O'REILLY®

Rust Atomics and Locks

Low-Level Concurrency in Practice

Early Release

RAW & UNEDITED

Mara Bos

Rust Atomics and Locks

Low-Level Concurrency in Practice

Mara Bos

Rust Atomics and Locks

by Mara Bos

Printed in the United States of America.

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

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

Acquisitions Editor: Suzanne McQuade

Development Editor: Shira Evans

Production Editor: Elizabeth Faerm

Copyeditor: TO COME

Proofreader: TO COME

Indexer: TO COME

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

November 2022: First Edition

Revision History for the Early Release

- 2022-04-13: First Release
- 2022-06-23: Second Release

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

978-1-098-11938-6

[LSI]

Chapter 1. Basics of Rust Concurrency

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

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

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

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

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

Threads in Rust

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

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

Let's take a look at an example:

```
use std::thread;

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

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

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

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

THREAD ID

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

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

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

Surprisingly, part of the output seems to be missing.

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

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

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

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

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

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

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

The `.join()` method will wait until the thread has finished executing, and returns a `thread::Result`. If the thread did not successfully finish its function because it panicked, this will contain the panic message.

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

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

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

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

OUTPUT LOCKING

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

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

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

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

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

Here, ownership of `numbers` is transferred to the newly spawned thread. We have to use a `move` closure to transfer ownership. Otherwise, the closure would capture `numbers` by reference even though the new thread might outlive it, resulting in a compiler error.

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

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

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

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

let average = t.join().unwrap(); ❷

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

Here, the value returned by the thread's closure (1) is sent back to the main thread through the `join` method (2).

If `numbers` had been empty, the thread would've panicked while trying to divide by zero (1), and `join` would've returned that panic message instead, causing the main thread to panic too because of `unwrap` (2).

THREAD BUILDER

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

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

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

Scoped Threads

WARNING

As of June 2022, scoped threads are not *yet* part of the Rust standard library. It has been proposed and implemented as unstable feature, which is available using `#![feature(scoped_threads)]` on nightly versions of Rust.

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

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

How it works is best shown with an example:

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

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

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

❷ We can use `s` to spawn threads, which can borrow local variables like `numbers`.

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

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

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

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

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

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

THE LEAKPOCALYPSE

Before Rust 1.0, the standard library had a function named `thread::scoped` that would directly spawn a thread, just like `thread::spawn`. It allowed non-`'static` captures, because instead of a `JoinHandle`, it returned a `JoinGuard` which joined the thread when dropped. Any borrowed data only needed to outlive this `JoinGuard`. This seemed safe, as long as the `JoinGuard` gets dropped at some point.

Just before the release of Rust 1.0, it slowly became clear it's not possible to guarantee that something will be dropped. There are many ways, such as a cycle of reference counted nodes, that make it possible to forget about something without dropping it.

Eventually, in what some people refer to as "The Leakpocalypse", `thread::scoped` was removed and `mem::forget` was made *safe* to emphasize that forgetting (or leaking) things is always possible, and that `Drop` cannot be relied upon for correctness.

Only much later, in Rust 1.63, a new `thread::scoped` function was added with a new design that does not rely on `Drop` for correctness.

Shared Ownership and Reference Counting

So far we've looked at transferring ownership of a value to a thread using a `move` closure ("Threads in Rust") and borrowing data from longer-living parent threads ("Scoped Threads"). When sharing data between two threads where neither thread is guaranteed to outlive the other, neither of them can be the owner of that data. Any data shared between them will need to live as long as the longest living thread.

Statics

There are several ways to create something that's not owned by a single thread. The simplest one is a `static` value, which is *`owned`* by the entire program, instead of an invidual thread. In the following example, both threads can access `X`, but neither of them owns it:

```
static X: [i32; 3] = [1, 2, 3];

thread::spawn(|| dbg!(&X));
thread::spawn(|| dbg!(&X));
```

A `static` item has a constant initializer, is never dropped, and already exists before the main function of the program even starts. Every thread can borrow it, since it's guaranteed to always exist.

Leaking

Another way to share ownership is by *leaking* an allocation. Using `Box::leak`, one can release ownership of a `Box`, promising to never drop it. From that point on, the `Box` will live forever, without owner, allowing it to be borrowed by any thread for as long as the program runs.

