This is because, as discussed previously, `enq` may not fully succeed. The second inner loop is hit when `head` and `tail` are not equal, meaning there's an element to be pulled. As the inline comment explains, we

give out the ownership of the element `T` when the first hasn't shifted on us but are careful not to dereference the pointer until we can be sure we're the only thread that will ever manage that. We can be on account of only one thread that will ever manage to swap the particular first and next pair the calling thread currently holds.

After all of that, the actual outer `Queue<T>` is a touch anti-climactic:

```
pub struct Queue<T> {
    inner: *mut InnerQueue<T>,
}

impl<T> Clone for Queue<T> {
    fn clone(&self) -> Queue<T> {
        Queue { inner: self.inner }
    }
}

impl<T> Queue<T> {
    pub fn new() -> Self {
        Queue {
            inner: Box::into_raw(Box::new(InnerQueue::new())),
        }
    }

    pub fn enq(&self, val: T) -> () {
        unsafe { (*self.inner).enq(val) }
    }

    pub fn deq(&self) -> Option<T> {
        unsafe { (*self.inner).deq() }
    }
```

We've already reasoned our way through the implementation and, hopefully, you, dear reader, are convinced that the idea should function. Where the rubber meets the road is in testing:

```
#[cfg(test)]
mod test {
    extern crate quickcheck;
```

```
    use super::*;
    use std::collections::VecDeque;
    use std::sync::atomic::AtomicUsize;
    use std::thread;
    use std::sync::Arc;
    use self::quickcheck::{Arbitrary, Gen, QuickCheck, TestResult};

    #[derive(Clone, Debug)]
    enum Op {
        Enq(u32),
        Deq,
    }

    impl Arbitrary for Op {
        fn arbitrary<G>(g: &mut G) -> Self
        where
            G: Gen,
        {
            let i: usize = g.gen_range(0, 2);
            match i {
                0 => Op::Enq(g.gen()),
                _ => Op::Deq,
            }
        }
    }
```

This is the usual `test` preamble that we've seen elsewhere in the book. We define an `Op` enumeration to drive an interpreter style `quickcheck test`, which we call here `sequential`:

```
#[test]
fn sequential() {
    fn inner(ops: Vec<Op>) -> TestResult {
        let mut vd = VecDeque::new();
        let q = Queue::new();

        for op in ops {
            match op {
                Op::Enq(v) => {
                    vd.push_back(v);
                    q.enq(v);
                }
                Op::Deq => {
                    assert_eq!(vd.pop_front(), q.deq());
                }
```

```
                }
            }
            TestResult::passed()
        }
        QuickCheck::new().quickcheck(inner as fn(Vec<Op>) ->
        TestResult);
    }
```

We have a `VecDeque` as the model; we know it's a proper queue. Then, without dipping into any kind of real concurrency, we confirm that `Queue` behaves similarly to a `VecDeque`. At least in a sequential setting, `Queue` will work. Now, for a parallel `test`:

```
    fn parallel_exp(total: usize, enqs: u8, deqs: u8) -> bool {
        let q = Queue::new();
        let total_expected = total * (enqs as usize);
        let total_retrieved = Arc::new(AtomicUsize::new(0));

        let mut ejhs = Vec::new();
        for _ in 0..enqs {
            let mut q = q.clone();
            ejhs.push(
                thread::Builder::new()
                    .spawn(move || {
                        for i in 0..total {
                            q.enq(i);
                        }
                    })
                    .unwrap(),
            );
        }

        let mut djhs = Vec::new();
        for _ in 0..deqs {
            let mut q = q.clone();
            let total_retrieved = Arc::clone(&total_retrieved);
            djhs.push(
                thread::Builder::new()
                    .spawn(move || {
                        while total_retrieved.load(Ordering::Relaxed)
                        != total_expected {
                            if q.deq().is_some() {
                                total_retrieved.fetch_add(1,
                                Ordering::Relaxed);
                            }
```

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

    for jh in ejhs {
        jh.join().unwrap();
    }
    for jh in djhs {
        jh.join().unwrap();
    }

    assert_eq!(total_retrieved.load(Ordering::Relaxed),
    total_expected);
    true
}
```

We set up two groups of threads, one responsible for enqueing and the other for dequeing. The enqueuing threads push a `total` number of items through the `Queue` and the dequeuers pull until a counter—shared between each of the dequeuers—hits bingo. Finally, back in the main `test` thread, we confirm that the total number of retrieved items is the same as the expected number of items. It's possible that our dequeing threads will read *past the end* of the queue because of a race between the check on the while loop and the call of `q.deq`, which works in our favor because confirming the queue allows the deque of no more elements than were enqueued. That, and there are no double-free crashes when the `test` is run. This inner `test` function is used twice, once in repeated runs and then again in a `quickcheck` setup:

```
#[test]
fn repeated() {
    for i in 0..10_000 {
        println!("{}", i);
        parallel_exp(73, 2, 2);
    }
}

#[test]
fn parallel() {
    fn inner(total: usize, enqs: u8, deqs: u8) -> TestResult {
```

```
            if enqs == 0 || deqs == 0 {
                TestResult::discard()
            } else {
                TestResult::from_bool(parallel_exp(total, enqs, deqs))
            }
        }
        QuickCheck::new().quickcheck(inner as fn(usize, u8, u8) ->
        TestResult);
    }
}
```

What's wrong here? If you run the `test` suite, you may or may not hit one of the issues. They are fairly improbable, though we'll shortly see a way to reliably trigger the worst. The first issue our implementation runs into is the ABA problem. In a compare-and-swap operation, pointer `A` is to be swapped by some thread with `B`. Before the check can be completed in the first thread, another thread swaps `A` with `C` and then `C` back again to `A`. The first thread is then rescheduled and performs its compare-and-swap of `A` to `B`, none the wiser that `A` is not really the `A` it had at the start of the swap. This will cause chunks of the queue's linked list to point incorrectly, possibly into the memory that the queue does not rightly own. That's bad enough. What could be worse?

Let's cause a use-after-free violation with this structure. Our demonstration program is short and lives at `src/bin/queue_spin.rs`:

```
extern crate synchro;

use synchro::Queue;
use std::thread;

fn main() {
    let q = Queue::new();

    let mut jhs = Vec::new();

    for _ in 0..4 {
        let eq = q.clone();
        jhs.push(thread::spawn(move || {
            let mut i = 0;
```

```
                loop {
                    eq.enq(i);
                    i += 1;
                    eq.deq();
                }
            }))
        }

        for jh in jhs {
            jh.join().unwrap();
        }
    }
```

The program creates four threads, each of which enqueue and dequeue in sequence as rapidly as possible with no coordination between them. It's important to have at least two threads, else the queue is used sequentially and the issue does not exist:

```
> time cargo run --bin queue_spin
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/queue_spin`
Segmentation fault

real    0m0.588s
user    0m0.964s
sys     0m0.016s
```

Ouch. That took no time at all. Let's have a look at the program in a debugger. We'll use `lldb`, but if you're using `gdb`, the results will be the same:

```
> lldb-3.9 target/debug/queue_spin
(lldb) target create "target/debug/queue_spin"
Current executable set to 'target/debug/queue_spin' (x86_64).
(lldb) run
Process 12917 launched:
'/home/blt/projects/us/troutwine/concurrency_in_rust/external_projects/
synchro/target/debug/queue_spin' (x86_64)
Process 12917 stopped
* thread #2: tid = 12920, 0x0000555555560585
queue_spin`_$LT$synchro..queue..InnerQueue$LT$T$GT$$GT$::deq::heefaa8c9
b1d410ee(self=0x00007ffff6c2a010) + 261 at queue.rs:78, name =
```

```
'queue_spin', stop reason = signal SIGSEGV: invalid address (fault
address: 0x0)
    frame #0: 0x0000555555560585
queue_spin`_$LT$synchro..queue..InnerQueue$LT$T$GT$$GT$::deq::heefaa8c9
b1d410ee(self=0x00007ffff6c2a010) + 261 at queue.rs:78
   75                       }
   76                       self.tail.compare_and_swap(tail, next,
                            Ordering::Relaxed);
   77                   } else {
-> 78                       let val: *mut T = (*next).value as *mut T;
   79                       if self.head.compare_and_swap(head, next,
                            Ordering::Release) == head {
   80                           value = *Box::from_raw(val);
   81                           break;
  thread #3: tid = 12921, 0x0000555555560585
queue_spin`_$LT$synchro..queue..InnerQueue$LT$T$GT$$GT$::deq::heefaa8c9
b1d410ee(self=0x00007ffff6c2a010) + 261 at queue.rs:78, name =
'queue_spin', stop reason = signal SIGSEGV: invalid address (fault
address: 0x0)
    frame #0: 0x0000555555560585
queue_spin`_$LT$synchro..queue..InnerQueue$LT$T$GT$$GT$::deq::heefaa8c9
b1d410ee(self=0x00007ffff6c2a010) + 261 at queue.rs:78
   75                       }
   76                       self.tail.compare_and_swap(tail, next,
                            Ordering::Relaxed);
   77                   } else {
-> 78                       let val: *mut T = (*next).value as *mut T;
   79                       if self.head.compare_and_swap(head, next,
                            Ordering::Release) == head {
   80                           value = *Box::from_raw(val);
   81                           break;
```

Well look at that! And, just to confirm:

```
(lldb) p next
(synchro::queue::Node<i32> *) $0 = 0x0000000000000000
```

Can we turn up another? Yes:

```
(lldb) Process 12893 launched:
'/home/blt/projects/us/troutwine/concurrency_in_rust/external_projects/
synchro/target/debug/queue_spin' (x86_64)
Process 12893 stopped
* thread #2: tid = 12896, 0x000055555555fb3e
```

```
queue_spin`core::sync::atomic::atomic_load::hd37078e3d501f11f(dst=0x000
0000000000008, order=Relaxed) + 78 at atomic.rs:1502, name =
'queue_spin', stop reason = signal SIGSEGV: invalid address (fault
address: 0x8)
    frame #0: 0x000055555555fb3e
queue_spin`core::sync::atomic::atomic_load::hd37078e3d501f11f(dst=0x000
0000000000008, order=Relaxed) + 78 at atomic.rs:1502
  thread #3: tid = 12897, 0x000055555555fb3e
queue_spin`core::sync::atomic::atomic_load::hd37078e3d501f11f(dst=0x000
0000000000008, order=Relaxed) + 78 at atomic.rs:1502, name =
'queue_spin', stop reason = signal SIGSEGV: invalid address (fault
address: 0x8)
    frame #0: 0x000055555555fb3e
queue_spin`core::sync::atomic::atomic_load::hd37078e3d501f11f(dst=0x000
0000000000008, order=Relaxed) + 78 at atomic.rs:1502
  thread #4: tid = 12898, 0x000055555555fb3e
queue_spin`core::sync::atomic::atomic_load::hd37078e3d501f11f(dst=0x000
0000000000008, order=Relaxed) + 78 at atomic.rs:1502, name =
'queue_spin', stop reason = signal SIGSEGV: invalid address (fault
address: 0x8)
    frame #0: 0x000055555555fb3e
queue_spin`core::sync::atomic::atomic_load::hd37078e3d501f11f(dst=0x000
0000000000008, order=Relaxed) + 78 at atomic.rs:1502
  thread #5: tid = 12899, 0x000055555555fb3e
queue_spin`core::sync::atomic::atomic_load::hd37078e3d501f11f(dst=0x000
0000000000008, order=Relaxed) + 78 at atomic.rs:1502, name =
'queue_spin', stop reason = signal SIGSEGV: invalid address (fault
address: 0x8)
    frame #0: 0x000055555555fb3e
```

```
queue_spin`core::sync::atomic::atomic_load::hd37078e3d501f11f(dst=0x000
0000000000008, order=Relaxed) + 78 at atomic.rs:1502
```

In the first example, the head is pointing to null, which will happen when the queue is empty, but this particular branch is only hit when the queue is not empty. What's going on here? It turns out there's a nasty race and its down to deallocation. Let's look at `deq` again, this time with line numbers:

```
64    pub unsafe fn deq(&mut self) -> Option<T> {
65        let mut head: *mut Node<T>;
66        let value: T;
67        loop {
68            head = self.head.load(Ordering::Acquire);
```

```
69            let tail: *mut Node<T> =
              self.tail.load(Ordering::Relaxed);
70            let next: *mut Node<T> =
              (*head).next.load(Ordering::Relaxed);
71            if head == self.head.load(Ordering::Relaxed) {
72                if head == tail {
73                    if next.is_null() {
74                        return None;
75                    }
76                    self.tail.compare_and_swap(tail, next,
                      Ordering::Relaxed);
77                } else {
78                    let val: *mut T = (*next).value as *mut T;
79                    if self.head.compare_and_swap(head, next,
                      Ordering::Release) == head {
80                        value = *Box::from_raw(val);
81                        break;
82                    }
83                }
84            }
85        }
86        let head: Node<T> = *Box::from_raw(head);
87        drop(head);
88        Some(value)
89    }
```

Say we have four threads, A through D. Thread A gets to line 78 and is stopped. Thread A is now in possession of a `head`, a `tail`, and a `next`, which point to a sensible linked-list in memory. Now, threads B, C, and D each perform multiple `enq` and `deq` operations such that when A wakes up the linked list pointed to by the head of A is long gone. In fact, head itself is actually deallocated, but A gets lucky and the OS hasn't overwritten its memory yet.

Thread A wakes up and performs line 78 but next now points off to nonsense and the whole thing crashes. Alternatively, say we have two threads, A and B. Thread A wakes up and advances through to line 70 and is stopped. Thread B wakes up and is very fortunate and advances all the way through to line 88, deallocating head. The OS is feeling its oats and overwrites the memory that A is pointing at. Thread A then wakes up, fires `(*head).next.load(Ordering::Relaxed)`, and subsequently crashes. There are many alternatives here. What's common between

them all is deallocation happening while there's still outstanding references to one or more nodes. In fact, Michael and Scott's paper does mention this as a problem, but briefly in a way that's easy to overlook:

"To obtain consistent values of various pointers we rely on sequences of reads that re-check earlier values to be sure they haven't changed. These sequences of reads are similar to, but simpler than, the snapshots of Prakash et al. (we need to check only one shared variable rather than two). A similar technique can be used to prevent the race condition in Stone's blocking algorithm. We use Treiber's simple and efficient non-blocking stack algorithm to implement a non-blocking free list."

The key ideas here are *sequences of reads that re-check earlier values* and *free list*. What we've seen by inspection is that it's entirely possible to compare-and-swap a value that has changed—the ABA problem—which leaves our implementation pointing off into space. Also, immediately deallocating nodes will leave us open to crashes even absent a compare-and-swap. What Michael and Scott have done is create a minimal kind of memory management; rather than delete nodes, they move them into a *free list* to be reused or deleted at a later time. Free lists can be thread-local, in which case you avoid expensive synchronization, but it's still kind of tricky to be sure your thread-local free list doesn't have the same pointer as another thread.

Options to correct the incorrect queue

What should we do? Well, it turns out there's lots of options in the literature. Hart, McKenney, Brown and Walpole's 2007 *Performance of Memory Reclamation for Lockless Synchronization* discusses four, along with references to their originating papers:

- **Quiescent-state-based reclamation** (**QSRB**): The application's signal quiescent'periods in which reclamation is allowed
- **Epoch-based reclamation** (**EBR**): The applications make use of structures that determine when reclamation is allowed by moving memory through epochs
- **Hazard-pointer-based reclamation** (**HPBR**): Threads cooperate and signal pointers as being hazardous for reclamation
- **Lock-free reference counting (LFRC**): Pointers carry an atomic counter of uses alongside them, which threads use to determine safe reclamation periods

Quiescent state reclamation works by having threads that operate on a shared concurrent structure signal that have entered a *quiescent state* for a period of time, meaning that for the signaled state the thread does not claim any stake over a reference from inside the shared structure. A separate reclamation actor, which may be the structure itself or a thread in a quiescent state, searched for *grace periods* in

which it's safe to reclaim storage, either because all threads have gone quiescent or because some relevant subsets have. It's quite tricky and involves peppering your code with notifications of entering and exiting quiescent states. Hart et al note that this reclamation method was commonly used in the Linux kernel at the time of writing, which remains true at the time of *this* writing. Interested readers are referred to the available literature on the Linux kernel's read-copy update mechanism. The key difficulty of QSRB as a technique is that it's not always clear in user-space when a natural quiescent period exists; if you signal one incorrectly you're going to get data-races. Context switches in an operating system are a natural quiescent period—the relevant contexts won't be manipulating anything until they get called back in—but it's less clear to me in our queue where such a period would comfortably exist. The benefit of QSRB is that it's a very *fast* technique, requiring less thread synchronization than any of the other methods below.

Epoch-based reclamation is similar to QSRB, except that it's application independent. That is, the application interacts with storage through an intermediary layer and this layer stores enough information to decide when a grace period has come. In particular, every thread ticks a flag, indicating that it means to interact with shared data, interacts, and then ticks the flag the other direction on its way out. The number of flag cycles is referred to as an 'epoch'. Once shared data has gone through enough epochs, the thread will attempt to reclaim the storage. This method has the benefit of being more automatic than QSRB—the application developer uses specially designed intermediary types—but has the downside of requiring some synchronization on either side of the data access—those flag ticks. (The paper actually introduces a fifth, new epoch-based reclamation, which improves on epoch-based replication but does require hints from the application developer. In some sense, it is an EBR/QSRB hybrid approach.)

The hazard pointer reclamation scheme is entirely data structure dependent. The idea is that every involved thread will keep a collection of hazard pointers, one for each lockless operation it intended to perform. This thread-local collection of hazard pointers is mirrored into some reclamation system—perhaps a special-purpose garbage collector or plug-in allocator—and this system is able to check the status of the hazard pointers. When a thread is intended to perform a lockless operation on some shared data, it signals the hazard pointer as protected, disallowing its reclamation. When the thread is done, it signals that the shared data is no longer protected. Depending on context, the shared data may now be free for reclamation or may be subject to later reclamation. Like EBR, HPBR requires access to shared data through special structures and has overheads in terms of synchronization. Also, there maybe be a higher burden of implementation compared to EBR, but less than QSRB. Harder yet, if your data structure passes references to its internal storage, that also requires a hazard pointer. The more threads, the more hazard pointer lists you require. Memory overheads can get non-trivial real quick. Compared to grace period approaches, hazard pointers can be much faster at reclaiming memory, due to there being no need to search for a time when a reference is not hazardous. It is or it isn't. HPBR has higher overheads per operation and higher memory requirements overall, but is potentially better at reclamation. That's a non-trivial detail. If QSBR or EBR are run on systems where allocation blocks the CPU, it's possible for these methods to *never* find a grace period, eventually exhausting system memory.

The last of the bunch is atomic reference counting, akin to standard library's `Arc` but for atomics. This method is not especially fast, incurring a cost per access to the reference counter as well as the branch on that counter, but it is simple and reclamation occurs promptly.

Aside from reference counting, each of these methods are tricky to implement. But we're in luck; the crate ecosystem has an

implementation of epoch-based reclamation *and* hazard pointer reclamation. The relevant libraries are crossbeam (`https://crates.io/crates/crossbeam`) and conc (`https://crates.io/crates/conc`). We'll discuss them in great detail in the next chapter. Crossbeam aims to be the `libcds` (`https://github.com/khizmax/libcds`) of Rust and has a Michael and Scott queue already implemented for us. Let's give it a spin. The relevant file is `src/bin/crossbeam_queue_spin.rs`:

```
extern crate crossbeam;

use crossbeam::sync::MsQueue;
use std::sync::Arc;
use std::thread;

fn main() {
    let q = Arc::new(MsQueue::new());

    let mut jhs = Vec::new();

    for _ in 0..4 {
        let q = Arc::clone(&q);
        jhs.push(thread::spawn(move || {
            let mut i = 0;
            loop {
                q.push(i);
                i += 1;
                q.pop();
            }
        }))
    }

    for jh in jhs {
        jh.join().unwrap();
    }
}
```

This is fairly similar to `queue_spin`, save that `enq` is now push and `deq` is `pop`. It's worth noting as well that `MsQueue::pop` is blocking. The mechanism for that is very neat; we'll discuss that in the next chapter.

Semaphore

We discussed semaphores in passing in `Chapter 5`, *Locks – Mutex, Condvar, Barriers, and RWLock*, especially with regard to the concurrency puzzles from *The Little Semaphore*. It's now, as promised, time to implement a semaphore. What *exactly* is a semaphore? Similar to our analysis of mutex as an *atomic object*, let's consider it:

- Semaphore supports two operations, `wait` and `signal`.
- A semaphore has an isize `value` that is used to track the available resource capacity of the semaphore. This value is only manipulable by `wait` and `signal`.
- The operation `wait` decrements `value`. If the value is less than zero, 'wait' blocks the calling thread until such time as a value becomes available. If the value is not less than zero, the thread does not block.
- The operation `signal` increments `value`. After increment, if the value is less than or equal to zero then there are one or more waiting threads. One of these threads is woken up. If the value is greater than zero after increment, there are no waiting threads.
- All loads and stores that occur after and before a wait in program order must not be moved prior to or after the wait.

- All loads and stores that occur before an signal in program order must not be moved to after the signal.

If this seems familiar, there's a good reason for that. Viewed a certain way, a mutex is a semaphore with only one resource available; a locking mutex maps to wait and signaling maps to unwait. We have specified the behaviour of loads and stores around the waiting and signaling of the semaphore to avoid the same deadlock behaviour identified previously in the mutex analysis.

There are some subtleties not captured in the preceding breakdown that don't affect the analysis of the semaphore but do affect the programming model. We'll be building a semaphore with fixed resources. That is, when the semaphore is created, the programmer is responsible for setting the maximum total resources available in the semaphore *and* ensuring that the semaphore starts with these resources available. Some semaphore implementations allow for the resource capacity to shift over time. These are commonly called *counting semaphores*. Our variant is called a *bounded semaphore;* the subvariant of this sort with only a single resource is called a *binary semaphore*. Behavior around the signaling of waiting threads may vary. We will signal our threads on a first-come, first-serve basis.

Let's dig in. Our semaphore implementation is in `src/semaphore.rs` and it's very short:

```
use crossbeam::sync::MsQueue;

unsafe impl Send for Semaphore {}
unsafe impl Sync for Semaphore {}

pub struct Semaphore {
    capacity: MsQueue<()>,
}

impl Semaphore {
```

```
    pub fn new(capacity: usize) -> Self {
        let q = MsQueue::new();
        for _ in 0..capacity {
            q.push(());
        }
        Semaphore { capacity: q }
    }

    pub fn wait(&self) -> () {
        self.capacity.pop();
    }

    pub fn signal(&self) -> () {
        self.capacity.push(());
    }
}
```

Well! Crossbeam's `MsQueue` has the correct ordering semantics when `MsQueue::push` and then `MsQueue::pop` are done in that sequence by the same thread, where pop has the added bonus of blocking until such a time where the queue is empty. So, our semaphore is an `MsQueue` filled with the capacity total `()`. The operation `wait` decreases the `value`—the total number of `()` in the queue—and does so with `Acquire`/`Release` ordering. The operation signal increases the *value* of the semaphore by pushing an additional `()` onto the queue with `Release` semantics. It is possible for a programming error to result in `wait`/`signal` invocations that are not one-to-one, and we can resolve this with the same `Guard` approach taken by Mutex and `SwapMutex`. The underlying queue linearizes `Guard`—see the next chapter for that discussion—and so our semaphore does so also. Let's try this thing out. We've got a program in-project to demonstrate the use of `Semaphore`, `src/bin/status_demo.rs`:

```
extern crate synchro;

use std::sync::Arc;
use synchro::Semaphore;
use std::{thread, time};

const THRS: usize = 4;
static mut COUNTS: &'static mut [u64] = &mut [0; THRS];
static mut STATUS: &'static mut [bool] = &mut [false; THRS];
```

```
fn worker(id: usize, gate: Arc<Semaphore>) -> () {
    unsafe {
        loop {
            gate.wait();
            STATUS[id] = true;
            COUNTS[id] += 1;
            STATUS[id] = false;
            gate.signal();
        }
    }
}

fn main() {
    let semaphore = Arc::new(Semaphore::new(1));

    for i in 0..THRS {
        let semaphore = Arc::clone(&semaphore);
        thread::spawn(move || worker(i, semaphore));
    }

    let mut counts: [u64; THRS] = [0; THRS];
    loop {
        unsafe {
            thread::sleep(time::Duration::from_millis(1_000));
            print!("|");
            for i in 0..THRS {
                print!(" {:>5}; {:010}/sec |", STATUS[i],
                       COUNTS[i] - counts[i]);
                counts[i] = COUNTS[i];
            }
            println!();
        }
    }
}
```

We make `THRS` total worker threads, whose responsibilities are to wait on the semaphore, flip their `STATUS` to true, add one to their `COUNT`, and flip their `STATUS` to false before signaling the semaphore. Mutable static arrays is kind of a goofy setup for any program, but it's a neat trick and causes no harm here, except for interacting oddly with the optimizer. If you compile this program under release mode, you may find that the optimizer determines worker to be a no-op. The `main` function creates a semaphore with a capacity of two, carefully offsets

the workers, and then spins, forever printing out the contents of STATUS and COUNT. A run on my x86 test article looks like the following:

```
> ./target/release/status_demo
| false; 0000170580/sec |  true; 0000170889/sec |  true; 0000169847/sec
| false; 0000169220/sec |
| false; 0000170262/sec | false; 0000170560/sec |  true; 0000169077/sec
|  true; 0000169699/sec |
| false; 0000169109/sec | false; 0000169333/sec | false; 0000168422/sec
| false; 0000168790/sec |
```

```
| false; 0000170266/sec |  true; 0000170653/sec | false; 0000168184/sec
|  true; 0000169570/sec |
| false; 0000170907/sec | false; 0000171324/sec |  true; 0000170137/sec
| false; 0000169227/sec |
...
```

And from my ARMv7 machine:

```
> ./target/release/status_demo
| false; 0000068840/sec |  true; 0000063798/sec | false; 0000063918/sec
| false; 0000063652/sec |
| false; 0000074723/sec | false; 0000074253/sec |  true; 0000074392/sec
|  true; 0000074485/sec |
|  true; 0000075138/sec | false; 0000074842/sec | false; 0000074791/sec
|  true; 0000074698/sec |
| false; 0000075099/sec | false; 0000074117/sec | false; 0000074648/sec
| false; 0000075083/sec |
| false; 0000075070/sec |  true; 0000074509/sec | false; 0000076196/sec
|  true; 0000074577/sec |
|  true; 0000075257/sec |  true; 0000075682/sec | false; 0000074870/sec
| false; 0000075887/sec |
...
```

Binary semaphore, or, a less wasteful mutex

How does our semaphore stack up against standard library's Mutex? How about our spinlock `Mutex`? Apply this `diff` to `status_demo`:

```
diff --git a/external_projects/synchro/src/bin/status_demo.rs
b/external_projects/synchro/src/bin/status_demo.rs
index cb3e850..fef2955 100644
--- a/external_projects/synchro/src/bin/status_demo.rs
+++ b/external_projects/synchro/src/bin/status_demo.rs
@@ -21,7 +21,7 @@ fn worker(id: usize, gate: Arc<Semaphore>) -> () {
 }

 fn main() {
-    let semaphore = Arc::new(Semaphore::new(2));
+    let semaphore = Arc::new(Semaphore::new(1));

     for i in 0..THRS {
         let semaphore = Arc::clone(&semaphore);
```

You'll create a mutex variant of the semaphore status demo. The numbers for that on my x86 test article look like this:

```
| false; 0000090992/sec |  true; 0000090993/sec |  true; 0000091000/sec
| false; 0000090993/sec |
| false; 0000090469/sec | false; 0000090468/sec |  true; 0000090467/sec
|  true; 0000090469/sec |
|  true; 0000090093/sec | false; 0000090093/sec | false; 0000090095/sec
| false; 0000090093/sec |
```

The numbers at capacity two were 170,000 per second, so there's quite a dip. Let's compare that against standard library mutex first. The adaptation is in `src/bin/mutex_status_demo.rs`:

```
use std::sync::{Arc, Mutex};
use std::{thread, time};

const THRS: usize = 4;
static mut COUNTS: &'static mut [u64] = &mut [0; THRS];
static mut STATUS: &'static mut [bool] = &mut [false; THRS];

fn worker(id: usize, gate: Arc<Mutex<()>>) -> () {
    unsafe {
        loop {
            let guard = gate.lock().unwrap();
            STATUS[id] = true;
            COUNTS[id] += 1;
            STATUS[id] = false;
            drop(guard);
        }
    }
}

fn main() {
    let mutex = Arc::new(Mutex::new(()));

    for i in 0..THRS {
        let mutex = Arc::clone(&mutex);
        thread::spawn(move || worker(i, mutex));
    }

    let mut counts: [u64; THRS] = [0; THRS];
    loop {
        unsafe {
            thread::sleep(time::Duration::from_millis(1_000));
            print!("|");
            for i in 0..THRS {
                print!(" {:>5}; {:010}/sec |", STATUS[i],
                       COUNTS[i] - counts[i]);
                counts[i] = COUNTS[i];
            }
            println!();
        }
    }
}
```

Straightforward, yeah? The numbers from that on the same x86 test article are as follows:

```
| false; 0001856267/sec | false; 0002109238/sec |  true; 0002036852/sec
|  true; 0002172337/sec |
| false; 0001887803/sec |  true; 0002072647/sec | false; 0002065467/sec
|  true; 0002143558/sec |
| false; 0001848387/sec | false; 0002044828/sec |  true; 0002098595/sec
|  true; 0002178304/sec |
```

Let's be a little generous and say that the standard library mutex is clocking in at 2,000,000 per second, while our binary semaphore is getting 170,000 per second. How about the spinlock? At this point, I will avoid the source listing. Just be sure to import `SwapMutex` and adjust it from the standard library `Mutex` accordingly. The numbers for that are as follows:

```
|  true; 0012527450/sec | false; 0011959925/sec |  true; 0011863078/sec
| false; 0012509126/sec |
|  true; 0012573119/sec | false; 0011949160/sec |  true; 0011894659/sec
| false; 0012495174/sec |
| false; 0012481696/sec |  true; 0011952472/sec |  true; 0011856956/sec
| false; 0012595455/sec |
```

Which, you know what, makes sense. The spinlock is doing the least amount of work and every thread is burning up CPU time to be right there when it's time to enter their critical section. Here is a summary of our results:

- Spinlock mutex: 11,000,000/sec
- Standard library mutex: 2,000,000/sec
- Binary semaphore: 170,000/sec

The reader is warmly encouraged to investigate each of these implementations in Linux perf. The key thing to understand, from these results, is not that any of these implementations is better or

worse than the other. Rather, that they are suitable for different purposes.

We could, for instance, use techniques from the next chapter to reduce the CPU consumption of the spinlock mutex. This would slow it down some, likely bringing it within the range of standard library Mutex. In this case, use the standard library Mutex.

Atomics programming is not a thing to take lightly. It's difficult to get right, and an implementation that may *seem* right might well not obey the causal properties of the memory model. Furthermore, just because a thing is lock-free does not mean that it is *faster* to execute.

Writing software is about recognizing and adapting to trade-offs. Atomics demonstrates that in abundance.

Summary

In this chapter, we discussed the atomic primitives available on modern CPU architectures and their reflection in Rust. We built higher-level synchronization primitives of the sort discussed in `Chapter 4`, *Sync and Send – the Foundation of Rust Concurrency*, and `Chapter 5`, *Locks – Mutex, Condvar, Barriers, and RWLock*, as well as our own semaphore, which does not exist in Rust. The semaphore implementation could be improved, depending on your needs, and I warmly encourage the readers to give that a shot. We also ran into a common problem of atomic programming, memory reclamation, which we discussed in terms of a Michael Scott queue. We'll discuss approaches to this problem in-depth in the next chapter.

Further reading

- *Axioms for Concurrent Objects*, Maurice Herlihy and Jeannette Wing. This paper by Herlihy and Wing introduced the formal definition of linearizability and the formal analysis framework. Subsequent work has expanded or simplified on this paper, but the reader is warmly encouraged to digest the first half of the paper, at least.
- *Distributed Algorithms*, Nancy Lynch. To my knowledge, Lynch's work was the first general overview of distributed algorithms in textbook form. It is incredibly accessible and Chapter 13 of Lynch's book is especially relevant to this discussion.
- *In C++, are acquire-release memory order semantics transitive?*, available at `https://softwareengineering.stackexchange.com/questions/253537/in-c-are-acquire-release-memory-order-semantics-transitive`. The consequences of memory orderings are not always clear. This question on `StackExchange`, which influenced the writing of this chapter, is about the consequence of splitting `Acquire` and `Release` across threads. The reader is encouraged to ponder the question before reading the answer.
- *std::memory_order, CppReference*, available at `http://en.cppreference.com/w/cpp/atomic/memory_order`. Rust's memory model is LLVM's, which is, in turn, influenced by the C++ memory

model. This discussion of memory ordering is particularly excellent, as it has example code and a less language-lawyer approach to explaining orders.

- *Peterson's lock with C++0x atomics*, available at `https://www.justsoftwaresolutions.co.uk/threading/petersons_lock_with_C++0x_atomics.html`. This post discusses Bartosz Milewski's implementation of Peterson's algorithm for Mutex, demonstrates how and why it is incorrect, and describes a functional alternative. Milewski is a well-known C++ expert. It just goes to show, atomic programming is difficult and easy to get subtly wrong.
- *Algorithms for Mutual Exclusion*, Michael Raynal. This book by Raynal was written in 1986, well before the x86_64 architecture we've discussed was introduced and a mere year after ARMv1 was introduced. Raynal's book remains useful, both as a historical overview of mutual exclusion algorithms and for environments where synchronization primitives are not available, such as on file systems.
- *Review of many Mutex implementations*, available at `http://cbloomrants.blogspot.com/2011/07/07-15-11-review-of-many-mutex.html`. As it says on the tin, this post is a review of many mutex implementations that are given a more modern context than Raynal's book. Some explanations rely on Microsoft Windows features, which may be welcome to the reader as this book is heavily invested in a Unix-like environment.
- *Encheapening Cernan Metrics*, available at `http://blog.troutwine.us/2017/08/31/encheapening-cernan-internal-metrics/`. In this

chapter we have discussed atomics largely in terms of synchronization. There are many other use cases. This post discusses the application of atomics to providing cheap self-telemetry to a complicated software project.

- *Roll Your Own Lightweight Mutex*, available at `http://preshing.com/20120226/roll-your-own-lightweight-mutex/`. Preshing On Programming has a run of excellent atomics material, focused on C++ and Microsoft environments. This particular post is to do with implementing mutexes—a topic dear to this chapter—and has an excellent follow-up conversation in the comments. The post discusses a variant of semaphores called benaphores.
- *Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms*, Maged Michael and Michael Scott. This paper introduces the Michael and Scott Queue, discussed in this chapter, and that's what it's best known for. You may also recognize their queue with two locks from the previous chapter, adapted through the Erlang VM, of course.
- *Lock-Free Data Structures. The Evolution of a Stack*, available at `https://kukuruku.co/post/lock-free-data-structures-the-evolution-of-a-stack/`. This post discusses the `libcds` implementation of Trieber stacks, as well as making reference to other areas of the literature. Readers will note that previous *Further reading* have introduced some of the same literature; it's always good to seek alternate takes. Readers are especially encouraged to investigate the Hendler, Shavit, and Yerushalmi paper referenced in the post.

Atomics – Safely Reclaiming Memory

In the previous chapter, we discussed the atomic primitives available to the Rust programmer, implementing higher-level synchronization primitives and some data structures built entirely of atomics. A key challenge with atomic-only programming, compared to using higher-level synchronization primitives, is memory reclamation. It is only safe to free memory once. When we build concurrent algorithms only from atomic primitives, it's very challenging to do something only once and keep performance up. That is, safely reclaiming memory requires some form of synchronization. But, as the total number of concurrent actors rise, the cost of synchronization dwarfs the latency or throughput benefits of atomic programming.

In this chapter, we will discuss three techniques to resolve the memory reclamation issue of atomic programming—reference counting, hazard pointers, and epoch-based reclamation. These methods will be familiar to you from previous chapters, but here we will do a deep-dive on them, investigating their trade-offs. We will introduce two new libraries, `conc` and crossbeam, which implement hazard pointers and epoch-based reclamation, respectively. By the close of this chapter, you should have a good handle on the three approaches presented and be in a good place to start laying down production-quality code.

By the end of this chapter, we will have:

- Discussed reference counting and its associated tradeoffs

- Discussed the hazard pointer approach to memory reclamation
- Investigated a Treiber stack using the hazard pointer approach via the `conc` crate
- Discussed the epoch-based reclamation strategy
- Done a deep investigation of the `crossbeam_epoch` crate
- Investigated a Treiber stack using the epoch-based reclamation approach

Technical requirements

This chapter requires a working Rust installation. The details of verifying your installation are covered in `Chapter 1`, *Preliminaries – Machine Architecture and Getting Started with Rust*. No additional software tools are required.

You can find the source code for this book's projects on GitHub: `https://github.com/PacktPublishing/Hands-On-Concurrency-with-Rust`. The source code for this chapter is under `Chapter07`.

Approaches to memory reclamation

In the discussion that follows, the three techniques are laid out roughly in order of their speed, slowest to fastest, as well as their difficulty, easiest to hardest. You should not be discouraged—you are now at one of the forefronts of the software engineering world. Welcome. Go boldly.

Reference counting

Reference-counting memory reclamation associates a piece of protected data with an atomic counter. Every thread reading or writing to that protected data increases the counter on acquisition and decreases the counter on de-acquisition. The thread to decrease the counter and find it as zero is the last to hold the data and may either mark the data as available for reclamation or deallocate it immediately. This should, hopefully, sound familiar. The Rust standard library ships with `std::sync::Arc`—discussed in `Chapter 4`, *Sync and Send – the Foundation of Rust Concurrency*, and `Chapter 5`, *Locks – Mutex, Condvar, Barriers and RWLock*, especially—to fulfill this exact need. While we discussed the usage of `Arc`, we did not discuss its internals previously, owing to our need to reach `Chapter 6`, *Atomics – the Primitives of Synchronization*, before we had the necessary background. Let's dig into `Arc` now.

The Rust compiler's code for `Arc` is `src/liballoc/arc.rs`. The structure definition is compact enough:

```
pub struct Arc<T: ?Sized> {
    ptr: Shared<ArcInner<T>>,
    phantom: PhantomData<T>,
}
```

We encountered `Shared<T>` in `Chapter 3`, *The Rust Memory Model – Ownership, References and Manipulation*, with it being a pointer with the same characteristics as `*mut X` except that it is non-null. There are two functions for creating a new `Shared<T>`, which are `const unsafe fn new_unchecked(ptr: *mut X) -> Self` and `fn new(ptr: *mut X) -> Option<Self>`. In the first variant, the caller is responsible for ensuring that the pointer is non-null, in the second, the nulled nature of the pointer is

checked. Please refer back to `Chapter 3`, *The Rust Memory Model – Ownership, References and Manipulation*, *Rc* section for a deep-dive. We know, then, that `ptr` in `Arc<T>` is non-null and, owing to the phantom data field, that the store of `T` is not held in `Arc<T>`. That'd be in `ArcInner<T>`:

```
struct ArcInner<T: ?Sized> {
    strong: atomic::AtomicUsize,

    // the value usize::MAX acts as a sentinel for temporarily
    // "locking" the ability to upgrade weak pointers or
    // downgrade strong ones; this is used to avoid races
    // in `make_mut` and `get_mut`.
    weak: atomic::AtomicUsize,

    data: T,
}
```

We see two inner `AtomicUsize` fields: `strong` and `weak`. Like `Rc`, `Arc<T>` allows two kinds of strength references to the interior data. A weak reference does not own the interior data, allowing for `Arc<T>` to be arranged in a cycle without causing an impossible memory-reclamation situation. A strong reference owns the interior data. Here, we see the creation of a new strong reference via `Arc<T>::clone() -> Arc<T>`:

```
fn clone(&self) -> Arc<T> {
    let old_size = self.inner().strong.fetch_add(1, Relaxed);

    if old_size > MAX_REFCOUNT {
        unsafe {
            abort();
        }
    }

    Arc { ptr: self.ptr, phantom: PhantomData }
}
```

Here, we can see that on every clone, the `strong` of `ArcInner<T>` is increased by one when you use relaxed ordering. The code comment

to this block—dropped for brevity in this text—asserts that relaxed ordering is sufficient owing to the circumstance of the use of `Arc`. Using a relaxed ordering is alright here, as knowledge of the original reference prevents other threads from erroneously deleting the object. This is an important consideration. Why does clone not require a more strict ordering? Consider a thread, `A`, that clones some `Arc<T>` and passes it to another thread, `B`, and then, having passed to `B`, immediately drops its own copy of `Arc`. We can determine from the inspection of `Arc<T>::new() -> Arc<T>` that, at creation time, `Arc` always has one strong and one weak reference:

```
pub fn new(data: T) -> Arc<T> {
    // Start the weak pointer count as 1 which is the weak
    // pointer that's held by all the strong pointers
    // (kinda), see std/rc.rs for more info
    let x: Box<_> = box ArcInner {
        strong: atomic::AtomicUsize::new(1),
        weak: atomic::AtomicUsize::new(1),
        data,
    };
    Arc { ptr: Shared::from(Box::into_unique(x)),
          phantom: PhantomData }
}
```

From the perspective of thread `A`, the cloning of `Arc` is composed of the following operations:

```
clone Arc
    INCR strong, Relaxed
    guard against strong wrap-around
    allocate New Arc
move New Arc into B
drop Arc
```

We know from discussion in the previous chapter, that `Release` ordering does not offer a causality relationship. It's entirely possible that a hostile CPU could reorder the increment of `strong` to after the movement of `New Arc` into `B`. Would we be in trouble if `B` then immediately dropped `Arc`? We need to inspect `Arc<T>::drop()`:

```
fn drop(&mut self) {
    if self.inner().strong.fetch_sub(1, Release) != 1 {
        return;
    }

    atomic::fence(Acquire);

    unsafe {
        let ptr = self.ptr.as_ptr();

        ptr::drop_in_place(&mut self.ptr.as_mut().data);

        if self.inner().weak.fetch_sub(1, Release) == 1 {
            atomic::fence(Acquire);
            Heap.dealloc(ptr as *mut u8, Layout::for_value(&*ptr))
        }
    }
}
```

Here we go. `atomic::fence(Acquire)` is new to us. `std::sync::atomic::fence` prevents memory migration around itself, according to the causality relationship established by the provided memory ordering. The fence applies to same-thread and other-thread memory operations. Recall that `Release` ordering disallows loads and stores from migrating downward in the source-code order. Here, we see, that the load and store of strong will disallow migrations downward but will not be reordered after the `Acquire` fence. Therefore, the deallocation of the `T` interior to `Arc` will not happen until both the `A` and `B` threads have synchronized and removed all strong references. An additional thread, `C`, cannot come through and increase the strong references to the interior `T` while this is ongoing, owing to the causality relationship established—neither `A` nor `B` can give `C` a strong reference without increasing the strong counter, causing the drops of `A` or `B` to bail out early. A similar analysis holds for weak references.

Doing an immutable dereference of `Arc` does not increase the strong or weak counts:

```
    impl<T: ?Sized> Deref for Arc<T> {
        type Target = T;

        #[inline]
        fn deref(&self) -> &T {
            &self.inner().data
        }
    }
```

Because drop requires a mutable self, it is impossible to free `Arc<T>` while there is a valid `&T`. Getting `&mut T` is more involved, which is done via `Arc<T>::get_mut(&mut Self) -> Option<&mut T>`. Note that the return is an `Option`. If there are other strong or weak references to the interior `T`, then it's not safe to consume `Arc`. The implementation of `get_mut` is as follows:

```
    pub fn get_mut(this: &mut Self) -> Option<&mut T> {
        if this.is_unique() {
            // This unsafety is ok because we're guaranteed that the
            // pointer returned is the *only* pointer that will ever
            // be returned to T. Our reference count is guaranteed
            // to be 1 at this point, and we required the Arc itself
            // to be `mut`, so we're returning the only possible
            // reference to the inner data.
            unsafe {
                Some(&mut this.ptr.as_mut().data)
            }
        } else {
            None
        }
    }
```

Where `is_unique` is:

```
    fn is_unique(&mut self) -> bool {
        if self.inner().weak.compare_exchange(1, usize::MAX,
                                              Acquire, Relaxed).is_ok()
{
            let unique = self.inner().strong.load(Relaxed) == 1;
            self.inner().weak.store(1, Release);
            unique
```

```
        } else {
            false
        }
    }
```

The compare and exchange operation on weak ensures that there is only one weak reference outstanding—implying uniqueness—and does so on success with `Acquire` ordering. This ordering will force the subsequent check of the strong references to occur after the check of weak references in the code order and above the release store to the same. This exact technique will be familiar from our discussion on mutual exclusion in the previous chapter.

Tradeoffs

Reference counting has much to recommend it as an approach for atomic memory reclamation. The programming model of reference counting is straightforward to understand and memory is reclaimed as soon as it's possible to be safely reclaimed. Programming reference-counted implementations without the use of `Arc<T>` remains difficult, owing to the lack of double-word compare and swap in Rust. Consider a Treiber stack in which the reference counter is internal to stack nodes. The head of the stack might be snapshotted, then freed by some other thread, and the subsequent read to the reference counter will be forced through an invalid pointer.

The following reference-counted Treiber stack is flawed:

```
use std::sync::atomic::{fence, AtomicPtr, AtomicUsize, Ordering};
use std::{mem, ptr};

unsafe impl<T: Send> Send for Stack<T> {}
unsafe impl<T: Send> Sync for Stack<T> {}

struct Node<T> {
    references: AtomicUsize,
    next: *mut Node<T>,
    data: Option<T>,
}

impl<T> Node<T> {
    fn new(t: T) -> Self {
        Node {
            references: AtomicUsize::new(1),
            next: ptr::null_mut(),
            data: Some(t),
        }
    }
}
```

```
pub struct Stack<T> {
    head: AtomicPtr<Node<T>>,
}

impl<T> Stack<T> {
    pub fn new() -> Self {
        Stack {
            head: AtomicPtr::default(),
        }
    }

    pub fn pop(&self) -> Option<T> {
        loop {
            let head: *mut Node<T> =
self.head.load(Ordering::Relaxed);

            if head.is_null() {
                return None;
            }
            let next: *mut Node<T> = unsafe { (*head).next };

            if self.head.compare_and_swap(head, next,
            Ordering::Relaxed) == head {
                let mut head: Box<Node<T>> = unsafe {
                    Box::from_raw(head)
                };
                let data: Option<T> = mem::replace(&mut
(*head).data,
                                                   None);
                unsafe {
                    assert_eq!(
                        (*(*head).next).references.fetch_sub(1,
                             Ordering::Release),
                        2
                    );
                }
                drop(head);
                return data;
            }
        }
    }

    pub fn push(&self, t: T) -> () {
        let node: *mut Node<T> =
Box::into_raw(Box::new(Node::new(t)));
        loop {
            let head = self.head.load(Ordering::Relaxed);
```

```
                unsafe {
                    (*node).next = head;
                }

                fence(Ordering::Acquire);
                if self.head.compare_and_swap(head, node,
                Ordering::Release) == head {
                    // node is now self.head
                    // head is now self.head.next
                    if !head.is_null() {
                        unsafe {
                            assert_eq!(1,
                                       (*head).references.fetch_add(1,
                                           Ordering::Release)
                            );
                        }
                    }
                    break;
                }
            }
        }
    }
```

This is one of the few (intentionally) flawed examples in this book. You are encouraged to apply the techniques discussed earlier in this book to identify and correct the flaws with this implementation. It should be interesting. J.D. Valois' 1995 paper, *Lock-Free Linked Lists Using Compare-and-Swap*, will be of use. You are further warmly encouraged to attempt a Treiber stack using only `Arc`.

Ultimately, where reference counting struggles is contention, as the number of threads operating on the same reference-counted data increase., those acquire/release pairs don't come cheaply. Reference counting is a good model when you expect the number of threads to be relatively low or where absolute performance is not a key consideration. Otherwise, you'll need to investigate the following methods, which have tradeoffs of their own.

Hazard pointers

We touched on hazard pointers as a method for safe memory reclamation in the previous chapter. We'll now examine the concept in-depth and in the context of a more or less ready-to-use implementation via the Redox project (`https://crates.io/crates/conc`). We'll be inspecting `conc` as a part of its parent project, tfs (`https://github.com/redox-os/tfs`), at SHA `3e7dcdb0c586d0d8bb3f25bfd948d2f418a4ab10`. Incidentally, if you're unfamiliar with this, Redox is a Rust microkernel-based operating system. The allocator, coreutils, and netutils are all encouraged reading.

> *The conc crate is not listed in its entirety. You can find the full listing in this book's source repository.*

Hazard pointers were introduced by Maged Michael—of Michael and Scott Queue fame—in his 2004 book, *Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects*. A *hazard*, in this context, is any pointer that is subject to races or ABA issues in the rendezvous of multiple treads participating in some shared data structure. In the previous section's Treiber stack, the hazard is the head pointer inside of the Stack struct, and in the Michael and Scott queue, it is the head and tail pointers. A hazard pointer associates these hazards with *single-writer multireader shared pointers*, owned by a single writer thread and read by any other threads participating in the data structure.

Every participating thread maintains a reference to its own hazard pointers and all other participating threads in addition to private, thread-local lists for the coordination of deallocation. The algorithm description in section 3 of Michael's paper is done at a high level and is difficult to follow with an eye towards producing a concrete

implementation. Rather than repeat that here, we will instead examine a specific implementation, `conc`. The reader is encouraged to read Michael's paper ahead of the discussion of `conc`, but this is not mandatory.

A hazard-pointer Treiber stack

The introduction of the Treiber stack in the previous section on reference counting was not an idle introduction. We'll examine hazard pointers and epoch-based reclamation here, through the lens of an effectively reclaimed Treiber stack. It so happens that `conc` ships with a Treiber stack, in `tfs/conc/src/sync/treiber.rs`. The preamble is mostly familiar:

```
use std::sync::atomic::{self, AtomicPtr};
use std::marker::PhantomData;
use std::ptr;
use {Guard, add_garbage_box};
```

Both `Guard` and `add_garbage_box` are new, but we'll get to them directly. The `Treiber` struct is as you might have imaged it:

```
pub struct Treiber<T> {
    /// The head node.
    head: AtomicPtr<Node<T>>,
    /// Make the `Sync` and `Send` (and other OIBITs) transitive.
    _marker: PhantomData<T>,
}
```

As is the node:

```
struct Node<T> {
    /// The data this node holds.
    item: T,
    /// The next node.
    next: *mut Node<T>,
}
```

This is where we need to understand `conc::Guard`. Much like `MutexGuard` in the standard library, `Guard` exists here to protect the contained data from being multiply mutated. `Guard` is the hazard pointer interface for `conc`. Let's examine `Guard` in detail and get to the hazard pointer algorithm. `Guard` is defined in `tfs/conc/src/guard.rs`:

```
pub struct Guard<T: 'static + ?Sized> {
    /// The inner hazard.
    hazard: hazard::Writer,
    /// The pointer to the protected object.
    pointer: &'static T,
}
```

As of writing this book, all `T` protected by `Guard` have to be static, but we'll ignore that as the codebase has references to a desire to relax that restriction. Just be aware of it should you wish to make immediate use of `conc` in your project. Like `MutexGuard`, `Guard` does not own the underlying data but merely protects a reference to it, and we can see that our `Guard` is the writer-end of the hazard. What is `hazard::Writer`? It's defined in `tfs/conc/src/hazard.rs`:

```
pub struct Writer {
    /// The pointer to the heap-allocated hazard.
    ptr: &'static AtomicPtr<u8>,
}
```

Pointer to the heap-allocated hazard? Okay, let's back out a bit. We know from the algorithm description that there has to be thread-local storage happening in coordination with, apparently, heap storage. We also know that `Guard` is our primary interface to the hazard pointers. There are three functions to create new instances of `Guard`:

- `Guard<T>::new<F: FnOnce() -> &'static T>(ptr: F) -> Guard<T>`

- `Guard<T>::maybe_new<F: FnOnce() -> Option<&'static T>>(ptr: F) -> Option<Guard<T>>`
- `Guard<T>::try_new<F: FnOnce() -> Result<&'static T, E>>(ptr: F) -> Result<Guard<T>, E>`

The first two are defined in terms of `try_new`. Let's dissect `try_new`:

```
pub fn try_new<F, E>(ptr: F) -> Result<Guard<T>, E>
where F: FnOnce() -> Result<&'static T, E> {
    // Increment the number of guards currently being created.
    #[cfg(debug_assertions)]
    CURRENT_CREATING.with(|x| x.set(x.get() + 1));
```

The function takes `FnOnce` whose responsibility is to kick out a reference to `T` or fail. The first call in `try_new` is an increment of `CURRENT_CREATING`, which is `thread-local Cell<usize>`:

```
thread_local! {
    /// Number of guards the current thread is creating.
    static CURRENT_CREATING: Cell<usize> = Cell::new(0);
}
```

We've seen `Cell` in `Chapter 3`, *The Rust Memory Model – Ownership, References and Manipulation*, but `thread_local!` is new. This macro wraps one or more static values into `std::thread::LocalKey`, Rust's take on thread-local stores. The exact implementation varies from platform to platform but the basic idea holds: the value will act as a static global but will be confined to a single thread. In this way, we can program as if the value were global but without having to manage coordination between threads. Once `CURRENT_CREATING` is incremented:

```
    // Get a hazard in blocked state.
    let hazard = local::get_hazard();
```

The `local::get_hazard` function is defined in `tfs/conc/src/local.rs`:

```
pub fn get_hazard() -> hazard::Writer {
    if STATE.state() == thread::LocalKeyState::Destroyed {
        // The state was deinitialized, so we must rely on the
        // global state for creating new hazards.
        global::create_hazard()
    } else {
        STATE.with(|s| s.borrow_mut().get_hazard())
    }
}
```

The `STATE` referenced in this function is:

```
thread_local! {
    /// The state of this thread.
    static STATE: RefCell<State> = RefCell::new(State::default());
}
```

The `State` is type defined like so:

```
#[derive(Default)]
struct State {
    garbage: Vec<Garbage>,
    available_hazards: Vec<hazard::Writer>,
    available_hazards_free_before: usize,
}
```

The field garbage maintains a list of `Garbage` that has yet to be moved to the global state—more on that in a minute—for reclamation. `Garbage` is a pointer to bytes and a function pointer to bytes called `dtor`, for destructor. Memory reclamation schemes must be able to deallocate, regardless of the underlying type. The common approach, and the approach taken by `Garbage`, is to build a monomorphized destructor function when the type information is available, but otherwise work on byte buffers. You are encouraged to thumb through the implementation of `Garbage` yourself, but the primary trick

is `Box::from_raw(ptr as *mut u8 as *mut T)`, which we've seen repeatedly throughout this book.

The `available_hazards` field stores the previously allocated hazard writers that aren't currently being used. The implementation keeps this as a cache to avoid allocator thrash. We can see this in action in `local::State::get_hazard`:

```
fn get_hazard(&mut self) -> hazard::Writer {
    // Check if there is hazards in the cache.
    if let Some(hazard) = self.available_hazards.pop() {
        // There is; we don't need to create a new hazard.
        //
        // Since the hazard popped from the cache is not
        // blocked, we must block the hazard to satisfy
        // the requirements of this function.
        hazard.block();
        hazard
    } else {
        // There is not; we must create a new hazard.
        global::create_hazard()
    }
}
```

The final field, `available_hazards_free_before`, stores hazards in the freed state, prior to the actual deallocation of the underlying type. We'll discuss this more later. Hazards are in one of four states: free, dead, blocked, or protecting. A dead hazard can be deallocated safely, along with the protected memory. A dead hazard should not be read. A free hazard is protecting nothing and may be reused. A blocked hazard is in use by some other thread can and will cause reads of the hazard to stall. A protecting hazard is, well, protecting some bit of memory. Now, jump back to this branch in `local::get_hazard`:

```
if STATE.state() == thread::LocalKeyState::Destroyed {
    // The state was deinitialized, so we must rely on the
    // global state for creating new hazards.
    global::create_hazard()
} else {
```

What is `global::create_hazard`? This module is `tfs/conc/src/global.rs` and the function is:

```
pub fn create_hazard() -> hazard::Writer {
    STATE.create_hazard()
}
```

The variable names are confusing. This `STATE` is not the thread-local `STATE` but a globally scoped `STATE`:

```
lazy_static! {
    /// The global state.
    ///
    /// This state is shared between all the threads.
    static ref STATE: State = State::new();
}
```

Let's dig in there:

```
struct State {
    /// The message-passing channel.
    chan: mpsc::Sender<Message>,
    /// The garbo part of the state.
    garbo: Mutex<Garbo>,
}
```

The global `STATE` is a mpsc `Sender` of `Message` and a mutex-guarded `Garbo`. `Message` is a simple enumeration:

```
enum Message {
    /// Add new garbage.
    Garbage(Vec<Garbage>),
    /// Add a new hazard.
    NewHazard(hazard::Reader),
}
```

`Garbo` is something we'll get into directly. Suffice it to say for now that `Garbo` acts as the global garbage collector for this implementation. The

global state sets up a channel, maintaining the sender side in the global state and feeding the receiver into `Garbo`:

```
impl State {
    /// Initialize a new state.
    fn new() -> State {
        // Create the message-passing channel.
        let (send, recv) = mpsc::channel();

        // Construct the state from the two halfs of the channel.
        State {
            chan: send,
            garbo: Mutex::new(Garbo {
                chan: recv,
                garbage: Vec::new(),
                hazards: Vec::new(),
            })
        }
    }
```

The creation of a new global hazard doesn't take much:

```
fn create_hazard(&self) -> hazard::Writer {
    // Create the hazard.
    let (writer, reader) = hazard::create();
    // Communicate the new hazard to the global state
    // through the channel.
    self.chan.send(Message::NewHazard(reader));
    // Return the other half of the hazard.
    writer
}
```

This establishes a new hazard via `hazard::create()` and feeds the reader side down through to `Garbo`, returning the writer side back out to `local::get_hazard()`. While the names writer and reader suggest that hazard is itself an MPSC, this is not true. The hazard module is `tfs/conc/src/hazard.rs` and creation is:

```
pub fn create() -> (Writer, Reader) {
    // Allocate the hazard on the heap.
```

```
        let ptr = unsafe {
            &*Box::into_raw(Box::new(AtomicPtr::new(
                            &BLOCKED as *const u8 as *mut u8)))
        };

        // Construct the values.
        (Writer {
            ptr: ptr,
        }, Reader {
            ptr: ptr,
        })
    }
```

Well, look at that. What we have here are two structs, `Writer` and `Reader`, which each store the same raw pointer to a heap-allocated atomic pointer to a mutable byte pointer. Phew! We've seen this trick previously but what's special here is the leverage of the type system to provide for different reading and writing interfaces over the same bit of raw memory.

What about `Garbo`? It's defined in the global module and is defined as:

```
struct Garbo {
    /// The channel of messages.
    chan: mpsc::Receiver<Message>,
    /// The to-be-destroyed garbage.
    garbage: Vec<Garbage>,
    /// The current hazards.
    hazards: Vec<hazard::Reader>,
}
```

`Garbo` defines a `gc` function that reads all `Messages` from its channel, storing the garbage into the garbage field and free hazards into hazards. Dead hazards are destroyed, freeing its storage as the other holder is guaranteed to have hung up already. Protected hazards also make their way into hazards, which are to be scanned during the next call of gc. Garbage collections are sometimes performed when a thread calls `global::tick()` or when `global::try_gc()` is called. A tick is performed whenever `local::add_garbage` is called, which is what `whatconc::add_garbage_box` calls.

We first encountered `add_barbage_box` at the start of this section. Every time a thread signals a node as garbage, it rolls the dice and potentially becomes responsible for performing a global garbage collection over all of the threads' hazard-pointed memory.

Now that we understand how memory reclamation works, all that remains is to understand how hazard pointers protect memory from reads and writes. Let's finish `guard::try_new` in one large jump:

```
        // This fence is necessary for ensuring that `hazard` does not
        // get reordered to after `ptr` has run.
        // TODO: Is this fence even necessary?
        atomic::fence(atomic::Ordering::SeqCst);

        // Right here, any garbage collection is blocked, due to the
        // hazard above. This ensures that between the potential
        // read in `ptr` and it being protected by the hazard, there
        // will be no premature free.

        // Evaluate the pointer through the closure.
        let res = ptr();

        // Decrement the number of guards currently being created.
        #[cfg(debug_assertions)]
        CURRENT_CREATING.with(|x| x.set(x.get() - 1));

        match res {
            Ok(ptr) => {
                // Now that we have the pointer, we can protect it by
                // the hazard, unblocking a pending garbage collection
                // if it exists.
                hazard.protect(ptr as *const T as *const u8);

                Ok(Guard {
                    hazard: hazard,
                    pointer: ptr,
                })
            },
            Err(err) => {
                // Set the hazard to free to ensure that the hazard
                // doesn't remain blocking.
                hazard.free();

                Err(err)
```

```
            }
        }
    }
```

We can see that the conc authors have inserted a sequentially consistent fence that they question. The model laid out by Michael does not require sequential consistency and I believe that this fence is not needed, being a significant drag on performance. The key things here to note are the call to `hazard::protect` and `'hazard::free`. Both calls are part of `hazard::Writer`, the former setting the internal pointer to the byte pointer fed to it, the latter marking the hazard as free. Both states interact with the garbage collector, as we've seen. The remaining bit has to do with `hardard::Reader::get`, the function used to retrieve the state of the hazard. Here it is:

```
impl Reader {
    /// Get the state of the hazard.
    ///
    /// It will spin until the hazard is no longer in a blocked state,
    /// unless it is in debug mode, where it will panic given enough
    /// spins.
    pub fn get(&self) -> State {
        // In debug mode, we count the number of spins. In release
        // mode, this should be trivially optimized out.
        let mut spins = 0;

        // Spin until not blocked.
        loop {
            let ptr = self.ptr.load(atomic::Ordering::Acquire)
                          as *const u8;

            // Blocked means that the hazard is blocked by another
            // thread, and we must loop until it assumes another
            // state.
            if ptr == &BLOCKED {
                // Increment the number of spins.
                spins += 1;
                debug_assert!(spins < 100_000_000, "\
                    Hazard blocked for 100 millions rounds. Panicking
                    as chances are that it will \
                    never get unblocked.\
                ");
```

```
                continue;
            } else if ptr == &FREE {
                return State::Free;
            } else if ptr == &DEAD {
                return State::Dead;
            } else {
                return State::Protect(ptr);
            }
```

Only if the hazard is blocked does the get of the state spin until it's dead, free, or merely protected. What blocks the hazard? Recall that they're created blocked. By creating the hazards in a blocked state, it is not possible to perform garbage collection over a pointer that has not been fully initialized—a problem we saw with the reference-counting implementation—nor is it possible to read from a partially-initialized hazarded pointer. Only once the pointer is moved into the protected state can reads move forward.

And there you go—garbage collection and atomic isolation.

Let's go all the way back up to the stack and look at its push implementation:

```
pub fn push(&self, item: T)
where T: 'static {
    let mut snapshot = Guard::maybe_new(|| unsafe {
        self.head.load(atomic::Ordering::Relaxed).as_ref()
    });

    let mut node = Box::into_raw(Box::new(Node {
        item: item,
        next: ptr::null_mut(),
    }));

    loop {
        let next = snapshot.map_or(ptr::null_mut(),
                                   |x| x.as_ptr() as *mut _);
        unsafe { (*node).next = next; }

        match Guard::maybe_new(|| unsafe {
            self.head.compare_and_swap(next, node,
```

```
                    atomic::Ordering::Release).as_ref()
            }) {
                Some(ref new) if new.as_ptr() == next => break,
                None if next.is_null() => break,
                // If it fails, we will retry the CAS with updated
                // values.
                new => snapshot = new,
            }
        }
    }
```

At the top of the function, the implementation loads the hazard pointer for the head node into the snapshot. `Guard::as_ptr(&self) -> *const T` retrieves the current pointer for the hazard on each invocation, adapting as the underlying data shifts forward. The node is the allocated and raw-pointered `Node` containing `item: T`. The remainder of the loop is the same compare-and-swap we've seen for other data structures of this kind, merely in terms of a hazard `Guard` instead of raw `AtomicPtrs` or the like. The programming model is very direct, as it is for `pop` as well:

```
    pub fn pop(&self) -> Option<Guard<T>> {
        let mut snapshot = Guard::maybe_new(|| unsafe {
            self.head.load(atomic::Ordering::Acquire).as_ref()
        });

        while let Some(old) = snapshot {
            snapshot = Guard::maybe_new(|| unsafe {
                self.head.compare_and_swap(
                    old.as_ptr() as *mut _,
                    old.next as *mut Node<T>,
                    atomic::Ordering::Release,
                ).as_ref()
            });

            if let Some(ref new) = snapshot {
                if new.as_ptr() == old.as_ptr() {
                    unsafe { add_garbage_box(old.as_ptr()); }
                    return Some(old.map(|x| &x.item));
                }
            } else {
                break;
            }
```

```
        }

        None
    }
```

Note that when the old node is removed from the stack, `add_garbage_box` is called on it, adding the node to be garbage-collected at a later date. We know, further, from inspection, that this later date might well be exactly the moment of invocation, depending on luck.

The hazard of Nightly

Unfortunately, conc relies on an unstable feature of the Rust compiler and will not compile with a recent `rustc`. The last update to `conc`, and to TFS, its parent project, was in August of 2017. We'll have to travel back in time for a nightly `rustc` from that period. If you followed the instructions laid out in `Chapter 1`, *Preliminaries – Machine Architecture and Getting Started with Rust*, and are using `rustup`, that's a simple `rustup` default nightly-2017-08-17 away. Don't forget to switch back to a more modern Rust when you're done playing with conc.

Exercizing the hazard-pointer Treiber stack

Let's put the conc, Treiber stack through the wringer to get an idea of the performance of this approach. Key areas of interest for us will be:

- Push/pop cycles per second
- Memory behavior, high-water mark, and what not
- Cache behavior

We'll run our programs on x86 and ARM, as in previous chapters. One thing to note here, before we proceed, is that conc—at least as of version 0.5—requires the nightly channel to be compiled. Please read the section just before this one if you're jumping around in the book for full details.

Because we have to run on an old version of nightly, we have to stick static `AtomicUsize` into `lazy_static!`, a technique you'll see in older Rust code but not in this book, usually. With that in mind, here is our exercise program:

```
extern crate conc;
#[macro_use]
extern crate lazy_static;
extern crate num_cpus;
extern crate quantiles;

use conc::sync::Treiber;
use quantiles::ckms::CKMS;
use std::sync::Arc;
```

```
use std::sync::atomic::{AtomicUsize, Ordering};
use std::{thread, time};

lazy_static! {
    static ref WORKERS: AtomicUsize = AtomicUsize::new(0);
    static ref COUNT: AtomicUsize = AtomicUsize::new(0);
}
static MAX_I: u32 = 67_108_864; // 2 ** 26

fn main() {
    let stk: Arc<Treiber<(u64, u64, u64)>> =
Arc::new(Treiber::new());

    let mut jhs = Vec::new();

    let cpus = num_cpus::get();
    WORKERS.store(cpus, Ordering::Release);

    for _ in 0..cpus {
        let stk = Arc::clone(&stk);
        jhs.push(thread::spawn(move || {
            for i in 0..MAX_I {
                stk.push((i as u64, i as u64, i as u64));
                stk.pop();
                COUNT.fetch_add(1, Ordering::Relaxed);
            }
            WORKERS.fetch_sub(1, Ordering::Relaxed)
        }))
    }

    let one_second = time::Duration::from_millis(1_000);
    let mut iter = 0;
    let mut cycles: CKMS<u32> = CKMS::new(0.001);
    while WORKERS.load(Ordering::Relaxed) != 0 {
        let count = COUNT.swap(0, Ordering::Relaxed);
        cycles.insert((count / cpus) as u32);
        println!(
            "CYCLES PER SECOND({}):\n  25th: \
            {}\n  50th: {}\n  75th: \
            {}\n  90th: {}\n  max:  {}\n",
            iter,
            cycles.query(0.25).unwrap().1,
            cycles.query(0.50).unwrap().1,
            cycles.query(0.75).unwrap().1,
            cycles.query(0.90).unwrap().1,
            cycles.query(1.0).unwrap().1
        );
        thread::sleep(one_second);
```

```
            iter += 1;
        }

        for jh in jhs {
            jh.join().unwrap();
        }
    }
```

We have a number of worker threads equal to the total number of CPUs in the target machine, each doing one push and then an immediate pop on the stack. A `COUNT` is kept and the main thread swaps that value for `0` every second or so, tossing the value into a quantile estimate structure—discussed in more detail in `Chapter 4`, *Sync and Send – the Foundation of Rust Concurrency*—and printing out a summary of the recorded cycles per second, scaled to the number of CPUs. The worker threads the cycle up through to `MAX_I`, which is arbitrarily set to the smallish value of `2**26`. When the worker is finished cycling, it decreases `WORKERS` and exits. Once `WORKERS` hits zero, the main loop also exits.

On my x86 machine, it takes approximately 58 seconds for this program to exit, with this output:

```
CYCLES PER SECOND(0):
  25th: 0
  50th: 0
  75th: 0
  90th: 0
  max:  0

CYCLES PER SECOND(1):
  25th: 0
  50th: 0
  75th: 1124493
  90th: 1124493
  max:  1124493

...

CYCLES PER SECOND(56):
  25th: 1139055
```

```
  50th: 1141656
  75th: 1143781
  90th: 1144324
  max:  1145284

CYCLES PER SECOND(57):
  25th: 1139097
  50th: 1141656
  75th: 1143792
  90th: 1144324
  max:  1145284
```

```
CYCLES PER SECOND(58):
  25th: 1139097
  50th: 1141809
  75th: 1143792
  90th: 1144398
  max:  1152384
```

The x86 perf run is as follows:

```
> perf stat --event task-clock,context-switches,page-
faults,cycles,instructions,branches,branch-misses,cache-
references,cache-misses target/release/conc_stack > /dev/null

 Performance counter stats for 'target/release/conc_stack':

     230503.932161      task-clock (msec)         #    3.906 CPUs
utilized
               943      context-switches          #    0.004 K/sec
             9,140      page-faults               #    0.040 K/sec
   665,734,124,408      cycles                    #    2.888 GHz
   529,230,473,047      instructions              #    0.79  insn per
cycle
    99,146,219,517      branches                  #  430.128 M/sec
     1,140,398,326      branch-misses             #    1.15% of all
branches
    12,759,284,944      cache-references          #   55.354 M/sec
       124,487,844      cache-misses              #    0.976 % of all
cache refs

      59.006842741 seconds time elapsed
```

The memory-allocation behavior of this program is very favorable as well.

On my ARM machine, it takes approximately 460 seconds for the program to run to completion, with this output:

```
CYCLES PER SECOND(0):
  25th: 0
  50th: 0
  75th: 0
  90th: 0
  max:  0
```

```
CYCLES PER SECOND(1):
  25th: 0
  50th: 0
  75th: 150477
  90th: 150477
  max:  150477

...

CYCLES PER SECOND(462):
  25th: 137736
  50th: 150371
  75th: 150928
  90th: 151129
  max:  151381

CYCLES PER SECOND(463):
  25th: 137721
  50th: 150370
  75th: 150928
  90th: 151129
  max:  151381
```

This is about 7.5 times slower than the x86 machine, even though both machines have the same number of cores. The x86 machine has a faster clock and a faster memory bus than my ARM:

```
> perf stat --event task-clock,context-switches,page-
faults,cycles,instructions,branches,branch-misses,cache
```

```
-references,cache-misses target/release/conc_stack > /dev/null

 Performance counter stats for 'target/release/conc_stack':

    1882745.172714      task-clock (msec)         #    3.955 CPUs
utilized
                 0      context-switches          #    0.000 K/sec
             3,212      page-faults               #    0.002 K/sec
 2,109,536,434,019      cycles                    #    1.120 GHz
 1,457,344,087,067      instructions              #    0.69  insn per
cycle
   264,210,403,390      branches                  #  140.333 M/sec
    36,052,283,984      branch-misses             #   13.65% of all
branches
   642,854,259,575      cache-references          #  341.445 M/sec
     2,401,725,425      cache-misses              #    0.374 % of all
cache refs

     476.095973090 seconds time elapsed
```

The fact that the branch misses on ARM are 12% higher than on x86 is an interesting result.

Before moving on, do remember to execute the rustup default stable.

Tradeoffs

Assuming that conc was brought up to speed with modern Rust, where would you want to apply it? Well, let's compare it to reference counting. In a reference-counting data structure, every read and modification requires manipulating atomic values, implying some level of synchronization between threads with an impact on performance. A similar situation exists with hazard pointers, though to a lesser degree. As we've seen, the hazard requires synchronization, but reading anything apart from the hazard, through it, is going to be at machine speed. That's a significant improvement over reference counting. Furthermore, we know that the hazard pointer approach does accumulate garbage—and the implementation of conc is done via an especially neat trick—but this garbage is bounded in size, no matter the frequency of updates to the hazardous structure. The cost of garbage collection is placed on one thread, incurring potentially high latency spikes for some operations while benefiting from bulk-operation calls to the deallocator. This is distinct from reference counting approaches where every thread is responsible for deallocating dead memory as soon as it is dead, keeping operation cost overheads similar but higher than baseline, and incurred on each operation without the benefit of bulk accumulation of deallocations. The hazard pointer approach struggles as the number of threads increase: recall that the number of hazards needed grows linearly with the number of threads, but also consider that an increase in thread count increases the traversal cost of any structure. Furthermore, while many structures have easily identifiable hazards, this is not universally true. Where do you put the hazard pointers for a b-tree?

In summary—hazard pointers are an excellent choice when you have relatively few hazardous memory references to make and need bounded garbage, no matter the load on your structure. Hazard-based memory reclamation does not have the raw speed of epoch-based reclamation, but bounded memory use in all cases is hard to pass up.

Epoch-based reclamation

We touched on epoch-based memory reclamation in the last chapter. We'll now examine the concept in depth and in the context of a ready-to-use implementation via the crossbeam project (`https://github.com/crossbeam-rs`). Some of the developers of conc overlap with crossbeam, but there's been more work done to crossbeam in the pursuit of a general-purpose framework for low-level atomic programming in Rust. The developers note that they plan to support other memory management schemes, for example hazard pointers (HP) and quiescent state-based reclamation (QSBR). The crate is re-exported as the epoch module. Future editions of this book may well only cover crossbeam and simply discuss the various memory reclamation techniques embedded in it. The particular crate we'll be discussing in this section is crossbeam-epoch, at SHA `3179e799f384816a0a9f2a3da82a147850e14d38`. This coincides with the 0.4.1 release of the project.

> *The crossbeam crates under discussion here are not listed in their entirety. You can find the full listings in this book's source repository.*

Epoch-based memory reclamation was introduced by Keir Fraser in his 2004 PhD thesis, *Practical lock-freedom*. Fraser's work builds on previous limbo list approaches, whereby garbage is placed in a global list and reclaimed through periodic scanning when it can be demonstrated that the garbage is no longer referenced. The exact mechanism varies somewhat by author, but limbo lists have some serious correctness and performance downsides. Fraser's improvement is to separate limbo lists into three epochs, a current epoch, a middle, and an epoch from which garbage may be collected. Every participating thread is responsible for observing the current epoch on start, adds its garbage to the current epoch, and

attempts to increase the epoch count when its operations complete. The last thread to complete is the one that succeeds in increasing the epoch and is responsible for performing garbage collection on the final epoch. Three total epochs are needed as threads move from epochs independently from one another: a thread at epoch n may still hold a reference from n-1, but not n-2, and so only n-2 is safe to reclaim. Fraser's thesis is quite long—being, well, being that it is sufficient work to acquire a PhD—but Hart et al's 2007 paper, *Performance of Memory Reclamation for Lockless Synchronization*, is warmly recommended to the reader pressed for time.

An epoch-based Treiber stack

We inspected reference counting and hazard-pointer-based Treiber stacks earlier in this chapter and will follow this approach in this section as well. Fortunately, crossbeam also ships with a Treiber stack implementation, in the crossbeam-rs/crossbeam project, which we'll examine at SHA `89bd6857cd701bff54f7a8bf47ccaa38d5022bfb`, which is the source for `crossbeam::sync::TreiberStack` is in `src/sync/treiber_stack.rs`. The preamble is almost immediately interesting:

```
use std::sync::atomic::Ordering::{Acquire, Relaxed, Release};
use std::ptr;

use epoch::{self, Atomic, Owned};
```

What, for instance, are `epoch::Atomic` and `epoch::Owned`? Well, we'll explore these here. Unlike the section on hazard pointers, which explored the implementation of conc in a depth-first fashion, we'll take a breadth-first approach here, more or less. The reason being, crossbeam-epoch is intricate—it's easy to get lost. Setting aside the somewhat unknown nature of `epoch::Atomic`, the `TreiberStack` and `Node` structs are similar to what we've seen in other implementations:

```
#[derive(Debug)]
pub struct TreiberStack<T> {
    head: Atomic<Node<T>>,
}

#[derive(Debug)]
struct Node<T> {
    data: T,
```

```
    next: Atomic<Node<T>>,
}
```

As is the creation of a new stack:

```
impl<T> TreiberStack<T> {
    /// Create a new, empty stack.
    pub fn new() -> TreiberStack<T> {
        TreiberStack {
            head: Atomic::null(),
        }
    }
```

Pushing a new element also looks familiar:

```
pub fn push(&self, t: T) {
    let mut n = Owned::new(Node {
        data: t,
        next: Atomic::null(),
    });
    let guard = epoch::pin();
    loop {
        let head = self.head.load(Relaxed, &guard);
        n.next.store(head, Relaxed);
        match self.head.compare_and_set(head, n, Release, &guard) {
            Ok(_) => break,
            Err(e) => n = e.new,
        }
    }
}
```

Except, what is `epoch::pin()`? This function `pins` the current thread, returning a `crossbeam_epoch::Guard`. Much like previous guards we've seen throughout the book, crossbeam_epoch's `Guard` protects a resource. In this case, the guard protects the pinned nature of the thread. Once the guard drops, the thread is no longer pinned. Okay, great, so what does it mean for a thread to be pinned? Recall that epoch-based reclamation works by a thread entering an epoch, doing some stuff with memory, and then potentially increasing the epoch after finishing up fiddling with memory. This process is observed in

crossbeam by pinning. `Guard` in hand—implying a safe epoch has been entered—the thread is able to take a heap allocation and get a stack reference to it. Just any old heap allocation and stack reference would be unsafe, however. There's no magic here. That's where `Atomic` and `Owned` come in. They're analogs to `AtomicPtr` and `Box` from the standard library, except that the operations done on them require a reference to `crossbeam_epoch::Guard`. We saw this technique in `Chapter 4`, *Sync and Send – the Foundation of Rust Concurrency*, when discussing hopper, using the type-system to ensure that potentially thread-unsafe operations are done safely by requiring the caller to pass in a guard of some sort. Programmer error can creep in by using `AtomicPtr`, `Box`, or any non-epoch unprotected memory accesses, but these will tend to stand out.

Let's look at popping an element off the stack, where we know marking memory as safe to delete will have to happen:

```
pub fn try_pop(&self) -> Option<T> {
    let guard = epoch::pin();
    loop {
        let head_shared = self.head.load(Acquire, &guard);
        match unsafe { head_shared.as_ref() } {
            Some(head) => {
                let next = head.next.load(Relaxed, &guard);
                if self.head
                    .compare_and_set(head_shared, next, Release,
                     &guard)
                    .is_ok()
                {
                    unsafe {
                        guard.defer(move ||
                        head_shared.into_owned());
                        return Some(ptr::read(&(*head).data));
                    }
                }
            }
            None => return None,
        }
    }
}
```

The key things to note are the creation of `head_shared` by performing a load on an Atomic, the usual compare and set operation, and then this:

```
    guard.defer(move || head_shared.into_owned());
    return Some(ptr::read(&(*head).data));
```

`head_shared` is moved into a closure, converted, and that closure is then passed into the as-yet unexamined defer function of `Guard`. But, head is dereferenced and the data from it is read out and returned. Absent some special trick, that's dangerous. We need to know more.

crossbeam_epoch::Atomic

Let's dig in with the holder of our data, `Atomic`. This structure is from the crossbeam_epoch library and its implementation is in `src/atomic.rs`:

```
pub struct Atomic<T> {
    data: AtomicUsize,
    _marker: PhantomData<*mut T>,
}

unsafe impl<T: Send + Sync> Send for Atomic<T> {}
unsafe impl<T: Send + Sync> Sync for Atomic<T> {}
```

The representation is a little odd: why not `data: *mut T`? Crossbeam's developers have done something neat here. On modern machines, there's a fair bit of space inside of pointer to store information, in the least significant bits. Consider that if we point only to aligned data, the address of a bit of memory will be multiples of 4 on a 32-bit system, or multiples of 8 on a 64-bit system. This leaves either two zero bits at the end of a pointer on 32-bit systems or three zero bits on a 64-bit system. Those bits can store information, called a tag. The fact that `Atomic` can be tagged will come into play. But, Rust doesn't allow us to manipulate pointers in the fashion of other systems programming languages. To that end, and because `usize` is the size of the machine word, crossbeam stores its `*mut T` as a `usize`, allowing for tagging of the pointer. Null `Atomics` are equivalent to tagged pointers to zero:

```
pub fn null() -> Atomic<T> {
    Self {
        data: ATOMIC_USIZE_INIT,
        _marker: PhantomData,
```

```
        }
    }
```

New `Atomic`s are created by passing `T` down through `Owned`—which we saw previously—and then converting from that `Owned`:

```
pub fn new(value: T) -> Atomic<T> {
    Self::from(Owned::new(value))
}
```

`Owned` boxes `T`, performing a heap allocation:

```
impl<T> Owned<T> {
    pub fn new(value: T) -> Owned<T> {
        Self::from(Box::new(value))
    }
```

And subsequently converts that box into `*mut T`:

```
impl<T> From<Box<T>> for Owned<T> {
    fn from(b: Box<T>) -> Self {
        unsafe { Self::from_raw(Box::into_raw(b)) }
    }
}
```

And converts that into a tagged pointer:

```
pub unsafe fn from_raw(raw: *mut T) -> Owned<T> {
    ensure_aligned(raw);
    Self::from_data(raw as usize)
}
```

That's a bit of a trek, but what that tells us is that `Atomic` is no different in terms of data representation compared to `AtomicPtr`, except that the pointer itself is tagged.

Now, what about the usual operations on an atomic pointer, loading and the like? How do those differ in `Atomic`? Well, for one, we know that an epoch `Guard` has to be passed in to each call, but there are other important differences. Here's `Atomic::load`:

```
pub fn load<'g>(&self, ord: Ordering, _: &'g Guard)
    -> Shared<'g, T>
{
    unsafe { Shared::from_data(self.data.load(ord)) }
}
```

We can see `self.data.load(ord)`, so the underlying atomic load is performed as expected. But, what is `Shared`?

```
pub struct Shared<'g, T: 'g> {
    data: usize,
    _marker: PhantomData<(&'g (), *const T)>,
}
```

It's a reference to `Atomic` with an embedded reference to `Guard`. So long as `Shared` exists, the `Guard` that makes memory operations on it safe will also exist and cannot, importantly, cease to exist until `Shared` has been dropped. `Atomic::compare_and_set` introduces a few more traits:

```
pub fn compare_and_set<'g, O, P>(
    &self,
    current: Shared<T>,
    new: P,
    ord: O,
    _: &'g Guard,
) -> Result<Shared<'g, T>, CompareAndSetError<'g, T, P>>
where
    O: CompareAndSetOrdering,
    P: Pointer<T>,
{
    let new = new.into_data();
    self.data
        .compare_exchange(current.into_data(), new,
                          ord.success(), ord.failure())
        .map(|_| unsafe { Shared::from_data(new) })
```

```
            .map_err(|current| unsafe {
                CompareAndSetError {
                    current: Shared::from_data(current),
                    new: P::from_data(new),
                }
            })
    }
```

Notice that `compare_and_set` is defined in terms of `compare_exchange`. This CAS primitive is equivalent to a comparative exchange, but that exchange allows failure to be given more relaxed semantics, offering a performance boost on some platforms. Implementation of compare-and-set, then, requires understanding of which success `Ordering` matches with which failure `Ordering`, from which need comes `CompareAndSetOrdering`:

```
pub trait CompareAndSetOrdering {
    /// The ordering of the operation when it succeeds.
    fn success(&self) -> Ordering;

    /// The ordering of the operation when it fails.
    ///
    /// The failure ordering can't be `Release` or `AcqRel` and must
be
    /// equivalent or weaker than the success ordering.
    fn failure(&self) -> Ordering;
}

impl CompareAndSetOrdering for Ordering {
    #[inline]
    fn success(&self) -> Ordering {
        *self
    }

    #[inline]
    fn failure(&self) -> Ordering {
        strongest_failure_ordering(*self)
    }
}
```

Where `strongest_failure_ordering` is:

```
fn strongest_failure_ordering(ord: Ordering) -> Ordering {
    use self::Ordering::*;
    match ord {
        Relaxed | Release => Relaxed,
        Acquire | AcqRel => Acquire,
        _ => SeqCst,
    }
}
```

The final new trait is `Pointer`, a little utility trait to provide functions over both `Owned` and `Shared`:

```
pub trait Pointer<T> {
    /// Returns the machine representation of the pointer.
    fn into_data(self) -> usize;

    /// Returns a new pointer pointing to the tagged pointer `data`.
    unsafe fn from_data(data: usize) -> Self;
}
```

crossbeam_epoch::Guard::defer

Now, again, how is the following in `TreiberStack::try_pop` a safe operation?