```
let x: &'static [i32; 3] = Box::leak(Box::new([1, 2, 3]));

thread::spawn(move || dbg!(x));
thread::spawn(move || dbg!(x));
```

The `move` closure might make it look like we're moving ownership into the threads, but a closer look at the type of `x` reveals that we're only giving the threads *a reference* to the data.

TIP

References are `Copy`, meaning that when you *`move`* them, the original still exists, just like with an integer or boolean.

Note how the `'static` lifetime doesn't mean that the value lived since the start of the program, but only that it lives to the end of the program. The past is simply not relevant.

The downside of leaking a `Box` is that we're *leaking memory*. We allocate something, but never drop and deallocate it. This can be fine if this only happens a limited number of times. But if we kept doing this, the program would slowly run out of memory.

Reference Counting

To be able to drop and deallocate something, we can't completely give up its ownership. Instead, we can *share ownership*. By keeping track of the number of owners, we can make sure the value is only dropped when there are no owners left.

The Rust standard library provides this functionality through the `Rc` type, short for "`Reference Counted`". It is very similar to a `Box`, except cloning it will not allocate anything new, but instead increment a counter stored next to the contained value. Both the original and cloned `Rc` will refer to the same allocation; they *share ownership*.

```
let a = Rc::new([1, 2, 3]);
let b = a.clone();

assert_eq!(a.as_ptr(), b.as_ptr()); // Same allocation!
```

Dropping an `Rc` will decrement the counter. Only the last `Rc` will see the counter drop to zero, and will be the one dropping and deallocating the contained data.

If we try to send an `Rc` to another thread however, we are faced with the following compiler error:

```
error[E0277]: `Rc` cannot be sent between threads safely
    |
8   |     thread::spawn(move || dbg!(b));
    |                   ^^^^^^^^^^^^^^^
    |
```

As it turns out, `Rc` is not *thread safe* (more on that in "Thread Safety: Send and Sync"). If multiple threads had an `Rc` to the same allocation, they might try to modify the reference counter at the same time, which can give unpredictable results.

Instead, we can use `Arc`, which stands for "`Atomically Reference Counted`". It's identical to `Rc`, except it guarantees that modifications to the reference counter are indivisible — *atomic* — operations, making it safe to use it with multiple threads. (More on that in Chapter 2.)

```
let a = Arc::new([1, 2, 3]); ❶
let b = a.clone(); ❷

thread::spawn(move || dbg!(a)); ❸
thread::spawn(move || dbg!(b)); ❸
```

❶ We put an array in a new allocation together with a reference counter, which starts at one..

❷ Cloning the `Arc` will increment the reference count to two, and provides us with a second `Arc` to the same allocation.

❸ Both threads get their own `Arc` through which they can access the shared array. Both will decrement the reference counter when they drop their `Arc`. The last thread to drop its `Arc` will see the counter drop to zero, and will be the one to drop and deallocate the array.

NAMING CLONES

Having to give every clone of an `Arc` a different name can quickly make the code quite cluttered and hard to follow. While every clone of an `Arc` is a separate object, each clone represents the same shared value, which is not well reflected by naming each one differently.

Rust allows (and encourages) you to *shadow* variables by defining a new variable with the same name. If you do that in the same scope, the original variable cannot be named anymore. But by opening a new scope, a statement like `let a = a.clone();` can be used to re-use the same name within that scope, while leaving the original variable available outside the scope.

By wrapping a closure in a new scope (with `{}`), we can clone variables before moving them into the closure, without having to rename them.

```
let a = Arc::new([1, 2, 3]);

let b = a.clone();

thread::spawn(move || {
    dbg!(b);
});

dbg!(a);
```

The clone of the `Arc` lives in the same scope.
Each thread gets its own clone with a different name.

```
let a = Arc::new([1, 2,
3]);

thread::spawn({
    let a = a.clone();
    move || {
        dbg!(a);
    }
});

dbg!(a);
```

The clone of the `Arc` lives in a different scope.
We can use the same name in each thread.

Because ownership is shared, reference counting pointers (`Rc<T>` and `Arc<T>`) have the same restrictions as shared references (`&T`). They do not give you mutable access to their contained value, since the value might be borrowed by other code at the same time.