```
guard.defer(move || head_shared.into_owned());
return Some(ptr::read(&(*head).data));
```

What we need to explore now is `defer`, which lives in crossbeam-epoch at `src/guard.rs`:

```
pub unsafe fn defer<F, R>(&self, f: F)
where
    F: FnOnce() -> R,
{
    let garbage = Garbage::new(|| drop(f()));

    if let Some(local) = self.local.as_ref() {
        local.defer(garbage, self);
    }
}
```

We need to understand `Garbage`. It's defined in `src/garbage.rs` as:

```
pub struct Garbage {
    func: Deferred,
}


unsafe impl Sync for Garbage {}
unsafe impl Send for Garbage {}
```

The implementation of `Garbage` is very brief:

```
impl Garbage {
    /// Make a closure that will later be called.
    pub fn new<F: FnOnce()>(f: F) -> Self {
        Garbage { func: Deferred::new(move || f()) }
    }
}

impl Drop for Garbage {
    fn drop(&mut self) {
        self.func.call();
    }
}
```

`Deferred` is a small structure that wraps `FnOnce()`, storing it inline or on the heap as size allows. You are encouraged to examine the implementation yourself, but the basic idea is to maintain the same properties of `FnOnce` in a heap-allocated structure that finds use in the `Garbage` drop implementation. What drops `Garbage`? This is where the following comes into play:

```
        if let Some(local) = self.local.as_ref() {
            local.defer(garbage, self);
        }
    }
```

The `self.local` of `Guard` is `*const Local`, a struct called a participant for garbage collection in crossbeam's documentation. We need to understand where and how this `Local` is created. Let's understand the `Guard` internals, first:

```
pub struct Guard {
    pub(crate) local: *const Local,
}
```

`Guard` is a wrapper around a const pointer to `Local`. We know that instances of `Guard` are created by `crossbeam_epoch::pin`, defined in

`src/default.rs`:

```
pub fn pin() -> Guard {
    HANDLE.with(|handle| handle.pin())
}
```

`HANDLE` is a thread-local static variable:

```
thread_local! {
    /// The per-thread participant for the default garbage
collector.
    static HANDLE: Handle = COLLECTOR.register();
}
```

`COLLECTOR` is a static, global variable:

```
lazy_static! {
    /// The global data for the default garbage collector.
    static ref COLLECTOR: Collector = Collector::new();
}
```

This is a lot of new things all at once. Pinning is a non-trivial activity! As its name implies and its documentation states, `COLLECTOR` is the entry point for crossbeam-epoch's global garbage collector, called `Global`. `Collector` is defined in `src/collector.rs` and has a very brief implementation:

```
pub struct Collector {
    pub(crate) global: Arc<Global>,
}

unsafe impl Send for Collector {}
unsafe impl Sync for Collector {}

impl Collector {
    /// Creates a new collector.
    pub fn new() -> Self {
        Collector { global: Arc::new(Global::new()) }
    }
```

```
    /// Registers a new handle for the collector.
    pub fn register(&self) -> Handle {
        Local::register(self)
    }
}
```

We know that `Collector` is a global static, implying that `Global::new()` will be called only once. `Global`, defined in `src/internal.rs`, is the data repository for the global garbage collector:

```
pub struct Global {
    /// The intrusive linked list of `Local`s.
    locals: List<Local>,

    /// The global queue of bags of deferred functions.
    queue: Queue<(Epoch, Bag)>,

    /// The global epoch.
    pub(crate) epoch: CachePadded<AtomicEpoch>,
}
```

List is an intrusive list, which we saw reference to in the definition of `Local`. An intrusive list is a linked list in which the pointer to the next node in the list is stored in the data itself, the `entry: Entry` field of `Local`. Intrusive lists pop up fairly rarely, but they are quite useful when you're concerned about small memory allocations or need to store an element in multiple collections, both of which apply to crossbeam. `Queue` is a Michael and Scott queue. The epoch is a cache-padded `AtomicEpoch`, cache-padding being a technique to disable write contention on cache lines, called false sharing. `AtomicEpoch` is a wrapper around `AtomicUsize`. `Global` is, then, a linked-list of instances of `Local` – which, themselves, are associated with thread-pinned `Guards`—a queue of `Bag`s, which we haven't investigated yet, associated with some epoch number (a `usize`) and an atomic, global `Epoch`. This layout is not unlike what the algorithm description suggests. Once the sole `Global` is initialized, every thread-local `Handle` is created by calling `Collector::register`, which is `Local::register` internally:

```
    pub fn register(collector: &Collector) -> Handle {
        unsafe {
            // Since we dereference no pointers in this block, it is
            // safe to use `unprotected`.

            let local = Owned::new(Local {
                entry: Entry::default(),
                epoch: AtomicEpoch::new(Epoch::starting()),
                collector: UnsafeCell::new(
                    ManuallyDrop::new(collector.clone())
                ),
                bag: UnsafeCell::new(Bag::new()),
                guard_count: Cell::new(0),
                handle_count: Cell::new(1),
                pin_count: Cell::new(Wrapping(0)),
            }).into_shared(&unprotected());
            collector.global.locals.insert(local, &unprotected());
            Handle { local: local.as_raw() }
        }
    }
```

Note, specifically, that the `collector.global.locals.insert(local, &unprotected())` call is inserting the newly created `Local` into the list of `Global` locals. (`unprotected` is a `Guard` that points to a null, rather than some valid `Local`). Every pinned thread has a `Local` registered with the global garbage collector's data. In fact, let's look at what happens when `Guard` is dropped, before we finish `defer`:

```
impl Drop for Guard {
    #[inline]
    fn drop(&mut self) {
        if let Some(local) = unsafe { self.local.as_ref() } {
            local.unpin();
        }
    }
}
```

The `unpin` method of `Local` is called:

```
    pub fn unpin(&self) {
        let guard_count = self.guard_count.get();
```

```
        self.guard_count.set(guard_count - 1);

        if guard_count == 1 {
            self.epoch.store(Epoch::starting(), Ordering::Release);

            if self.handle_count.get() == 0 {
                self.finalize();
            }
        }
    }
```

The `guard_count` field, recall, is the total number of participating threads or otherwise arranged for guards that keep the thread pinned. The `handle_count` field is a similar mechanism, but one used by `Collector` and `Local`. `Local::finalize` is where the action is:

```
    fn finalize(&self) {
        debug_assert_eq!(self.guard_count.get(), 0);
        debug_assert_eq!(self.handle_count.get(), 0);

        // Temporarily increment handle count. This is required so that
        // the following call to `pin` doesn't call `finalize` again.
        self.handle_count.set(1);
        unsafe {
            // Pin and move the local bag into the global queue. It's
            // important that `push_bag` doesn't defer destruction
            // on any new garbage.
            let guard = &self.pin();
            self.global().push_bag(&mut *self.bag.get(), guard);
        }
        // Revert the handle count back to zero.
        self.handle_count.set(0);
```

`Local` contains a `self.bag` field of the `UnsafeCell<Bag>` type. Here's `Bag`, defined in `src/garbage.rs`:

```
    pub struct Bag {
        /// Stashed objects.
        objects: ArrayVec<[Garbage; MAX_OBJECTS]>,
    }
```

`ArrayVec` is new to this book. It's defined in the `ArrayVec` crate and is a `Vec` but with a capped maximum size. It's the same growable vector we know and love, but one that can't allocate to infinity. `Bag`, then, is a growable vector of `Garbage`, capped at `MAX_OBJECTS` total size:

```
#[cfg(not(feature = "strict_gc"))]
const MAX_OBJECTS: usize = 64;
#[cfg(feature = "strict_gc")]
const MAX_OBJECTS: usize = 4;
```

Attempting to push garbage into a bag above `MAX_OBJECTS` will fail, signaling to the caller that it's time to collect some garbage. What `Local::finalize` is doing, specifically with `self.global().push_bag(&mut *self.bag.get(), guard)`, is taking the `Local`'s bag of garbage and pushing it into the `Global`'s bag of garbage as a part of shutting `Local` down. `Global::push_bag` is:

```
pub fn push_bag(&self, bag: &mut Bag, guard: &Guard) {
    let bag = mem::replace(bag, Bag::new());

    atomic::fence(Ordering::SeqCst);

    let epoch = self.epoch.load(Ordering::Relaxed);
    self.queue.push((epoch, bag), guard);
}
```

Unpinning `Guard` potentially shuts down a `Local`, which pushes its garbage into the queue of epoch-tagged garbage in `Global`. Now that we understand that, let's finish `Guard::defer` by inspecting `Local::defer`:

```
pub fn defer(&self, mut garbage: Garbage, guard: &Guard) {
    let bag = unsafe { &mut *self.bag.get() };

    while let Err(g) = bag.try_push(garbage) {
        self.global().push_bag(bag, guard);
        garbage = g;
    }
}
```

The pinned caller signals garbage to its `Guard` by calling `defer`. The `Guard`, in turn, defers this garbage to its `Local`, which enters a while loop wherein it attempts to push garbage onto the local bag of garbage but will shift garbage into `Global` so long as the local bag is full.

When does garbage get collected from `Global`? The answer is in a function we've not yet examined, `Local::pin`.

crossbeam_epoch::Local::pin

Recall that `crossbeam_epoch::pin` calls `Handle::pin`, which in turn calls `pin` on its `Local`. What happens during the execution of `Local::pin`? A lot:

```
pub fn pin(&self) -> Guard {
    let guard = Guard { local: self };

    let guard_count = self.guard_count.get();
    self.guard_count.set(guard_count.checked_add(1).unwrap());
```

The `Local` guard count is increased, which is done amusingly by creating a `Guard` with `self`. If the guard count was previously zero:

```
    if guard_count == 0 {
        let global_epoch =
     self.global().epoch.load(Ordering::Relaxed);
        let new_epoch = global_epoch.pinned();
```

This means that there are no other active participants in `Local`, and `Local` needs to query for the global epoch. The global epoch is loaded as the `Local` epoch:

```
            self.epoch.store(new_epoch, Ordering::Relaxed);
            atomic::fence(Ordering::SeqCst);
```

Sequential consistency is used here to ensure that ordering is maintained with regard to future and past atomic operations. Finally, the `Local`'s `pin_count` is increased and this kicks off, potentially, `Global::collect`:

```
            let count = self.pin_count.get();
            self.pin_count.set(count + Wrapping(1));

            // After every `PINNINGS_BETWEEN_COLLECT` try advancing the
            // epoch and collecting some garbage.
            if count.0 % Self::PINNINGS_BETWEEN_COLLECT == 0 {
                self.global().collect(&guard);
            }
        }

        guard
    }
```

It is possible for `Local` to be pinned and unpinned repeatedly, which is where `pin_count` comes into play. This mechanism doesn't find use in our discussion here, but the reader is referred to `Guard::repin` and `Guard::repin_after`. The latter function is especially useful when mixing network calls with atomic data structures as, if you'll recall, garbage cannot be collected until epochs advance and the epoch is only advanced by unpinning, usually. `Global::collect` is brief:

```
    pub fn collect(&self, guard: &Guard) {
        let global_epoch = self.try_advance(guard);
```

The global epoch is potentially advanced by `Global::try_advance`, the advancement only happening if every `Local` in the global list of `Local` is not in another epoch *or* the list of locals is not modified during examination. The detection of concurrent modification is an especially neat trick, being part of crossbeam-epoch's private `List` iteration. The tagged pointer plays a part in this iteration and the reader is warmly encouraged to read and understand the `List` implementation crossbeam-epoch uses:

```
        let condition = |item: &(Epoch, Bag)| {
            // A pinned participant can witness at most one epoch
            advancement. Therefore, any bag
            // that is within one epoch of the current one cannot be
            destroyed yet.
```

```
            global_epoch.wrapping_sub(item.0) >= 2
        };

        let steps = if cfg!(feature = "sanitize") {
            usize::max_value()
        } else {
            Self::COLLECT_STEPS
        };

        for _ in 0..steps {
            match self.queue.try_pop_if(&condition, guard) {
                None => break,
                Some(bag) => drop(bag),
            }
        }
    }
```

The condition will be passed into the Global's `Queue::try_pop_if`, which only pops an element from the queue that matches the conditional. We can see the algorithm at play again here. Recall `self.queue :: Queue<(Epoch, Bag)>`? Bags of garbage will only be pulled from the queue if they are greater than two epochs away from the current epoch, otherwise there may still be live and dangerous references to them. The steps control how much garbage will end up being collected, doing a tradeoff for time versus memory use. Recall that every newly pinned thread is participating in this process, deallocating some of the global garbage.

To summarize, when the programmer calls for their thread to be pinned, this creates a `Local` for storing thread-local garbage that is linked into a list of all `Local`s in a `Global` context. The thread will attempt to collect `Global` garbage, potentially pushing the epoch forward in the process. Any garbage the programmer defers during execution is preferentially pushed into the `Local` bag of garbage or, if that's full, causes some garbage to shift into the `Global` bag. Any garbage in the `Local` bag is shifted onto the `Global` bag when the `Local` is unpinned. Every atomic operation, by reason of requiring a reference to `Guard`, is implicitly associated with some `Local`, some epoch, and can

be safely reclaimed by the method outlined in the preceding algorithm.

Phew!

Exercising the epoch-based Treiber stack

Let's put the crossbeam-epoch Treiber stack through the wringer to get an idea of the performance of this approach. Key areas of interest for us will be:

- Push/pop cycles per second
- Memory behavior, high-water mark, and what not
- cache behavior

We'll run our programs on x86 and ARM, like we did in previous chapters. Our exercise program, similar to the hazard pointer program from the previous section:

```
extern crate crossbeam;
#[macro_use]
extern crate lazy_static;
extern crate num_cpus;
extern crate quantiles;

use crossbeam::sync::TreiberStack;
use quantiles::ckms::CKMS;
use std::sync::Arc;
use std::sync::atomic::{AtomicUsize, Ordering};
use std::{thread, time};

lazy_static! {
    static ref WORKERS: AtomicUsize = AtomicUsize::new(0);
    static ref COUNT: AtomicUsize = AtomicUsize::new(0);
}
static MAX_I: u32 = 67_108_864; // 2 ** 26
```

```
fn main() {
    let stk: Arc<TreiberStack<(u64, u64, u64)>> =
Arc::new(TreiberStack::new());

    let mut jhs = Vec::new();

    let cpus = num_cpus::get();
    WORKERS.store(cpus, Ordering::Release);

    for _ in 0..cpus {
        let stk = Arc::clone(&stk);
        jhs.push(thread::spawn(move || {
            for i in 0..MAX_I {
                stk.push((i as u64, i as u64, i as u64));
                stk.pop();
                COUNT.fetch_add(1, Ordering::Relaxed);
            }
            WORKERS.fetch_sub(1, Ordering::Relaxed)
        }))
    }

    let one_second = time::Duration::from_millis(1_000);
    let mut iter = 0;
    let mut cycles: CKMS<u32> = CKMS::new(0.001);
    while WORKERS.load(Ordering::Relaxed) != 0 {
        let count = COUNT.swap(0, Ordering::Relaxed);
        cycles.insert((count / cpus) as u32);
        println!(
            "CYCLES PER SECOND({}):\n  25th: \
             {}\n  50th: {}\n  75th: \
             {}\n  90th: {}\n  max:  {}\n",
            iter,
            cycles.query(0.25).unwrap().1,
            cycles.query(0.50).unwrap().1,
            cycles.query(0.75).unwrap().1,
            cycles.query(0.90).unwrap().1,
            cycles.query(1.0).unwrap().1
        );
        thread::sleep(one_second);
        iter += 1;
    }

    for jh in jhs {
        jh.join().unwrap();
    }
}
```

We have a number of worker threads equal to the total number of CPUs in the target machine, each doing one push and then an immediate pop on the stack. A `COUNT` is kept and the main thread swaps that value for 0 every second or so, tossing the value into a quantile estimate structure—discussed in more detail in `Chapter 4`, *Sync and Send – the Foundation of Rust Concurrency*—and printing out a summary of the recorded cycles per second, scaled to the number of CPUs. The worker threads cycle up through to `MAX_I`, which is arbitrarily set to the smallish value of `2**26`. When the worker is finished cycling, it decreases `WORKERS` and exits. Once `WORKERS` hits zero, the main loop also exits.

On my x86 machine, it takes approximately 38 seconds for this program to exit, with this output:

```
CYCLES PER SECOND(0):
  25th: 0
  50th: 0
  75th: 0
  90th: 0
  max:  0

CYCLES PER SECOND(1):
  25th: 0
  50th: 0
  75th: 1739270
  90th: 1739270
  max:  1739270

...

CYCLES PER SECOND(37):
  25th: 1738976
  50th: 1739528
  75th: 1740474
  90th: 1757650
  max:  1759459

CYCLES PER SECOND(38):
  25th: 1738868
  50th: 1739528
  75th: 1740452
```

```
  90th: 1757650
  max:  1759459
```

Compare this to the x86 hazard implementation, which takes 58 total seconds. The x86 perf run is as follows:

```
> perf stat --event task-clock,context-switches,page-
faults,cycles,instructions,branches,branch-misses,cache-
references,cache-misses target/release/epoch_stack > /dev/null

 Performance counter stats for 'target/release/epoch_stack':

     148830.337380      task-clock (msec)         #    3.916 CPUs
utilized
             1,043      context-switches          #    0.007 K/sec
             1,039      page-faults               #    0.007 K/sec
   429,969,505,981      cycles                    #    2.889 GHz
   161,901,052,886      instructions              #    0.38  insn per
cycle
    27,531,697,676      branches                  #  184.987 M/sec
       627,050,474      branch-misses             #    2.28% branches
    11,885,394,803      cache-references          #   79.859 M/sec
         1,772,308      cache-misses              #    0.015 % cache
refs

      38.004548310 seconds time elapsed
```

The total number of executed instructions is much, much lower in the epoch-based approach, which squares well with the analysis in the opening discussion of this section. Even with a single hazardous pointer, epoch-based approaches do less work. On my ARM machine, it takes approximately 72 seconds for the program to run to completion, with this output:

```
CYCLES PER SECOND(0):
  25th: 0
  50th: 0
  75th: 0
  90th: 0
  max:  0

CYCLES PER SECOND(1):
```

```
  25th: 0
  50th: 0
  75th: 921743
  90th: 921743
  max:  921743

...

CYCLES PER SECOND(71):
  25th: 921908
  50th: 922326
  75th: 922737
  90th: 923235
  max:  924084

CYCLES PER SECOND(72):
  25th: 921908
  50th: 922333
  75th: 922751
  90th: 923235
  max:  924084
```

Compared to the ARM hazard implementation, which takes 463 total seconds! The ARM perf run is as follows:

```
> perf stat --event task-clock,context-switches,page-
faults,cycles,instructions,branches,branch-misses,cache-
references,cache-misses target/release/epoch_stack > /dev/null

 Performance counter stats for 'target/release/epoch_stack':

     304880.898057      task-clock (msec)         #    3.959 CPUs
utilized
                 0      context-switches          #    0.000 K/sec
               248      page-faults               #    0.001 K/sec
   364,139,169,537      cycles                    #    1.194 GHz
   215,754,991,724      instructions              #    0.59  insn per
cycle
    28,019,208,815      branches                  #   91.902 M/sec
     3,525,977,068      branch-misses             #   12.58% branches
   105,165,886,677      cache-references          #  344.941 M/sec
     1,450,538,471      cache-misses              #    1.379 % cache
refs

      77.014340901 seconds time elapsed
```

Again, drastically fewer instructions are executed compared to the hazard-pointer implementation on the same processor architecture. In passing, it's worth noting that the memory use of both the x86 and ARM versions' `epoch_stack` were lower than the hazard pointer stack implementations, though neither were memory hogs, consuming only a few kilobytes each. Had one of our epoch threads slept for a long while—say, a second or so—before leaving its epoch, memory use would have grown during execution. The reader is encouraged to wreak havoc on their own system.

Tradeoffs

Of the three approaches outlined in this chapter, crossbeam-epoch is the fastest, both in terms of our measurements and from discussions in the literature. The faster technique, quiescent-state-based memory reclamation, is not discussed in this chapter as it's an awkward fit for user-space programs and there is no readily-available implementation in Rust. Furthermore, the authors of the library have put years of work into the implementation: it is well-documented and runs on modern Rust across multiple CPU architectures.

Traditional issues with the approach—namely, threads introducing long delays to pinned critical sections, resulting in long-lived epochs and garbage buildup—are addressed in the API by the abiltiy of `Guard` to be arbitrarily repinned. The transition of standard-library `AtomicPtr` data structures to crossbeam is a project, to be sure, but an approachable one. As we've seen, crossbeam-epoch does introduce some overhead, both in terms of cache padding and thread synchronization. This overhead is minimal and should be expected to improve with time.

Summary

In this chapter, we discussed three memory reclamation techniques: reference counting, hazard pointers, and epoch-based reclamation. Each, in turn, is faster than the last, though there are tradeoffs with each approach. Reference counting incurs the most overhead and has to be incorporated carefully into your data structure, unless Arc fits your needs, which it may well. Hazard pointers require the identification of hazards, memory accesses that result in memory that cannot be reclaimed without some type of coordination. This approach incurs overhead on each access to the hazard, which is costly if traversal of a hazardous structure must be done. Finally, epoch-based reclamation incurs overhead at thread-pinning, which denotes the start of an epoch and may require the newly pinned thread to participate in garage collection. Additional overhead is not incurred on memory accesses post-pin, a big win if you're doing traversal or can otherwise include many memory operations in a pinned section.

The crossbeam library is exceptionally well done. Unless you have an atomic data structure specifically designed to not require allocations or to cope with allocations on a relaxed memory system without recourse to garbage collection, you are warmly encouraged to consider crossbeam as part and parcel with doing atomic programming in Rust, as of this writing.

In the next chapter, we will leave the realm of atomic programming and discuss higher-level approaches to parallel programming, using thread pooling and data parallel iterators.

Further reading

- *Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects*, Maged Michael. This paper introduces the hazard-pointer reclamation technique discussed in this chapter. The paper specifically discusses the newly invented technique in comparison to reference counting and demonstrates the construction of a safe Michael and Scott queue using the hazard pointer technique.
- *Practical Lock-Freedom*, Keir Fraser. This is Keir Fraser's PhD thesis and is quite long, being concerned with the introduction of abstractions to ease the writing of lock-free structures—one of which is epoch-based reclamation—and the introduction of lock-free search structures, skip-lists, binary search trees, and red-black trees. Warmly recommended.
- *Performance of Memory Reclamation for Lockless Synchronization*, Thomas Hart et al. This paper provides an overview of four techniques, all of which were discussed in passing in `Chapter 6`, *Atomics – the Primitives of Synchronization*, of this book, and three of which were discussed in-depth in this chapter. Further, the paper introduces an optimization to epoch-based reclamation, which has influenced crossbeam, if your author is not

mistaken. The paper is an excellent overview of the algorithms and provides comparative measurements of the algorithms' performance in a well-defined manner.

- *RFCs for changes to Crossbeam*, available at `https://github.com/crossbeam-rs/rfcs`. This Github repository is the primary discussion point for large-scale changes to the crossbeam library and is an especially important read for its discussions. For instance, RFC 2017-07-23-relaxed-memory lays out the changes necessary to crossbeam to operate on relaxed-memory systems, a topic rarely discussed in the literature.
- *CDSCHECKER: Checking Concurrent Data Structures Written with C/C++ Atomics*, Brian Norris and Brian Demsky. This paper introduces a tool for checking the behavior of concurrent data structures according to the C++11/LLVM memory model. Given how utterly confounding relaxed-memory ordering can be, this tool is extremely useful when doing C++ work. I am unaware of a Rust analog but hope this will be seen as an opportunity for some bright spark reading this. Good luck, you.
- *A Promising Semantics for Relaxed-Memory Concurrency*, Jeehoon Kang et al. Reasoning about the memory model proposed in C++/LLVM is very difficult. Which, despite how many people have thought about it in-depth over the years, there's still active research into formalizing this model to validate the correct function of optimizers and algorithms. Do read this paper in conjunction with CDSCHECKER by Norris and Demsky.

High-Level Parallelism – Threadpools, Parallel Iterators and Processes

In previous chapters, we introduced the basic mechanisms of concurrency in the Rust—programming language. In `Chapter 4`, *Sync and Send – the Foundation of Rust Concurrency*, we discussed the interplay of the type system of Rust with concurrent programs, how Rust ensures memory safety in this most difficult of circumstances. In `Chapter 5`, *Locks – Mutex, Condvar, Barriers and RWLock*, we discussed the higher, so-called coarse, synchronization mechanisms available to us, common among many languages. In `Chapter 6`, *Atomics – the Primitives of Synchronization*, and `Chapter 7`, *Atomics – Safely Reclaiming Memory*, we discussed the finer synchronization primitives available on modern machines, exposed through Rust's concurrent memory model. This has all been well and good but, though we've done deep-dives into select libraries or data structures, we have yet to see the *consequences* of all of these tools on the structure of programs, or how you might choose to split up your workloads across CPUs depending on need.

In this chapter, we will explore higher-level techniques for exploiting concurrent machines without dipping into manual locking or atomic synchronization. We'll examine thread pooling, a technique common in other programming languages, data parallelism with the rayon library, and demonstrate multiprocessing in the context of a genetic programming project that will carry us into the next chapter, as well.

By the end of this chapter, we will have:

- Explored the implementation of thread pool
- Understood how thread pooling relates to the operation of rayon
- Explored rayon's internal mechanism in-depth
- Demonstrated the use of rayon in a non-trivial exercise

Technical requirements

This chapter requires a working Rust installation. The details of verifying your installation are covered in `Chapter 1`, *Preliminaries – Machine Architecture and Getting Started with Rust.* A pMARS executable is required and must be on your PATH. Please follow the instructions in the pMARS (`http://www.koth.org/pmars/`) source tree for building instructions.
You can find the source code for this book's projects on GitHub: `https://github.com/PacktPublishing/Hands-On-Concurrency-with-Rust`. The source code for this chapter is under `Chapter08`.

Thread pooling

To this point in the book, whenever we have needed a thread, we've simply called `thread::spawn`. This is not necessarily a safe thing to do. Let's inspect two projects that suffer from a common defect—potential over-consumption of OS threads. The first will be obviously deficient, the second less so.

Slowloris – attacking thread-per-connection servers

The thread-per-connection architecture is a networking architecture that allocates one OS thread per inbound connection. This works well for servers that will receive a total number of connections relatively similar to the number of CPUs available to the server. Depending on the operating system, this architecture tends to reduce time-to-response latency of network transactions, as a nice bonus. The major defect with thread-per-connection systems has to do with slowloris attacks. In this style of attack, a malicious user opens a connection to the server–a relatively cheap operation, requiring only a single file-handler and simply holds it. Thread-per-connection systems can mitigate the slowloris attack by aggressively timing out idle connections, which the attacker can then mitigate by sending data through the connection at a very slow rate, say one byte per 100 milliseconds. The attacker, which could easily be just buggy software if you're deploying solely inside a trusted network, spends very little resources to carry out their attack. Meanwhile, for every new connection into the system, the server is forced to allocate a full stack frame and forced to cram another thread into the OS scheduler. That's not cheap.

The server

Let's put together a vulnerable server, then blow it up. Lay out your `Cargo.toml` like so:

```
[package]
name = "overwhelmed_tcp_server"
version = "0.1.0"
authors = ["Brian L. Troutwine <brian@troutwine.us>"]

[dependencies]
clap = "2.31"
slog = "2.2"
slog-term = "2.4"
slog-async = "2.3"

[[bin]]
name = "server"

[[bin]]
name = "client"
```

The library slog (`https://crates.io/crates/slog`) is a structured logging library that I highly recommend in production systems. Slog itself is very flexible, being more of a framework for composing a logging system rather than a set thing. Here we'll be logging to terminal, which is what slog-term is for. The `slog-async` dependency defers actual writing to the terminal, in our case, to a dedicated thread. This dedication is more important when slog is emitting logs over a network, rather than quickly to terminal. We won't go in-depth on slog in this book but the documentation is comprehensive and I imagine you'll appreciate slog if you aren't already using it. The library clap (`https://crates.io/crates/clap`) is a command-line parsing library, which I also highly recommend for use in production systems. Note, finally, that we produce two binaries in this project,

called `server` and `client`. Let's dig into `server` first. As usual, our projects start with a preamble:

```
   #[macro_use]
   extern crate slog;
   extern crate clap;
   extern crate slog_async;
   extern crate slog_term;

   use clap::{App, Arg};
   use slog::Drain;
   use std::io::{self, BufRead, BufReader, BufWriter, Write};
   use std::net::{TcpListener, TcpStream};
```

```
   use std::sync::atomic::{AtomicUsize, Ordering};
   use std::thread;

   static TOTAL_STREAMS: AtomicUsize = AtomicUsize::new(0);
```

By this point in the book, there's nothing surprising here. We import the external libraries we've just discussed as well as a host of standard library material. The networking-related imports are unfamiliar, with regard to the previous content of this book, but we'll go into that briefly below. We finally create a static `TOTAL_STREAMS` at the top of the program that the `server` will use to track the total number of TCP streams it has connected. The `main` function starts off setting up slog:

```
fn main() {
    let decorator = slog_term::TermDecorator::new().build();
    let drain =
slog_term::CompactFormat::new(decorator).build().fuse();
    let drain = slog_async::Async::new(drain).build().fuse();
    let root = slog::Logger::root(drain, o!());
```

The exact details here are left to the reader to discover in slog's documentation. Next, we set up the clap and `parse` program arguments:

```
let matches = App::new("server")
    .arg(
        Arg::with_name("host")
            .long("host")
            .value_name("HOST")
            .help("Sets which hostname to listen on")
            .takes_value(true),
    )
    .arg(
        Arg::with_name("port")
            .long("port")
            .value_name("PORT")
            .help("Sets which port to listen on")
            .takes_value(true),
    )
    .get_matches();

let host: &str = matches.value_of("host").unwrap_or("localhost");
let port = matches
    .value_of("port")
    .unwrap_or("1987")
    .parse::<u16>()
    .expect("port-no not valid");
```

The details here are also left to the reader to discover in clap's documentation, but hopefully the intent is clear enough. We're setting up two arguments, `host` and `port`, which the server will listen for connections on. Speaking of:

```
let listener = TcpListener::bind((host, port)).unwrap();
```

Note that we're unwrapping if it's not possible for the server to establish a connection. This is user-hostile and in a production-worthy application you should match on the error and print out a nice, helpful message to explain the error. Now that the server is listening for new connections, we make a server-specific logger and start handling incoming connections:

```
let server = root.new(o!("host" => host.to_string(),
                         "port" => port));
```

```
    info!(server, "Server open for business! :D");

    let mut joins = Vec::new();
    for stream in listener.incoming() {
        if let Ok(stream) = stream {
            let stream_no = TOTAL_STREAMS.fetch_add(1,
            Ordering::Relaxed);
            let log = root.new(o!("stream-no" => stream_no,
                   "peer-addr" => stream.peer_addr()
                                  .expect("no peer address")
                                  .to_string()));
            let writer = BufWriter::new(
                            stream.try_clone()
                            .expect("could not clone stream"));
            let reader = BufReader::new(stream);
            match handle_client(log, reader, writer) {
                Ok(handler) => {
                    joins.push(handler);
                }
                Err(err) => {
                    error!(server,
                           "Could not make client handler. {:?}",
                           err);
                }
            }
        } else {
            info!(root, "Shutting down! {:?}", stream);
        }
    }

    info!(
        server,
        "No more incoming connections. Draining existing connections."
    );
    for jh in joins {
        if let Err(err) = jh.join() {
            info!(server,
                  "Connection handler died with error: {:?}",
                  err);
        }
    }
}
```

The key thing here is `handle_client(log, reader, writer)`. This function accepts the newly created `stream :: TcpStream`— in its buffered reader and writer guise—and returns `std::io::Result<JoinHandler<()>>`. We'll see

the implementation of this function directly. Before that and somewhat as an aside, it's very important to remember to add buffering to your IO. If we did not have `BufWriter` and `BufReader` in place here, every read and write to `TcpStream` would result in a system call on most systems doing per-byte transmissions over the network. It is *significantly* more efficient for everyone involved to do reading and writing in batches, which the `BufReader` and `BufWriter` implementations take care of. I have lost count of how many overly slow programs I've seen fixed with a judicious application of buffered-IO. Unfortunately, we won't dig into the implementations of `BufReader` and `BufWriter` here as they're outside the scope of this book. If you've read this far and I've done alright job explaining, then you've learned everything you need to understand the implementations and you are encouraged to dig in at your convenience. Note that here we're also allocating a vector for `JoinHandler<()>`s returned by `handle_client`. This is not necessarily ideal. Consider if the first connection were to be long-lived and every subsequent connection short. None of the handlers would be cleared out, though they were completed, and the program would gradually grow in memory consumption. The resolution to this problem is going to be program-specific but, at least here, it'll be sufficient to ignore the handles and force mid-transaction exits on worker threads. Why? Because the server is only echoing:

```
fn handle_client(
    log: slog::Logger,
    mut reader: BufReader<TcpStream>,
    mut writer: BufWriter<TcpStream>,
) -> io::Result<thread::JoinHandle<()>> {
    let builder = thread::Builder::new();
    builder.spawn(move || {
        let mut buf = String::with_capacity(2048);

        while let Ok(sz) = reader.read_line(&mut buf) {
            info!(log, "Received a {} bytes: {}", sz, buf);
            writer
                .write_all(&buf.as_bytes())
                .expect("could not write line");
            buf.clear();
        }
```

```
            TOTAL_STREAMS.fetch_sub(1, Ordering::Relaxed);
        })
    }
```

A network protocol must, anyway, be resilient to hangups on either end, owing to the unreliable nature of networks. In fact, the reader who has enjoyed this book and especially `Chapter 6`, *Atomics – the Primitives of Synchronization*, and `Chapter 7`, *Atomics – Safely Reclaiming Memory*, will be delighted to learn that distributed systems face many of the same difficulties with the added bonus of unreliable transmission. Anyway, note that `handle_client` isn't doing all that much, merely using the `thread::Builder` API to construct threads, which we discussed in `Chapter 5`, *Locks – Mutex, Condvar, Barriers and RWLock*. Otherwise, it's a fairly standard TCP echo.

Let's see the server in operation. In the top of the project, run `cargo run --release --bin server` and you should be rewarded with something much like:

```
> cargo run --release --bin server
    Finished release [optimized] target(s) in 0.26 secs
     Running `target/release/server`
host: localhost
 port: 1987
  Apr 22 21:31:14.001 INFO Server open for business! :D
```

So far so good. This server is listening on localhost, port 1987. In some other terminal, you can run a telnet to the server:

```
> telnet localhost 1987
Trying ::1...
Connected to localhost.
Escape character is '^]'.
hello server
```

I've sent `hello server` to my running instance and have yet to receive a response because of the behavior of the write buffer. An explicit flush

would correct this, at the expense of worse performance. Whether a flush should or should not be placed will depend entirely on setting. Here? Meh.

The server dutifully logs the interaction:

```
stream-no: 0
 peer-addr: [::1]:65219
  Apr 22 21:32:54.943 INFO Received a 14 bytes: hello server
```

The client

The real challenge for our server comes with a specially constructed client. Let's go through it before we see the client in action. The preamble is typical:

```
#[macro_use]
extern crate slog;
extern crate clap;
extern crate slog_async;
extern crate slog_term;

use clap::{App, Arg};
use slog::Drain;
use std::net::TcpStream;
use std::sync::atomic::{AtomicUsize, Ordering};
use std::{thread, time};

static TOTAL_STREAMS: AtomicUsize = AtomicUsize::new(0);
```

In fact, this client preamble hews fairly close to that of the server. So, too, the `main` function, as we'll see shortly. As with many other programs in this book, we dedicate a thread to reporting on the behavior of the program. In this client, this thread runs the long-lived `report`:

```
fn report(log: slog::Logger) {
    let delay = time::Duration::from_millis(1000);
    let mut total_streams = 0;
    loop {
        let streams_per_second = TOTAL_STREAMS.swap(0,
Ordering::Relaxed);
        info!(log, "Total connections: {}", total_streams);
        info!(log, "Connections per second: {}", streams_per_second);
        total_streams += streams_per_second;
        thread::sleep(delay);
```

```
    }
}
```

Every second `report` swaps `TOTAL_STREAMS`, maintaining its own `total_streams`, and uses the reset value as a per-second gauge. Nothing we haven't seen before in this book. Now, in the `main` function, we set up the logger and parse the same command options as in the server:

```
fn main() {
    let decorator = slog_term::TermDecorator::new().build();
    let drain =
slog_term::CompactFormat::new(decorator).build().fuse();
    let drain = slog_async::Async::new(drain).build().fuse();
    let root = slog::Logger::root(drain, o!());

    let matches = App::new("server")
        .arg(
            Arg::with_name("host")
                .long("host")
                .value_name("HOST")
                .help("Sets which hostname to listen on")
                .takes_value(true),
        )
        .arg(
            Arg::with_name("port")
                .long("port")
                .value_name("PORT")
                .help("Sets which port to listen on")
                .takes_value(true),
        )
        .get_matches();

    let host: &str = matches.value_of("host").unwrap_or("localhost");
    let port = matches
        .value_of("port")
        .unwrap_or("1987")
        .parse::<u16>()
        .expect("port-no not valid");

    let client = root.new(o!("host" => host.to_string(), "port" =>
port));
    info!(client, "Client ready to be mean. >:)");
```

In `main`, we start the reporter thread and ignore its handle:

```
    let reporter_log = root.new(o!("meta" => "report"));
    let _ = thread::spawn(|| report(reporter_log));
```

That leaves only committing the slowloris attack, which is disappointingly easy for the client to do:

```
    let mut streams = Vec::with_capacity(2048);
    loop {
        match TcpStream::connect((host, port)) {
            Ok(stream) => {
                TOTAL_STREAMS.fetch_add(1, Ordering::Relaxed);
                streams.push(stream);
            }
            Err(err) => error!(client,
                               "Connection rejected with error: {:?}",
                               err),
        }
    }
}
```

Yep, that's it, one simple loop connecting as quickly as possible. A more sophisticated setup would send a trickle of data through the socket to defeat aggressive timeouts or limit the total number of connections made to avoid suspicion, coordinating with other machines to do the same.

Mischief on networks is an arms race that nobody really wins, is the moral.

Anyway, let's see this client in action. If you run `cargo run --release --bin client`, you should be rewarded with:

```
> cargo run --release --bin client
    Finished release [optimized] target(s) in 0.22 secs
     Running `target/release/client`
host: localhost
 port: 1987
  Apr 22 21:50:49.200 INFO Client ready to be mean. >:)
meta: report
```

```
Apr 22 21:50:49.200 INFO Total connections: 0
Apr 22 21:50:49.201 INFO Connections per second: 0
Apr 22 21:50:50.200 INFO Total connections: 0
Apr 22 21:50:50.200 INFO Connections per second: 1160
Apr 22 21:50:51.204 INFO Total connections: 1160
Apr 22 21:50:51.204 INFO Connections per second: 1026
Apr 22 21:50:52.204 INFO Total connections: 2186
Apr 22 21:50:52.204 INFO Connections per second: 915
```

You'll find a spew of errors in the server log as well as high-CPU load due to the OS scheduler thrash. It's not a good time.

The ultimate problem here is that we've allowed an external party, the client, to decide the allocation patterns of our program. Fixed sizes make for safer software. In this case, we'll need to fix the total number of threads at startup, thereby capping the number of connections the server is able to sustain to a relatively low number. Though, it will be a safe low number. Note that the deficiency of allocating a whole OS thread to a single network connection is what inspired C10K Problem in the late 1990s, leading to more efficient polling capabilities in modern operating systems. These polling syscalls are the basis of Mio and thereby Tokio, though the discussion of either is well outside the scope of this book. Look them up. It's the future of Rust networking.

What this program needs is a way of fixing the number of threads available for connections. That's exactly what a thread pool is.

A thread-pooling server

Let's adapt our overwhelmed TCP server to use a fixed number of threads. We'll start a new project whose `Cargo.toml` looks like:

```
[package]
name = "fixed_threads_server"
version = "0.1.0"
authors = ["Brian L. Troutwine <brian@troutwine.us>"]

[dependencies]
clap = "2.31"
slog = "2.2"
slog-term = "2.4"
slog-async = "2.3"
threadpool = "1.7"

[[bin]]
name = "server"

[[bin]]
name = "client"
```

This is almost exactly the same as the unbounded thread project, save that the name has been changed from `overwhelmed_tpc_server` to `fixed_threads_tcp_server` and we've added a new dependency– threadpool. There are a few different, stable thread-pool libraries available in crates, each with a slightly different feature set or focus. The workerpool project (`https://crates.io/crates/workerpool`), for example, sports almost the same API as threadpool, save that it comes rigged up so in/out communication with the worker thread is enforced at the type-level, where threadpool requires the user to establish this themselves if they want. In this project, we don't have a need to communicate with the worker thread.

The server preamble is only slightly altered from the previous project:

```
#[macro_use]
extern crate slog;
extern crate clap;
extern crate slog_async;
extern crate slog_term;
extern crate threadpool;

use clap::{App, Arg};
use slog::Drain;
use std::io::{BufRead, BufReader, BufWriter, Write};
use std::net::{TcpListener, TcpStream};
use std::sync::atomic::{AtomicUsize, Ordering};
use threadpool::ThreadPool;

static TOTAL_STREAMS: AtomicUsize = AtomicUsize::new(0);
```

The only changes here are the introduction of the thread-pooling library and the removal of some unused imports. Our `handle_client` implementation is also basically the same, but is notably not spawning a thread:

```
fn handle_client(
    log: slog::Logger,
    mut reader: BufReader<TcpStream>,
    mut writer: BufWriter<TcpStream>,
) -> () {
    let mut buf = String::with_capacity(2048);

    while let Ok(sz) = reader.read_line(&mut buf) {
        info!(log, "Received a {} bytes: {}", sz, buf);
        writer
            .write_all(&buf.as_bytes())
            .expect("could not write line");
        buf.clear();
    }
    TOTAL_STREAMS.fetch_sub(1, Ordering::Relaxed);
}
```

Our main function, likewise, is very similar. We set up the logger and parse application options, adding in a new `max_connections` option:

```
fn main() {
    let decorator = slog_term::TermDecorator::new().build();
    let drain =
slog_term::CompactFormat::new(decorator).build().fuse();
    let drain = slog_async::Async::new(drain).build().fuse();
    let root = slog::Logger::root(drain, o!());

    let matches = App::new("server")
        .arg(
            Arg::with_name("host")
                .long("host")
                .value_name("HOST")
                .help("Sets which hostname to listen on")
                .takes_value(true),
        )
        .arg(
            Arg::with_name("port")
                .long("port")
                .value_name("PORT")
                .help("Sets which port to listen on")
                .takes_value(true),
        )
        .arg(
            Arg::with_name("max_connections")
                .long("max_connections")
                .value_name("CONNECTIONS")
                .help("Sets how many connections (thus,\
                       threads) to allow simultaneously")
                .takes_value(true),
        )
        .get_matches();

    let host: &str = matches.value_of("host").unwrap_or("localhost");
    let port = matches
        .value_of("port")
        .unwrap_or("1987")
        .parse::<u16>()
        .expect("port-no not valid");
    let max_connections = matches
        .value_of("max_connections")
        .unwrap_or("256")
        .parse::<u16>()
        .expect("max_connections not valid");
```

Note that `max_connections` is `u16`. This is arbitrary and will under-consume some of the larger machines available in special circumstances. But this is a reasonable range for most hardware today. Just the same as before, we adapt the logger to include the host and port information:

```
    let listener = TcpListener::bind((host, port)).unwrap();
    let server = root.new(o!("host" => host.to_string(), "port" =>
port));
    info!(server, "Server open for business! :D");
```

So far, so good. Before the server can start accepting incoming connections, it needs a thread pool. Like so:

```
    let pool: ThreadPool = threadpool::Builder::new()
        .num_threads(max_connections as usize)
        .build();
```

Exactly what `threadpool::Builder` does and how it works we defer to the next section, relying only on documentation to guide us. As such, we now have `ThreadPool` with `max_connection` possible active threads. This pool implementation does not bound the total number of jobs that can be run against the pool, collecting them in a queue, instead. This is not ideal for our purposes, where we aim to reject connections that can't be serviced. This is something we can address:

```
    for stream in listener.incoming() {
        if let Ok(stream) = stream {
            if pool.active_count() == (max_connections as usize) {
                info!(
                    server,
                    "Max connection condition reached, rejecting
incoming"
                );
            } else {
                let stream_no = TOTAL_STREAMS.fetch_add(
                                    1,
```

```
                                    Ordering::Relaxed
                                );
                let log = root.new(o!("stream-no" => stream_no,
                   "peer-addr" => stream.peer_addr()
                                    .expect("no peer address")
                                    .to_string()));
                let writer = BufWriter::new(stream.try_clone()
                                .expect("could not clone stream"));
                let reader = BufReader::new(stream);
                pool.execute(move || handle_client(log, reader,
writer));
            }
        } else {
            info!(root, "Shutting down! {:?}", stream);
        }
    }
```

The accept loop is more or less the same as in the previous server, except at the top of the loop we check `pool.active_count()` against the configured `max_connections`, reject the connection if the number of running workers is at capacity; otherwise we accept the connection and execute it against the pool. Finally, much as before, the server drains connections when the incoming socket is closed:

```
    info!(
        server,
        "No more incoming connections. \
         Draining existing connections."
    );
    pool.join();
}
```

The client presented in the previous section can be applied to this server without any changes. If the reader inspects the book's source repository, they will find that the clients are identical between projects.

This server implementation does not suffer from an ever-growing `JoinHandle` vector, as the last server did. The pool takes care to remove panicked threads from its internal buffers and we take care to never

allow more workers than there are threads available to work them. Let's find out how that works!

Looking into thread pool

Let's look into threadpool and understand its implementation. Hopefully by this point in the book, you have a sense of how you'd go about building your own thread-pooling library. Consider that for a moment, before we continue, and see how well your idea stacks up against this particular approach. We'll inspect the library (`https://crates.io/crates/threadpool`) at SHA `a982e060ea2e3e85113982656014b212c5b88ba2`.

> *The thread pool crate is not listed in its entirety. You can find the full listing in the book's source repository.*

Let's look first at the project's `Cargo.toml`:

```
[package]

name = "threadpool"
version = "1.7.1"
authors = ["The Rust Project Developers", "Corey Farwell
<coreyf@rwell.org>", "Stefan Schindler <dns2utf8@estada.ch>"]
license = "MIT/Apache-2.0"
readme = "README.md"
repository = "https://github.com/rust-threadpool/rust-threadpool"
homepage = "https://github.com/rust-threadpool/rust-threadpool"
documentation = "https://docs.rs/threadpool"
description = """
A thread pool for running a number of jobs on a fixed set of worker
threads.
"""
```

```
keywords = ["threadpool", "thread", "pool", "threading", "parallelism"]
categories = ["concurrency", "os"]

[dependencies]
num_cpus = "1.6"
```

Fairly minimal. The only dependency is `num_cpus`, a little library to determine the number of logical and physical cores available on the machine. On linux, this reads `/proc/cpuinfo`. On other operating systems, such as the BSDs or Windows, the library makes system calls. It's a clever little library and well worth reading if you need to learn how to target distinct function implementations across OSes. The key thing to take away from the threadpool's `Cargo.toml` is that it is almost entirely built from the tools available in the standard library. In fact, there's only a single implementation file in the library, `src/lib.rs`. All the code we'll discuss from this point can be found in that file.

Now, let's understand the builder pattern we saw in the previous section. The `Builder` type is defined like so:

```
#[derive(Clone, Default)]
pub struct Builder {
    num_threads: Option<usize>,
    thread_name: Option<String>,
    thread_stack_size: Option<usize>,
}
```

We only populated `num_threads` in the previous section. `thread_stack_size` is used to control the stack size of pool threads. As of writing, thread stack sizes are by default two megabytes. Standard library's `std::thread::Builder::stack_size` allows us to manually set this value. We could have, for instance, in our thread-per-connection example, set the stack size significantly lower, netting us more threads on the same hardware. After all, each thread allocated very little storage, especially if we had taken steps to read only into a fixed buffer. The `thread_name` field, like the stack size, is a toggle for `std::thread::Builder::name`. That is, it allows the user to set the thread name for all threads in the pool, a name they will all share. In my experience, naming threads is a relatively rare thing to do, especially in a pool, but it can sometimes be useful for logging purposes.

The pool builder works mostly as you'd expect, with the public functions setting the fields in the `Builder` struct. Where the implementation gets interesting is `Builder::build`:

```
    pub fn build(self) -> ThreadPool {
        let (tx, rx) = channel::<Thunk<'static>>();

        let num_threads =
self.num_threads.unwrap_or_else(num_cpus::get);

        let shared_data = Arc::new(ThreadPoolSharedData {
            name: self.thread_name,
            job_receiver: Mutex::new(rx),
            empty_condvar: Condvar::new(),
            empty_trigger: Mutex::new(()),
            join_generation: AtomicUsize::new(0),
            queued_count: AtomicUsize::new(0),
            active_count: AtomicUsize::new(0),
            max_thread_count: AtomicUsize::new(num_threads),
            panic_count: AtomicUsize::new(0),
            stack_size: self.thread_stack_size,
        });

        // Threadpool threads
        for _ in 0..num_threads {
            spawn_in_pool(shared_data.clone());
        }

        ThreadPool {
            jobs: tx,
            shared_data: shared_data,
        }
    }
```

There's a lot going on here. Let's take it a bit at a time. First, that channel is the `std::sync::mpsc::channel` we discussed at length in `Chapter 4`, *Sync and Send – the Foundation of Rust Concurrency*. What is unfamiliar there is `Thunk`. Turns out, it's a type synonym:

```
type Thunk<'a> = Box<FnBox + Send + 'a>;
```

Now, what is `FnBox`? It's a small trait:

```
trait FnBox {
    fn call_box(self: Box<Self>);
}

impl<F: FnOnce()> FnBox for F {
    fn call_box(self: Box<F>) {
        (*self)()
    }
}
```

It is the `FnOnce` trait we encountered in `Chapter 3`, *The Rust Memory Model – Ownership, References and Manipulation*, so if you read that chapter, you know that `F` will only be called once. The boxing of the function closure means its captured variables are available on the heap and don't get deallocated when the pool caller moves the closure out of scope. Great. Now, let's jump back to build and look at `shared_data`:

```
        let shared_data = Arc::new(ThreadPoolSharedData {
            name: self.thread_name,
            job_receiver: Mutex::new(rx),
            empty_condvar: Condvar::new(),
            empty_trigger: Mutex::new(()),
            join_generation: AtomicUsize::new(0),
            queued_count: AtomicUsize::new(0),
            active_count: AtomicUsize::new(0),
            max_thread_count: AtomicUsize::new(num_threads),
            panic_count: AtomicUsize::new(0),
            stack_size: self.thread_stack_size,
        });
```

Alright, several atomic usizes, a condvar, a mutex to protect the receiver side of the thunk channel, and a mutex that'll be paired with the condvar. There's a fair bit you can tell from reading the population of a struct, with a little bit of background information. The implementation of `ThreadPoolSharedData` is brief:

```
impl ThreadPoolSharedData {
    fn has_work(&self) -> bool {
```

```
        self.queued_count.load(Ordering::SeqCst) > 0 ||
        self.active_count.load(Ordering::SeqCst) > 0
    }

    /// Notify all observers joining this pool if there
    /// is no more work to do.
    fn no_work_notify_all(&self) {
        if !self.has_work() {
            *self.empty_trigger
                .lock()
                .expect("Unable to notify all joining threads");
            self.empty_condvar.notify_all();
        }
    }
}
```

The `has_work` function does a sequentially consistent read of the two indicators of work, an operation that is not exactly cheap considering the two sequentially consistent reads, but implies a need for accuracy in the response. The `no_work_notify_all` function is more complicated and mysterious. Happily, the function is used in the implementation of the next chunk of `Build::build` and will help clear up that mystery. The next chunk of `build` is:

```
        for _ in 0..num_threads {
            spawn_in_pool(shared_data.clone());
        }

        ThreadPool {
            jobs: tx,
            shared_data: shared_data,
        }
    }
}
```

For each of the `num_threads`, the `spawn_in_pool` function is called over a clone of the `Arc`ed `ThreadPoolSharedData`. Let's inspect `spawn_in_pool`:

```
fn spawn_in_pool(shared_data: Arc<ThreadPoolSharedData>) {
    let mut builder = thread::Builder::new();
    if let Some(ref name) = shared_data.name {
```

```
        builder = builder.name(name.clone());
    }
    if let Some(ref stack_size) = shared_data.stack_size {
        builder = builder.stack_size(stack_size.to_owned());
    }
```

As you might have expected, the function creates `std::thread::Builder` and pulls references to the name and stack size embedded in the `ThreadPoolSharedData` shared data. With these variables handy, the `builder.spawn` is called, passing in a closure in the usual fashion:

```
    builder
        .spawn(move || {
            // Will spawn a new thread on panic unless it is cancelled.
            let sentinel = Sentinel::new(&shared_data);
```

Well, let's take a look at `Sentinel`:

```
struct Sentinel<'a> {
    shared_data: &'a Arc<ThreadPoolSharedData>,
    active: bool,
}
```

It holds a reference to `Arc<ThreadPoolSharedData>`—avoiding increasing the reference counter—and an `active` boolean. The implementation is likewise brief:

```
impl<'a> Sentinel<'a> {
    fn new(shared_data: &'a Arc<ThreadPoolSharedData>) -> Sentinel<'a>
{
        Sentinel {
            shared_data: shared_data,
            active: true,
        }
    }

    /// Cancel and destroy this sentinel.
    fn cancel(mut self) {
        self.active = false;
```

```
    }
}
```

The real key here is the `Drop` implementation:

```
impl<'a> Drop for Sentinel<'a> {
    fn drop(&mut self) {
        if self.active {
            self.shared_data.active_count.fetch_sub(1,
Ordering::SeqCst);
            if thread::panicking() {
                self.shared_data.panic_count
                    .fetch_add(1, Ordering::SeqCst);
            }
            self.shared_data.no_work_notify_all();
            spawn_in_pool(self.shared_data.clone())
        }
    }
}
```

Recall how in the previous section, our join vector grew without bound, even though threads in that vector had panicked and were no longer working. There is an interface available to determine whether the current thread is in a panicked state, `std::thread::panicking()`, in use here.

When a thread panics, it drops its storage, which, in this pool, includes the allocated `Sentinel`. `drop` then checks the `active` flag on the `Sentinel`, decrements the `active_count` of the `ThreadPoolSharedData`, increases its `panic_count`, calls the as-yet mysterious `no_work_notify_all` and then adds an additional thread to the pool. In this way, the pool maintains its appropriate size and there is no need for any additional monitoring of threads to determine when they need to be recycled: the type system does all the work.

Let's hop back into `spawn_in_pool`:

```
            loop {
                // Shutdown this thread if the pool has become smaller
                let thread_counter_val = shared_data
                                         .active_count
                                         .load(Ordering::Acquire);
                let max_thread_count_val = shared_data
                                         .max_thread_count
                                         .load(Ordering::Relaxed);
                if thread_counter_val >= max_thread_count_val {
                    break;
                }
```

Here, we see the start of the infinite loop of the `builder.spawn` function, plus a check to shut down threads if the pool size has decreased since the last iteration of the loop. `ThreadPool` exposes the `set_num_threads` function to allow changes to the pool's size at runtime. Now, why an infinite loop? Spawning a new thread is not an entirely fast operation on some systems and, besides, it's not a free operation. If you can avoid the cost, you may as well. Some pool implementations in other languages spawn a new thread for every bit of work that comes in, fearing memory pollution. This is less a problem in Rust, where unsafe memory access has to be done intentionally and `FnBox` is effectively a trap for such behavior anyway, owing to the fact that the closure will have no pointers to the pool's private memory. The loop exists to pull `Thunks` from the receiver channel side in `ThreadPoolSharedData`:

```
                let message = {
                    // Only lock jobs for the time it takes
                    // to get a job, not run it.
                    let lock = shared_data
                        .job_receiver
                        .lock()
                        .expect("Worker thread unable to \
                                 lock job_receiver");
                    lock.recv()
                };
```

The message may be an error, implying that the `ThreadPool` was dropped, closing the sender channel side. But, should the message be `Ok`, we'll have our `FnBox` to call:

```
                let job = match message {
                    Ok(job) => job,
                    // The ThreadPool was dropped.
                    Err(..) => break,
                };
```

The final bit of `spawn_in_pool` is uneventful:

```
                shared_data.active_count.fetch_add(1,
Ordering::SeqCst);
                shared_data.queued_count.fetch_sub(1,
Ordering::SeqCst);

                job.call_box();

                shared_data.active_count.fetch_sub(1,
Ordering::SeqCst);
                shared_data.no_work_notify_all();
            }

            sentinel.cancel();
        })
        .unwrap();
}
```

The `FnBox` called job is called via `call_box` and if this panics, killing the thread, the `Sentinel` cleans up the atomic references as appropriate and starts a new thread in the pool. By leaning on Rust's type system and memory model, we get a cheap thread-pool implementation that spawns threads only when needed, with no fears of accidentally polluting memory between jobs.

`ThreadPool::execute` is a quick boxing of `FnOnce`, pushed into the sender side of the `Thunk` channel:

```
    pub fn execute<F>(&self, job: F)
    where
        F: FnOnce() + Send + 'static,
    {
        self.shared_data.queued_count.fetch_add(1, Ordering::SeqCst);
        self.jobs
            .send(Box::new(job))
            .expect("ThreadPool::execute unable to send job into
queue.");
    }
```

The last piece here is `ThreadPool::join`. This is where `ThreadPoolSharedData::no_work_notify_all` comes into focus. Let's look at `join`:

```
    pub fn join(&self) {
        // fast path requires no mutex
        if self.shared_data.has_work() == false {
            return ();
        }

        let generation = self.shared_data
                             .join_generation
                             .load(Ordering::SeqCst);
        let mut lock = self.shared_data.empty_trigger.lock().unwrap();

        while generation == self.shared_data
                                .join_generation
                                .load(Ordering::Relaxed)
            && self.shared_data.has_work()
        {
            lock = self.shared_data.empty_condvar.wait(lock).unwrap();
        }

        // increase generation if we are the first thread to
        // come out of the loop
        self.shared_data.join_generation.compare_and_swap(
            generation,
            generation.wrapping_add(1),
            Ordering::SeqCst,
        );
    }
```

The function calls first to `has_work`, bailing out early if there are no active or queued threads in the pool. No reason to block the caller there. Then, `generation` is set up to act as a condition variable in the loop surrounding `empty_condvar`. Every thread that joins to the pool checks that the pool `generation` has not shifted, implying some other thread has unjoined, and that there's work yet to do. Recall that it's `no_work_notify_all` that calls `notify_all` on condvar, this function in turn being called when either a `Sentinel` drops or the inner-loop of `spawn_in_pool` returns from a job. Any joined thread waking on those two conditions—a job being completed or crashing—checks their condition, incrementing the `generation` on their the way to becoming unjoined.

That's it! That's a thread pool, a thing built out of the pieces we've discussed so far in this book. If you wanted to make a thread pool without queuing, you'd push an additional check into `execute`. Some of the sequentially consistent atomic operations could likely be relaxed, as well, as potentially the consequence of making a simple implementation more challenging to reason with. It's potentially worth it for the performance gain if your executed jobs are very brief. In fact, we'll discuss a library later in this chapter that ships an alternative thread pool implementation for just such a use case, though it is quite a bit more complex than the one we've just discussed.

The Ethernet sniffer

Of equal importance to understanding a technique in-depth is understanding when not to apply it. Let's consider another thread-per-unit-of-work system, but this time we'll be echoing Ethernet packets rather than lines received over a TCP socket. Our project's `Cargo.toml`:

```
[package]
name = "sniffer"
version = "0.1.0"
authors = ["Brian L. Troutwine <brian@troutwine.us>"]

[dependencies]
pnet = "0.21"

[[bin]]
name = "sniffer"

[[bin]]
name = "poor_threading"
```

Like the TCP example, we'll create two binaries, one that is susceptible to thread overload–`poor_threading`–and another–`sniffer`–that is not. The premise here is that we want to sniff a network interface for Ethernet packets, reverse the source and destination headers on that packet, and send the modified packet back to the original source. Simultaneously, we'll collect summary statistics of the packets we collect. On a saturated network, our little programs will be very busy and there will have to be trade-offs made somewhere in terms of packet loss and receipt.

The only dependency we're pulling in here is `pnet`. libpnet is a low-level packet manipulation library, useful for building network utilities or for prototyping transport protocols. I think it's pretty fun to fuzz-

test transport implementations with libpnet on commodity network devices. Mostly, though, the reward is a crashed home router, but I find that amusing. Anyhow, let's look into `poor_threading`. It's preamble is fairly ho-hum:

```
extern crate pnet;

use pnet::datalink::Channel::Ethernet;
use pnet::datalink::{self, DataLinkReceiver, DataLinkSender,
                     MacAddr, NetworkInterface};
use pnet::packet::ethernet::{EtherType, EthernetPacket,
                             MutableEthernetPacket};
use pnet::packet::{MutablePacket, Packet};
use std::collections::HashMap;
use std::sync::mpsc;
use std::{env, thread};
```

We pull in a fair bit of pnet's facilities, which we'll discuss in due time. We don't pull in clap or similar argument-parsing libraries, instead requiring that the user pass in the interface name as the first argument to the program:

```
fn main() {
    let interface_name = env::args().nth(1).unwrap();
    let interface_names_match = |iface: &NetworkInterface| {
        iface.name == interface_name
    };

    // Find the network interface with the provided name
    let interfaces: Vec<NetworkInterface> = datalink::interfaces();
    let interface = interfaces
        .into_iter()
        .filter(interface_names_match)
        .next()
        .unwrap();
```

The user-supplied interface name is checked against the interfaces pnet is able to find, via its `datalink::interfaces()` function. We haven't done much yet in this book with iterators, though they've been omnipresent and assumed knowledge, at least minimally. We'll

discuss iteration in Rust in detail after this. Note here, though, that `filter(interface_names_match)` is applying a boolean function, `interface_names_match(&NetworkInterface) -> bool`, to each member of the determined interfaces.

If the function returns true, the member passes through the filter, otherwise it doesn't. The call to `next().unwrap()` cranks the filtered iterator forward one item, crashing if there are no items in the filtered iterator. This is a somewhat user-hostile way to determine whether the passed interface is, in fact, an interface that pnet could discover. That's alright here in our demonstration program.

Next, we establish a communication channel for the concurrent actors of this program:

```
    let (snd, rcv) = mpsc::sync_channel(10);

    let _ = thread::spawn(|| gather(rcv));
    let timer_snd = snd.clone();
    let _ = thread::spawn(move || timer(timer_snd));
```

The `timer` function pushes a time pulse through the channel we just established at regular intervals, as similarly is done in cernan, discussed previously in this book. The function is small:

```
fn timer(snd: mpsc::SyncSender<Payload>) -> () {
    use std::{thread, time};
    let one_second = time::Duration::from_millis(1000);

    let mut pulses = 0;
    loop {
        thread::sleep(one_second);
        snd.send(Payload::Pulse(pulses)).unwrap();
        pulses += 1;
    }
}
```

The `Payload` type is an enum with only two variants:

```
enum Payload {
    Packet {
        source: MacAddr,
        destination: MacAddr,
        kind: EtherType,
    },
    Pulse(u64),
}
```

The `Pulse(u64)` variant is the timer pulse, sent periodically by the `timer` thread. It's very useful to divorce time in a concurrent system from the actual wall-clock, especially with regard to testing individual components of the system. It's also helpful for the program structure to unify different message variants in a union. At the time of writing, Rust's MPSC implementation does not have a stable select capability, and so you'll have to manually implement that with `mpsc::Receiver::recv_timeout` and careful multiplexing. It's much better to unify it into one union type. Additionally, this gets the type system on your side, confirming that all incoming variants are handled in a single match, which likely optimizes better, too. This last benefit should be measured.

Let's look at `gather` now:

```
fn gather(rcv: mpsc::Receiver<Payload>) -> () {
    let mut sources: HashMap<MacAddr, u64> = HashMap::new();
    let mut destinations: HashMap<MacAddr, u64> = HashMap::new();
    let mut ethertypes: HashMap<EtherType, u64> = HashMap::new();

    while let Ok(payload) = rcv.recv() {
        match payload {
            Payload::Pulse(id) => {
                println!("REPORT {}", id);
                println!("    SOURCES:");
                for (k, v) in sources.iter() {
                    println!("        {}: {}", k, v);
                }
                println!("    DESTINATIONS:");
                for (k, v) in destinations.iter() {
                    println!("        {}: {}", k, v);
```

```
                }
                println!("    ETHERTYPES:");
                for (k, v) in ethertypes.iter() {
                    println!("        {}: {}", k, v);
                }
            }
            Payload::Packet {
                source: src,
                destination: dst,
                kind: etype,
            } => {
                let mut destination =
destinations.entry(dst).or_insert(0);
                *destination += 1;

                let mut source = sources.entry(src).or_insert(0);
                *source += 1;

                let mut ethertype =
ethertypes.entry(etype).or_insert(0);
                *ethertype += 1;
            }
        }
    }
}
```

Straightforward receiver loop, pulling off `Payload` enums. The `Payload::Packet` variant is deconstructed and its contents stored into three `HashMap`s, mapping MAC addresses to a counter or an EtherType to a counter. MAC addresses are the unique identifiers of network interfaces—or, ideally unique, as there have been goofs—and get used at the data-link layer of the OSI model. (This is not a book about networking but it is, I promise, a really fun domain.) EtherType maps to the two octet fields at the start of every Ethernet packet, defining the packet's, well, type. There's a standard list of types, which pnet helpfully encodes for us. When `gather` prints the EtherType, there's no special work needed to get human-readable output.

Now that we know how `gather` and `timer` work, we can wrap up the `main` function:

```
    let iface_handler = match datalink::channel(&interface,
                                                Default::default()) {
        Ok(Ethernet(tx, rx)) => {
            let snd = snd.clone();
            thread::spawn(|| watch_interface(tx, rx, snd))
        }
        Ok(_) => panic!("Unhandled channel type"),
        Err(e) => panic!(
            "An error occurred when creating the datalink channel: {}",
            e
        ),
    };

    iface_handler.join().unwrap();
}
```

The `datalink::channel` function establishes an MPSC-like channel for bi-directional packet reading and writing on the given interface. We only care about Ethernet packets here and match only on that variant. We spawn a new thread for `watch_interface,` which receives both read/write sides of the channel and the MPSC we made for `Payload.` In this way, we have one thread reading Ethernet packets from the network, stripping them into a normalized `Payload::Packet` and pushing them to the `gather` thread. Meanwhile, we have another `timer` thread pushing `Payload::Pulse` at regular intervals to the `gather` thread to force user reporting.

The sharp-eyed reader will have noticed that our `Payload` channel is actually `sync_channel(10)`, meaning that this program is not meant to store much transient information in memory. On a busy Ethernet network, this will mean it's entirely possible that `watch_interface` will be unable to push a normalized Ethernet packet into the channel. Something will have to be done and it all depends on how willing we are to lose information.

Let's look and see how this implementation goes about addressing this problem:

```
fn watch_interface(
    mut tx: Box<DataLinkSender>,
    mut rx: Box<DataLinkReceiver>,
    snd: mpsc::SyncSender<Payload>,
) {
    loop {
        match rx.next() {
            Ok(packet) => {
                let packet = EthernetPacket::new(packet).unwrap();
```

So far, so good. We see the sides of `datalink::Channel` that we discussed earlier, plus our internal MPSC. The infinite loop reads `&[u8]` off the receive side of the channel and our implementation has to convert this into a proper EthernetPacket. If we were being very thorough, we'd guard against malformed Ethernet packets but, since we aren't, the creation is unwrapped. Now, how about that normalization into `Payload::Packet` and transmission to `gather`:

```
                {
                    let payload: Payload = Payload::Packet {
                        source: packet.get_source(),
                        destination: packet.get_destination(),
                        kind: packet.get_ethertype(),
                    };
                    let thr_snd = snd.clone();
                    thread::spawn(move || {
                        thr_snd.send(payload).unwrap();
                    });
                }
```

Uh oh. The EthernetPacket has normalized into `Payload::Packet` just fine, but when we send the packet down the channel to `gather` we do so by spawning a thread. The decision being made here is that no incoming packet should be lost—implying we have to read them off the network interface as quickly as possible—and if the channel blocks, we'll need to store that packet in some pool, somewhere.

The *pool* is, in fact, a thread stack. Or, a bunch of them. Even if we were to drop the thread's stack size to a sensible low size on a

saturated network, we're still going to overwhelm the operating system at some point. If we absolutely could not stand to lose packets, we could use something such as hopper (`https://github.com/postmates/hopper`), discussed in detail in `Chapter 5`, *Locks – Mutex, Condvar, Barriers and RWLock*, which was designed for just this use case. Or, we could defer these sends to threadpool, allow it to queue the jobs, and run them on a finite number of threads. Anyway, let's set this aside for just a second and wrap up `watch_interface`:

```
                tx.build_and_send(1, packet.packet().len(),
                                  &mut |new_packet|
                {
                    let mut new_packet =

MutableEthernetPacket::new(new_packet).unwrap();

                    // Create a clone of the original packet
                    new_packet.clone_from(&packet);

                    // Switch the source and destination
                    new_packet.set_source(packet.get_destination());
                    new_packet.set_destination(packet.get_source());
                });
            }
            Err(e) => {
                panic!("An error occurred while reading: {}", e);
            }
        }
    }
}
```

The implementation takes the newly created EthernetPacket, swaps the source and destination of the original in a new EthernetPacket, and sends it back across the network at the original source. Now, let's ask ourselves, is it *really* important that we tally every Ethernet packet we can pull off the network? We could speed up the `HashMap`s in the `gather` thread in the fashion described in `Chapter 02`, *Sequential Rust Performance and Testing*, by inserting a faster hasher. Or, we could buffer in the `watch_interface` thread when the synchronous channel is full, in addition to using threadpool or hopper. Only speeding up

`gather` and using hopper don't potentially incur unbounded storage requirements. But hopper will require a filesystem and may still not be able to accept all incoming packets, only making it less likely that there won't be sufficient storage.

It's a tough problem. Given the nature of the network at layer 2, you might as well just shed the packets. The network card itself is going to be shedding packets. With that in mind, the implementation of `watch_interface` in the other binary in this project—`sniffer`—differs only marginally:

```
fn watch_interface(
    mut tx: Box<DataLinkSender>,
    mut rx: Box<DataLinkReceiver>,
    snd: mpsc::SyncSender<Payload>,
) {
    loop {
        match rx.next() {
            Ok(packet) => {
                let packet = EthernetPacket::new(packet).unwrap();

                let payload: Payload = Payload::Packet {
                    source: packet.get_source(),
                    destination: packet.get_destination(),
                    kind: packet.get_ethertype(),
                };
                if snd.try_send(payload).is_err() {
                    SKIPPED_PACKETS.fetch_add(1, Ordering::Relaxed);
                }
```

`Payload::Packet` is created and rather than calling `snd.send`, this implementation calls `snd.try_send`, ticking up a `SKIPPED_PACKETS` static `AtomicUsize` in the event a packet has to be shed. The gather implementation is likewise only slightly adjusted to report on this new `SKIPPED_PACKETS`:

```
fn gather(rcv: mpsc::Receiver<Payload>) -> () {
    let mut sources: HashMap<MacAddr, u64> = HashMap::new();
    let mut destinations: HashMap<MacAddr, u64> = HashMap::new();
    let mut ethertypes: HashMap<EtherType, u64> = HashMap::new();
```

```
    while let Ok(payload) = rcv.recv() {
        match payload {
            Payload::Pulse(id) => {
                println!("REPORT {}", id);
                println!(
                    "    SKIPPED PACKETS: {}",
                    SKIPPED_PACKETS.swap(0, Ordering::Relaxed)
                );
```

This program will use a moderate amount of storage for the `Payload` channel, no matter how busy the network.

In high-traffic domains or long-lived deployments, the `HashMap`s in the `gather` thread are going to be a concern, but this is a detail the intrepid reader is invited to address.

When you run `sniffer`, you should be rewarded with output very much like this:

```
REPORT 6
    SKIPPED PACKETS: 75
    SOURCES:
        ff:ff:ff:ff:ff:ff: 1453
        00:23:6a:00:51:6e: 2422
        5c:f9:38:8b:4a:b6: 1034
    DESTINATIONS:
        33:33:ff:0b:62:e8: 1
        33:33:ff:a1:90:82: 1
        33:33:00:00:00:01: 1
        00:23:6a:00:51:6e: 2414
        5c:f9:38:8b:4a:b6: 1032
        ff:ff:ff:ff:ff:ff: 1460
    ETHERTYPES:
        Ipv6: 4
        Ipv4: 1999
        Arp: 2906
```

This report came from the sixth second of my sniffer run and 75 packets were dropped for lack of storage. That's better than 75 threads.

Iterators

So far, we've assumed that the reader has been at least passingly familiar with Rust iterators. It's possible that you've used them extensively but have never written your own iterator implementation. That knowledge is important for what follows and we'll discuss Rust's iteration facility now.

A Rust iterator is any type that implements `std::iter::Iterator`. The `Iterator` trait has an interior `Item` type that allows the familiar iterator functions, such as `next(&mut self) -> Option<Self::Item>`, to be defined in terms of that generic `Item`. Many of the iterator functions are, well, functional in nature: `map`, `filter`, and `fold` are all higher-order functions on a stream of `Item`. Many of the iterator functions are searches on that stream of `Item`: `max`, `min`, `position`, and `find`. It is a versatile trait. If you find it limited in some manner, the community has put together a crate with more capability: itertools (`https://crates.io/crates/itertools`). Here, we're less concerned with the interface of `Iterator` than how to implement it for our own types.

Smallcheck iteration

In `Chapters 2`, *Sequential Rust Performance and Testing*, `Chapters 5`, *Locks – Mutex, Condvar, Barriers and RWLock*, and `Chapters 6`, *Atomics – the Primitives of Synchronization*, we've discussed the QuickCheck testing methodology, a structured way of inserting random, type-driven input into a function to search function property failures. Inspired by the work done by Claessen and Hughes in their QuickCheck paper, Runciman, Naylor and Lindblad observed that, in their experience, most failures were on small input and put forward the observation that a library that tested from small output to big first might find failures faster than purely random methods. Their hypothesis was more or less correct, for a certain domain of function, but the approach suffers from duplication of effort, resolved somewhat by the authors in the same paper with Lazy Smallcheck. We won't go into further detail here, but the paper is included in the *Further reading* section of this chapter and the reader is encouraged to check out the paper.

Anyway, producing all values of a *small* to *big* type sounds like iteration. In fact, it is. Let's walk through producing iterators for signed and unsigned integers. The project's `Cargo.toml` is minimal:

```
[package]
name = "smalliters"
version = "0.1.0"
authors = ["Brian L. Troutwine <brian@troutwine.us>"]

[dependencies]
```

No dependencies, just the default that `cargo new` lays down. The project has only one file, `src/lib.rs`. The rest of our discussion will take place

in the context of this file. First, the preamble:

```
use std::iter::Iterator;
use std::mem;
```

We pull in `Iterator`, as you might expect, and then `std::mem`, which we've seen throughout this book. Nothing unusual here. What is unusual for this book is this:

```
macro_rules! unsized_iter {
    ($name:ident, $int:ty, $max:expr) => {
```

A macro! We've been using macros throughout the entire book but have not written one yet. Macros in Rust are kind of a dark art and a touch on the fidgety side to get working. Our ambitions here are simple. The iterators for the Rust integer types are mostly the same, varying by signed/unsigned and total bit size of the type, hence `std::mem`. `($name:ident, $int:ty, and $max:expr)` declares three macro variables: `$name`, which is an identifier, `$int`, which is a type, and `$max`, which is an expression. Here is how we'll use the `unsigned_iter!` macro:

```
unsized_iter!(SmallU8, u8, u8::max_value());
unsized_iter!(SmallU16, u16, u16::max_value());
unsized_iter!(SmallU32, u32, u32::max_value());
unsized_iter!(SmallU64, u64, u64::max_value());
unsized_iter!(SmallUsize, usize, usize::max_value());
```

That's an identifier, followed by a type, followed by an expression. OK, now back to the macro definition:

```
macro_rules! unsized_iter {
    ($name:ident, $int:ty, $max:expr) => {
        #[derive(Default)]
        pub struct $name {
            cur: $int,
            done: bool,
        }
```

The first step in defining a small iterator for integer types is to produce some struct to store whatever data the Iterator implementation needs. We require two things, the current integer and a boolean flag to tell us when we're done iterating. When the macros expand, there will be structs called `SmallU8`, `SmallU16`, and so on in the expanded source code, like this:

```
#[derive(Default)]
pub struct SmallU8 {
    cur: u8,
    done: bool,
}
```

The `Iterator` implementation for unsigned types is straightforward: keep incrementing until you hit the maximum value for the type:

```
        impl Iterator for $name {
            type Item = $int;

            fn next(&mut self) -> Option<$int> {
                if self.done {
                    return None;
                }
                let old = self.cur;
                if old == $max {
                    self.done = true;
                }
                self.cur = self.cur.saturating_add(1);
                return Some(old);
            }
```

Technically, `saturating_add` is not needed and could be replaced with `+=`, but it's helpful to have to avoid wrap-around bugs. Finally, we also provide an `Iterator::size_hint` implementation:

```
            fn size_hint(&self) -> (usize, Option<usize>) {
                let size = mem::size_of::<$int>() * 8;
                let total = 2_usize.pow(size as u32);
```

```
                let remaining = total - (self.cur as usize);
                (remaining, Some(remaining))
            }
        }
    };
}
```

This function is an optional implementation, but it is neighborly to provide one. The return tuple estimates how many items remain in the iteration, the first element being a lower estimate and the second being an optional upper bound. We happen to know exactly how many elements remain to be iterated because our domain is finite. Implementing `Iterator::size_hint` will help out code with pre-allocating buffers, which includes other functions on `Iterator`.

The sized integer types have a similar implementation. Their macro, `sized_iter!`, in use:

```
sized_iter!(SmallI8, i8, i8::min_value());
sized_iter!(SmallI16, i16, i16::min_value());
sized_iter!(SmallI32, i32, i32::min_value());
sized_iter!(SmallI64, i64, i64::min_value());
sized_iter!(SmallIsize, isize, isize::min_value());
```

The macro itself starts out in the same way as its unsigned counterpart:

```
macro_rules! sized_iter {
    ($name:ident, $int:ty, $min:expr) => {
        #[derive(Default)]
        pub struct $name {
            cur: $int,
            done: bool,
        }

        impl Iterator for $name {
            type Item = $int;

            fn next(&mut self) -> Option<$int> {
                if self.done {
```

```
                    return None;
                }
```

Only the inner part of `next` is different:

```
                let old = self.cur;
                if self.cur == 0 {
                    self.cur = -1;
                } else if self.cur.is_negative() && self.cur == $min {
                    self.done = true;
                } else if self.cur.is_positive() {
                    self.cur *= -1;
                    self.cur -= 1;
                } else if self.cur.is_negative() {
                    self.cur *= -1;
                }
                return Some(old);
            }
```

This warrants some explanation. We take inspiration from the SmallCheck paper and iterate values out like so: 0, -1, 1, -2, 2, and so on. The implementation could follow the same method as the unsigned variant, proceeding up from `min_value()` to `max_value()`, but this would not be small to big, at least in terms of byte representation. Finally, `size_hint`:

```
            fn size_hint(&self) -> (usize, Option<usize>) {
                let size = mem::size_of::<$int>() * 8;
                let total = 2_usize.pow(size as u32);
                let remaining = total - ((self.cur.abs() * 2) as
usize);
                (remaining, Some(remaining))
            }
        }
    };
}
```

If these implementations work correctly, the `Iterator::count` for our new iterators ought to be equal to the two-power of the bit size of the type, `count` being a function that keeps a tally of elements iterated out.

The tests also rely on macros to cut down on duplication. First, the signed variants:

```
#[cfg(test)]
mod tests {
    use super::*;

    macro_rules! sized_iter_test {
        ($test_name:ident, $iter_name:ident, $int:ty) => {
            #[test]
            fn $test_name() {
                let i = $iter_name::default();
                let size = mem::size_of::<$int>() as u32;
                assert_eq!(i.count(), 2_usize.pow(size * 8));
            }
        };
    }

    sized_iter_test!(i8_count_correct, SmallI8, i8);
    sized_iter_test!(i16_count_correct, SmallI16, i16);
    sized_iter_test!(i32_count_correct, SmallI32, i32);
    // NOTE: These take forever. Uncomment if you have time to burn.
    // sized_iter_test!(i64_count_correct, SmallI64, i64);
    // sized_iter_test!(isize_count_correct, SmallIsize, isize);
```

And now the unsigned variants:

```
    macro_rules! unsized_iter_test {
        ($test_name:ident, $iter_name:ident, $int:ty) => {
            #[test]
            fn $test_name() {
                let i = $iter_name::default();
                let size = mem::size_of::<$int>() as u32;
                assert_eq!(i.count(), 2_usize.pow(size * 8));
            }
        };
    }

    unsized_iter_test!(u8_count_correct, SmallU8, u8);
    unsized_iter_test!(u16_count_correct, SmallU16, u16);
    unsized_iter_test!(u32_count_correct, SmallU32, u32);
    // NOTE: These take forever. Uncomment if you have time to burn.
    // unsized_iter_test!(u64_count_correct, SmallU64, u64);
```

```
    // unsized_iter_test!(usize_count_correct, SmallUsize, usize);
}
```

If you run the tests, you should find that they pass. It'll take a minute for the larger types. But, that's iterators. All they are is some kind of base type—in this case, the built-in integers—plus a state-tracking type—our `Small*` structs—and an implementation of `Iterator` for that state-tracking type. Remarkably useful, and the optimizer is reasonably good at turning iterator chains into loops and getting better at it all the time.

rayon – parallel iterators

In the previous section, we built an iterator over Rust integer types. The essential computation of each `next` call was small—some branch checks, and possibly a negation or an addition. If we were to try to parallelize this, we'd drown out any potential performance bump with the time it takes to allocate a new thread from the operating systems. No matter how fast that becomes, it will still be slower than an addition. But, say we've gone all-in on writing in iteration style and do have computations that would benefit from being run in parallel. How do you make `std::iter::Iterator` parallel?

You use rayon.

The rayon crate is a *data-parallelism* library for Rust. That is, it extends Rust's basic `Iterator` concept to include a notion of implicit parallelism. Yep, implicit. Rust's memory safety means we can pull some pretty fun tricks on modern machines. Consider that the thread pool implementation we investigated previously in the chapter, which had no concerns for memory pollution. Well, it's the same deal with rayon. Every element in an iterator is isolated from each other, an invariant enforced by the type system. Which means, the rayon developers can focus on building a data-parallelism library to be as fast as possible while we, the end users, can focus on structuring out code in an iterator style.

We'll examine rayon (`https://crates.io/crates/rayon`) at SHA `5f98c019abc4d40697e83271c072cb2b92966f46`. The rayon project is split into sub-crates:

- rayon-core

- rayon-futures

We'll discuss rayon—the top-level crate—and rayon-core in this chapter, touching on the fundamental technology behind rayon-futures in `Chapter 10`, *Futurism – Near-Term Rust*. The interested reader can flag rayon-futures on to play with that interface but please do refer to the `README` at the top of that sub-crate for details.

> *The rayon crate is not listed in its entirety. You can find the full listing in this book's source repository.*

The dependencies of rayon are minimal, being that rayon-core is the sole dependency by default. The rayon-core dependencies are:

```
[dependencies]
rand = ">= 0.3, < 0.5"
num_cpus = "1.2"
libc = "0.2.16"
lazy_static = "1"
```

We've seen these dependencies before, except `libc`. This library exposes the `libc` platform in a stable interface. From the project's documentation:

"This crate does not strive to have any form of compatibility across platforms, but rather it is simply a straight binding to the system libraries on the platform in question."

It's quite useful. Now, rayon is a large library and it can be a challenge to navigate. That's okay, though, because it's simple to use. To give ourselves an in, let's pull the example code from rayon's `README` and do an exploration from that point. The example is:

```
use rayon::prelude::*;

fn sum_of_squares(input: &[i32]) -> i32 {
    input.par_iter()
         .map(|&i| i * i)
```

```
        .sum()
}
```

Okay, so what's going on here? Well, let's first compare it against the sequential version of this example:

```
fn sum_of_squares(input: &[i32]) -> i32 {
    input.iter()
         .map(|&i| i * i)
         .sum()
}
```

Spot the key difference? The sequential version is `input.iter()`, the parallel version is `input.par_iter()`. Okay, first, we have to understand that every slice exposes `iter(&self) -> std::slice::Iter`. This `Iter` implements `std::iter::Iterator`, which we recently discussed. rayon implements something similar, so that:

- `input.iter() :: std::slice::Iter<'_, u32>`
- `input.par_iter() :: rayon::slice::Iter<'_, i32>`

Let's look at `rayon::slice::Iter`, in `src/slice/mod.rs`:

```
#[derive(Debug)]
pub struct Iter<'data, T: 'data + Sync> {
    slice: &'data [T],
}
```

Okay, nothing we haven't seen so far. There's a lifetime data, and every `T` has to be part of that lifetime plus `Sync`. Now, what is the type of rayon's `map`? By inspection, we can see it's `ParallelIterator::map<F, R>(self, map_op: F) -> Map<Self, F>` where `F: Fn(Self::Item) -> R + Sync + Send` and `R: Send`. Alright, `ParallelIterator`. That is new. There's an implementation for `Iter` just under the struct definition:

```
impl<'data, T: Sync + 'data> ParallelIterator for Iter<'data, T> {
    type Item = &'data T;

    fn drive_unindexed<C>(self, consumer: C) -> C::Result
        where C: UnindexedConsumer<Self::Item>
    {
        bridge(self, consumer)
    }

    fn opt_len(&self) -> Option<usize> {
        Some(self.len())
    }
}
```

Clearly, we need to understand `ParallelIterator`. This trait is defined in `src/iter/mod.rs` and is declared to be the parallel version of the standard iterator trait. Well, good! That gives us a pretty good anchor into its semantics. Now, we just have to figure out its implementation. The implementation of `ParallelIterator` for `Iter` defined two functions: `drive_unindexed` and `opt_len`. The latter function returns the optional length of the underlying structure, much like `Iterator::size_hint`.

The rayon documentation notes that some of their functions can trigger fast-paths when `opt_len` returns a value: always a win. `drive_unindexed` is more complicated and introduces two unknowns. Let's figure out what `consumer: UnindexedConsumer` is first. The trait definition, in `src/iter/plumbing/mod.rs`, is very brief:

```
pub trait UnindexedConsumer<I>: Consumer<I> {
    /// Splits off a "left" consumer and returns it. The `self`
    /// consumer should then be used to consume the "right" portion of
    /// the data. (The ordering matters for methods like find_first –
    /// values produced by the returned value are given precedence
    /// over values produced by `self`.) Once the left and right
    /// halves have been fully consumed, you should reduce the results
    /// with the result of `to_reducer`.
    fn split_off_left(&self) -> Self;

    /// Creates a reducer that can be used to combine the results from
    /// a split consumer.
```

```
    fn to_reducer(&self) -> Self::Reducer;
}
```

Okay, we're missing some key pieces of information here. What is `Consumer`? This trait is defined in the same file, like so:

```
pub trait Consumer<Item>: Send + Sized {
    /// The type of folder that this consumer can be converted into.
    type Folder: Folder<Item, Result = Self::Result>;

    /// The type of reducer that is produced if this consumer is split.
    type Reducer: Reducer<Self::Result>;

    /// The type of result that this consumer will ultimately produce.
    type Result: Send;

    /// Divide the consumer into two consumers, one processing items
    /// `0..index` and one processing items from `index..`. Also
    /// produces a reducer that can be used to reduce the results at
    /// the end.
    fn split_at(self, index: usize) -> (Self, Self, Self::Reducer);

    /// Convert the consumer into a folder that can consume items
    /// sequentially, eventually producing a final result.
    fn into_folder(self) -> Self::Folder;

    /// Hint whether this `Consumer` would like to stop processing
    /// further items, e.g. if a search has been completed.
    fn full(&self) -> bool;
}
```

There are some key unknowns here, but the picture is getting a little clearer. `Consumer` is able to take in `Item` and subdivide it at an `index`. rayon is performing a divide and conquer, taking small chunks of the stream and dividing them over *something*. What are `Folder` and `Reducer`? This trait *folds* into itself, like the higher-order fold function. The trait definition is:

```
pub trait Folder<Item>: Sized {
    type Result;

    fn consume(self, item: Item) -> Self;
```

```
    fn consume_iter<I>(mut self, iter: I) -> Self
        where I: IntoIterator<Item = Item>
    {
        for item in iter {
            self = self.consume(item);
            if self.full() {
                break;
            }
        }
        self
    }

    fn complete(self) -> Self::Result;

    fn full(&self) -> bool;
}
```

The important function here is `consume_iter`. Note that it calls `consume` for each element in the `iter` variable, folding the previous `self` into a new `self`, only stopping once the implementor of `Folder` signals that it is *full*. Note as well that a `Folder`, once complete is called on it, is turned into a `Result`. Recall that `Result` is set in `Consumer`. Now, what is `Reducer`? It is:

```
pub trait Reducer<Result> {
    fn reduce(self, left: Result, right: Result) -> Result;
}
```

`Reducer` combines two `Result`s into one single `Result`. `Consumer` is then a type that can take an `Item`, apply `Folder` over chunks of the `Item`, and then reduce the many `Result`s down into a single `Result`. If you squint, `Consumer` *is* a fold. `UnindexedConsumer` is a specialization of `Consumer` that splits at an arbitrary point, where the plain `Consumer` requires an index. We understand, then, that the parallel iterator for the slice iterator is being driven by `drive_unindexed`. This function is passed *some* `consumer: C`, which is minimally `UnindexedConsumer` and the function entirely defers to `bridge` to do its work. Two questions: *what* is calling `drive_unindexed` and what is `bridge`? Let's look into the `bridge` function, defined in

`src/iter/plumbing/mod.rs`. The header of the function immediately presents new information:

```
pub fn bridge<I, C>(par_iter: I, consumer: C) -> C::Result
    where I: IndexedParallelIterator,
          C: Consumer<I::Item>
```

What is `IndexedParallelIterator`? I'll spare you the exploration, but it's a further specialization of `ParallelIterator` that can be *split at arbitrary indices*. This specialization has more functions than the `ParallelIterator` trait, and `rayon::slice::Iter` implements both. The definition is brief and something we'll need to know to understand bridge:

```
impl<'data, T: Sync + 'data> IndexedParallelIterator for Iter<'data, T>
{
    fn drive<C>(self, consumer: C) -> C::Result
        where C: Consumer<Self::Item>
    {
        bridge(self, consumer)
    }

    fn len(&self) -> usize {
        self.slice.len()
    }

    fn with_producer<CB>(self, callback: CB) -> CB::Output
        where CB: ProducerCallback<Self::Item>
    {
        callback.callback(IterProducer { slice: self.slice })
    }
}
```

We see `bridge` backing a function `drive` and a new `with_producer` function with an unknown purpose. Back to `bridge`:

```
{
    let len = par_iter.len();
    return par_iter.with_producer(Callback {
                                      len: len,
```

```
                                        consumer: consumer,
                                    });
```

We see `bridge` calculate the length of the underlying slice and then call `with_producer` on some `Callback` struct. We know from the trait implementation on `Iter` that `Callback: ProducerCallback`, which is itself an analogue to `FnOnce`, is taking some `T`, a `Producer<T>`, and emitting an `Output` from `ProducerCallback::callback`. If you squint hard enough, that's a closure. A `Producer` is more or less an `std::iter::IntoIterator`, but one that can be split at an index into two `Producer`s. Again, we see that rayon is dividing work into sub-pieces and that this method of operation extends through multiple types. But what is `Callback`? It turns out, this struct is defined inline with the function body of `bridge`:

```
    struct Callback<C> {
        len: usize,
        consumer: C,
    }

    impl<C, I> ProducerCallback<I> for Callback<C>
        where C: Consumer<I>
    {
        type Output = C::Result;
        fn callback<P>(self, producer: P) -> C::Result
            where P: Producer<Item = I>
        {
            bridge_producer_consumer(self.len, producer, self.consumer)
        }
    }
}
```

I admit, that's pretty weird! The `producer: P` that is passed in the interior callback is an `IterProducer`, a type defined in `src/slice/mod.rs`, which holds a reference to the slice. The interesting thing is the implementation of `Producer` for `IterProducer`:

```
impl<'data, T: 'data + Sync> Producer for IterProducer<'data, T> {
    type Item = &'data T;
    type IntoIter = ::std::slice::Iter<'data, T>;
```

```
    fn into_iter(self) -> Self::IntoIter {
        self.slice.into_iter()
    }

    fn split_at(self, index: usize) -> (Self, Self) {
        let (left, right) = self.slice.split_at(index);
        (IterProducer { slice: left }, IterProducer { slice: right })
    }
}
```

Look at that `split_at`! Whenever rayon needs to split a slice Producer, it calls `std::slice::split_at` and makes two new `Producer`s. Alright, so now we know *how* rayon is splitting things up but still not *to what*. Let's look further into the implementation of `bridge_producer_consumer`:

```
pub fn bridge_producer_consumer<P, C>(len: usize, producer: P,
                                      consumer: C) -> C::Result
    where P: Producer,
          C: Consumer<P::Item>
{
    let splitter = LengthSplitter::new(producer.min_len(),
                                       producer.max_len(), len);
    return helper(len, false, splitter, producer, consumer);
```

Okay, we are mostly familiar with these types. `LengthSplitter` is new and is a type that informs us whether a split of a certain length is valid for some minimum size. This is where rayon is able to decide how small to make split work and whether or not to split a workload further. The `helper` function rounds things out:

```
    fn helper<P, C>(len: usize,
                    migrated: bool,
                    mut splitter: LengthSplitter,
                    producer: P,
                    consumer: C)
                    -> C::Result
        where P: Producer,
              C: Consumer<P::Item>
    {
        if consumer.full() {
            consumer.into_folder().complete()
```

```
        } else if splitter.try(len, migrated) {
            let mid = len / 2;
            let (left_producer, right_producer) =
producer.split_at(mid);
            let (left_consumer, right_consumer,
                 reducer) = consumer.split_at(mid);
            let (left_result, right_result) =
                join_context(|context| {
                    helper(mid, context.migrated(), splitter,
                           left_producer, left_consumer)
                }, |context| {
                    helper(len - mid, context.migrated(), splitter,
                           right_producer, right_consumer)
                });
            reducer.reduce(left_result, right_result)
        } else {
            producer.fold_with(consumer.into_folder()).complete()
        }
    }
}
```

This is dense. Most of this is to do with splitting the current `Producer` into appropriately sized chunks, but especially this block of code:

```
                join_context(|context| {
                    helper(mid, context.migrated(), splitter,
                           left_producer, left_consumer)
                }, |context| {
                    helper(len - mid, context.migrated(), splitter,
                           right_producer, right_consumer)
                });
```

Here we see newly split `Producers` that lazily make a recursive call to `helper` when executed by `join_context`. This function is defined in `rayon-core/src/join/mod.rs`:

```
pub fn join_context<A, B, RA, RB>(oper_a: A, oper_b: B) -> (RA, RB)
    where A: FnOnce(FnContext) -> RA + Send,
          B: FnOnce(FnContext) -> RB + Send,
          RA: Send,
          RB: Send
{
```

The new type here, `FnContext`, is a `Send + Sync` disabling boolean. The boolean interior to `FnContext` is called `migrated`, an interesting clue to the context's purpose. Let's continue in `join_context`:

```
    registry::in_worker(|worker_thread, injected| unsafe {
        log!(Join { worker: worker_thread.index() });

        // Create virtual wrapper for task b; this all has to be
        // done here so that the stack frame can keep it all live
        // long enough.
        let job_b = StackJob::new(|migrated|
                                  oper_b(FnContext::new(migrated)),
                                  SpinLatch::new());
        let job_b_ref = job_b.as_job_ref();
        worker_thread.push(job_b_ref);
```

Well hey, it's a thread pool! We *know* thread pools! There's a lot more detail to rayon's thread pool implementation compared to the one we investigated earlier in this chapter, but it's a known concept. Each thread in rayon's pool maintains its own queue of work. Every thread queues up two new jobs—in this case, calls to helper with a specific `FnContext`—that may execute the function that `Consumer` and `Producer` combined represent, or split the work up into small chunks, pushing onto the thread's queue. Each thread in the pool is able to steal jobs from the others in the pool, spreading the load around the pool. In fact, what we've seen so far is that the caller of `join_context` immediately constructs `StackJob` out of `oper_b` and calls `worker_thread.push`. This function is defined in `rayon-core/src/registry.rs`:

```
    #[inline]
    pub unsafe fn push(&self, job: JobRef) {
        self.worker.push(job);
        self.registry.sleep.tickle(self.index);
```

`sleep.tickle`, in addition to being amusingly named, is meant to notify threads waiting on condvar. This condvar is tracking whether or not there's work available, saving power when there's none, rather than have threads in the pool spin-looping. What happens when

`self.worker.push` is called? It turns out, the `WorkerThread::worker` field is of the `crossbeam_deque::Deque<job::JobRef>` type. We discussed crossbeam in the previous chapter! Things are starting to come together for us. Let's look at the definition of `WorkerThread`:

```
pub struct WorkerThread {
    /// the "worker" half of our local deque
    worker: Deque<JobRef>,

    index: usize,

    /// are these workers configured to steal breadth-first or not?
    breadth_first: bool,

    /// A weak random number generator.
    rng: UnsafeCell<rand::XorShiftRng>,

    registry: Arc<Registry>,
}
```

What is `Registry`? What's creating these `WorkerThread`s? Recall that in `join_context`, we're inside a call to `registry::in_worker`. That function is defined in `rayon-core/src/registry.rs` and is:

```
pub fn in_worker<OP, R>(op: OP) -> R
    where OP: FnOnce(&WorkerThread, bool) -> R + Send, R: Send
{
    unsafe {
        let owner_thread = WorkerThread::current();
        if !owner_thread.is_null() {
            // Perfectly valid to give them a `&T`: this is the
            // current thread, so we know the data structure won't be
            // invalidated until we return.
            op(&*owner_thread, false)
        } else {
            global_registry().in_worker_cold(op)
        }
    }
}
```

The `WorkerThread::current()` call is polling a thread-local static variable called `WORKER_THREAD_STATE`, a `Cell<*const WorkerThread>` that may or may not be null in the event that no `WorkerThread` has been created inside the current operating system thread or has ceased to exist for some reason. If the `WorkerThread` does exist as thread-local, the passed `op` function is called, which is what we're investigating inside `join_context`. But, if the thread-local is null, `global_registry().in_worker_cold(op)` is called. The `global_registry` is:

```
static mut THE_REGISTRY: Option<&'static Arc<Registry>> = None;
static THE_REGISTRY_SET: Once = ONCE_INIT;

/// Starts the worker threads (if that has not already happened). If
/// initialization has not already occurred, use the default
/// configuration.
fn global_registry() -> &'static Arc<Registry> {
    THE_REGISTRY_SET.call_once(|| unsafe {
init_registry(ThreadPoolBuilder::new()).unwrap() });
    unsafe {
        THE_REGISTRY.expect("The global thread pool \
                             has not been initialized.")
    }
}
```

That is, once `global_registry()` is called at least once, we've established a static `Registry` populated by the return of `init_registry`:

```
unsafe fn init_registry(builder: ThreadPoolBuilder) -> Result<(),
ThreadPoolBuildError> {
    Registry::new(builder).map(|registry|
                                THE_REGISTRY = Some(leak(registry))
                               )
}
```

That function further defers to `Registry::new` after having populated some builders for configuration purposes:

```
impl Registry {
    pub fn new(mut builder: ThreadPoolBuilder)
```

```
           -> Result<Arc<Registry>, ThreadPoolBuildError>
    {
        let n_threads = builder.get_num_threads();
        let breadth_first = builder.get_breadth_first();

        let inj_worker = Deque::new();
        let inj_stealer = inj_worker.stealer();
        let workers: Vec<_> = (0..n_threads)
            .map(|_| Deque::new())
            .collect();
        let stealers: Vec<_> = workers.iter().map(|d|
                                                  d.stealer()
                                              ).collect();

        let registry = Arc::new(Registry {
            thread_infos: stealers.into_iter()
                .map(|s| ThreadInfo::new(s))
                .collect(),
            state: Mutex::new(RegistryState::new(inj_worker)),
            sleep: Sleep::new(),
            job_uninjector: inj_stealer,
            terminate_latch: CountLatch::new(),
            panic_handler: builder.take_panic_handler(),
            start_handler: builder.take_start_handler(),
            exit_handler: builder.take_exit_handler(),
        });
```

Here, finally, we see threads being built:

```
        // If we return early or panic, make sure to terminate
        // existing threads.
        let t1000 = Terminator(&registry);

        for (index, worker) in workers.into_iter().enumerate() {
            let registry = registry.clone();
            let mut b = thread::Builder::new();
            if let Some(name) = builder.get_thread_name(index) {
                b = b.name(name);
            }
            if let Some(stack_size) = builder.get_stack_size() {
                b = b.stack_size(stack_size);
            }
            if let Err(e) = b.spawn(move || unsafe {
                main_loop(worker, registry, index, breadth_first)
            }) {
                return Err(ThreadPoolBuildError::new(
```

```
                                ErrorKind::IOError(e))
                            )
            }
        }

        // Returning normally now, without termination.
        mem::forget(t1000);

        Ok(registry.clone())
    }
```

Woo! Okay, a `Registry` is a static which, once created, spawns a number of threads and enters them into `main_loop`. This loop creates `WorkerThread`, notifies the `Registry` of `WorkerThread` being started, which marks the thread as alive. This is the `thread_infos` of the `Registry` and is why `WorkerThread` carries an index in itself. `main_thread` calls `WorkerThread::wait_until`, a function whose purpose is to probe the `Registry` for termination notice and, without that notice, calls `WorkerThread::wait_until_cold`. That cold condition is the status of `WorkerThread` when `take_local_job` or steal fail to return any items. Taking a local job looks like so:

```
    #[inline]
    pub unsafe fn take_local_job(&self) -> Option<JobRef> {
        if !self.breadth_first {
            self.worker.pop()
        } else {
            loop {
                match self.worker.steal() {
                    Steal::Empty => return None,
                    Steal::Data(d) => return Some(d),
                    Steal::Retry => {},
                }
            }
        }
    }
```

The `self.worker` here is the deque from crossbeam and the breadth first option simply controls which end of the deque work is retrieved from. Stealing is a little more complicated, using the random number generator of `WorkerThread` to choose arbitrary threads—indexed through

the `Registry`—to steal work from, seeking forward through the threads until work can be stolen from another thread's deque or until no work is found.

No more mysteries! As we descend further into rayon, we understand that it's got quite a bit of machinery for representing split workloads that then gradually transition into the execution of those chunks of work on a thread pool. That thread pool performs work-stealing to keep the CPUs saturated. This understanding helps make the remainder of `join_context` more understandable:

```
        // Execute task a; hopefully b gets stolen in the meantime.
        let status_a = unwind::halt_unwinding(move ||
                           oper_a(FnContext::new(injected)));
        let result_a = match status_a {
            Ok(v) => v,
            Err(err) => join_recover_from_panic(worker_thread,
                                                &job_b.latch,
                                                err),
        };
```

The caller of `join_context` has packaged `oper_b` up, pushed it to the worker's queue, and executes `oper_a` in the hopes that the other operation will be executed as well. The remainder of the function is a loop, either pulling `oper_b` from the local deque or stealing from some other thread:

```
        while !job_b.latch.probe() {
            if let Some(job) = worker_thread.take_local_job() {
                if job == job_b_ref {
                    // Found it! Let's run it.
                    //
                    // Note that this could panic, but it's ok if we
                    // unwind here.
                    log!(PoppedRhs { worker: worker_thread.index() });
                    let result_b = job_b.run_inline(injected);
                    return (result_a, result_b);
                } else {
                    log!(PoppedJob { worker: worker_thread.index() });
                    worker_thread.execute(job);
```

```
                }
            } else {
                // Local deque is empty. Time to steal from other
                // threads.
                log!(LostJob { worker: worker_thread.index() });
                worker_thread.wait_until(&job_b.latch);
                debug_assert!(job_b.latch.probe());
                break;
            }
        }

        return (result_a, job_b.into_result());
    })
```

Ordering is maintained through the latch mechanism, defined in `rayon-core/src/latch.rs`. You are encouraged to study this mechanism. It's very clever and well within the capabilities of the reader who has gotten this far in the book, bless you.

That is *almost* it. We still have yet to discuss `map(|&i| i * i).sum()`. The `map` there is defined on `ParallelIterator`, `IndexedParallelIterator`, and is:

```
fn map<F, R>(self, map_op: F) -> Map<Self, F>
    where F: Fn(Self::Item) -> R + Sync + Send,
          R: Send
{
    map::new(self, map_op)
}
```

`Map` is, in turn, a `ParallelIterator` that consumes the individual `map_op` functions, producing `MapConsumer`s out of them, the details of which we'll skip as we are already familiar enough with rayon's `Consumer` concept. Finally, sum:

```
fn sum<S>(self) -> S
    where S: Send + Sum<Self::Item> + Sum<S>
{
    sum::sum(self)
}
```

This is, in turn, defined in `src/iter/sum.rs` as:

```
pub fn sum<PI, S>(pi: PI) -> S
    where PI: ParallelIterator,
          S: Send + Sum<PI::Item> + Sum
{
    pi.drive_unindexed(SumConsumer::new())
}
```

The producer is driven into a `SumConsumer` that splits the summation work up over the thread pool and then eventually folds it into a single value.

And *that* is that. That's rayon. It's a thread pool that can automatically split workloads over the threads plus a series of `Iterator` adaptations that greatly reduce the effort required to exploit the thread-pooling model.

Now you know! That's not all that rayon can do—it's a *very* useful library—but that's the central idea. I warmly encourage you to read through rayon's documentation.

Data parallelism and OS processes – evolving corewars players

In this final section of the chapter, I'd like to introduce a project that will carry us over into the next chapter. This project will tie together the concepts we've introduced so far in this chapter and introduce a new one: processes. Compared to threads, processes have a lot to recommend them: memory isolation, independent priority scheduling, convenient integration of existing programs into your own, and a long history of standardized syscalls (in Unix) to deal with them. But, memory isolation is less ideal when your aim is to fiddle with the same memory from multiple concurrent actors—as in atomic programming—or when you otherwise have to set up expensive IPC channels. Worse, some operating systems are not fast to spawn new processes, limiting a multi-processesing program's ability to exploit CPUs on a few fronts. All that said, knowing how to spawn and manipulate OS processes *is* very useful and we'd be remiss not to cover it in this book.

Corewars

Corewars is a computer programmer game from the 1980s. Introduced in a 1984 Scientific American article, the game is a competition between two or more programs—or warriors—that are written in an obscure assembly language called Redcode. Realistically, there are effectively two variants of Redcode, one of which has nice features, such as labels or pseudo-opcodes for variable assignments. This is usually referred to as Redcode. The other variant is referred to as *load file Redcode* or simply *load file* and is a straight listening of opcodes. The machine that Redcode targets is the Memory Array Redcode Simulator (MARS). It is… peculiar. Memory is bounded and circular, the maximum address prior to wrap-around is `core_size-1`. Most players set `core_size` to 8000 but any value is valid. Each memory location is an instruction. There is no distinction between instruction cache and data storage. In fact, there's not *really* data storage per se, though you can store information in unused sections of an instruction. There is no such thing as absolute memory addressing in the MARS machine: every address is relative in the ring of memory to the current instruction. For instance, the following instruction will tell MARS to jump backward two spots in memory and continue execution from there:

```
JMP -2
```

MARS executes one instruction from each warrior in turn, in the order that the MARS loaded the warrior into memory. Warriors start at some offset from one another and are guaranteed not to overlap. A warrior may spawn sub-warriors and this sub-warrior will get executed like all the others. A single opcode `DAT` will cause a warrior to exit. The objective of the game is force opposing warriors to

execute DAT. The last warrior—plus any spawned warriors—under simulation wins the game.

There's quite a bit more detail to Corewars than has been presented here, in terms of opcode listing and memory modes and what not, but this is not a Corewars tutorial. There are many fine tutorials online and I suggest some in the *Further reading* section. What's important to understand is that Corewars is a game played between programmers with programs. There are many MARS simulators but the most commonly used as the *MARS* is pMARS (`http://www.koth.org/pmars/`), or Portable MARS. It'll build pretty easily even under a modern GCC, though I can't say I've managed to build it under clang.

Here's a pretty fun bomber from pmars 0.9.2's source distribution, to give you a sense of how these programs look:

```
;redcode-94
;name Rave
;author Stefan Strack
;strategy Carpet-bombing scanner based on Agony and Medusa's
;strategy (written in ICWS'94)
;strategy Submitted: @date@
;assert CORESIZE==8000

CDIST   equ 12
IVAL    equ 42
FIRST   equ scan+OFFSET+IVAL
OFFSET  equ (2*IVAL)
DJNOFF  equ -431
BOMBLEN equ CDIST+2

        org comp

scan    sub.f  incr,comp
comp    cmp.i  FIRST,FIRST-CDIST        ;larger number is A
        slt.a  #incr-comp+BOMBLEN,comp  ;compare to A-number
        djn.f  scan,<FIRST+DJNOFF       ;decrement A- and B-number
        mov.ab #BOMBLEN,count
split   mov.i  bomb,>comp               ;post-increment
count   djn.b  split,#0
        sub.ab #BOMBLEN,comp
```

```
        jmn.b  scan,scan
bomb    spl.a  0,0
        mov.i  incr,<count
incr    dat.f  <0-IVAL,<0-IVAL
        end
```

A bomber tosses a bomb at offset intervals into memory. Some toss DATs, which force warriors to exit if executed. This one tosses an imp. An imp is a Corewars program that creeps forward in memory some number of instructions at a time. The Imp, introduced in the 1984 article, is:

```
MOV.I 0 1
```

The opcode here is MOV with modifier I, source address 0, and destination address 1. MOV moves instructions at the source address, also traditionally called the *a-field*, to the destination address, also called the *b-field*. The opcode modifier—the .I—defines how the opcode interprets its mandate. Here, we're specifying that MOV moves the whole instruction from the current cell to the next one up the ring. MOV.A would have only moved the a-field of the current instruction into the a-field of the destination instruction. The bomber's imp `mov.i incr,<count` relies on previously computed values to drop imps in memory that advance at different rates with different offsets. The < on the b-field is an indirect with predecrement operator on the address but, like I said, this is not a Corewars tutorial.

The pMARS simulator can read in both load files and Redcode, will drop the warriors into the simulator, and report the results. Here we run the bomber against the imp for 100 simulated rounds:

```
> pmars -b -r 1000 imp.red rave.red
Imp by Alexander Dewdney scores 537
Rave by Stefan Strack scores 1926
Results: 0 463 537
```

The imp does not fare too well. It wins 0 times but does manage to tie 537 times. An imp is hard to kill.

Writing warriors is fun, but teaching a computer to write warriors is even more fun and so that's what we're going to set out to do. Specifically, we'll write an *evolver,* a program that uses a simulated evolution of a population of warriors to produce ever more fit specimens. We'll call out to the pMARS executable to evaluate our population and, with enough time, hopefully pop out something pretty impressive.

Feruscore – a Corewars evolver

I want to make sure we're all on the same page, before we start digging through source code. The way simulated evolution works is you make a population—maybe totally random, maybe not—and call it generation `0`. You then run some `fitness` function on the members of the population. Fitness may be defined in terms of some known absolute or as a relative value between members of the population. The fittest members of the population are then taken to form a subset of parents. Many algorithms take two members, but the sky's the limit and your species could well require six individuals for reproduction. That's what we're after—reproduction.

The individuals of the population are non-destructively recombined to form children. The population is then mutated, changing genes in an individual's genomes with some domain-specific probability. This may be *all* members of the population or just children, or just parents, or whatever you'd like. There's no perfect answer here. After mutation, the parents, their children, and any non-reproducing members of the population that did not die during fitness evaluation are then promoted to generation 2. The cycle continues:

- Fitness evaluation
- Parents reproduce
- Population is mutated

Exactly when the cycle stops is up to the programmer: perhaps when a fixed number of generations have passed, perhaps when an individual that meets some minimum threshold of fitness has evolved.

Simulated evolution is not magic. Much like QuickCheck, which uses randomness to probe programs for property violations, this algorithm is probing a problem space for the most fit solution. It's an optimization strategy, mimicking the biological process. As you can probably infer from the description I just gave, there are a lot of knobs and alternatives to the basic approach. We'll be putting together a straightforward approach here but the you are warmly encouraged to make your own modifications.

Representing the domain

Before we can start evolving anything, we have to figure out how to represent the individuals; we need to decide how to structure their chromosome. Many genetic algorithm implementations represent these as a [u8], serializing and deserializing as appropriate. There's a lot to recommend this representation. For one, modern computers have instructions specifically tailored to operate on many bytes in parallel—a topic we'll touch on in `Chapter 10`, *Futurism – Near-Term Rust*,—which is especially helpful in the mutation and reproduction stages of a genetic algorithm. You see, one of the key things to a successful simulation of evolution is *speed*. We need a large population and many, many generations to discover fit individuals. `[u8]` is a convenient representation for serialization and deserialization, especially with something such as serde (`https://crates.io/crates/serde`) and bincode (`https://crates.io/crates/bincode`) at hand. The downside to a `[u8]` representation is creating individuals that don't deserialize into something valid, in addition to eating up CPU time moving back and forth between `[u8]` and structs.

Program optimization comes down to carefully structuring one's computation to suit the computer or finding clever ways to avoid computation altogether. In this particular project, we do have a trick we can play. A valid individual is a finite list of instructions—feruscore targets producing *load files*—and all we have to do, then, is represent an instruction as a Rust struct and pop it into a vector. Not bad! This will save us a significant amount of CPU time, though our mutation step will be a tad slower. That's okay, though, as fitness evaluation will be the biggest time sink, as we'll see that shortly.

Exploring the source

Let's look at feruscore's `Cargo.toml` first. It is:

```
[package]
name = "feruscore"
version = "0.1.0"
authors = ["Brian L. Troutwine <brian@troutwine.us>"]

[dependencies]
rand = "0.4"
rayon = "1.0"
tempdir = "0.3"
```

We saw the tempdir crate back in `Chapter 5`, *Locks – Mutex, Condvar, Barriers and RWLock*; its purpose is to create temporary directories that get deleted when the directory handler is dropped. We discussed rayon earlier in the chapter. The rand crate is new, though we did mention it in passing in both `Chapter 2`, *Sequential Rust Performance and Testing*, and `Chapter 5`, *Locks – Mutex, Condvar, Barriers and RWLock*, when we produced our own XorShift. The rand crate implements many different pseudo-random algorithms, in addition to providing a convenient interface to OS facilities for randomness. We'll be putting rand to good use.

The project is a standard library/executable bundle. Unlike many other projects in this book, we are not creating multiple executables and so there's only a single `src/main.rs`. We'll talk through that in due course. The library root, in `src/lib.rs`:

```
extern crate rand;
extern crate rayon;
extern crate tempdir;
```

```
pub mod individual;
pub mod instruction;
```

Instructions

No surprises here. We import the project dependencies and expose two public modules: instruction and individual. The instruction module at `src/instruction.rs` is feruscore's version of `[u8]`, or, rather, the `u8` in the array. Let's take a look. The structure at the root is `Instruction`:

```
#[derive(PartialEq, Eq, Copy, Clone, Debug)]
pub struct Instruction {
    pub opcode: OpCode,
    pub modifier: Modifier,
    pub a_mode: Mode,
    pub a_offset: Offset,
    pub b_mode: Mode,
    pub b_offset: Offset,
}
```

An instruction in Redcode is an opcode, a modifier for that opcode, an a-offset plus modifier and a b-offset plus modifier. (The a-field and b-field are the combination of offset and mode.) That's exactly what we have here. No need to deserialize from a byte array; we'll work on a direct representation. The implementation of `Instruction` is small. To start, we need some way of creating random `Instruction`s:

```
impl Instruction {
    pub fn random(core_size: u16) -> Instruction {
        Instruction {
            opcode: OpCode::random(),
            modifier: Modifier::random(),
            a_mode: Mode::random(),
            a_offset: Offset::random(core_size / 32),
            b_mode: Mode::random(),
            b_offset: Offset::random(core_size / 32),
        }
    }
```

This sets up a pattern we'll use through the rest of this module. Every type exposes `random() -> Self` or some variant thereof. Note that the offset is restricted to 1/32 the passed `core_size`. Why? Well, we're trying to cut down the cardinality of the domain being explored. Say the core size is 8,000. If all possible instances of core size values were tried in an offset, there'd be 64,000,000 possible `Instruction`, not including any of the other valid combinations with the other structure fields. Based on what I know about high-scoring warriors, the explicit offsets are usually small numbers.

By cutting down the offset domain, we've eliminated 62,000,000 potential instructions, saving CPU time. This could well be a pessimistic restriction—forcing the population right into a bottleneck—but I doubt it. Every struct in this module also has a `serialize(&self, w: &mut Write) -> io::Result<usize>` function. Here's the one for `Instruction`:

```
pub fn serialize(&self, w: &mut Write) -> io::Result<usize> {
    let mut total_written = 0;
    self.opcode.serialize(w)?;
    total_written += w.write(b".")?;
    self.modifier.serialize(w)?;
    total_written += w.write(b" ")?;
    total_written += match self.a_mode {
        Mode::Immediate => w.write(b"#")?,
        Mode::Direct => w.write(b"$")?,
        Mode::Indirect => w.write(b"*")?,
        Mode::Decrement => w.write(b"{")?,
        Mode::Increment => w.write(b"}")?,
    };
    self.a_offset.serialize(w)?;
    total_written += w.write(b", ")?;
    total_written += match self.b_mode {
        Mode::Immediate => w.write(b"#")?,
        Mode::Direct => w.write(b"$")?,
        Mode::Indirect => w.write(b"@")?,
        Mode::Decrement => w.write(b"<")?,
        Mode::Increment => w.write(b">")?,
    };
    total_written += self.b_offset.serialize(w)?;
    total_written += w.write(b"\n")?;
```

```
        Ok(total_written)
    }
}
```

We're going to be calling out to the local system's pmars executable when we run feruscore and that program needs to find files on-disk. Every `Instruction` in every individual will be serialized to disk each time we check fitness. That load file will be deserialized by pmars, run in simulation and the results issued back to feruscore from pmars' stdout. That's not a fast process.

The remainder of the instruction module follows this general outline. Here, for instance, is the `Modifier` struct:

```
#[derive(PartialEq, Eq, Copy, Clone, Debug)]
pub enum Modifier {
    A,
    B,
    AB,
    BA,
    F,
    X,
    I,
}
```

Here's the implementation:

```
impl Modifier {
    pub fn random() -> Modifier {
        match thread_rng().gen_range(0, 7) {
            0 => Modifier::A,
            1 => Modifier::B,
            2 => Modifier::AB,
            3 => Modifier::BA,
            4 => Modifier::F,
            5 => Modifier::X,
            6 => Modifier::I,
            _ => unreachable!(),
        }
    }
```

```
    pub fn serialize(&self, w: &mut Write) -> io::Result<usize> {
        match *self {
            Modifier::A => w.write(b"A"),
            Modifier::B => w.write(b"B"),
            Modifier::AB => w.write(b"AB"),
            Modifier::BA => w.write(b"BA"),
            Modifier::F => w.write(b"F"),
            Modifier::X => w.write(b"X"),
            Modifier::I => w.write(b"I"),
        }
    }
}
```

Individuals

Nothing special, especially by this point in the book. The `random() -> Self` functions are a touch fragile because you can't query an enumeration for its total number of fields, meaning that if we add or remove a field from `Modifier`, for instance, we also have to remember to update `gen_range(0, 7)` appropriately. Removal is not so bad, the compiler will complain some, but addition is easy to forget and overlook.

Let's now look at the individual module. The bulk of the code is in src/individual/mod.rs. The struct for Individual is very short:

```
#[derive(Debug)]
pub struct Individual {
    chromosome: Vec<Option<Instruction>>,
}
```

An `Individual` is a chromosome and little else, a vector of instructions just as we discussed. The functions on the `Individual` provide what you might expect, given the discussion of the evolution algorithm. Firstly, we have to be able to make a new `Individual`:

```
impl Individual {
    pub fn new(chromosome_size: u16, core_size: u16) -> Individual {
        let mut chromosome = Vec::with_capacity(chromosome_size as
usize);
        chromosome.par_extend((0..(chromosome_size as usize))
                  .into_par_iter().map(|_| {
            if thread_rng().gen_weighted_bool(OpCode::total() * 2) {
                None
            } else {
                Some(Instruction::random(core_size))
            }
```

```
        }));
        Individual { chromosome }
    }
```

Ah! Here now we've run into something new. What in the world is `par_extend`? It's a rayon function, short for parallel extend. Let's work inside out. The interior map ignores its argument and produces a `None` with 1/28 chance—14 being the total number of `OpCode` variants—and some other random `Instruction` with 27/28th chance:

```
map(|_| {
    if thread_rng().gen_weighted_bool(OpCode::total() * 2) {
        None
    } else {
```

```
        Some(Instruction::random(core_size))
    }
})
```

The input that the interior map is so studiously ignoring is `(0..(chromosome_size as usize)).into_par_iter()`. That's a `ParallelIterator` over a range of numbers, from 0 to the maximum number of chromosomes an Individual is allowed to have. One creative solution in early Corewars was to submit a `MOV 0 1` imp followed by 7999 `DAT` instructions. If the MARS loaded your warrior and wasn't programmed to carefully check warrior offsets, your opponent would lose by being immediately overwritten. `Instruction` length limits were quickly put into place and we obey that here, too. Serialization of an `Individual` follows the pattern we're familiar with:

```
    pub fn serialize(&self, w: &mut Write) -> io::Result<usize> {
        let mut total_wrote = 0;
        for inst in &self.chromosome {
            if let Some(inst) = inst {
                total_wrote += inst.serialize(w)?;
            } else {
                break;
            }
        }
```

```
        Ok(total_wrote)
    }
```

Mutation and reproduction

Before we discuss competition between two `Individual`s, let's talk about mutation and reproduction. Mutation is the easier of the two to understand, in my opinion, because it operates on one `Individual` at a time. Our implementation:

```
    pub fn mutate(&mut self, mutation_chance: u32, core_size: u16) ->
() {
        self.chromosome.par_iter_mut().for_each(|gene| {
            if thread_rng().gen_weighted_bool(mutation_chance) {
                *gene = if thread_rng().gen::<bool>() {
                    Some(Instruction::random(core_size))
                } else {
                    None
                };
            }
        });
    }
```

The call to `par_iter_mut` creates a `ParallelIterator` in which the elements of the iterator are mutable, much like `iter_mut` from the standard library. The `for_each` operator applies the closure to each. The `for_each` operator is similar in purpose to `map`, except it's going to necessarily consume the input but will be able to avoid allocating a new underlying collection. Modifying a thing in-place? Use `for_each`. Creating a new thing from an existing bit of storage? Use `map`. Anyway, the mutation strategy that feruscore uses is simple enough. Every `Instruction` in the chromosome is going to be changed with probability `1/mutation_chance` and may be disabled with 1/2 probability, that is, be set to None. The careful reader will have noticed that the serializer stops serializing instructions when a `None` is found in the chromosome, making everything after that point junk DNA. The junk still comes into play during reproduction:

```
pub fn reproduce(&self, partner: &Individual,
                 child: &mut Individual) -> ()
{
    for (idx, (lgene, rgene)) in self.chromosome
        .iter()
        .zip(partner.chromosome.iter())
        .enumerate()
    {
        child.chromosome[idx] = if thread_rng().gen::<bool>() {
            *lgene
        } else {
            *rgene
        };
    }
}
```

Feruscore uses a reproduction strategy called half-uniform crossover. Each gene from a parent has even odds of finding its way into the child. The child is passed as a mutable reference, which is creepy if you think about it too hard. I'm not sure of any real species that takes over the body of a less fit individual and hot-swaps DNA into it to form a child in order to save on energy (or computation, in our case) but here we are. Note that, unlike mutation, reproduction is done serially. Remember that rayon is chunking work and distributing it through a threadpool. If we were to make this parallel, there'd have to be a mechanism in place to rectify the choices made from the zipper of the parents' chromosomes into the child. If the chromosomes were many multiple megabytes, this would be a good thing to tinker with. As is, the chromosomes are very small. Most Corewars games limit a warrior to 100 instructions. rayon is impressive but it is not *free*, either in runtime cost or programming effort.

Competition – calling out to pMARS

It's now time to consider how each `Individual` competes. Earlier, we discussed calling out to pmars on-system, which requires on-disk representations of the two competing Individual warriors. True to that, competition first serializes both warriors to temporary locations on the disk:

```
pub fn compete(&self, other: &Individual) -> Winner {
    let dir = TempDir::new("simulate")
                  .expect("could not make tempdir");

    let l_path = dir.path().join("left.red");
    let mut lfp = BufWriter::new(File::create(&l_path)
                      .expect("could not write lfp"));
    self.serialize(&mut lfp).expect("could not serialize");
    drop(lfp);

    let r_path = dir.path().join("right.red");
    let mut rfp = BufWriter::new(File::create(&r_path)
                      .expect("could not write rfp"));
    other.serialize(&mut rfp).expect("could not serialize");
    drop(rfp);
```

We haven't interacted much with the filesystem in this book but, suffice it to say, Rust has an *excellent* filesystem setup in the standard library, most of it under `std::fs`. You are warmly encouraged to take a gander at the documentation for any of the functions here if they are not already familiar with them. Anyway, with both warriors written to disk, we can run pmars on them:

```
    let output = Command::new("pmars")
        .arg("-r") // Rounds to play
```

```
            .arg(format!("{}", ROUNDS))
            .arg("-b") // Brief mode (no source listings)
            .arg(&l_path)
            .arg(&r_path)
            .output()
            .expect("failed to execute process");
```

Let's break down what's happening here. `std::process::Command` is a builder struct for running an OS process. If we did nothing but call `Command::output(&mut self)` on the new `Command` then:

- pmars would be run without arguments
- The working directory and environment of feruscore would become pmars
- The stdout of pmars will be piped back to the feruscore process

Each of these can be changed. The actual execution of pmars does not take place until the output is called: or, spawn, which is intended for long-lived processes. The `Command::stdin`, `Command::stderr`, and `Command::stdout` functions control the IO behavior of the process, whether feruscore's descriptors are inherited by the child-process—which is the case with spawn—or piped back to feruscore, the default for output. We *don't* want pmars to write to feruscore's stdout/sterr, so the defaults of `output` are ideal for our needs. The call to `args` adds a new argument to the `Command` and a new call is required even for arguments that associate. We write `.arg("-r").arg(format("{}", ROUNDS))` rather than `.arg(format!("-r {}", ROUNDS))`. Here, we're crashing the program if pmars fails to make appropriate output, which might happen if the executable can't be found or we trigger a crash-bug in pmars and don't have permissions to run the program. Crashing is a bit user-hostile, but good enough for our purposes here.

Now, `output: std::process::Output`, a struct with three members. We've ignored the exit status in favor of pulling the last stdio line:

```
        let result_line = ::std::str::from_utf8(&output.stdout)
            .expect("could not parse output")
            .lines()
            .last()
            .expect("no output");
```

If there was output on stdio, then pmars successfully ran the passed warriors, if not, then the warriors were invalid. The parser in pmars is very forgiving if the warriors failed to load the serializer in Individual or Instruction is faulty. Note that `result_line` is the last line of output. The last line is always of the form `Results: 10 55 32`, meaning that the left program won 10 times, the right program won 55 times, and they tied 32 times. A tie happens when neither warrior is able to force the other out of the game by some pre-determined number of executions, often around 10,000. Parsing that last line is a rugged affair:

```
        let pieces = result_line.split(' ').collect::<Vec<&str>>();

        let l_wins = pieces[1].parse::<u16>()
                        .expect("could not parse l_wins");
        let r_wins = pieces[2].parse::<u16>()
                        .expect("could not parse r_wins");
        let ties = pieces[3].parse::<u16>()
                        .expect("could not parse ties");
        assert_eq!((l_wins + r_wins + ties) as usize, ROUNDS);
```

Note the assertion. `ROUNDS` is a constant set in the module, set to 100. Our `Command` informed pmars to play 100 rounds with the passed warriors, if the result doesn't add up to 100, we've got a real problem. Placing these sanity checks in code sometimes feels a bit silly but, lo and behold, they do turn up bugs. Likewise, it's rarely a wasted effort to invest in diagnostics. For instance, in genetic algorithms, it's hard to know how things are going in your population. Is every generation getting fitter? Has the population bottlenecked somewhere? To that

end, the next step of compete is to tally the fitness of the left and right warriors:

```
        tally_fitness(l_wins as usize);
        tally_fitness(r_wins as usize);
```

The `tally_fitness` function is defined like so:

```
fn tally_fitness(score: usize) -> () {
    assert!(score <= ROUNDS);

    match score {
        0...10 => FITNESS_00010.fetch_add(1, Ordering::Relaxed),
        11...20 => FITNESS_11020.fetch_add(1, Ordering::Relaxed),
        21...30 => FITNESS_21030.fetch_add(1, Ordering::Relaxed),
        31...40 => FITNESS_31040.fetch_add(1, Ordering::Relaxed),
        41...50 => FITNESS_41050.fetch_add(1, Ordering::Relaxed),
        51...60 => FITNESS_51060.fetch_add(1, Ordering::Relaxed),
        61...70 => FITNESS_61070.fetch_add(1, Ordering::Relaxed),
        71...80 => FITNESS_71080.fetch_add(1, Ordering::Relaxed),
        81...90 => FITNESS_81090.fetch_add(1, Ordering::Relaxed),
        91...100 => FITNESS_91100.fetch_add(1, Ordering::Relaxed),
        _ => unreachable!(),
    };
}
```

What is this, you ask? Well, I'll tell you! `tally_fitness` takes the input score—the wins of the warrior—and buckets that score into a histogram. Usually this is done with a modulo operation but we're stuck here. The implementation has to assume that `tally_fitness` will be run by multiple threads at once. So, we've constructed a histogram from several discrete `AtomicUsize`s with names suggestive of their bucket purpose. The declaration of the statics is as repetitive as you'd imagine and we'll spare repeating it here. Maintaining this structure by hand is a real pain but it *does* get you an atomic histogram, so long as you're fine with that histogram also being static. This implementation is. The histogram is not used for decision making so much as it's meant for diagnostic display. We'll come to that when we discuss the main loop.

The final lines of `compete` are underwhelming:

```
        if l_wins > r_wins {
            Winner::Left(l_wins)
        } else if l_wins < r_wins {
            Winner::Right(r_wins)
        } else {
            Winner::Tie
        }
    }
}
```

The win counts of each warrior are compared and the winner is declared to be the one with the higher score, or neither wins and a tie is declared. Depending on your problem domain, this fitness' rubric might be too coarse. From the output, we don't know how much fitter an individual was compared to its competitor, only that it was. The warriors may have tied at 0 wins each, or gone 50/50. We believe this simple signal is enough, at least for our purposes now.

At least as far as creating an `Individual`, mutating it, reproducing it, and competing it, that's it! Now all that's left is to simulate evolution and for that we have to jump to `main.rs`. Before we do, though, I want to point out briefly that there's a submodule here, `individuals::ringers`. This module has functions such as:

```
pub fn imp(chromosome_sz: u16) -> Individual {
    let mut chromosome = vec![
        // MOV.I $0, $1
        Some(Instruction {
            opcode: OpCode::Mov,
            modifier: Modifier::I,
            a_mode: Mode::Direct,
            a_offset: Offset { offset: 0 },
            b_mode: Mode::Direct,
            b_offset: Offset { offset: 1 },
        }),
    ];
    for _ in 0..(chromosome_sz - chromosome.len() as u16) {
        chromosome.push(None);
```

```
    }
    Individual { chromosome }
}
```

Sometimes it's wise to nudge evolution along by sprinkling a little starter into the random mixture. Feruscore has no parser—at least, not until some kind reader contributes one—so the ringers are written out long-form.

Now, on to `main`!

Main

The preamble to `src/main.rs` is typical for the programs we've seen so far in this book:

```
extern crate feruscore;
extern crate rand;
extern crate rayon;

use feruscore::individual::*;
use rand::{thread_rng, Rng};
use rayon::prelude::*;
use std::collections::VecDeque;
use std::fs::File;
use std::fs::{self, DirBuilder};
use std::io::{self, BufWriter};
use std::path::{Path, PathBuf};
use std::sync::atomic::{AtomicUsize, Ordering};
use std::{thread, time};
```

The program does start, however, with an abnormally large block of constants and statics:

```
// configuration
const POPULATION_SIZE: u16 = 256; // NOTE must be powers of two
const CHROMOSOME_SIZE: u16 = 25; // Must be <= 100
const PARENTS: u16 = POPULATION_SIZE / 8;
const CHILDREN: u16 = PARENTS * 2;
const RANDOS: u16 = POPULATION_SIZE - (PARENTS + CHILDREN);
const CORE_SIZE: u16 = 8000;
const GENE_MUTATION_CHANCE: u32 = 100;

// reporting
static GENERATIONS: AtomicUsize = AtomicUsize::new(0);
static REGIONAL_BATTLES: AtomicUsize = AtomicUsize::new(0);
static FINALS_BATTLES: AtomicUsize = AtomicUsize::new(0);
```

```
static STATE: AtomicUsize = AtomicUsize::new(0);
static NEW_PARENTS: AtomicUsize = AtomicUsize::new(0);
```

Realistically, the configuration constants could be adjusted into actual configuration, via clap or some configuration-parsing library. The toml (`https://crates.io/crates/toml`) crate paired with serde is very useful for parsing straightforward configuration, where the validation steps are a few checks we now encode as comments. The reporting statics are the same static `AtomicUsize`s we've seen through the book. In fact, starting the reporting thread is the first thing that the `main` function does:

```
fn main() {
    let _ = thread::spawn(report);
```

The `report` thread works in a similar fashion to the threads that fill a comparable role elsewhere in the book: the global statics—including `FITNESS_*` from `feruscore::individuals`—are swapped for zero, a little bit of computation happens, then there's a block that prints to stdout, and finally the thread sleeps for a bit before looping around again, forever. It's a well-worn technique by this point.

With the reporting thread online, feruscore creates its initial population:

```
    let mut population: Vec<Individual> =
        Vec::with_capacity(POPULATION_SIZE as usize);
    let mut children: Vec<Individual> =
        Vec::with_capacity(CHILDREN as usize);
    let mut parents: VecDeque<Individual> =
        VecDeque::with_capacity(PARENTS as usize);
    population.par_extend(
        (0..POPULATION_SIZE)
            .into_par_iter()
            .map(|_| Individual::new(CHROMOSOME_SIZE, CORE_SIZE)),
    );
    population.pop();
    population.pop();
```

```
    population.push(ringers::imp(CHROMOSOME_SIZE));
    population.push(ringers::dwarf(CHROMOSOME_SIZE));
```

rayon's `par_extend` makes an appearance again, useful when `POPULATION_SIZE` is a large number but not harmful with its current value. As we saw, rayon will decline to split a parallel collection too small to be effectively parallelized, the distribution to threads overwhelming the time to compute. `Individual::new` is called with the actual chromosome and core sizes, and two unlucky, random members of the population are ejected and deallocated in favor of ringers from `individual::ringers`.

There are many ways to decide the fitness of individuals in a population. The method that feruscore takes is to run individuals in a tournament. The winner of the first tournament becomes the first parent, the winner of the second becomes the second parent, and so forth until each parent is filled up. That implementation is brief but contains multitudes:

```
    loop {
        // tournament, fitness and selection of parents
        STATE.store(0, Ordering::Release);
        while parents.len() < PARENTS as usize {
            thread_rng().shuffle(&mut population);
            let res = population
                .into_par_iter()
                .fold(|| (None, Vec::new()), regional_tournament)
                .reduce(|| (None, Vec::new()), finals_tournament);
            population = res.1;
            parents.push_back(res.0.unwrap());
            NEW_PARENTS.fetch_add(1, Ordering::Relaxed);
        }
```

What's going on here? The population is shuffled at the start to avoid pairing up the same individuals in each tournament. Indeterminacy in thread makes this somewhat unnecessary and the speed-obsessed evolver might do well to remove this. The main show is the fold and reduce of the population. Readers familiar with functional

programming will be surprised to learn that rayon's `fold` and `reduce` are not synonymous. The type of `ParallelIterator::fold` is:

```
fold<T, ID, F>(self, identity: ID, fold_op: F)
    -> Fold<Self, ID, F>
where
    F: Fn(T, Self::Item) -> T + Sync + Send,
    ID: Fn() -> T + Sync + Send,
    T: Send,
```

Readers familiar with functional programming will not be surprised to learn that `fold`'s type is somewhat involved. Jokes aside, note that the function does not return an instance of `T` but `Fold<Self, ID, F>` where `Fold: ParallelIterator`. rayon's `fold` does not produce a single `T` but, instead, a parallel iterator over chunks that will have `Fn(T, Self::Item) -> T` applied: an iterator over folded chunks. `ParallelIterator::reduce` has the following type:

```
reduce<OP, ID>(self, identity: ID, op: OP)
    -> Self::Item
where
    OP: Fn(Self::Item, Self::Item) -> Self::Item + Sync + Send,
    ID: Fn() -> Self::Item + Sync + Send,
```

The `OP` takes two `Item`s and combines them into one `Item`. `Reduce`, then, takes a `ParallelIterator` of `Item`s and reduces them down into a single instance of `Item` type. Our tournament implementation folds `regional_tournament` over the population, producing a `Fold` over `(Option<Individual>, Vec<Individual>)`, the first element of the tuple being the fittest individual in that subchunk of the population, the second element being the remainder of the population. The `reduce` step, then, takes the winners of the regional tournaments and reduces them into one final winner. The implementation of these two functions is similar. First, `regional_tournament`:

```
fn regional_tournament(
    (chmp, mut population): (Option<Individual>, Vec<Individual>),
```

```
    indv: Individual,
) -> (Option<Individual>, Vec<Individual>) {
    if let Some(chmp) = chmp {
        REGIONAL_BATTLES.fetch_add(1, Ordering::Relaxed);
        match chmp.compete(&indv) {
            Winner::Left(_) | Winner::Tie => {
                population.push(indv);
                (Some(chmp), population)
            }
            Winner::Right(_) => {
                population.push(chmp);
                (Some(indv), population)
            }
        }
    } else {
        (Some(indv), population)
    }
}
```

Note the call to compete and the choice to promote the existing `chmp`—champion—to the next tournament round in the event of a tie. `REGIONAL_BATTLES` sees an update, feeding the report thread information. The less fit individual is pushed back into the population. `finals_tournament` is a little more complicated but built along the same lines:

```
fn finals_tournament(
    (left, mut lpop): (Option<Individual>, Vec<Individual>),
    (right, rpop): (Option<Individual>, Vec<Individual>),
) -> (Option<Individual>, Vec<Individual>) {
    if let Some(left) = left {
        lpop.extend(rpop);
        if let Some(right) = right {
            FINALS_BATTLES.fetch_add(1, Ordering::Relaxed);
            match left.compete(&right) {
                Winner::Left(_) | Winner::Tie => {
                    lpop.push(right);
                    (Some(left), lpop)
                }
                Winner::Right(_) => {
                    lpop.push(left);
                    (Some(right), lpop)
                }
            }
```

```
        } else {
            (Some(left), lpop)
        }
    } else {
        assert!(lpop.is_empty());
        (right, rpop)
    }
}
```

This function is responsible for rerunning competitions and for joining up the previously split parts of the population. Note the call to `lpop.extend`. The right population—`rpop`—is always merged into the left population, as are less fit individuals. There's no special reason for this, we could equally have merged right and returned right.

Now, take a minute to look at these two functions. They are sequential. We are able to reason about them as sequential functions, we are able to program them like sequential functions. We can test them like sequential functions. rayon's internal model doesn't leak into this code. We have to understand only the types and, once that's done, rayon's able to do its thing. This sequential inside, parallel outside model is unique, compared to all the techniques we've discussed in the book so far. rayon's implementation is undoubtedly complicated, but the programming model it admits is quite straightforward, once you get the hang of iterator style.

Once the tournament selection is completed, the parents vector is filled with `PARENTS`, number of `Individual`s and the population has members who are all suitable for being turned into children. Why draw from the population? Well, the upside is we avoid having to reallocate a new `Individual` for each reproduction. Many small allocations can get expensive fast. The downside is, by drawing from the population, we're not able to easily turn a direct loop into a parallel iterator. Given that the total number of reproductions is small, the implementation does not attempt anything but a serial loop:

```
        STATE.store(1, Ordering::Release);
        while children.len() < CHILDREN as usize {
            let parent0 = parents.pop_front().expect("no parent0");
            let parent1 = parents.pop_front().expect("no parent1");
            let mut child = population.pop().expect("no child");
            parent0.reproduce(&parent1, &mut child);
            children.push(child);
            parents.push_back(parent0);
            parents.push_back(parent1);
        }
```

Two parents are popped off, a child is popped off, reproduction happens, and everyone is pushed into their respective collection. Children exist to avoid reproducing into the same `Individual` multiple times. Once reproduction is completed, `children` is merged back into the population, but not before some sanity-checks take place:

```
        assert_eq!(children.len(), CHILDREN as usize);
        assert_eq!(parents.len(), PARENTS as usize);
        assert_eq!(population.len(), RANDOS as usize);

        population.append(&mut children);
```

Once `children` is integrated into the population, mutation:

```
        STATE.store(2, Ordering::Release);
        population.par_iter_mut().for_each(|indv| {
            let _ = indv.mutate(GENE_MUTATION_CHANCE, CORE_SIZE);
        });
```

The careful reader will note that the parents are not included in the mutation step. This is intentional. Some implementations do mutate the parents in a given generation, and whether or not this is a good thing to do depends very much on domain. Here, the difference between a fit `Individual` and an unfit one is so small that I deemed it best to leave parents alone. A matter for experimentation.

Finally, we bump the `generation`, potentially `checkpoint`, and re-integrate the parents into the population:

```
        let generation = GENERATIONS.fetch_add(1, Ordering::Relaxed);
        if generation % 100 == 0 {
            checkpoint(generation, &parents)
                .expect("could not checkpoint");
        }

        for parent in parents.drain(..) {
            population.push(parent);
        }
    }
}
```

There is no stop condition in feruscore. Instead, every 100 generations, the program will write out its best warriors—the parents—to disk. The user can use this information as they want. It would be reasonable to stop simulation after a fixed number of generations or compare the parents against a host of known best-of-breed warriors, exiting when a warrior had been evolved that could beat them all. That last would be especially nifty but would require either a parser or a fair deal of hand-assembly. Checkpointing looks the way you may imagine it:

```
fn checkpoint(generation: usize, best: &VecDeque<Individual>) ->
io::Result<()> {
    let root: PathBuf = Path::new("/tmp/feruscore/checkpoints")
                            .join(generation.to_string());
    DirBuilder::new().recursive(true).create(&root)?;
    assert!(fs::metadata(&root).unwrap().is_dir());

    for (idx, indv) in best.iter().enumerate() {
        let path = root.join(format!("{:04}.red", idx));
        let mut fp = BufWriter::new(File::create(&path)
                        .expect("could not write lfp"));
        indv.serialize(&mut fp)?;
    }

    Ok(())
}
```

Okay! Let's evolve some warriors.

Running feruscore

Before you attempt to run feruscore, please make sure pmars is available on your PATH. Some operating systems bundle pmars in their package distribution, others, such as OS X, require gcc to compile the program. The pMARS project is *venerable* C code and getting it compiled can be a bit fiddly. On my Debian system, I had to tweak the makefile some, but on OS X I found a helpful brew cask to get pMARS installed.

Once you've got pMARS installed, you should be able to run `cargo run --release` at the head of the project and receive output like the following:

```
GENERATION(0):
    STATE:          Tournament
    RUNTIME (sec):  4
    GENS/s:         0
    PARENTS:        2
       PARENTS/s:   1
    BATTLES:        660
       R_BATTLES/s: 131
       F_BATTLES/s: 25
    FITNESS:
        00...10:    625
        11...20:    1
        21...30:    3
        31...40:    0
        41...50:    130
        51...60:    3
        61...60:    0
        71...70:    0
        81...80:    0
        91...100:   550
```

Woo! Look at it go. This instance of feruscore has been running for 4 seconds—RUNTIME (sec): 4—and has computed two parents so far. This generation has gone through 660 battles: 131 regional and 25 final. The fitness histogram shows that the population is pretty evenly bad, otherwise we'd expect to see a clump at the 0 end and a clump at the 100 end. You should see something similar. I bet you'll also find that your CPUs are pegged. But, there's a problem. This is *slow*. It gets worse. Once I evolved more fit warriors, the generations took longer and longer to compute:

```
GENERATION(45):
    STATE:          Tournament
    RUNTIME (sec):  25297
    GENS/s:         0
    PARENTS:        8
       PARENTS/s:   0
    BATTLES:        1960
       R_BATTLES/s: 12
       F_BATTLES/s: 1
    FITNESS:
        00...10:    3796
        11...20:    0
        21...30:    0
        31...40:    0
        41...50:    0
        51...60:    0
        61...60:    0
        71...70:    0
        81...80:    0
        91...100:   118
```

That makes sense though. The better the population is, the longer members will hold out in a round. It's not uncommon to run a genetic algorithm for *thousands* of generations before it settles on a really excellent solution. A generation in the present implementation takes, generously, 30 seconds at least. By the time we're able to evolve a halfway-decent warrior, it'll be the 30th anniversary of the '94 ICWS Corewars standard! That's no good.

Turns out, serializing an `Individual` to disk, spawning a pMARS process, and forcing it to parse the serialized format—the load file—is not a fast operation. We could also compact our `Individual` representation significantly, improving cache locality of simulation. There's also something fishy about performing a *regional* and then a *finals* tournament just to fit the iterator style rayon requires. We can fix all of these things, but it'll have to wait until the next chapter. We're going to embed a MARS written in C into feruscore and all sorts of changes will fall out as a result.

Should be fun!

Summary

In this chapter, we set the lower-level details of concurrency in Rust as a foundation. We discussed thread pools, which, it turns out, we had all the pieces in-hand from previous chapters, to understand a fairly sophisticated one. Then we looked into rayon and discovered that we could also understand an *extremely* sophisticated threadpool, hidden behind the type system to enable data parallelism in the programming model. We discussed architectural concerns with the thread-per-connection model and the challenges of splitting small datasets up into data parallel iterators. Finally, we did a walkthrough of a rayon and multi-processing-based genetics algorithm project. The `std::process` interface is lean compared to that exposed by some operating systems, but well-thought-out and quite useful, as demonstrated in the feruscore project that closed out the chapter. We'll pick up feruscore in the next chapter when we integrate C code into it, in lieu of calling out to a process.

Further reading

The notes for the chapter are a bit unusual for the book. Rather than call out papers the reader could look to for further research these notes are, overwhelmingly, suggestions of codebases to read. Data parallel iterators are *amazing* but take a little getting used to.

Nothing helps more than reading existing projects. Me, I figure every thousand lines of source code takes an hour to understand well. Makes for a peaceful afternoon:

- *rayon*, available at `https://github.com/rayon-rs/rayon`. We discussed this crate quite a bit in the chapter but only skimmed the surface of it. I highly, highly recommend that the motivated reader go through ParallelIterator and work to understand the operators exposed there.
- *xsv*, available at `https://github.com/BurntSushi/xsv`. Andrew Gallant is responsible for some of the fastest text-focused Rust code right now and xsv is no exception. This crate implements a toolkit for very fast, parallel CSV querying and manipulation. The threadpool crate discussed earlier drives the whole thing. Well worth reading if you've got ambitions for fast text processing and want to see the application of thread pooling to such.
- *ripgrep*, available at `https://github.com/BurntSushi/xsv`. Andrew Gallant's ripgrep is one of the fastest grep implementations in

the world. The reader will be interested to know that the implementation does not use any off-the-shelf threadpool crates nor rayon. Ripgrep spawns a result printing thread for every search, then a bounded many threads to perform actual searches on files. Each search thread communicates results to the print thread via an MPSC, thereby ensuring that printed results are not torn and search can exploit the available machine CPUs.

- *tokei*, available at `https://github.com/Aaronepower/tokei`. There are many source-code line-counting programs in the world, but few cover as many languages or are as fast as Aaron Power's tokei. The implementation is well worth reading if you're interested in parsing alone. But, tokei also notably makes use of rayon. Where the project has chosen sequential std iterators over rayon's parallel iterators is something readers should discover for themselves. Then, they should ponder through the reasons why such choices were made.

- *i8080*, available at `https://github.com/Aaronepower/i8080`. In addition to tokei, Aaron Power wrote an impressive Intel 8080 emulator in Rust. MARS is a deeply odd machine and the reader will probably have an excellent time discovering how an actual, simple CPU is emulated.

- *QuickCheck: Lightweight Tool for Random Testing of Haskell Programs*, 2000, Koen Claessen and John Hughes. This paper introduces the QuickCheck tool for Haskell and introduces property-based testing to the world. The research here builds on previous work into randomized testing, but is

novel for realizing that computers had got fast enough to support type-directed generation as well as shipping with the implementation in a single-page appendix. Many, many subsequent papers have built on this one to improve the probing ability of property testers.

- *Smallcheck and Lazy Smallcheck: Automatic Exhaustive Testing for Small Values*, 2008, Colin Runciman, Matthew Naylor, and Fredrik Lindblad. This paper takes direct inspiration from Claessen and Hughes work but works the hypothesis that many program defects are to be found on s*mall* input first. The Lazy Smallcheck discussion in the second half of the paper is especially interesting. I found myself reading this with a growing ambition to implement a Prolog.

- *Macros: The Rust Reference*, available at `https://doc.rust-lang.org/reference/macros.html`. Macro use in Rust is in a weird state. It's clearly useful but the current macro implementation is unloved and a new macro system is coming, gradually, to replace it. Meanwhile, if you want to build macros to use in code, you're going to need this reference.

- *The C10K Problem*, available at `http://www.kegel.com/c10k.html`. This page was much discussed when it first hit the internet. The problem discussed was that of getting 10,000 active connections to a single network server, processing them in a non-failing fashion. The author notes several resources for further reading, all of which are still useful today, and discusses the state-of-the-art operating systems at that time,

in terms of kernal versions and the inclusion of new, better polling APIs. Though old—the many-connection problem is often now stated as C1M—it's still an excellent and informative read.

- *Annotated Draft of the Proposed 1994 Core War Standard*, available at `http://corewar.co.uk/standards/icws94.htm`. Corewars is a subtle game, in terms of the effect of the assembly language, Redcode, on the machine and the function of the machine itself. This document is the attempt to standardize the behavior of both in 1994. The standard was not accepted—the standard body was dying—but the proposal document is very clear and even comes with a brief C implementation of a MARS.

FFI and Embedding – Combining Rust and Other Languages

Up until this point in the book, we've discussed Rust more or less in isolation. Rust was intentionally designed to integrate with other programming languages by calling external programming languages through its **Foreign Function Interface** (**FFI**) and by being embedded itself. Many modern programming languages offer FFI, easy embedding, or both. Python, for instance, can very conveniently call out to libraries with C calling conventions and can be embedded with a little forethought. Lua, a high-level and garbage-collected language like Python, has a convenient FFI and can be embedded without much trouble. Erlang has a small handful of FFI interfaces but Erlang is not, itself, easily embedded into user-space environments. Amusingly, it's fairly straightforward to compile Erlang into an RTOS image.

In this chapter, we'll discuss calling out to foreign code in Rust and embedding Rust into foreign programming languages. We'll start off by extending the corewars evolver program —feruscore—that we covered at the tail end of `Chapter 8`, *High-Level Parallelism – Threadpools, Parallel Iterators, and Processes*. You are encouraged to read that material before starting this chapter. After we've extended feruscore by embedding a C MARS simulator inside it, we'll move on to calling Rust functions from a C program. We'll demonstrate embedding Lua into a Rust system for convenient scripting and close the chapter by embedding Rust in high-level garbage-collected languages—Python and Erlang.

By the end of this chapter, we will have:

- Adapted the feruscore project from the last section to incorporate a C simulator
- Demonstrated the inclusion of Lua code into a Rust project
- Demonstrated the inclusion of Rust code into a C project
- Demonstrated the inclusion of Rust code into a Python project
- Demonstrated the inclusion of Rust code into an Erlang/Elixir project

Embedding C into Rust – feruscore without processes

When we wrapped up our discussion of feruscore in the previous chapter, we'd constructed a program that could discover corewars warriors through simulated natural selection. This was done by writing evolved warriors out to disk, using Rust's OS process interface to call out to pmars—the de facto standard MARS—and competing them to discover their relative fitness. We used Rayon—Rust's very convenient data parallelism library—to distribute the workload of competitions between available CPUs. Unfortunately, the implementation was pretty slow. Building a tournament selection criteria was maybe more difficult to express than we might have hoped—though I'm sure there a bright-spark of a reader out there who will improve that substantially and wow me. The real pain point was serializing *every* warrior to disk multiple times, allocating similar structures repeatedly to establish each round, and then eating pmars' allocation and parsing overhead. It's this last step, calling out to an external program to run competitions, is something we'll address in this chapter. We will also address the other two issues because, well, why not go all-in?

> *Not all of feruscore's source code appears in this chapter. Some of it was discussed in-depth in the previous chapter, some of it—such as benchmarking code—would be a rehash of material already covered in the book. The C code is not printed in its entirety as it's very dense and very long. You can find the full listing in the book's source repository.*

The MARS C interface

Corewars is a venerable game and there are many implementations of MARS out there. pMARS, like I mentioned, is the defacto standard, implementing the '94 ICWS draft and adding later additions, such as new instructions and a thing called *p-space*, that we won't go into here. Corewars has never been serious business. Many of the MARS available online today have traded code back and forth over the years, have been written with varying levels of attention to correctness, and sometimes programmed for older machines where multiprocessing was a concern for research labs. Most make no distinction between their internal library interface and executable consumer, like Rust programs do. Some blessed few are designed to be embedded, or, at least, are extractions of older MARS codebases that can be extended. Some blessed few of those don't use static globals in their implementations and can be embedded in a multiprocessing environment, such as feruscore.

In the grand tradition of MARS implementations, we will take an existing, public domain MARS, whittle it down to a manageable core, and put a new spin on it. Specifically, the feruscore introduced in this chapter embeds exhaust 1.9.2 (`http://corewar.co.uk/pihlaja/exhaust/`), written by M. Joonas Pihlaja in the early 2000s. Pihlaja seems to have extracted select code from pMARS, especially in and around exhaust's main and parser functions. We aren't after any of that code. What we need is the simulator. This means that we can toss out anything to do with parsing, and any support code needed for exhaust's `main` function. The extracted code we require lives in the feruscore project root, in `c_src/`. The functions we'll embed are all implemented in `c_src/sim.c`. These are from `c_src/sim.h`:

```
void sim_free_bufs(mars_t* mars);
void sim_clear_core(mars_t* mars);

int sim_alloc_bufs(mars_t* mars);
int sim_load_warrior(mars_t* mars, uint32_t pos,
                     const insn_t* const code, uint16_t len);
int sim_mw(mars_t* mars, const uint16_t * const war_pos_tab,
           uint32_t *death_tab );
```

The `sim_alloc_bufs` function is responsible for allocating the internal storage of a blank `mars_t`, the structure on which simulation is performed. `sim_clear_core` clears `mars_t` between rounds, setting core memory to `DAT`, and resetting any warrior queues. `sim_load_warrior` loads the warrior into core memory, the warrior really just being a pointer to an array of instructions—`insn_t`—with the `len` passed length. `sim_mw` runs the simulation, reading the warrior positions from `war_pos_tab`, and writing to `death_tab` when a warrior completely dies.

Creating C-structs from Rust

Now, how do we call these functions from Rust? Moreover, how do we create instances of `mars_t` or `insn_t`? Well, recall back in `Chapter 03`, *The Rust Memory Model – Ownership, References, and Manipulation*, that Rust allows control over memory layout in structures. Specifically, all Rust structures are implicitly `repr(Rust)`, types aligned to byte boundaries with structure fields being reordered as the compiler sees fit, among other details. We also noted the existence of a `repr(C)` for structures, in which Rust would lay out a structure's memory representation in the same manner as C. That knowledge now comes to bear.

What we will do is this. First, we'll compile our C code as a library and rig it to link into feruscore. That's done by placing a `build.rs` at the root of the project and using the cc (`https://crates.io/crates/cc`) crate to produce a static archive, like so:

```
extern crate cc;

fn main() {
    cc::Build::new()
        .file("c_src/sim.c")
        .flag("-std=c11")
        .flag("-O3")
        .flag("-Wall")
        .flag("-Werror")
        .flag("-Wunused")
        .flag("-Wpedantic")
        .flag("-Wunreachable-code")
        .compile("mars");
}
```

Cargo will produce `libmars.a` into `target/` when the project is built. But, how do we make `insn_t`? We copy the representation. The C side of this project defines `insn_t` like so:

```
typedef struct insn_st {
  uint16_t a, b;
  uint16_t in;
} insn_t;
```

`uint16_t a` and `uint16_t b` are the *a*-field and *b*-field of the instruction, where `uint16_t` is a compressed representation of the `OpCode`, `Modifier`, and `Modes` in an instruction. The Rust side of the project defines an instruction like so:

```
#[derive(PartialEq, Eq, Copy, Clone, Debug, Default)]
#[repr(C)]
pub struct Instruction {
    a: u16,
    b: u16,
    ins: u16,
}
```

This is the exact layout of the `inst_t` C. The reader will note that this is quite different from the definition of `Instruction` we saw in the previous chapter. Also, note that the field names do not matter, only the bit representation. The C structure calls the last field of the struct in, but this is a reserved keyword in Rust, so it is `ins` in the Rust side. Now, what is going on with that `ins` field? Recall that the `Mode` enumeration only had five fields. All we really need to encode a mode is three bits, converting the enumeration into numeric representation. A similar idea holds for the other components of an instruction. The layout of the `ins` field is:

```
bit         15 14 13 12 11 10 09 08 07 06 05 04 03 02 01 00
field       |-flags-| |--opcode--| |-mod-| |b-mode| |a-mode|
```

The `Mode` for a-field is encoded in bits 0, 1 and, 2. The `Mode` for b-field is in bits 3, 4, and 5, and so on for the other instruction components. The last two bits, 14 and 15, encode a `flag` that is almost always zero. A non-zero flag is an indicator to the simulator that the non-zero instruction is the `START` instruction—the 0^{th} instruction of a warrior is not necessarily the one executed first by MARS. This compact structure requires a little more work on the part of the programmer to support it. For instance, the `Instruction` can no longer be created directly by the programmer but has to be constructed through a builder. The `InstructionBuilder`, defined in `src/instruction.rs`, is:

```
pub struct InstructionBuilder {
    core_size: u16,
    ins: u16,
    a: u16,
    b: u16,
}
```

As always, we have to keep track of the core size. Building the builder is straightforward enough, by this point:

```
impl InstructionBuilder {
    pub fn new(core_size: u16) -> Self {
        InstructionBuilder {
            core_size,
            ins: 0_u16,
            a: 0,
            b: 0,
        }
    }
```

Writing a field into the instruction requires a little bit manipulation. Here's writing a `Modifier`:

```
    pub fn modifier(mut self, modifier: Modifier) -> Self {
        let modifier_no = modifier as u16;
        self.ins &= !MODIFIER_MASK;
        self.ins |= modifier_no << MODIFIER_MASK.trailing_zeros();
```

```
        self
    }
```

The constant `MODIFIER_MASK` is defined in a block at the top of the source file with the other field masks:

```
const AMODE_MASK: u16 = 0b0000_0000_0000_0111;
const BMODE_MASK: u16 = 0b0000_0000_0011_1000;
const MODIFIER_MASK: u16 = 0b0000_0001_1100_0000;
const OP_CODE_MASK: u16 = 0b0011_1110_0000_0000;
const FLAG_MASK: u16 = 0b1100_0000_0000_0000;
```

Observe that the relevant bits in the masks are 1 bits. In `InstructionBuilder::modifier` we `&=` the negation of the mask, which boolean-ands `ins` with the negation of the modifier mask, zero-ing the `Modifier` that was previously there. That done, the `Modifier` encoded as u16 is shifted left and boolean-or'ed into place. The `trailing_zeros()` function returns the total number of contiguous zeros in the lower end of a word, the exact number we need to shift by for each mask. Those readers that have done bit-manipulation work in other languages may find this to be very clean. I think so as well. Rust's explicit binary form for integers makes writing, and later, understanding, masks a breeze. Common bit-manipulation operations and queries are implemented on every basic integer type. Very useful.

The `OpCode` layout has changed somewhat. We don't `repr(C)` the enum, as the bit representation does not matter. What does matter, since this is enumeration is field-less, is which integer the variants cast to. First in the source maps to 0, the second to 1, and so forth. The C code has op-codes defined like so in `c_src/insn.h`:

```
enum ex_op {
    EX_DAT,             /* must be 0 */
    EX_SPL,
    EX_MOV,
    EX_DJN,
    EX_ADD,
```

```
    EX_JMZ,
    EX_SUB,
    EX_SEQ,
    EX_SNE,
    EX_SLT,
    EX_JMN,
    EX_JMP,
    EX_NOP,
    EX_MUL,
    EX_MODM,
    EX_DIV,             /* 16 */
};
```

The Rust version is as follows:

```
#[derive(PartialEq, Eq, Copy, Clone, Debug, Rand)]
pub enum OpCode {
    Dat,  // 0
    Spl,  // 1
    Mov,  // 2
    Djn,  // 3
    Add,  // 4
    Jmz,  // 5
    Sub,  // 6
    Seq,  // 7
    Sne,  // 8
    Slt,  // 9
    Jmn,  // 10
    Jmp,  // 11
    Nop,  // 12
    Mul,  // 13
    Modm, // 14
    Div,  // 15
}
```

The other instruction components have been shuffled around just a little bit to cope with the changes required by the C code. The good news is, this representation is more compact than the one from the previous chapter and should probably be maintained, even if all the C code were ported into Rust, a topic we'll get into later. But—and I'll spare you the full definition of `InstructionBuilder` because once you've seen one set-function you've seen them all—all this bit fiddling does

make the implementation harder to see, and to correct at a glance. The instruction module now has QuickCheck tests to verify that all the fields get set correctly, meaning they can be ready right back out again no matter how many times fields are set and reset. You are encouraged to examine the QuickCheck tests yourself.

The high-level idea is this—a blank `Instruction` is made and a sequence of change orders is run over that Instruction—momentarily shifted into an `InstructionBuilder` to allow for modification—and then the changed field is read and confirmed to have become the value it was changed to. The technique is inline with what we've seen before elsewhere.

Now, what about that `mars_t`? The C definition, in `c_src/sim.h`, is:

```
typedef struct mars_st {
  uint32_t nWarriors;

  uint32_t cycles;
  uint16_t coresize;
  uint32_t processes;

  uint16_t maxWarriorLength;

  w_t* warTab;
  insn_t* coreMem;
  insn_t** queueMem;
} mars_t;
```

The `nWarriors` field sets how many warriors will be in the simulation, which for feruscore is always two cycles controls the number of cycles a round will take before ending if both warriors are still alive, processes the maximum number of processes available, and `maxWarriorLength` shows the maximum number of instructions a warrior may be. All of these are more or less familiar from the last chapter, just in a new programming language and with different names. The final three fields are pointers to arrays and are effectively private to the simulation function. These are allocated and deallocated by

`sim_alloc_bufs` and `sim_free_bufs`, respectively. The Rust side of this structure looks like so, from `src/mars.rs`:

```
#[repr(C)]
pub struct Mars {
    n_warriors: u32,
    cycles: u32,
    core_size: u16,
    processes: u32,
    max_warrior_length: u16,
    war_tab: *mut WarTable,
    core_mem: *mut Instruction,
    queue_mem: *const *mut Instruction,
}
```

The only new type here is `WarTable`. Even though our code will never explicitly manipulate the warrior table, we do still have to be bit-compatible with C. The definition of `WarTable` is:

```
#[repr(C)]
struct WarTable {
    tail: *mut *mut Instruction,
    head: *mut *mut Instruction,
    nprocs: u32,
    succ: *mut WarTable,
    pred: *mut WarTable,
    id: u32,
}
```

We could have maybe got away with just making these private fields in `Mars` pointers to void in `mars_st`, but that would have reduced type information on the C side of the project and this approach might hamper future porting efforts. With the type explicit on the Rust side of the project, it's much easier to consider rewriting the C functions in Rust.

Calling C functions

Now that we can create Rust structs with C-bit layout, we can start passing that memory down into C. It's important to understand, this is an inherently unsafe activity. The C code might fiddle with memory in a way that breaks Rust's invariants and the only way we can be sure this isn't true is auditing the C code ahead of time. This is the same with fuzzing, which we'll do too. How do we link with `libmars.a`? A small `extern` block will do it, in `src/mars.rs`:

```
#[link(name = "mars")]
extern "C" {
    fn sim_alloc_bufs(mars: *mut Mars) -> isize;
    fn sim_free_bufs(mars: *mut Mars) -> ();
    fn sim_clear_core(mars: *mut Mars) -> ();
    fn sim_load_warrior(mars: *mut Mars, pos: u32,
                        code: *const Instruction,
                        len: u16) -> isize;
    fn sim_mw(mars: *mut Mars, war_pos_tab: *const u16,
              death_tab: *mut u32) -> isize;
}
```

With `#[link(name = "mars")]`, we're instructing the compiler to link to the `libmars.a` produced in the build process. If we were linking to a system library, the approach would be the same. The Rust Nomicon section on FFI—referenced in the *Further reading* section at the end of this chapter—links to libsnappy, for example. The extern block informs the compiler that these functions will need to be called with the C ABI, not Rust's. Let's compare `sim_load_warrior` side by side. Here is the C version:

```
int sim_load_warrior(mars_t* mars, uint32_t pos,
                     const insn_t* const code, uint16_t len);
```

Here is the Rust version:

```
fn sim_load_warrior(mars: *mut Mars, pos: u32,
                    code: *const Instruction,
                    len: u16) -> isize;
```

These are similar. C doesn't have the same mutability concept as Rust, though there are const annotations for code that Rust picks up for free. We've enhanced that by making all of Instruction's query functions take `&self`. Rule of thumb—unless the C is const-correct, you should probably assume the Rust needs a `*mut`. This isn't always true, but it saves a fair deal of headache. Now, on that notion, the careful reader may note that as I ported exhaust's code into feruscore, I adapted it to use `stdint.h`. In the unaltered code, `pos` has the unsigned int type or `usize`. The simulator C code assumes `pos` will be at least 32 bits wide, hence the explicit conversation to a 32-bit integer. This wasn't entirely necessary, but it's good form to used fixed-width types as they minimize surprises moving between CPU architectures.

Managing cross-language ownership

Rust is now aware of the C functions, and we have our internal structs reworked to use C layout: it is time to link up the C functions that manipulate `mars_t` with our `Mars`. Because we're sharing ownership of `Mars` between the Rust allocator and the C allocator, we've got to be careful to initialize the `Mars` struct with null pointers in the fields that C will own, like so:

```
impl MarsBuilder {
    pub fn freeze(self) -> Mars {
        let mut mars = Mars {
            n_warriors: 2,
            cycles: u32::from(self.cycles.unwrap_or(10_000)),
            core_size: self.core_size.unwrap_or(8_000),
            processes: self.processes.unwrap_or(10_000),
            max_warrior_length: self.max_warrior_length.unwrap_or(100),
            war_tab: ptr::null_mut(),
            core_mem: ptr::null_mut(),
            queue_mem: ptr::null_mut(),
        };
        unsafe {
            sim_alloc_bufs(&mut mars);
        }
        mars
    }
```

This `MarsBuilder` is a builder-pattern for producing a `Mars`. The remainder of the implementation works how you might expect:

```
    pub fn core_size(mut self, core_size: u16) -> Self {
        self.core_size = Some(core_size);
        self
    }
```

```
    pub fn cycles(mut self, cycles: u16) -> Self {
        self.cycles = Some(cycles);
        self
    }

    pub fn processes(mut self, processes: u32) -> Self {
        self.processes = Some(processes);
        self
    }

    pub fn max_warrior_length(mut self, max_warrior_length: u16) ->
Self {
        self.max_warrior_length = Some(max_warrior_length);
        self
    }
}
```

Back to `MarsBuilder::freeze(self) -> Mars`. This function creates a new `Mars` and then passes it immediately into `sim_alloc_bufs`:

```
unsafe {
    sim_alloc_bufs(&mut mars);
}
```

`&mut mars` automatically coerces into `*mut mars` and, as we know from previous chapters, dereferencing a raw pointer is unsafe. The fact that it's *C* dereferencing the raw pointer is icing on the cake. Now, let's take a look at `sim_alloc_bufs` and get a sense of what's going on. In `c_src/sim.c`:

```
int sim_alloc_bufs(mars_t* mars) {
    mars->coreMem = (insn_t*)malloc(sizeof(insn_t) * mars->coresize);
    mars->queueMem = (insn_t**)malloc(sizeof(insn_t*) *
                        (mars->nWarriors * mars->processes + 1));
    mars->warTab = (w_t*)malloc(sizeof(w_t)*mars->nWarriors);

    return (mars->coreMem
            && mars->queueMem
            && mars->warTab);
}
```

The three fields owned by C—`coreMem`, `queueMem`, and `warTab`—are allocated according to the size of the structs being stored and the return is a boolean-and of the `malloc` returns, a short way of determining whether `malloc` ever returned `NULL`, signaling that no more memory was available on-system. Had we decided to add a new field into the Rust struct and not updated the C struct to reflect this change, these stores would be too small. Eventually, some code somewhere would reach past the bounds of an array and crash. No good, that.

But! We've just called C code from Rust.

Let's talk about ownership for a second. `Mars` is a structure not wholly owned by Rust, and not wholly owned by C. That's fine and is also not uncommon, especially if you're partially (or completely) porting a C codebase into Rust. It does mean we've got to be careful about `Drop`:

```
impl Drop for Mars {
    fn drop(&mut self) {
        unsafe { sim_free_bufs(self) }
    }
}
```

As we've seen in previous chapters, we have to arrange for explicit `Drop` when there's raw memory in use. `Mars` is no exception. The call here to `sim_free_bufs` clears up the C-owned memory and Rust takes care of the rest. If cleanup were more difficult—as sometimes happens—you'd have to take care to avoid deferencing the C-owned pointers post-free. `sim_free_bufs` is a brief implementation:

```
void sim_free_bufs(mars_t* mars)
{
    free(mars->coreMem);
    free(mars->queueMem);
    free(mars->warTab);
}
```

Running the simulation

All that's left to do is run warriors in simulation. In the previous chapter, an `Individual` was responsible for managing its own pmars process. Now, `Mars` will be responsible for hosting the competitors, which you may have gleaned from the C API. We simply don't have any space in `Individual` to store something, and by pushing competition into `Mars`, we can avoid allocation churn on temporary `Mars` structures.

The compete function that was formerly bound to `Individual` has now been moved to `Mars`:

```
impl Mars {
    /// Competes two Individuals at random locations
    ///
    /// The return of this function indicates the winner.
    /// `Winner::Right(12)` will mean that the 'right' player
    /// won 12 more rounds than did 'left'. This may mean they
    /// tied 40_000 times or maybe they only played 12 rounds
    /// and right won each time.
    pub fn compete(&mut self, rounds: u16, left: &Individual,
                   right: &Individual) -> Winner {
        let mut wins = Winner::Tie;
        for _ in 0..rounds {
            let core_size = self.core_size;
            let half_core = (core_size / 2) - self.max_warrior_length;
            let upper_core = core_size - self.max_warrior_length;
            let left_pos = thread_rng().gen_range(0, upper_core);
            let right_pos = if (left_pos + self.max_warrior_length)
                            < half_core {
                thread_rng().gen_range(half_core + 1, upper_core)
            } else {
                thread_rng().gen_range(0, half_core)
            };
            wins = wins + self.compete_inner(left, left_pos,
                                             right, right_pos);
```

```
        }
        tally_fitness(wins);
        BATTLES.fetch_add(1, Ordering::Relaxed);
        wins
    }
```

The C API requires that we calculate the warrior offsets and take care to not overlap them. The approach taken here is to randomly place the left Individual, determine whether it's in the upper or lower core, and then place the right Individual, taking care with both to not place them past the `end` of the core. The actual implementation of the competition is `compete_inner`:

```
    pub fn compete_inner(
        &mut self,
        left: &Individual,
        left_pos: u16,
        right: &Individual,
        right_pos: u16,
    ) -> Winner {
        let (left_len, left_code) = left.as_ptr();
        let (right_len, right_code) = right.as_ptr();

        let warrior_position_table: Vec<u16> = vec![left_pos,
right_pos];
        let mut deaths: Vec<u32> = vec![u32::max_value(),
                                        u32::max_value()];
```

We call `Individual::as_ptr() -> (u16, *const Instruction)` to get a raw view of the chromosome of the Individual and its length. Without this, we've got nothing to pass down to the C functions. `warrior_position_table` informs MARS which instructions are the start of its competitors. We could search out the `START` flag in the warriors and place that in `warrior_position_table`. This is an improvement left for the reader. The deaths table will be populated by the simulator code. If both warriors die during competition, the array will be `[0, 1]`. The death table is populated with `u32::max_value()` to make distinguishing no-result from result easy enough. Before starting the competition,

we have to clear the simulator—which might be filled with instructions from a previous bout:

```
        unsafe {
            sim_clear_core(self);
```

If you pull up the `sim_clear_core` implementation, you'll find a memset to 0 over `core_mem`. Recall that `DAT` is the 0^{th} variant of the `Instruction` enumeration. Unlike `pmars`, this simulator must use `DAT` as a default instruction, but it does make resetting the field very fast. `sim_clear_core` also clears up the process queue and other C-owned storage. Loading the warriors is a matter of plugging in the information we've already computed:

```
            assert_eq!(
                0,
                sim_load_warrior(self, left_pos.into(),
                                 left_code, left_len)
            );
            assert_eq!(
                0,
                sim_load_warrior(self, right_pos.into(),
                                 right_code, right_len)
            );
```

If the result is non-zero, that's an indicator that some serious and non-recoverable fault has happened. `sim_load_warrior` is an array operation, writing the warrior Instructions into `core_mem` at the defined offsets. We could, very conceivably, rewrite the functions of `sim_clear_core` and `sim_load_warrior` in Rust if we wanted to. Finally, field cleared and warriors loaded, we are able to simulate:

```
            let alive = sim_mw(self,
                               warrior_position_table.as_ptr(),
                               deaths.as_mut_ptr());
            assert_ne!(-1, alive);
        }
```

The `sim_mw` function returns the total number of warriors left alive at the end of the simulation run. If this value is `-1`, there's been some catastrophic, unrecoverable error.

Now, because we have a nice type system to play with, we don't really want to signal results back to the user with a vector of integers. We preserve the `Winner` type seen

in the previous chapter, doing a quick conversion before returning:

```
        let left_dead = deaths[0] != u32::max_value();
        let right_dead = deaths[1] != u32::max_value();
        match (left_dead, right_dead) {
            (false, false) | (true, true) => Winner::Tie,
            (true, false) => Winner::Right(1),
            (false, true) => Winner::Left(1),
        }
    }
}
```

You may have noticed that `Winner` implements `Add`, allowing compete to signal how many rounds the warriors won. `impl Add for Winner` is also in `src/mars.rs`, should you be curious to see it. As a final sanity test around `Mars`, we confirm that the Imp will lose to Dwarf more often than other outcomes:

```
#[cfg(test)]
mod tests {
    use super::*;
    use individual::*;

    #[test]
    fn imp_vs_dwarf() {
        let core_size = 8_000;
        let rounds = 100;
        let mut mars = MarsBuilder::default()
                            .core_size(core_size).freeze();
        let imp = ringers::imp(core_size);
        let dwarf = ringers::dwarf(core_size);
        let res = mars.compete(rounds, &imp, &dwarf);
```

```
        println!("RES: {:?}", res);
        match res {
            Winner::Left(_) | Winner::Tie => {
                panic!("imp should lose to dwarf")
            },
            Winner::Right(_) => {}
        }
    }
}
```

Recall that an Imp can't self-kill, only propagate. The Dwarf bombs core memory at regular, increasing offsets. An Imp versus Dwarf competition, then, is a race between Imp finding Dwarf's instructions and Dwarf dropping a DAT in the path of an Imp.

Fuzzing the simulation

Feruscore has been changed in this chapter to include raw memory access in both Rust and through FFI. The responsible thing to do is fuzz `Mars` and make sure we're not causing segmentation faults. We'll use AFL, which we discussed back in `Chapter 2`, *Sequential Rust Performance and Testing*, and again in `Chapter 5`, *Locks – Mutex, Condvar, Barriers, and RWLock*. The fuzz target is `src/bin/fuzz_target.rs`. The trick with fuzzing is ensuring stability. That is, AFL can't really do its work if for some input applied multiple times multiple paths come out. Fuzzing is more efficient in the case of a deterministic system. We were careful to make `Mars::compete_inner` deterministic, where `Mars::compete` uses randomness to determine warrior positions. Fuzzing, then, will go through `compete_inner` only. The preamble for `fuzz_target` doesn't contain any new crates:

```
extern crate byteorder;
extern crate feruscore;

use byteorder::{BigEndian, ReadBytesExt};
use feruscore::individual::*;
use feruscore::instruction::*;
use feruscore::mars::*;
use std::io::{self, Cursor, Read};
```

Remember that AFL passes a byte slice through stdin and the fuzzing target is responsible for deserializing that array into something sensible for itself. We'll build up a `Config` structure:

```
#[derive(Debug)]
struct Config {
    pub rounds: u16,
    pub core_size: u16,
```

```
    pub cycles: u16,
    pub processes: u16,
    pub max_warrior_length: u16,
    pub left_chromosome_size: u16,
    pub right_chromosome_size: u16,
    pub left: Individual,
    pub right: Individual,
    pub left_pos: u16,
    pub right_pos: u16,
}
```

Hopefully, these fields are familiar. The `rounds`, `core_size`, `cycles`, and `processes` each affect the MARS environment. The `max_warrior_length`, `left_chromosome_size`, and `right_chromosome_size` affect the two individuals that will be made to compete. `left` and `right` are those `Individual` instances. `left_pos` and `right_pos` set where the warriors will be placed in the MARS core memory. The numbers that we'll deserialize from the byte slice won't always be entirely sensible, so there'll be some cleanup needed, like so:

```
impl Config {
    pub fn new(rdr: &mut Cursor<Vec<u8>>) -> io::Result<Config> {
        let rounds = (rdr.read_u16::<BigEndian>()? % 1000).max(1);
        let core_size = (rdr.read_u16::<BigEndian>()? %
24_000).max(256);
        let cycles = (rdr.read_u16::<BigEndian>()? % 10_000).max(100);
        let processes = (rdr.read_u16::<BigEndian>()? % 1024).max(2);
        let max_warrior_length = (rdr.read_u16::<BigEndian>()? %
256).max(4);
        let left_chromosome_size = (rdr.read_u16::<BigEndian>()?
                                        % max_warrior_length).max(2);
        let right_chromosome_size = (rdr.read_u16::<BigEndian>()?
                                         % max_warrior_length).max(2);
```

It doesn't make any sense for there to be 0 rounds, as an example, so we say thatthere must be at least one round. Likewise, we need two processes, desire at least four warrior instructions, and so forth. Creating the left and right warriors is a matter of passing the byte reader into `Config::mk_individual`:

```
    let left = Config::mk_individual(rdr,
                                     max_warrior_length,
                                     left_chromosome_size,
                                     core_size)?;
    let right =
        Config::mk_individual(rdr,
                              max_warrior_length,
                              right_chromosome_size,
                              core_size)?;
```

`Config::mk_individual` deserializes into `InstructionBuilder`. The whole thing is kind of awkward. While we can convert a field-less Enum into an integer, it's not possible to go from an integer to a field-less Enum without some hairy match statements:

```
fn mk_individual(
    rdr: &mut Cursor<Vec<u8>>,
    max_chromosome_size: u16,
    chromosome_size: u16,
    core_size: u16,
) -> io::Result<Individual> {
    assert!(chromosome_size <= max_chromosome_size);
    let mut indv = IndividualBuilder::new();
    for _ in 0..(chromosome_size as usize) {
        let builder = InstructionBuilder::new(core_size);
        let a_field = rdr.read_i8()?;
        let b_field = rdr.read_i8()?;
        let a_mode = match rdr.read_u8()? % 5 {
            0 => Mode::Direct,
            1 => Mode::Immediate,
            2 => Mode::Indirect,
            3 => Mode::Decrement,
            _ => Mode::Increment,
        };
        let b_mode = match rdr.read_u8()? % 5 {
            0 => Mode::Direct,
            1 => Mode::Immediate,
            2 => Mode::Indirect,
            3 => Mode::Decrement,
            _ => Mode::Increment,
        };
```

Here, we've established the `InstructionBuilder` and read the `Mode` for a-field and b-field out from the byte slice. If a field is added, we'll have to come through here and update the fuzzing code. It's a real pain. Reading the `Modifier` out works the same way:

```
            let modifier = match rdr.read_u8()? % 7 {
                0 => Modifier::F,
                1 => Modifier::A,
                2 => Modifier::B,
                3 => Modifier::AB,
                4 => Modifier::BA,
                5 => Modifier::X,
                _ => Modifier::I,
            };
```

As does reading out the `OpCode`:

```
            let opcode = match rdr.read_u8()? % 16 {
                0 => OpCode::Dat,   // 0
                1 => OpCode::Spl,   // 1
                2 => OpCode::Mov,   // 2
                3 => OpCode::Djn,   // 3
                4 => OpCode::Add,   // 4
                5 => OpCode::Jmz,   // 5
                6 => OpCode::Sub,   // 6
                7 => OpCode::Seq,   // 7
                8 => OpCode::Sne,   // 8
                9 => OpCode::Slt,   // 9
                10 => OpCode::Jmn,  // 10
                11 => OpCode::Jmp,  // 11
                12 => OpCode::Nop,  // 12
                13 => OpCode::Mul,  // 13
                14 => OpCode::Modm, // 14
                _ => OpCode::Div,   // 15
            };
```

Producing an instruction is simple enough, thanks to the builder pattern in use here:

```
            let inst = builder
                .a_field(a_field)
```

```
                .b_field(b_field)
                .a_mode(a_mode)
                .b_mode(b_mode)
                .modifier(modifier)
                .opcode(opcode)
                .freeze();
            indv = indv.push(inst);
        }
        Ok(indv.freeze())
    }
```

Moving back up to `Config::new`, we create the left and right positions:

```
        let left_pos =
            Config::adjust_pos(core_size,
                               rdr.read_u16::<BigEndian>()?,
                               max_warrior_length);
        let right_pos =
            Config::adjust_pos(core_size,
                               rdr.read_u16::<BigEndian>()?,
                               max_warrior_length);
```

The `adjust_pos` function is a small thing, intended to keep warrior positions properly in bounds:

```
fn adjust_pos(core_size: u16, mut pos: u16, space: u16) -> u16 {
    pos %= core_size;
    if (pos + space) > core_size {
        let past = (pos + space) - core_size;
        pos - past
    } else {
        pos
    }
}
```

It's entirely possible that the warriors will overlap with this calculation. That is okay. Our ambition with fuzzing is not to check the logic of the program, only to seek out crashes. In fact, if overlapping two warriors causes a crash, that's a fact we need to know. The close of `Config::new` is fairly straightforward:

```
        Ok(Config {
            rounds,
            core_size,
            cycles,
            processes,
            max_warrior_length,
            left_chromosome_size,
            right_chromosome_size,
            left,
            right,
            left_pos,
            right_pos,
        })
    }
```

After all that, the main function of `fuzz_target` is minimal:

```
fn main() {
    let mut input: Vec<u8> = Vec::with_capacity(1024);
    let result = io::stdin().read_to_end(&mut input);
    if result.is_err() {
        return;
    }
    let mut rdr = Cursor::new(input);
    if let Ok(config) = Config::new(&mut rdr) {
        let mut mars = MarsBuilder::default()
            .core_size(config.core_size)
            .cycles(config.cycles)
            .processes(u32::from(config.processes))
            .max_warrior_length(config.max_warrior_length as u16)
            .freeze();
        mars.compete_inner(
            &config.left,
            config.left_pos,
            &config.right,
            config.right_pos,
        );
    }
}
```

`Stdin` is captured and a `Cursor` built from it, which we pass into `Config::new`, as explained earlier. The resulting `Config` is used to fuel `MarsBuilder`, and the `Mars` is then the arena for competition between the

two fuzzing `Individual` instances that may or may not overlap. Remember, before running AFL, be sure to run `cargo afl build`—release and not cargo build —release. Both will work, but the first is significantly faster to discover crashes, as AFL's instrumentation will be inlined in the executable. I've found that even a single instance of `cargo afl fuzz -i /tmp/in -o /tmp/out target/release/fuzz_target` will run through AFL cycles at a good clip. There aren't many branches in the code and, so, few paths for AFL to probe.

The feruscore executable

The final thing left to cover is `src/bin/feruscore.rs`. The careful reader will have noted that there's been a suspicious lack of rayon in the implementation so far. In fact, rayon is not in use in this version. Here's the full `Cargo.toml` for the project:

```
[package]
name = "feruscore"
version = "0.2.0"
authors = ["Brian L. Troutwine <brian@troutwine.us>"]

[dependencies]
rand = "0.4"
rand_derive = "0.3"
libc = "0.2.0"
byteorder = "1.0"
num_cpus = "1.0"

[build-dependencies]
cc = "1.0"

[dev-dependencies]
quickcheck = "0.6"
criterion = "0.2"

[[bench]]
name = "mars_bench"
harness = false

[[bin]]
name = "feruscore"

[[bin]]
name = "fuzz_target"
```

As mentioned at the start of the chapter, there were two issues with feruscore aside from calling out to an OS process: repeat allocation of

similar structures and poor control over tournaments. Viewed a certain way, comparing `Individual` fitness is a sorting function. Rayon does have a parallel sorting capability, `ParallelSliceMut<T: Send>::par_sort(&mut self)`, where `T: Ord`. We could make use of that, defining an `Ord` for `Individual` that allocated a new `Mars` for every comparison. Many tiny allocations is a killer for speed, though. A thread-local `Mars` could reduce that to a single allocation per thread, but then we're still giving up some control here. For instance, without inspecting rayon's source, can we be sure that population chunks are going to be of roughly equal size? Usually, this is not a concern, but it is for us here. Rayon's requiring that we perform a fold and then a reduce step is also extra work we don't necessarily have to do if we adjust our ambitions some.

One common method to deal with parallelizing genetic algorithms, and the one we'll take now, is to make islands that undergo evolution in parallel. The user sets a global population, where this population is split among islands, and threads are assigned an island to simulate for some number of generations. After that limit of generations is up, the island populations are merged, shuffled, and redistributed to islands. This has the benefit of reducing cross-thread communication, which potentially comes with cache locality issues.

The preamble of `src/bin/feruscore.rs` is straightforward:

```
extern crate feruscore;
extern crate num_cpus;
extern crate rand;

use feruscore::individual::*;
use feruscore::mars::*;
use rand::{thread_rng, Rng};
use std::fs::{self, DirBuilder, File};
use std::io::{self, BufWriter};
use std::path::{Path, PathBuf};
use std::sync::atomic::{AtomicUsize, Ordering};
use std::sync::mpsc;
use std::{thread, time};
```

The configuration and reporting globals are somewhat reduced, compared to the last chapter:

```
// configuration
const POPULATION_SIZE: usize = 1_048_576 / 8;
const CHROMOSOME_SIZE: u16 = 100;
const CORE_SIZE: u16 = 8000;
const GENE_MUTATION_CHANCE: u32 = 100;
const ROUNDS: u16 = 100;

// reporting
static GENERATIONS: AtomicUsize = AtomicUsize::new(0);
```

The `report` function is almost entirely the same as in last chapter, except there are a few less atomic variables to read from. checkpoint is also almost entirely the same. We'll skip reprinting both functions in the interest of space. What is new is `sort_by_tournament`:

```
fn sort_by_tournament(mars: &mut Mars,
                      mut population: Vec<Individual>)
    -> Vec<Individual>
{
    let mut i = population.len() - 1;
    while i >= (population.len() / 2) {
        let left_idx = thread_rng().gen_range(0, i);
        let mut right_idx = left_idx;
        while left_idx == right_idx {
            right_idx = thread_rng().gen_range(0, i);
        }
        match mars.compete(ROUNDS,
                           &population[left_idx],
                           &population[right_idx])
        {
            Winner::Right(_) => {
                population.swap(i, right_idx);
            }
            Winner::Left(_) => {
                population.swap(i, left_idx);
            }
            Winner::Tie => {}
        }
        i -= 1;
    }
```

```
    population
}
```

This function sorts a population by fitness, based on the result of competition inside the passed `Mars`. The reader will note that it's not a true sort in that the last element is the most fit, but that the last element is the winner of the first tournament, the second to last the winner of the second tournament, and so forth. Only `population.len() / 2` competitions are held and the resulting champions compare favorably to a population precisely sorted by fitness. The reader is encouraged to experiment with their own implementation of `sort_by_tournament`. Now, let's look at the `main` function:

```
fn main() {
    let mut out_recvs = Vec::with_capacity(num_cpus::get());
    let mut in_sends = Vec::with_capacity(num_cpus::get());

    let total_children = 128;

    let thr_portion = POPULATION_SIZE / num_cpus::get();
    for _ in 0..num_cpus::get() {
        let (in_snd, in_rcv) = mpsc::sync_channel(1);
        let (out_snd, out_rcv) = mpsc::sync_channel(1);
        let mut population: Vec<Individual> = (0..thr_portion)
            .into_iter()
            .map(|_| Individual::new(CHROMOSOME_SIZE))
            .collect();
        population.pop();
        population.pop();
        population.push(ringers::imp(CORE_SIZE));
        population.push(ringers::dwarf(CORE_SIZE));
        let _ = thread::spawn(move || island(in_rcv,
                                             out_snd,
                                             total_children));
        in_snd.send(population).unwrap();
        in_sends.push(in_snd);
        out_recvs.push(out_rcv);
    }
```

The total number of islands in a feruscore run will vary by the number of CPUs available on-system. Each island causes two

synchronous MPSC channels to be created, one for the main thread to push populations into the worker thread and one for the worker thread to push populations back to the main. The implementation refers to these as `in_*` and `out_*` senders and receivers. You can see again that we're building up a population of random `Individual` warriors and pushing the ringers in, though the island population is not `POPULATION_SIZE`, but an even split of `POPULATION_SIZE` by the number of available CPUs. The reporting thread is started after the island threads have their populations, mostly just to avoid UI spam:

```
    let _ = thread::spawn(report);
```

The report function is much the same as it was in the previous chapter's discussion of feruscore and I'll avoid listing it here for brievity's sake.

The final chunk of the `main` function is the recombining loop. When the island threads finish their competitions, they write into their out sender, which gets picked up by the recombination loop:

```
    let mut mars =
MarsBuilder::default().core_size(CORE_SIZE).freeze();
    let mut global_population: Vec<Individual> =
        Vec::with_capacity(POPULATION_SIZE);
    loop {
        for out_rcv in &mut out_recvs {
            let mut pop = out_rcv.recv().unwrap();
            global_population.append(&mut pop);
        }
```

Once all the islands have been merged together, the whole lot is shuffled:

```
        assert_eq!(global_population.len(), POPULATION_SIZE);
        thread_rng().shuffle(&mut global_population);
```

That's one generation done. We `checkpoint`, this time doing an additional tournament to pull the save winners from the global population:

```
        let generation = GENERATIONS.fetch_add(1, Ordering::Relaxed);
        if generation % 100 == 0 {
            global_population = sort_by_tournament(&mut mars,
                                                   global_population);
            checkpoint(generation, &global_population)
                .expect("could not checkpoint");
        }
```

Finally, the population is split back up and sent to the island threads:

```
        let split_idx = global_population.len() / num_cpus::get();

        for in_snd in &mut in_sends {
            let idx = global_population.len() - split_idx;
            let pop = global_population.split_off(idx);
            in_snd.send(pop).unwrap();
        }
    }
}
```

Now, what is island doing? Well, it's an infinite loop pulling from the population assignment receiver:

```
fn island(
    recv: mpsc::Receiver<Vec<Individual>>,
    snd: mpsc::SyncSender<Vec<Individual>>,
    total_children: usize,
) -> () {
    let mut mars =
MarsBuilder::default().core_size(CORE_SIZE).freeze();

    while let Ok(mut population) = recv.recv() {
```

Now a `Mars` is allocated above the loop, meaning we'll only ever do `num_cpu::get()` allocations of this structure per feruscore run. Also recall that `Receiver::recv` blocks when there is no data in the channel,

so island threads don't burn up CPU when there's no work for them. The interior of the loop should be familiar. First, the `Individual` warriors are put into competition:

```
        // tournament, fitness and selection of parents
        population = sort_by_tournament(&mut mars, population);
```

Reproduction is done by taking high-ranking members of the population and producing two children into the low reaches of the population until the total number of children needed per generation is reached:

```
        // reproduce
        let mut child_idx = 0;
        let mut parent_idx = population.len() - 1;
        while child_idx < total_children {
            let left_idx = parent_idx;
            let right_idx = parent_idx - 1;
            parent_idx -= 2;

            population[child_idx] = population[left_idx]
                                    .reproduce(&population[right_idx]);
            child_idx += 1;
            population[child_idx] = population[left_idx]
                                    .reproduce(&population[right_idx]);
            child_idx += 1;
        }
```

New to this implementation, we also introduce random new population members just before the newly introduced children:

```
        for i in 0..32 {
            population[child_idx + i] =
Individual::new(CHROMOSOME_SIZE)
        }
```

Random infusion can help tamp down premature convergence in a population. Remember, simulated evolution is a kind of state-space

search. Something else that has changed is that all members of the population have a chance of mutation:

```
        // mutation
        for indv in population.iter_mut() {
            indv.mutate(GENE_MUTATION_CHANCE);
        }
```

Finally, we push the population back up to the recombination thread, having worked the population over thoroughly:

```
        snd.send(population).expect("could not send");
    }
}
```

With all the changes made here—cutting back on small allocations, reducing total number of competitions per generation, removing pmars parsing plus spawn overhead—the implementation is substantially faster. Recall that the implementation in the last chapter struggled to peak 500 competitions—or BATTLES as the UI has it—per second. Here's a report diagnostic from a recent eight-hour run I did:

```
GENERATION(5459):
    RUNTIME (sec):  31248
    BATTLES:        41181
       BATTLES/s:   14340
    FITNESS:
        00...10:    151
        11...20:    2
        21...30:    0
        31...40:    1
        41...50:    0
        51...60:    0
        61...60:    2
        71...70:    1
        81...80:    0
        91...100:   41024
```

That's 14,000 battles per second on an 8,000-sized core memory, or around 650 generations per hour. Still not blazingly fast, but enough that with some time you can start to get pretty mediocre players being produced. Reducing core memory size will improve the runtime, as will limiting the maximum length of the warriors. Those who are interested in building a better evolver would do well to investigate different ways of assessing fitness, of introducing more ringers, and seeing whether porting `sim_mw` into Rust wouldn't improve runtime some. This simulator doesn't support the full scope of instructions that pMARS does, so that's also an area for improvement.

I'd love to hear about what you come up with.

Embedding Lua into Rust

Many of the programs we've discussed in this book use a *report* thread to notify the user of the running behavior of the program. Each of these report functions have been coded in Rust, an unchanging part of the executable. What if, however, we wanted end-users to be able to supply their own reporting routine? Or, consider the cernan project (`https://crates.io/crates/cernan`), discussed previously in this book, which supports a *programmable filter*, an online data-stream filter that can be programmed by end-users without changing the cernan binary. How do you pull such a trick off?

A common answer, not just in Rust but in many compiled languages, is to embed a Lua interpreter (`https://www.lua.org/`) and read user programs in at startup. It's such a common answer, in fact, that there are many Lua embeddings to choose from in the crates ecosystem. We'll use rlua (`https://crates.io/crates/rlua`) here as it's a safe choice and the project documentation is very good. Other Lua embeddings suit different ambitions. Cernan uses a different, not necessarily safe embedding, for example, because we need to allow end-users to define their own functions.

In the previous chapter, we wrote a project called `sniffer` whose purpose was threefold—collect Ethernet packets on an interface, report about them, and echo the Ethernet packet back. Let's take that program and adapt it so that users can decide how to report with custom scripts. The `Cargo.toml` file of the project is a little different, including the rlua dependency and dropping the thread-hungry alternative executable:

```
[package]
name = "sniffer"
version = "0.2.0"
authors = ["Brian L. Troutwine <brian@troutwine.us>"]

[dependencies]
pnet = "0.21"
rlua = "0.13"

[[bin]]
name = "sniffer"
```

The preamble to the sniffer program, defined in `src/bin/sniffer.rs`, holds no surprises for us:

```
   extern crate pnet;
   extern crate rlua;

   use pnet::datalink::Channel::Ethernet;
   use pnet::datalink::{self, DataLinkReceiver, MacAddr,
                        NetworkInterface};
   use pnet::packet::ethernet::{EtherType, EthernetPacket};
   use rlua::{prelude, Function, Lua};
   use std::fs::File;
   use std::io::{prelude::*, BufReader};
   use std::path::Path;
   use std::sync::atomic::{AtomicUsize, Ordering};
   use std::sync::mpsc;
   use std::{env, thread};
```

We're importing `Function` and `Lua` from the rlua crate, but that's about all that is new. The `SKIPPED_PACKETS` and `Payload` details are unchanged:

```
static SKIPPED_PACKETS: AtomicUsize = AtomicUsize::new(0);

#[derive(Debug)]
enum Payload {
    Packet {
        source: MacAddr,
        destination: MacAddr,
        kind: EtherType,
    },
```

```
    Pulse(u64),
}
```

I have removed the echoing of Ethernet packets in this example as it's not necessarily a kind thing to do on a busy Ethernet network. `watch_interface`, as a result, is a little more svelte than it used to be:

```
fn watch_interface(mut rx: Box<DataLinkReceiver>, snd:
mpsc::SyncSender<Payload>) {
    loop {
        match rx.next() {
            Ok(packet) => {
                let packet = EthernetPacket::new(packet).unwrap();

                let payload: Payload = Payload::Packet {
                    source: packet.get_source(),
                    destination: packet.get_destination(),
                    kind: packet.get_ethertype(),
                };
                if snd.try_send(payload).is_err() {
                    SKIPPED_PACKETS.fetch_add(1, Ordering::Relaxed);
                }
            }
            Err(e) => {
                panic!("An error occurred while reading: {}", e);
            }
        }
    }
}
```

The `timer` function is also unchanged:

```
fn timer(snd: mpsc::SyncSender<Payload>) -> () {
    use std::{thread, time};
    let one_second = time::Duration::from_millis(1000);

    let mut pulses = 0;
    loop {
        thread::sleep(one_second);
        snd.send(Payload::Pulse(pulses)).unwrap();
        pulses += 1;
    }
}
```

The `main` function now picks up an additional argument, intended to be the on-disk path of a Lua function:

```
fn main() {
    let interface_name = env::args().nth(1).unwrap();
    let pulse_fn_file_arg = env::args().nth(2).unwrap();
    let pulse_fn_file = Path::new(&pulse_fn_file_arg);
    assert!(pulse_fn_file.exists());
    let interface_names_match = |iface: &NetworkInterface| {
        iface.name == interface_name
    };

    // Find the network interface with the provided name
    let interfaces = datalink::interfaces();
    let interface = interfaces
        .into_iter()
        .filter(interface_names_match)
        .next()
        .unwrap();
```

The function is intended to handle the `Payload::Pulse` that comes in from the timer thread. The gather function from the previous chapter no longer exists. To actually do something with the user-supplied function, we'll have to create a new Lua VM via `rlua::Lua::new()` and then load the function into it, like so:

```
    let lua = Lua::new();
    let pulse_fn = eval(&lua, pulse_fn_file, Some("pulse")).unwrap();
```

The `eval` function is brief and mostly to do with reading from disk:

```
fn eval<'a>(lua: &'a Lua, path: &Path, name: Option<&str>)
    -> prelude::LuaResult<Function<'a>>
{
    let f = File::open(path).unwrap();
    let mut reader = BufReader::new(f);

    let mut buf = Vec::new();
    reader.read_to_end(&mut buf).unwrap();
```

```
    let s = ::std::str::from_utf8(&buf).unwrap();

    lua.eval(s, name)
}
```

The lua-specific bit there is `lua.eval(s, name)`. This evaluates the Lua code read from disk and returns function, a Rust-callable bit of Lua. Any user-supplied name in the Lua source itself is ignored and the name exists to add context to error messages on the Rust side of things. rlua does not expose loadfile in its Rust API, though other Lua embeddings do.

The next bit of the `main` function is mostly unchanged, though the packet buffer channel has been increased from 10 elements to `100`:

```
    let (snd, rcv) = mpsc::sync_channel(100);

    let timer_snd = snd.clone();
    let _ = thread::spawn(move || timer(timer_snd));

    let _iface_handler = match datalink::channel(&interface,
                                                 Default::default())
    {
        Ok(Ethernet(_tx, rx)) => {
            let snd = snd.clone();
            thread::spawn(|| watch_interface(rx, snd))
        }
        Ok(_) => panic!("Unhandled channel type"),
        Err(e) => panic!(
            "An error occurred when creating the datalink channel: {}",
            e
        ),
    };
```

Now, things are going to change. First, we create three Lua tables for storing the components of the `Payload::Packet`:

```
    let destinations = lua.create_table().unwrap();
    let sources = lua.create_table().unwrap();
    let kinds = lua.create_table().unwrap();
```

In the previous chapter, we used three `HashMap`s but, now, we need something we can easily pass into Lua. The main thread is responsible for the role that gather used to play: collecting `Payload` instances. This saves a thread and means we don't have to carefully arrange for sending the Lua VM across thread boundaries:

```
    while let Ok(payload) = rcv.recv() {
        match payload {
            Payload::Packet {
                source,
                destination,
                kind,
            } => {
                let d_cnt = destinations
                                .get(destination.to_string())
                                .unwrap_or(0);
                destinations
                    .set(destination.to_string(), d_cnt + 1)
                    .unwrap();

                let s_cnt =
sources.get(source.to_string()).unwrap_or(0);
                sources.set(source.to_string(), s_cnt + 1).unwrap();

                let k_cnt = kinds.get(kind.to_string()).unwrap_or(0);
                kinds.set(kind.to_string(), k_cnt + 1).unwrap();
            }
```

Here, we've started pulling payloads from the receiver and have handled `Payload::Packets` coming across. Notice how the tables we created just before this loop are now being populated. The same basic aggregation is in play; keep a running tally of the `Packet` components coming in. A more adventurous reader might extend this program to allow for the Lua side to build its own aggregation. Now all that remains is to handle `Payload::Pulse`:

```
            Payload::Pulse(id) => {
                let skipped_packets = SKIPPED_PACKETS
                                      .swap(0, Ordering::Relaxed);
                pulse_fn
                    .call::<_, ()>((
```

```
                        id,
                        skipped_packets,
                        destinations.clone(),
                        sources.clone(),
                        kinds.clone(),
                    ))
                    .unwrap()
            }
        }
    }
}
```

The `pulse_fn` created earlier is called with a four-tuple of arguments—the pulse ID, the `SKIPPED_PACKETS`, and the aggregation tables. We don't expect `pulse_fn` to have any kind of return value, hence the `<_, ()>` bit. This version of sniffer includes an example packet function, defined in `examples/pulse.lua`:

```
   function (id, skipped_packets, dest_tbl, src_tbl, kind_tbl)
      print("ID: ", id)
      print("SKIPPED PACKETS: ", skipped_packets)
      print("DESTINATIONS:")
      for k,v in pairs(dest_tbl) do
         print(k, v)
      end
      print("SOURCES:")
      for k,v in pairs(src_tbl) do
         print(k, v)
      end
      print("KINDS:")
      for k,v in pairs(kind_tbl) do
         print(k, v)
      end
   end
```

The reader will note that it does basically the same work as the former gather function once did. Running sniffer works the same way as before. Be sure to `cargo build --release` and then:

```
> ./target/release/sniffer en0 examples/pulse.lua
ID:     0
SKIPPED PACKETS:    0
```

```
DESTINATIONS:
5c:f9:38:8b:4a:b6   11
8c:59:73:1d:c8:73   16
SOURCES:
5c:f9:38:8b:4a:b6   16
8c:59:73:1d:c8:73   11
KINDS:
Ipv4    27
ID:     1
SKIPPED PACKETS:    0
DESTINATIONS:
5c:f9:38:8b:4a:b6   14
8c:59:73:1d:c8:73   20
SOURCES:
5c:f9:38:8b:4a:b6   20
8c:59:73:1d:c8:73   14
KINDS:
Ipv4    34
```

The allocation-conscious reader will have noticed there's a lot of cloning going on here. That's true. The primary compilation is interop with Lua's GC. A safe Rust interface must assume that data passed down to Lua will be garbage-collected. But, Lua does support a concept of user data, in which the programmer associates opaque blobs with Lua functions to manipulate it. Lua also supports light user data, which is very similar to user data except that the light variant is associated with a pointer to memory. The UserData type in RLua is quite well done and the ambitious reader might do well to build a `PacketAggregation` type that implements `UserData` to avoid all the cloning.

Combining a high-level language into a systems language is often a game of trade-offs between memory management complexity, end-user burden, and initial programming difficulty. rlua does an excellent job of landing on the safer side of these trade-offs. Something like mond, in use in cernan, less so, but with more flexibility in use.

Embedding Rust

So far, we've seen how to embed C and Lua into Rust. But, what if you want to combine Rust into other programming languages? Doing so is a very handy technique for improving runtime performance in interpreted languages, making memory-safe extensions where once you might have been using C or C++. If your target high-level language has difficulty with concurrency embedding, Rust is a further win. Python programs suffer in this regard—at least those implemented on CPython or PyPy—because of the Global Interpreter Lock, an internal mutex that locks objects' bytecode. Offloading computation of large blocks of data into a Rust + Rayon extension, for example, can be both straightforward to program and improve computation speed.

Well, great. How do we make this sort of thing happen? Rust's approach is simple: if you can embed C, you can embed Rust.

Into C

Let's embed some Rust into C. The quantiles library (`https://crates.io/crates/quantiles`)—discussed in `Chapter 4`, *Sync and Send – the Foundation of Rust Concurrency*—implements online summarization algorithms. These algorithms are easy to get wrong, in terms of producing incorrect results, but more often in storing more points than strictly necessary. A good deal of work has gone into quantiles to ensure the algorithms implemented there are near the theoretical minimum storage requirements, and so it makes sense to reuse this library for online summarization in C, rather than redo all that work.

Specifically, let's expose a `quantiles::ckms::CKMS<f32>` to C (`https://docs.rs/quantiles/0.7.1/quantiles/ckms/struct.CKMS.html`). We have to make the type concrete as C lacks any manner of generics in its types, but that's okay.

The Rust side

There are new things in the `Cargo.toml` file that we need to discuss:

```
[package]
name = "embed_quantiles"
version = "0.1.0"
authors = ["Brian L. Troutwine <brian@troutwine.us>"]

[dependencies]
quantiles = "0.7"

[lib]
name = "embed_quantiles"
crate-type = ["staticlib"]
```

Note specifically that we're building the embed_quantiles library with `crate-type = ["staticlib"]`. What does that mean? The Rust compiler is capable of making many kinds of linkages, and usually they're just implicit. For instance, a binary is `crate-type = ["bin"]`. There's a longish list of different link types, which I've included in the *Further reading* section. The interested reader is encouraged to look through them. What we'd like to produce here is a statically linked library, otherwise every C programmer that tries to make use of embed_quantiles is going to need Rust's shared libraries installed on their system. Not… not great. A staticlib crate will archive all the bits of the Rust standard library needed by the Rust code. The archive can then be linked to C as normal.

Okay, now, if we're going to produce a static archive that C can call, we've got to export Rust functions with a C ABI. Put another way, we've got to put a C skin on top of Rust. C++ programmers will be

familiar with this tactic. The sole Rust file in this project is `src/lib.rs` and its preamble is what you might expect:

```
extern crate quantiles;

use quantiles::ckms::CKMS;
```

We've pulled in the quantiles library and have imported `CKMS`. No big deal. Check this out, though:

```
#[no_mangle]
pub extern "C" fn alloc_ckms(error: f64) -> *mut CKMS<f32> {
    let ckms = Box::new(CKMS::new(error));
    Box::into_raw(ckms)
}
```

Hey! There's a bunch of new things. First, `#[no_mangle]`? A static library has to export a symbol for the linker to, well, link. These symbols are functions, static variables, and so forth. Now, usually Rust is free to fiddle with the names that go into a symbol to include information, such as module location or, really, whatever else the compiler wants to do. The exact semantics of mangling are undefined, as of this writing. If we're going to be calling a function from C, we have to have the exact symbol name to refer to. `no_mangle` turns off mangling, leaving us with our name as written. It does mean we have to be careful not to cause symbol collisions. Similar to importing functions, the `extern C` here means that this function should be written out to obey the C ABI. Technically, we could also have written this as `extern fn`, leaving the C off as the C ABI is the implicit default.

`alloc_ckms` allocates a new `CKMS`, returning a mutable pointer to it. Interop with C requires raw pointers, which, you know, makes sense. We do have to be very conscious of memory ownership when embedding Rust—does Rust own the memory, implying we need to provide a free function? Or, does the other language own the

memory? More often than not, it's easier to keep ownership with Rust, because to free memory, the compiler will need to know the type's size in memory. By passing a pointer out, as we're doing here, we've kept C in the dark about the size of `CKMS`. All the C side of this project knows is that it has an *opaque struct* to deal with. This is a common tactic in C libraries, for good reason. Here's freeing a `CKMS`:

```
#[no_mangle]
pub extern "C" fn free_ckms(ckms: *mut CKMS<f32>) -> () {
    unsafe {
        let ckms = Box::from_raw(ckms);
        drop(ckms);
    }
}
```

Notice that in `alloc_ckms`, we're boxing the `CKMS`—forcing it to the heap—and in `free_ckms` we're building a boxed `CKMS` from its pointer. We discussed boxing and freeing memory in the context of raw pointers extensively in `Chapter 3`, *The Rust Memory Model – Ownership, References and Manipulation*. Inserting a value into the `CKMS` is straightforward enough:

```
#[no_mangle]
pub extern "C" fn ckms_insert(ckms: &mut CKMS<f32>, value: f32) ->
() {
    ckms.insert(value)
}
```

Querying requires a little explanation:

```
#[no_mangle]
pub extern "C" fn query(ckms: &mut CKMS<f32>, q: f64,
                        quant: *mut f32)
    -> i8
{
    unsafe {
        if let Some((_, res)) = ckms.query(q) {
            *quant = res;
            0
```

```
            } else {
                -1
            }
        }
    }
```

Signaling error conditions in a C API is tricky. In Rust, we return some kind of compound type, such as an `Option`. There's no such thing in C without building an error signaling struct for your API. In addition to the error-struct approach it's common to either write well-known nonsense into the pointer where the answer will be written or return a negative value. C expects to have its answer written into a 32-bit float being pointed to by quant, and there's no easy way to write nonsense into a numeric value. So, query returns an `i8`; zero on success, negative on a failure. A more elaborate API would differentiate failures by returning different negative values.

That's it! When you run `cargo build --release`, a static library obeying the C ABI will get kicked out into `target/release`. We're ready to link it into a C program.

The C side

Our C program is `c_src/main.c`. We need a few system headers before we can define the `embed_quantiles` interface:

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
```

The only slightly unusual one is `time.h`. We pull that in because we're going to be pushing random floats into the `CKMS` structure. Not cryptographically secure randomness, mind. Here's the C view of the Rust API we just created:

```
  struct ckms;

  struct ckms* alloc_ckms(double error);
  void free_ckms(struct ckms* ckms);
  void ckms_insert(struct ckms* ckms, float value);
  int8_t query(struct ckms* ckms, double q, float* quant);
```

Notice the `CKMS` struct . That's the opaque struct. C is now aware there is a structure but it doesn't know how big it is. That's okay. Each of our functions only operate on a pointer to this structure, which C does know the size of. The `main` function is brief:

```
  int main(void) {
    srand(time(NULL));

    struct ckms* ckms = alloc_ckms(0.001);

    for (int i=1; i <= 1000000; i++) {
      ckms_insert(ckms, (float)rand()/(float)(RAND_MAX/10000));
```

```
    }

    float quant = 0.0;
    if (query(ckms, 0.75, &quant) < 0) {
      printf("ERROR IN QUERY");
      return 1;
    }
    printf("75th percentile: %f\n", quant);

    free_ckms(ckms);
    return 0;
  }
```

We allocate a `CKMS`, whose error bound is 0.001, load 1 million random floats into the `CKMS`, and then query for the 75th quantile, which should be around 7,500. Finally, the function frees the `CKMS` and exits. Short and sweet.

Now, building and linking `libembed_quantiles.a` is fairly easy with a clang compiler, and a little fiddly with GCC. I've gone ahead and included a Makefile, which has been tested on OS X and Linux:

```
CC ?= $(shell which cc)
CARGO ?= $(shell which cargo)
OS := $(shell uname)

run: clean build
	./target/release/embed_quantiles

clean:
	$(CARGO) clean
	rm -f ./target/release/embed_quantiles

build:
	$(CARGO) build --release
ifeq ($(OS),Darwin) # Assumption: CC == Clang
	$(CC) -std=c11 -Wall -Werror -Wpedantic -O3 c_src/main.c \
		-L target/release/ -l embed_quantiles \
		-o target/release/embed_quantiles
else # Assumption: CC == GCC
	$(CC) -std=c11 -Wall -Werror -Wpedantic -O3 c_src/main.c \
		-L target/release/ -lpthread -Wl,–no-as-needed -ldl \
		-lm -lembed_quantiles \
```

```
                -o target/release/embed_quantiles
endif

.PHONY: clean run
```

I don't have a Windows machine to test with so, uh, when this inevitably doesn't work I really do apologize. Hopefully there's enough here to figure it out quickly. Once you have the Makefile in place, you should be able to run make run and see something similar to the following:

```
> make run
/home/blt/.cargo/bin/cargo clean
rm -f ./target/release/embed_quantiles
/home/blt/.cargo/bin/cargo build --release
   Compiling quantiles v0.7.1
   Compiling embed_quantiles v0.1.0
    Finished release [optimized] target(s) in 0.91 secs
cc -std=c11 -Wall -Werror -Wpedantic -O3 c_src/main.c \
        -L target/release/ -lpthread -Wl,–no-as-needed -ldl \
        -lm -lembed_quantiles \
        -o target/release/embed_quantiles
./target/release/embed_quantiles
75th percentile: 7499.636230
```

The percentile value will move around some, but it ought to be close to 7,500, as it is here.

Into Python

Let's look at embedding Rust into Python. We'll write a function to sum up the trailing zeros in a cytpes array, built by Python. We can't link statically into Python as the interpreter is already compiled and linked, so we'll need to create a dynamic library. The `Cargo.toml` project reflects this:

```
[package]
name = "zero_count"
version = "0.1.0"
authors = ["Brian L. Troutwine <brian@troutwine.us>"]

[dependencies]

[lib]
name = "zero_count"
crate-type = ["dylib"]
```

The sole Rust file, `src/lib.rs`, has a single function in it:

```
   #[no_mangle]
   pub extern "C" fn tail_zero_count(arr: *const u16, len: usize) ->
   u64 {
       let mut zeros: u64 = 0;
       unsafe {
           for i in 0..len {
               zeros += (*arr.offset(i as isize)).trailing_zeros() as
   u64
           }
       }
       zeros
   }
```

The `tail_zero_count` function takes a pointer to an array of `u16`s and the length of that array. It zips through the array, calling `trailing_zeros()`

on each `u16` and adding the value to a global sum: `zeros`. This global sum is then returned. In the top of the project, run `cargo build --release` and you'll find the project's dynamic library—possibly `libzero_count.dylib`, `libzero_count.dll`, or `libzero_count.so`, depending on your host—in `target/release`. So far, so good.

Calling this function is now up to Python. Here's a small example, which lives at `zero_count.py` in the root of the project:

```
import ctypes
import random

length = 1000000
lzc = ctypes.cdll.LoadLibrary("target/release/libzero_count.dylib")
arr = (ctypes.c_uint16 * length)(*[random.randint(0, 65000) for _ in
range(0, length)])
print(lzc.tail_zero_count(ctypes.pointer(arr), length))
```

We import the cytpes and random libraries, then load the shared library—here, pegged to OS X's naming convention—and bind it to `lzc`. Do edit this to read `[so|dll]` if you're running this on an operating system other than OS X. Once `lzc` is bound, we use the ctypes API to create a random array of `u16` values and call `tail_zero_count` on that array. Python is forced to allocate the full array before passing it into Rust using this approach, so don't increase the length too much. Running the program is a matter of calling Python's `zero_count.py`, like so:

```
> python zero_count.py
1000457
>  python zero_count.py
999401
> python zero_count.py
1002295
```

Coping with opaque struct pointers—as we did in the C example—is well-documented in the ctypes documentation. Rust will not know

the difference; it's just kicking out objects conforming to the C ABI.

Into Erlang/Elixir

In this last section of the chapter, we'll investigate embedding Rust into the BEAM, the virtual machine that underpins Erlang and Elixir. The BEAM is a sophisticated work-stealing scheduler system that happens to be programmable. Erlang processes are small C structs that bounce around between scheduler threads and carry enough internal information to allow interpretation for a fixed number of instructions. There's no concept of shared memory in Erlang/Elixir: communication between different concurrent actors in the system *must* happen via message passing. There are many benefits to this and the VM does a great deal of work to avoid copying when possible. Erlang/Elixir processes receive messages into a message queue, a double-ended threadsafe queue of the kind we've discussed throughout the book.

Erlang and Elixir are rightly known for their efficiency at handling high-scale IO in a real-time fashion. Erlang was invented at Ericsson to serve as telephony control software, after all. What these languages are not known for is serial performance. Sequential computations in Erlang are relatively slow, it's just that it's so *easy* to get concurrent computations going that this is sort of made up for. Sort of. Sometimes, you need raw, serial performance.

Erlang has a few answers for this. A *Port* opens up an external OS process with a bidirectional byte stream. We saw an approach much like this in the previous chapter with feruscore embedding pmars. A *Port Driver* is a linked shared object, often written in C. The Port Driver interface has facilities for giving interrupt hints to BEAM's schedulers and has much to recommend it. A Port Driver *is* a process and must be able to handle the things an Erlang process normally

does: servicing its message queue, dealing with special interrupt signals, cooperatively scheduling itself, and so on. These are non-trivial to write. Finally, Erlang supports a Natively Implemented Function (NIF) concept. These are simpler than Port Drivers and are synchronous, being simply callable functions that happen to be written in a language other than Erlang. NIFs are shared libraries and are often implemented in C. Both Port Drivers and NIFs have a serious downside: memory issues will corrupt the BEAM and knock your application offline. Erlang systems tend to be deployed where fault-tolerance is a major factor and segfaulting the VM is a big no-no.

As a result, there's a good deal of interest in the Erlang/Elixir communities towards Rust. The Rustler project (`https://crates.io/crates/rustler`) aims to make combining Rust into an Elixir project as NIFs a simple matter. Let's take a look at a brief example project, presented by Sonny Scroggin at Code BEAM 2018 in San Francisco—beamcoin (`https://github.com/blt/beamcoin`). We'll discuss the project at SHA `3f510076990588c51e4feb1df990ce54ff921a06`.

> *The beamcoin project is not listed in its entirety. We've mostly dropped the build configuration. You can find the full listing in this book's source repository.*

The build system for Elixir—the BEAM language that Rustler targets natively—is called Mix. Its configuration file is `mix.exs` at the root of the project:

```
defmodule Beamcoin.Mixfile do
  use Mix.Project

  def project do
    [app: :beamcoin,
     version: "0.1.0",
     elixir: "~> 1.5",
     start_permanent: Mix.env == :prod,
     compilers: [:rustler] ++ Mix.compilers(),
     deps: deps(),
     rustler_crates: rustler_crates()]
```

```
  end

  def application do
    [extra_applications: [:logger]]
  end

  defp deps do
    [{:rustler, github: "hansihe/rustler", sparse: "rustler_mix"}]
  end

  defp rustler_crates do
    [beamcoin: [
      path: "native/beamcoin",
      mode: mode(Mix.env)
    ]]
  end

  defp mode(:prod), do: :release
  defp mode(_), do: :debug
end
```

There's a fair bit here that we won't get into. Note, however, this section:

```
  defp rustler_crates do
    [beamcoin: [
      path: "native/beamcoin",
      mode: mode(Mix.env)
    ]]
  end
```

Embedded in the project under `native/beamcoin` is the Rust library we'll be exploring. Its cargo configuration, at `native/beamcoin/Cargo.toml`, is:

```
[package]
name = "beamcoin"
version = "0.1.0"
authors = []

[lib]
name = "beamcoin"
path = "src/lib.rs"
crate-type = ["dylib"]
```

```
[dependencies]
rustler = { git = "https://github.com/hansihe/rustler", branch =
"master" }
rustler_codegen = { git = "https://github.com/hansihe/rustler", branch
= "master" }
lazy_static = "0.2"
num_cpus = "1.0"
scoped-pool = "1.0.0"
sha2 = "0.7"
```

Nothing too surprising here. A dynamic library, called libbeamcoin, will be produced when Rust compiles this and we've seen almost all the dependencies before. rustler and `rustler_codegen` are the interface and compiler generators for Rustler, respectively. `rustler_codegen` removes a lot of boilerplate C extern work we might otherwise have to do. sha2 is a crate from the RustCrypto project that, well, implements the sha2 hashing algorithm. Beamcoin is kind of a joke project. The idea is to distribute numbers between threads and count the zeros at the back of the sha2-256 hash, mining those with a pre-set number of zeros. This would be a *very* slow thing to do in Erlang but, as we'll see, is a relatively fast computation in Rust. The scoped-pool crate is a threadpooling library that is Sendable, meaning it can be placed in a `lazy_static!`

The preamble for `src/lib.rs` is straightforward enough:

```
#[macro_use]
extern crate lazy_static;
extern crate num_cpus;
extern crate scoped_pool;
extern crate sha2;
#[macro_use]
extern crate rustler;

use rustler::{thread, Encoder, Env, Error, Term};
use scoped_pool::Pool;
use sha2::Digest;
use std::mem;
use std::sync::atomic::{AtomicBool, Ordering};
use std::sync::{mpsc, Arc};
```

We've seen much of this before. The Rustler imports are analogs of the Erlang C NIF API. The difficulty of mining is controlled by a top-level usize, called `DIFFICULTY`:

```
const DIFFICULTY: usize = 6;
```

This value controls the total number of zeros that are required to be found at the back of a hash before it is declared minable. Here's our thread pool:

```
lazy_static! {
    static ref POOL: Pool = Pool::new(num_cpus::get());
}
```

I mentioned earlier that the BEAM maintains its own scheduler threads. This is true. The beamcoin NIF also maintains its own, separate pool of threads to distribute mining over. Now, Rustler reduces boilerplate but cannot totally remove it. We must, for instance, tell BEAM which interpreted functions to associate with that symbol and pre-define *atoms* for use:

```
   rustler_atoms! {
       atom ok;
       atom error;
   }

   rustler_export_nifs! {
       "Elixir.Beamcoin",
       [("native_mine", 0, mine)],
       None
   }
```

An Erlang atom is a named constant. These are extremely common data types in Erlang/Elixir programs. The threads in the pool are each given a chunk of the positive integers to search, stripped according to

their thread number. The pool workers use an MPSC channel to communicate back to the mining thread that a result has been found:

```
#[derive(Debug)]
struct Solution(u64, String);

fn mine<'a>(caller: Env<'a>, _args: &[Term<'a>])
    -> Result<Term<'a>, Error>
{
    thread::spawn::<thread::ThreadSpawner, _>(caller, move |env| {
        let is_solved = Arc::new(AtomicBool::new(false));
        let (sender, receiver) = mpsc::channel();

        for i in 0..num_cpus::get() {
            let sender_n = sender.clone();
            let is_solved = is_solved.clone();

            POOL.spawn(move || {
                search_for_solution(i as u64, num_cpus::get() as
u64,
                                    sender_n, is_solved);
            });
        }

        match receiver.recv() {
            Ok(Solution(i, hash)) => (atoms::ok(), i,
hash).encode(env),
            Err(_) => (
                atoms::error(),
                "Worker threads disconnected before the \
                 solution was found".to_owned(),
            ).encode(env),
        }
    });

    Ok(atoms::ok().encode(caller))
}
```

The `search_for_solution` function is a small loop:

```
fn search_for_solution(
    mut number: u64,
    step: u64,
    sender: mpsc::Sender<Solution>,
```

```
    is_solved: Arc<AtomicBool>,
) -> () {
    let id = number;
    while !is_solved.load(Ordering::Relaxed) {
        if let Some(solution) = verify_number(number) {
            if let Ok(_) = sender.send(solution) {
                is_solved.store(true, Ordering::Relaxed);
                break;
            } else {
                println!("Worker {} has shut down without \
                         finding a solution.", id);
            }
        }
        number += step;
    }
}
```

At the top of the loop, every thread in the pool checks to see whether any other thread has mined a beamcoin. If one has, the function exits and the thread is available for new work in the pool. Otherwise, `verify_number` is called. That function is:

```
fn verify_number(number: u64) -> Option<Solution> {
    let number_bytes: [u8; 8] = unsafe {
        mem::transmute::<u64, [u8; 8]>(number)
    };
    let hash = sha2::Sha256::digest(&number_bytes);

    let top_idx = hash.len() - 1;
    // Hex chars are 16 bits, we have 8 bits. /2 is conversion.
    let trailing_zero_bytes = DIFFICULTY / 2;

    let mut jackpot = true;
    for i in 0..trailing_zero_bytes {
        jackpot &= hash[top_idx - i] == 0;
    }

    if jackpot {
        Some(Solution(number, format!("{:X}", hash)))
    } else {
        None
    }
}
```

The number is passed in, transmuted into a byte array of 8 members, and hashed. If the hash has the appropriate number of trailing zero bytes, jackpot is true at the end of the function, and the number plus its hash are returned. The careful reader will have noted that the Rust module exports a NIF named `native_mine`. Erlang NIFs, generally speaking, have a system language component with a BEAM-native implementation as a fallback. The system language NIF implementation is called `native_*` by tradition.

The final piece here is the Elixir module to wrap the native NIF bits. This module is called `Beamcoin` and is `lib/beamcoin.ex`:

```
defmodule Beamcoin do
  use Rustler, otp_app: :beamcoin

  require Logger

  def mine do
    :ok = native_mine()

    start = System.system_time(:seconds)
    receive do
      {:ok, number, hash} ->
        done = System.system_time(:seconds)
        total = done - start
        Logger.info("Solution found in #{total} seconds")
        Logger.info("The number is: #{number}")
        Logger.info("The hash is: #{hash}")
      {:error, reason} ->
        Logger.error(inspect reason)
    end
  end

  def native_mine, do: :erlang.nif_error(:nif_not_loaded)
end
```

After installing Elixir (`https://elixir-lang.org/install.html`), you can move to the root of the project, execute `mix deps.get` to get the project's dependencies, and then use `MIX_ENV=prod iex -S mix` to bring up the Elixir repl. That latter command should look something like this:

```
> MIX_ENV=prod iex -S mix
Erlang/OTP 20 [erts-9.3] [source] [64-bit] [smp:4:4] [ds:4:4:10]
[async-threads:10] [hipe] [kernel-poll:false] [dtrace]

Compiling NIF crate :beamcoin (native/beamcoin)...
    Finished release [optimized] target(s) in 0.70 secs
Interactive Elixir (1.6.4) - press Ctrl+C to exit (type h() ENTER for
help)
iex(1)>
```

At the prompt, type `Beamcoin.mine` and hit *Enter*. After a few seconds, you should see:

```
iex(1)> Beamcoin.mine

23:30:45.243 [info]  Solution found in 3 seconds

23:30:45.243 [info]  The number is: 10097471

23:30:45.243 [info]  The hash is:
2354BB63E90673A357F53EBC96141D5E95FD26B3058AFAD7B1F7BACC9D000000
:ok
```

Or, something like it.

Building NIFs in the BEAM is a complicated topic, one we've hardly touched on here. Sonny Scroggin's talk, included in the *Further reading* section, covers that nuance in detail. If you're curious about how the BEAM functions, I've included a talk of mine on that subject in the *Further reading* section as well. Decades of careful effort have gone into the BEAM and it really shows.

Summary

In this chapter, we discussed embedding languages in Rust and vice versa. Rust is an incredibly useful programming language in its own right, but has been designed with care to interoperate with the existing language ecosystem. Delegating difficult concurrency work in a memory-unsafe environment into Rust is a powerful model. Mozilla's work on Firefox has shown that path to be fruitful. Likewise, there are decades' worth of well-tested libraries niche domains—weather modeling, physics, amusing programming games from the 1980s—that could, theoretically, be rewritten in Rust but are probably better incorporated behind safe interfaces.

This chapter is the last that aims to teach you a new, broad skill. If you've made it this far in the book, thank you. It's been a real pleasure writing for you. You should, hopefully, now have a solid foundation for doing low-level concurrency in Rust and the confidence to read through most Rust codebases you come across. There's a lot going on in Rust and I hope it seems more familiar now. In the next, and last, chapter of the book, we'll discuss the future of Rust, what language features apropos this book are coming soon, and how they might bring new capabilities to us.

Further reading

- *FFI examples written in Rust*, available at `https://github.com/alexcrichton/rust-ffi-examples`. This repository by Alex Crichton is a well-done collection of FFI examples in Rust. The documentation in this book on this topic is quite good, but it never hurts to pour through working code.
- *Hacker's Delight*, Henry Warren Jr. If you enjoyed the bit fiddling present in this chapter's take on feruscore, you'll love Hacker's Delight. It's old now and some of its algorithms no longer function on 64-bit words, but it's still well worth reading, especially if you, like me, work to keep fixed-width types as small as possible.
- *Foreign Function Interface*, available at `https://doc.rust-lang.org/nomicon/ffi.html`. The Nomicon builds a higher-level wrapper for the compression library snappy. This wrapper is extended in ways we did not touch on here, specifically with regard to C callbacks and vardic function calls.
- *Global Interpreter Lock*, available at `https://wiki.python.org/moin/GlobalInterpreterLock`. The GIL has long been the bane of multiprocessing software written in Python. This wiki entry discusses the technical details mandating the GIL and the historical attempts to remedy the situation.

- *Guided Tour*, available at `https://github.com/chucklefish/rlua/blob/master/examples/guided_tour.rs`. The rlua crate includes a guided tour module, which is well-documented and runnable. I haven't seen this approach to documentation in other projects and I warmly encourage you to check it out. First, it's helpful for learning rlua. Second, it's well-written and empathetic to the reader: a fine example of technical writing.
- *Linkage*, available at `https://doc.rust-lang.org/reference/linkage.html`. This is the Rust Reference chapter on linking. The details here are very specific, but that's often necessary when being explicit about linking. The common reader will more or less use the information we've covered in this chapter but there's always some new domain requiring specific knowledge.
- *Rust Inside Other Languages*, available at `https://doc.rust-lang.org/1.2.0/book/rust-inside-other-languages.html`. This chapter in the Rust Book covers similar ground to this chapter—embedding Rust—but at a faster clip and with different high-level languages. Specifically, The Book covers embedding Rust into both Ruby and NodeJS, which we did not.
- *FFI in Rust - writing bindings for libcpuid*, available at `http://siciarz.net/ffi-rust-writing-bindings-libcpuid/`. Zbigniew Siciarz has been writing about Rust and writing in Rust for a good while. You may know him from his *24 days of Rust* series. In this post, Sicarz documents the process of building a safe wrapper for libcpuid, a library whose job is to poll the OS for information about the user's CPU.

- *Taking Elixir to the Metal with Rust*, Sonny Scroggin, available at `https://www.youtube.com/watch?v=lSLTwWqTbKQ`. In this chapter we demonstrated Beamcoin, a combination of Elixir and Rust in the same project. Integrating NIFs into a BEAM system is a complicated subject. This talk, presented at NDC London 2017, is warmly recommended as an introduction to the subject.
- *Piecemeal Into Space: Reliability, Safety and Erlang Principles*, Brian L. Troutwine, available at `https://www.youtube.com/watch?v=pwoaJvrJE_U`. There's a great deal of work that's gone into the BEAM over the decades, earning those languages a key place in fault-tolerant software deployments. Exactly how the BEAM functions is something of a mystery without inspection. In this talk, I cover the BEAM's semantic model and then discuss it's implementation at a high-level.

Futurism – Near-Term Rust

We've covered a lot of material in this book.

If you already knew parallel programming well from another systems programming language, I hope that now you've got a confident handle on the way Rust operates. Its ownership model is strict, and can feel foreign at first. Unsafe Rust, if my own experiences are any kind of general guide, feels like more familiar ground. I hope that you too have come to appreciate the general safety of Rust, as well as its ability to do fiddly things in memory when needed. I find being able to implement unsafe code that is then wrapped up in a safe interface nothing short of incredible.

If you knew parallel programming fairly well from a high-level, garbage-collected programming language, then I hope that this book has served as an introduction to parallel programming at a systems level. As you know, memory safety does not guarantee correct operation, hence this book's continual focus on testing—generative and fuzz, in addition to performance inspection—and careful reasoning about models. Rust, I have argued, is one of the easier systems languages to do parallel programming well in because of the language's focus on ownership management with lifetimes and the strong, static type system. Rust is not immune to bugs—in fact, it's vulnerable to a wide variety—but these are common to all languages on modern hardware, and Rust does manage to solve a swath of memory-related issues.

If you already knew Rust, but didn't know much about parallel programming, I very much hope that this book has managed to convince you of one thing—concurrency is not magic. Parallel

programming is a wide, wide field of endeavor, one that takes a great deal of study, but it can be understood and mastered. Moreover, it need not be mastered all at once. The shape of this book argues for one path among several. May this book be the first of many on your journey.

Whatever your background, thank you for reading through to the end. I know that, at times, the material covered was not easy and it must have taken some real willpower to get through it all. I appreciate it. I wrote this book to be what I would have wanted for myself ten years ago. It was a real pleasure getting the opportunity to write it out for you.

By the end of this chapter, we will have:

- Discussed the future of SIMD in Rust
- Discussed the future of async IO in Rust
- Discussed the possibility of specialization being stablized and in what fashion
- Discussed more extensive testing methods for Rust
- Introduced various avenues to get involved with the Rust community

Technical requirements

This chapter has no software requirements.

You can find the source code for this book's projects at `https://github.com/PacktPublishing/Hands-On-Concurrency-with-Rust`. Chapter 10 has its source code under `Chapter10/`.

Near-term improvements

The focus of 2018's Rust development has been stabilization. Since the 1.0 days in 2015, many important crates were nightly-only, whether because of modifications to the compiler or because of the fear of standardizing too quickly on awkward APIs. This fear was reasonable. At the time, Rust was a new language and it changed *very* drastically in the lead-up to 1.0. A new language takes time to resolve into its natural style. By the end of 2017, the community had come to a general feeling that a stabilization cycle was in order, that some natural expression of the language had more or less been established, and that in areas where this was not true, it could be established, with some work.

Let's discuss this stabilization work with regards to the topics we've followed throughout the book.

SIMD

In this book, we discussed thread-based concurrency. In `Chapter 8`, *High-Level Parallelism – Threadpools, Parallel Iterators, and Processes*, we took to a higher level of abstraction with Rayon's parallel iterators. Underneath, as we saw, rayon uses a sophisticated work-stealing threadpool. Threads are merely one approach to concurrency on modern CPUs. In a sense, serial programs are parallel on CPUs with deep lookahead pipelines and sophisticated branch predictors, as we saw in `Chapter 2`, *Sequential Rust Performance and Testing*. Exploiting this parallelism is a matter of carefully structuring your data and managing access to it, among the other details we discussed. What we have not gone into in this book is how to exploit a modern CPUs' ability to perform the same operation on contiguous memory in parallel *without* resorting to threads—vectorization. We haven't looked at this because, as I write this, the language is only now getting minimal support in the standard library for vectorization instructions.

Vectorization comes in two variants—MIMD and SIMD. MIMD stands for multiple instructions, multiple data. SIMD stands for single instructions, single data. The basic idea is this: A modern CPU can apply an instruction to a contiguous, specifically sized block of data in parallel. That's it. Now, consider how many loops we've written in this book where we've done the exact same operation to every element of the loop. More than a few! Had we investigated string algorithms or looked into doing numeric calculations—encryption, physics simulations, or the like—we would have had more such loops in this book.

As I write this, the merger of SIMD vectorization into the standard library is held up behind RFC 2366 (`https://github.com/rust-lang/rfcs/pull/2366`), with ambitions to having x86 SIMD in a stable channel by year's end, with other architectures to follow. The reader may remember from `Chapter 1`, *Preliminaries – Machine Architecture and Getting Started with Rust*, that the memory systems of x86 and ARM are quite different. Well, this difference extends to their SIMD systems. It *is* possible to exploit SIMD instructions using stable Rust via the stdsimd (`https://crates.io/crates/stdsimd`) and simd (`https://crates.io/crates/simd`) crates. The stdsimd crate is the best for programmers to target as it will eventually be the standard library implementation.

A discussion of SIMD is beyond the scope of this book, given the remaining space. Suffice it to say that getting the details right with SIMD is roughly on a par with the difficulty of getting the details right in atomic programming. I expect that, post stabilization, the community will build higher-level libraries to exploit SIMD approaches without requiring a great deal of pre-study, though potentially with less chance for finely tuned optimizations. The faster (`https://crates.io/crates/faster`) crate, for example, exposes SIMD-parallel iterators in the spirit of rayon. The programming model there, from an end-user perspective, is extremely convenient.

Hex encoding

Let's look at an example from the stdsimd crate. We won't dig in too deeply here; we're only out to get a sense of the structure of this kind of programming. We've pulled stdsimd at `2f86c75a2479cf051b92fc98273daaf7f151e7a1`. The file we'll examine is `examples/hex.rs`. The program's purpose is to `hex` encode stdin. For example:

```
> echo hope | cargo +nightly run --release --example hex -p stdsimd
    Finished release [optimized + debuginfo] target(s) in 0.52s
     Running `target/release/examples/hex`
686f70650a
```

Note that this requires a nightly channel, and not a stable one, we've used so far in this book. I tested this on nightly, 2018-05-10. The details of the implementation are shifting so rapidly and rely on such unstable compiler features that you should make sure that you use the specific nightly channel that is listed, otherwise the example code may not compile.

The preamble to `examples/hex.rs` contains a good deal of information we've not seen in the book so far. Let's take it in two chunks and tease out the unknowns:

```
   #![feature(stdsimd)]
   #![cfg_attr(test, feature(test))]
   #![cfg_attr(feature = "cargo-clippy",
               allow(result_unwrap_used, print_stdout,
   option_unwrap_used,
                     shadow_reuse, cast_possible_wrap, cast_sign_loss,
                     missing_docs_in_private_items))]
```

```
    #[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
    #[macro_use]
    extern crate stdsimd;

    #[cfg(not(any(target_arch = "x86", target_arch = "x86_64")))]
    extern crate stdsimd;
```

Because we've stuck to stable Rust, we've not seen anything like `#![feature(stdsimd)]` before. What is this? Rust allows the adventurous user to enable in-progress features that are present in rustc, but that are not yet exposed in the nightly variant of the language. I say adventurous because an in-progress feature may maintain a stable API through its development, or it may change every few days, requiring updates on the consumer side. Now, what is `cfg_attr`? This is a conditional compilation attribute, turning on features selectively. You can read all about it in *The Rust Reference*, linked in the *Further reading* section. There are many details, but the idea is simple—turn bits of code on or off depending on user directives—such as testing—or the running environment.

This later is what `#[cfg(any(target_arch = "x86", target_arch = "x86_64"))]` means. On either x86 or x86_64, stdsimd will be externed with macros enabled and without them enabled when on any other platform. Now, hopefully, the rest of the preamble is self-explanatory:

```
#[cfg(test)]
#[macro_use]
extern crate quickcheck;

use std::io::{self, Read};
use std::str;

#[cfg(target_arch = "x86")]
use stdsimd::arch::x86::*;
#[cfg(target_arch = "x86_64")]
use stdsimd::arch::x86_64::*;
```

The `main` function of this program is quite brief:

```
fn main() {
    let mut input = Vec::new();
    io::stdin().read_to_end(&mut input).unwrap();
    let mut dst = vec![0; 2 * input.len()];
    let s = hex_encode(&input, &mut dst).unwrap();
    println!("{}", s);
}
```

The whole of stdin is read into the input and a brief vector to hold the hash, called `dst`. `hex_encode`, is also used. The first portion performs input validation:

```
fn hex_encode<'a>(src: &[u8], dst: &'a mut [u8]) -> Result<&'a str,
usize> {
    let len = src.len().checked_mul(2).unwrap();
    if dst.len() < len {
        return Err(len);
    }
```

Note the return—a `&str` on success and a `usize`—the faulty length—on failure. The rest of the function is a touch more complicated:

```
    #[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
    {
        if is_x86_feature_detected!("avx2") {
            return unsafe { hex_encode_avx2(src, dst) };
        }
        if is_x86_feature_detected!("sse4.1") {
            return unsafe { hex_encode_sse41(src, dst) };
        }
    }

    hex_encode_fallback(src, dst)
}
```

This introduces conditional compilation, which we saw previously, and contains a new thing too—`is_x86_feature_detected!`. This macro is new to the SIMD feature, and tests at runtime whether the given CPU feature is available at runtime via libcpuid. If it is, the feature is used. On an ARM processor, for instance, `hex_encode_fallback` will be

executed. On an x86 processor, one or the other avx2 or sse4.1, implementations will be followed, depending on the chip's exact capabilities. Which implementation is called is determined at runtime. Let's look into the implementation of `hex_encode_fallback` first:

```
fn hex_encode_fallback<'a>(
    src: &[u8], dst: &'a mut [u8]
) -> Result<&'a str, usize> {
    fn hex(byte: u8) -> u8 {
        static TABLE: &[u8] = b"0123456789abcdef";
        TABLE[byte as usize]
    }

    for (byte, slots) in src.iter().zip(dst.chunks_mut(2)) {
        slots[0] = hex((*byte >> 4) & 0xf);
        slots[1] = hex(*byte & 0xf);
    }

    unsafe { Ok(str::from_utf8_unchecked(&dst[..src.len() * 2])) }
}
```

The interior function `hex` acts as a static lookup table, and the for loop zips together the bytes from the `src` and a two-chunk pull from `dst`, updating `dst` as the loop proceeds. Finally, `dst` is converted into a `&str` and returned. It's safe to use `from_utf8_unchecked` here as we can verify that no non-utf8 characters are possible in `dst`, saving us a check.

Okay, now that's the fallback. How do the SIMD variants read? We'll look at `hex_encode_avx2`, leaving the sse4.1 variant to the ambitious reader. First, we see the reappearance of conditional compilation and the familiar function types:

```
#[target_feature(enable = "avx2")]
#[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
unsafe fn hex_encode_avx2<'a>(
    mut src: &[u8], dst: &'a mut [u8]
) -> Result<&'a str, usize> {
```

So far, so good. Now, look at the following:

```
    let ascii_zero = _mm256_set1_epi8(b'0' as i8);
    let nines = _mm256_set1_epi8(9);
    let ascii_a = _mm256_set1_epi8((b'a' - 9 - 1) as i8);
    let and4bits = _mm256_set1_epi8(0xf);
```

Well, the function `_mm256_set1_epi8` is certainly new! The documentation (`https://rust-lang-nursery.github.io/stdsimd/x86_64/stdsimd/arch/x86_64/fn._mm256_set1_epi8.html`) states the following:

"Broadcast 8-bit integer a to all elements of returned vector."

Okay. What vector? Many SIMD instructions are carried out in terms of vectors with implicit sizes, the size being defined by the CPU architecture. For instance, the function that we have been looking at returns an `stdsimd::arch::x86_64::__m256i`, a 256-bit-wide vector. So, `ascii_zero` is 256-bits-worth of zeros, `ascii_a` 256-bits-worth of `a`, and so forth. Each of these functions and types, incidentally, follows Intel's naming scheme. It's my understanding that the LLVM uses their own naming scheme, but Rust—in an effort to reduce developer confusion—maintains a mapping from the architecture names to LLVM's names. By keeping the architecture names, Rust makes it real easy to look up details about each intrinsic. You can see this in the *Intel Intrinsics Guide* at `mm256_set1_epi8` (`https://software.intel.com/sites/landingpage/IntrinsicsGuide/#text=_mm256_set1_epi8&expand=4676`).

The implementation then enters a loop, processing `src` in 256-bit-wide chunks until there are only 32 or fewer bits left:

```
    let mut i = 0_isize;
    while src.len() >= 32 {
        let invec = _mm256_loadu_si256(src.as_ptr() as *const _);
```

The variable `invec` is now 256 bits of `src`, with `_mm256_loadu_si256` being responsible for performing a load. SIMD instructions have to operate on their native types, you see. In the actual reality of the

machine, there is no type, but specialized registers. What comes next is complex, but we can reason through it:

```
        let masked1 = _mm256_and_si256(invec, and4bits);
        let masked2 = _mm256_and_si256(
            _mm256_srli_epi64(invec, 4),
            and4bits
        );
```

Okay, `_mm256_and_si256` performs a bitwise boolean-and of two `__m256i`, returning the result. `masked1` is the bitwise boolean-and of `invec` and `and4bits`. The same is true for `masked2`, except we need to determine what `_mm256_srli_epi64` does. The Intel document state the following:

"Shift packed 64-bit integers in a right by imm8 while shifting in zeros, and store the results in dst."

`invec` is subdivided into four 64-bit integers, and these are shifted right by four bits. That is then boolean-and'ed with `and4bits`, and `masked2` pops out. If you refer back to `hex_encode_fallback` and squint a little, you can start to see the relationship to `(*byte >> 4) & 0xf`. Next up, two comparison masks are put together:

```
        let cmpmask1 = _mm256_cmpgt_epi8(masked1, nines);
        let cmpmask2 = _mm256_cmpgt_epi8(masked2, nines);
```

`_mm256_cmpgt_epi8` greater-than compares 8-bit chunks of the 256-bit vector, returning the larger byte. The comparison masks are then used to control the blending of `ascii_zero` and `ascii_a`, and the result of that is added to `masked1`:

```
        let masked1 = _mm256_add_epi8(
            masked1,
            _mm256_blendv_epi8(ascii_zero, ascii_a, cmpmask1),
        );
        let masked2 = _mm256_add_epi8(
            masked2,
```

```
            _mm256_blendv_epi8(ascii_zero, ascii_a, cmpmask2),
        );
```

Blending is done bytewise, with the high bit in each byte determining whether to move a byte from the first vector in the `arg` list or the second. The computed masks are then interleaved, choosing high and low halves:

```
        let res1 = _mm256_unpacklo_epi8(masked2, masked1);
        let res2 = _mm256_unpackhi_epi8(masked2, masked1);
```

In the high half of the interleaved masks, the destination's first 128 bits are filled with the top 128 bits from the source, the remainder being filled in with zeros. In the lower half of the interleaved masks, the destination's first 128 bits are filled from the bottom 128 bits from the source, the remainder being zeros. Then, at long last, a pointer is computed into `dst` at 64-bit strides and further subdivided into 16-bit chunks, and the `res1` and `res2` are stored there:

```
        let base = dst.as_mut_ptr().offset(i * 2);
        let base1 = base.offset(0) as *mut _;
        let base2 = base.offset(16) as *mut _;
        let base3 = base.offset(32) as *mut _;
        let base4 = base.offset(48) as *mut _;
        _mm256_storeu2_m128i(base3, base1, res1);
        _mm256_storeu2_m128i(base4, base2, res2);
        src = &src[32..];
        i += 32;
    }
```

When the `src` is less than 32 bytes, `hex_encode_sse4` is called on it, but we'll leave the details of that to the adventurous reader:

```
    let i = i as usize;
    let _ = hex_encode_sse41(src, &mut dst[i * 2..]);

    Ok(str::from_utf8_unchecked(
        &dst[..src.len() * 2 + i * 2],
```

```
    ))
}
```

That was tough! If our aim here was to do more than get a flavor of the SIMD style of programming, then we'd probably need to work out by hand how these instructions work on a bit level. Is it worth it? This example comes from RFC 2325 (`https://github.com/rust-lang/rfcs/blob/master/text/2325-stable-simd.md`), *Stable SIMD*. Their benchmarking demonstrates that the fallback implementation we looked at first hashes at around 600 megabytes per second. The avx2 implementation hashes at 15,000 MB/s, a 60% performance bump. Not too shabby. And keep in mind that each CPU core comes equipped with SIMD capability. You could split such computations up among threads, too.

Like atomic programming, SIMD is not a panacea or even especially easy. If you need the speed, though, it's worth the work.

Futures and async/await

Asynchronous I/O is a hot topic in the Rust ecosystem right now. The C10K challenge we discussed briefly in `Chapter 8`, *High-Level Parallelism – Threadpools, Parallel Iterators, and Processes*, was resolved when operating systems introduced scalable I/O notification syscalls such as kqueue in the BSDs and epoll in Linux. Prior to epoll, kqueue, and friends, I/O notification suffered linear growth in processing time as the number of file descriptors increased—a real problem. The aggressively naive approach to networking we've taken in this book also suffers from linear blowup. Every TCP socket we opened for listening would need to be polled in a loop, underserving very high-volume connections and overserving all others.

Rust's abstraction for scalable network I/O is mio (`https://crates.io/crates/mio`). If you go poking around inside of mio, you'll find that it is a clean adapter for the I/O notification subsystems of a few common platforms: Windows, Unixen, and that's about it. The user of mio doesn't get much in the way of additional support. Where do you store your mio sockets? That's up to the programmer. Can you include multithreading? Up to you. Now, that's great for very specific use cases—cernan, for example, uses mio to drive its ingress I/O because we wanted very fine control over all of these details—but unless you have very specific needs, this is probably going to be a very tedious situation for you. The community has been investing very heavily in tokio (`https://crates.io/crates/tokio`), a framework for doing scalable I/O with essential things such as back-pressure, transaction cancellation, and higher-level abstractions to simplify request/response protocols. Tokio's been a fast-moving project, but will almost surely be one of the *key* projects in the coming years. Tokio is, fundamentally, a reactor-based (`https://en.wikipedia.org/wiki/R`

eactor_pattern) architecture; an I/O event becomes available and handlers you've registered are then called.

Reactor-based systems are simple enough to program if you have only one event to respond to. But, more often than you'd imagine, there exist dependency relationships between I/O events in a real system—an ingressing UDP packet results in a TCP round-trip to a network memory cache, which results in an egress UDP packet to the original ingress system, and another as well. Coordinating that is hard. Tokio—and the Rust ecosystem somewhat generally—has gone all-in on a promise library called futures (https://crates.io/crates/futures). Promises are a fairly tidy solution to asynchronous I/O: the underlying reactor calls a well-established interface, you implement a closure inside (or a trait to then slot inside) that interface, and everyone's code stays loosely coupled. Rust's futures are pollable, meaning you can hit the poll function on a future and it might resolve into a value or a notification that the value isn't ready yet.

This is similar to promise systems in other languages. But, as anyone who remembers the early introduction of NodeJS into the JavaScript ecosystem can attest to, promises without language support get weird and deeply nested fast. Rust's borrow checker made the situation harder, requiring boxing or full-on arcs where, in straight-line code, this would not have been necessary.

Languages with promises as a first-class concern generally evolve async/await syntaxes. An async function is a function in Rust that will return a future and little else. A Rust future is simple trait, extracted from the futures project. The actual implementation will live outside the language's standard library and allow for alternative implementations. The interior of an async function is not executed immediately, but only when the future is polled. The interior of the function executes through the polling until an await point is hit, resulting in a notification that the value is not yet ready and that whatever is polling the future should move on.

As in other languages, Rust's proposed async / await syntax really is a sugar over the top of an explicit callback chaining structure. But, because the async/await syntax and futures trait is moving into the compiler, rules can be added to the borrow-checking system to remove the current boxing concerns. I expect you'll see more futures in stable Rust once they become more convenient to interact with. Indeed, as we saw in `Chapter 8`, *High-Level Parallelism – Pools, Iterators, and Processes*, rayon is investing time in a futures-based interface.

Specialization

In `Chapter 2`, *Sequential Rust Performance and Testing*, we noted that the technique of *specialization* would be cut off from data structure implementors if they intended to target stable Rust. Specialization is a means by which a generic structure—say, a `HashMap`—can be implemented specifically for one type. Usually, this is done to improve performance—say, by turning a `HashMap` with `u8` keys into an array—or provide a simpler implementation when possible. Specialization has been in Rust for a good while, but limited to nightly projects, such as the compiler, because of soundness issues when interacting with lifetimes. The canonical example has always been the following:

```
trait Bad1 {
    fn bad1(&self);
}

impl<T> Bad1 for T {
    default fn bad1(&self) {
        println!("generic");
    }
}

// Specialization cannot work: trans doesn't know if T: 'static
impl<T: 'static> Bad1 for T {
    fn bad1(&self) {
        println!("specialized");
    }
}

fn main() {
    "test".bad1()
}
```

Lifetimes are erased at a certain point in the compilation process. String literals are static, but by the time dispatch to specialized type would occur there's not enough information in the compiler to decide. Lifetime equality in specialization also poses difficulties.

It seemed for a good while that specialization would miss the boat for inclusion into the language in 2018. Now, *maybe*, a more limited form of specialization will be included. A specialization will only apply if the implementation is generic with regard to lifetimes, does not introduce duplication into trait lifetimes, does not repeat generic type parameters, and all trait bounds are themselves applicable for specialization. The exact details are in flux as I write this, but do keep an eye out.

Interesting projects

There are many interesting projects in the Rust community right now. In this section, I'd like to look at two that I didn't get a chance to discuss at length in the book.

Fuzzing

In previous chapters we've used AFL to validate that our programs did not exhibit crashing behavior. While AFL is very commonly used, it's not the only fuzzer available for Rust. LLVM has a native library—libfuzzer (`https://llvm.org/docs/LibFuzzer.html`)—covering the same space, and the cargo-fuzz (`https://crates.io/crates/cargo-fuzz`) project acts as an executor. You might also be interested in honggfuzz-rs (`https://crates.io/crates/honggfuzz`), a fuzzer developed at Google for searching out security related violations. It is natively multithreaded—there is no need to spin up multiple processes manually—and can do network fuzzing. My preference, traditionally, has been to fuzz with AFL. The honggfuzz project has real momentum, and readers should give it a try in their own projects.

Seer, a symbolic execution engine for Rust

Fuzzing, as we discussed previously, explores the state space of a program using random inputs and binary instrumentation. This can be, as we've seen, slow. The ambition of symbolic execution (`https://en.wikipedia.org/wiki/Symbolic_execution`) is to allow the same exploration of state space, but without random probing. Searching for program crashes is one area of application, but it can also be used with proof tools. Symbolic execution, in a carefully written program, can let you demonstrate that your program can never reach error states. Rust has a partially implemented symbolic execution tool, seer (`https://github.com/dwrensha/seer`). The project uses z3, a constraint solver, to generate branching inputs at program branches. Seer's README, as of SHA `91f12b2291fa52431f4ac73f28d3b18a0a56ff32`, amusingly decodes a hash. This is done by defining the decoding of the hashed data to be a crashable condition. Seer churns for a bit and then decodes the hash, crashing the program. It reports the decoded value among the error report.

It's still early days for seer, but the potential is there.

The community

The Rust community is large and multi-faceted. It's so large, in fact, that it can be hard to know where to go with questions or ideas. More, books at the intermediate to advanced level will often assume that readers know as much about the language community as the author does. Having been mostly on the other side of the author, reader relationship I've always found this assumption frustrating. As an author, though, I now understand the hesitancy—none of the community information will stay up to date.

Oh well. Some of this information may be out of date by the time you get to it. Reader beware.

Throughout this book, we've referred to the crates ecosystem; crates.io (`https://crates.io/`) is *the* location for Rust source projects. The docs.rs (`https://docs.rs/`) is a vital resource for understanding crates, and is run by the Rust team. You can also `cargo docs` and get a copy of your project's dependency documentation locally. I'm often without Wi-Fi, and I find this very useful.

As with almost all things in Rust, the community itself is documented on this web page (`https://www.rust-lang.org/en-US/community.html`). IRC is fairly common for real-time, loose conversations, but there's a real focus on the web-facing side of Rust's communication. The Users Forum (`https://users.rust-lang.org/`) is a web forum intended for folks who use the language and have questions or announcements. The compiler people—as well as the standard library implementors, and so on—hang out on the Internals Forum(`https://internals.rust-lang.org/`). There is a good deal of overlap in the two communities, as you might expect.

Throughout this book, we've referenced various RFCs. Rust evolves through a request-for-comments system, all of which are centralized (`https://github.com/rust-lang/rfcs`). That's just for the Rust compiler and standard library. Many community projects follow a similar system—for example, crossbeam RFCs (`https://github.com/crossbeam-rs/rfcs`). These are well worth reading, by the way.

Should I use unsafe?

It's not uncommon to hear some variant of the following position—*I won't use any library that has an* `unsafe` *block in it*. The reasoning behind this position is that `unsafe`, well, advertises that the crate is potentially unsafe and might crash your otherwise carefully crafted program. That's true—kind of. As we've seen in this book, it's entirely possible to put together a project using `unsafe` that is totally safe at runtime. We've also seen that it's entirely possible to put together a project without `unsafe` blocks that flame out at runtime. The existence or absence of `unsafe` blocks shouldn't reduce the original programmer's responsibilities for due diligence—writing tests, probing the implementation with fuzzing tools, and so on. Moreover, the existence or absence of `unsafe` blocks does not relieve the user of a crate from that same responsibility. Any software, at some level, should be considered suspect unless otherwise demonstrated to be safe.

Go ahead and use the `unsafe` keyword. Do so when it's necessary. Keep in mind that the compiler won't be able to help you as much as it normally would—that's all.

Summary

In this last chapter of the book, we discussed near and midterm improvements coming to the Rust language—features being stabilized after a while out in the wilderness, the foundations of larger projects being integrated, research that may or may not pan out, and other topics. Some of these topics will surely have chapters of their own in the second edition of this book, should it get a new edition (Tell your friends to pick up a copy!). We also touched on the community, which is the place to find ongoing discussions, ask questions, and get involved. Then, we affirmed that, yes indeed, you should write in unsafe Rust when you have reason to do so.

Okay, that's it. That's the end of the book.

Further reading

- *Shipping specialization: a story of soundness*, available at https://aturon.github.io/blog/2017/07/08/lifetime-dispatch/. There's long been an ambition to see specialization available in stable Rust, and it's not for want of trying that it hasn't happened yet. In this blog post, Aaron Turon discusses the difficulties of specialization in 2017, introducing the Chalk logic interpreter in the discussion. Chalk is interesting in its own right, especially if you are interested in compiler internals or logic programming.
- *Maximally minimal specialization: always applicable impls*, available at http://smallcultfollowing.com/babysteps/blog/2018/02/09/maximally-minimal-specialization-always-applicable-impls/. This blog post by Niko Matsakis extends the topics covered in Turon's *Shipping specialization* article, discussing a *min-max* solution to the specialization soundness issue. The approach seemed to be the most likely candidate for eventual implementation, but flaws were discovered. Happily, not unresolvable flaws. This post is an excellent example of the preRFC conversation in the Rust community.
- *Sound and ergonomic specialization for Rust*, available at http://aturon.github.io/2018/04/05/sound-specialization/. This blog post addresses the issues in the min-max article and proposes

the implementation discussed in this chapter. It is, as of writing this book, the most likely to be implemented, unless some bright spark finds a flaw in it.

- *Chalk*, available at `https://github.com/rust-lang-nursery/chalk`. Chalk is really a very interesting Rust project. It is, according to the project description, a *PROLOG-ish interpreter written in Rust*. To date, Chalk is being used to reason about specialization in the Rust language, but there are plans to merge Chalk into `rustc` itself someday. The project README, as of `SHA 94a1941a021842a5fcb35cd043145c8faae59f08`, has a list of excellent articles on the applications of chalk.
- *The Unstable Book*, available at `https://doc.rust-lang.org/nightly/unstable-book/`. `rustc` has a large number of in-flight features. *The Unstable Book* is intended to be a best-effort collection of these features, the justifications behind them, and any relevant tracking issues. It is well worth a read, especially if you're looking to contribute to the compiler project.
- *Zero-cost futures in Rust*, available at `http://aturon.github.io/blog/2016/08/11/futures/`. This post introduced the Rust community to the futures project and explains the motivation behind the introduction well. The actual implementation details of futures have changed with time, but this article is still well worth a read.
- *Async / Await*, available at `https://github.com/rust-lang/rfcs/blob/master/text/2394-async_await.md`. RFC 2394 introduces the motivation for simplifying the ergonomics of futures in Rust, as well as laying out the implementation approach for it. The

RFC is, itself, a fine example of how Rust evolves—community desire turns into experimentation which then turns into support from the compiler.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

Rust Cookbook
Vigneshwer Dhinakaran

ISBN: 978-1-78588-025-4

- Understand system programming language problems and see how Rust provides unique solutions
- Get to know the core concepts of Rust to develop fast and safe applications
- Explore the possibility of integrating Rust units into existing applications to make them more efficient
- Achieve better parallelism, security, and performance
- Explore ways to package your Rust application and ship it for deployment in a production environment

- Discover how to build web applications and services using Rust to provide high-performance to the end user

Rust Standard Library Cookbook
Jan Nils Ferner, Daniel Durante

ISBN: 978-1-78862-392-6

- How to use the basic modules of the library: strings, command line access, and more.
- Implement collections and folding of collections using vectors, Deque, linked lists, and more.
- Handle various file types , compressing and decompressing data.
- Search for files with glob patterns.
- Implement parsing through various formats such as CSV, TOML, and JSON.
- Utilize drop trait , the Rust version of destructor.
- Resource locking with Bilocks.

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!