```
error[E0596]: cannot borrow data in an `Arc` as mutable
  |
6 |     a.sort();
  |     ^^^^^^^^
```

Borrowing and Data Races

In Rust, values can be borrowed in two ways:

Immutable Borrowing

> Borrowing something with `&` gives an *immutable reference*. Such a reference can be copied. Access to the data it references is shared between all copies of such a reference. As the name implies, the compiler does normally not allow you to *mutate* something through such a reference, since that might affect other code that's currently borrowing the same data.

Mutable Borrowing

> Borrowing something with `&mut` gives a *mutable reference*. A mutable borrow guarantees it's the only active borrow of that data, making sure that mutating the data will not change anything other code is currently looking at.

These two concepts together fully prevent *data races*: situations where one thread is mutating data while another is concurrently accessing it. Data races are generally *undefined behavior*, which means the compiler does not need to take these situations into account. It will simply assume they do not happen.

To clarify what that means, let’s take a look at an example:

```
fn f(a: &i32, b: &mut i32) {
    let before = *a;
    *b += 1;
    let after = *a;
    if before != after {
        x(); // never happens
    }
}
```

Here, we get an immutable reference to an integer, and store the value of the integer both before and after incrementing the integer that `b` refers to. The compiler is free to assume the fundamental rules about borrowing and data races are upheld, which means that `b` can’t possibly refer to the same integer as `a` does. In fact, nothing in the entire program can mutably borrow the integer that `a` refers to as long as `a` is borrowing it. Therefore, it can easily conclude that `*a` will not change and the condition of the `if` statement will never be true, and can completely remove the call to `x` from the program as an optimization.

It’s impossible to write a Rust program that breaks the compiler’s assumptions, other than by using an `unsafe` block to disable some of the compiler’s safety checks.

UNDEFINED BEHAVIOR

Languages like C, C++, and Rust have a set of rules that need to be followed to avoid something called *undefined behavior*. For example, one of Rust's rules is that there may never be more than one mutable reference to any object.

In Rust, it's only possible to break any of these rules when using `unsafe` code. That word doesn't mean that the code is incorrect or never safe to use, but rather that the compiler is not validating for you that the code is safe. If the code does violate these rules, it is called *unsound*.

The compiler is allowed to assume, without checking, that these rules are never broken. When broken, this results in something called *undefined behavior*, which we need to avoid at all costs. If we allow the compiler to make an assumption that is not actually true, it can easily result in more wrong conclusions about different parts of your code, affecting your whole program.

As a concrete example, let's take a look at a small snippet that uses the `get_unchecked` method on a slice.

```
let a = [123, 456, 789];
let b = unsafe { a.get_unchecked(index) };
```

The `get_unchecked` method gives us an element of the slice given its index, just like `a[index]`, but allows the compiler to assume the index is always within bounds, without any checks.

This mean that in this code snippet, because `a` is of length 3, the compiler may assume that `index` is less than three. It's up to us to make sure its assumption holds.

If we break this assumption, for example if we run this with `index` equal to 3, anything might happen. It might result in reading from memory whatever was stored in the bytes right after `a`. It might cause the program to crash. It might end up executing some entirely unrelated part of the program. It can cause all kinds of havoc.

Perhaps surprisingly, undefined behavior can even `travel back in time`, causing problems in code that precedes it. To understand how that can happen, imagine we had a `match` statement before our previous snippet, as follows:

```
match index {
    0 => x(),
    1 => y(),
    _ => z(index),
}

let a = [123, 456, 789];
let b = unsafe { a.get_unchecked(index) };
```

Becuase of the unsafe code, the compiler is allowed to assume `index` is only ever 0, 1, or 2. It may logically conclude that the last arm of our `match` statement will only ever match a 2, and thus that `z` is only ever called as `z(2)`. That conclusion might be used not only to optimize the `match`, but also to optimize `z` itself. This can include throwing out unused parts of the code.

If we execute this with an `index` of `3`, our program might attempt to execute parts that have been optimized away, resulting in completely unpredictable things, far before we get to the `unsafe` block on the last line. Just like that, undefined behavior can propagate through a whole program, both backwards and forwards, in often very unexpected ways.

When calling any `unsafe` functions, read its documentation carefully and make sure you fully understand its *safety requirements*: the assumptions you need to uphold, as the caller, to avoid undefined behavior.

Interior Mutability

The borrowing rules as introduced in the previous section are simple, but can be quite limiting. Especially when multiple threads are involved. Following these rules makes communication between threads extremely limited and almost impossible, since no data that's accessible by multiple threads can be mutated.

Luckily, there is an escape hatch: something called *interior mutability*. A data type with interior mutability slightly bends the borrowing rules. Under certain conditions, those types can allow mutation through an "immutable" reference.

In "Shared Ownership and Reference Counting", we've already seen one subtle example involving interior mutability. Both `Rc` and `Arc` mutate a reference counter, even though there might be multiple clones all using the same reference counter.

As soon as interior mutable types are involved, calling a reference "immutable" or "mutable" become confusing and inaccurate, since some things can be mutated though both. The more accurate terms are "shared" and "exclusive": a *shared reference* (`&T`) can be copied and shared with others, while a *exclusive reference* (`&mut T`) guarantees it's the only *exclusive borrowing* of that `T`. For most types, shared references do not allow mutation, but there are exceptions. Since in this book we will mostly be working with these exceptions, we'll use the more accurate terms in the rest of this book.

CAUTION

Keep in mind that interior mutability only bends the rules of shared borrowing, to allow mutation when shared. It does not change anything about exclusive borrowing. Exclusive borrowing still guarantees that there are no other active borrows. Creating multiple exclusive references to something (through unsafe code) is always undefined behavior, regardless of interior mutability.

Let's take a look at a few types with interior mutability, and how they can allow mutation through shared references without causing undefined behavior.

Cell

A `Cell<T>` simply wraps a `T`, but allows mutations through a shared reference. To avoid undefined behavior, it only allows you to copy the value out (if `T` is `Copy`), or replace it with another value as a whole. In addition, it can only be used within a single thread.

Let’s take a look at similar example as in the previous section, but this time using `Cell<i32>` instead of `i32`:

```
fn f(a: &Cell<i32>, b: &Cell<i32>) {
    let before = a.get();
    b.set(b.get() + 1);
    let after = a.get();
    if before != after {
        x(); // might happen
    }
}
```

Unlike last time, it is now possible for the `if` condition to be true. Because a `Cell<i32>` has interior mutability, the compiler can no longer assume its value won’t change as long as we have a shared reference to it. Both `a` and `b` might refer to the same value, such that mutating through `b` might affect `a` as well. It may still assume, however, that no other threads are accessing the cells concurrently.

The restrictions on a `Cell` are not always easy to work with. Since it can’t directly let us borrow the value it holds, we need to move a value out (leaving something in its place), modify it, then put it back, to mutate its contents.

```
fn f(v: &Cell<Vec<i32>>) {
    let mut v2 = v.take(); // Replaces the contents of the Cell
with an empty Vec
    v2.push(1);
    v.set(v2); // Put the modified Vec back
}
```

RefCell

Unlike a regular `Cell`, a `RefCell` does allow you to borrow its contents, at a small run-time cost. A `RefCell<T>` does not only hold a `T`, but also holds a counter that keeps track of any outstanding borrows. If you try to borrow it while it is already mutably borrowed, it will panic, which avoids undefined behavior. Just like a `Cell`, a `RefCell` can only be used within a single thread.

```
fn f(v: &RefCell<Vec<i32>>) {
    v.borrow_mut().push(1); // We can modify the `Vec` directly.
}
```

While `Cell` and `RefCell` can be very useful, they become rather useless when we need to do something with multiple threads. So let's move on to the types that are relevant for concurrency.

Mutex and RwLock

An `RwLock` or *reader-writer lock* is the concurrent version of a `RefCell`. An `RwLock<T>` holds a `T`, and tracks any outstanding borrows. However, unlike a `RefCell`, it does not panic on conflicting borrows. Instead, it blocks the current thread—putting it to sleep—while waiting for conflicting borrows to disappear. We'll just have to patiently wait for our turn with the data, after the other threads are done with it.

Borrowing the contents of an `RwLock` is called *locking*. By *locking* it we temporarily block concurrent conflicting borrows, allowing us to borrow it without causing data races.

A `Mutex` (short for *mutual exclusion*) is very similar, but conceptually slightly simpler. Instead of keeping track of the number of shared and exclusive borrows like an `RwLock`, it only allows exclusive borrows.

We'll go more into detail on these types in "Locking: Mutexes and RwLocks".

Atomics

The atomic types represent the concurrent version of a `Cell`, and are the main topic of Chapter 2 and X REF HERE FOR CH03. Like a `Cell`, they avoid undefined behavior by making us copy values in and out as a whole, without letting us borrow the contents directly.

Unlike a `Cell` though, they cannot be of arbitrary size. Because of this, there is no generic `Atomic<T>` type for any `T`, but there are only specific atomic types such as `AtomicU32` and `AtomicPtr`. Which ones are available depends on the platform, since they require support from the processor to avoid data races. (We'll dive into that in X REF HERE FOR CH04.)

Since they are so limited in size, atomics often don't directly contain the information that needs to be shared between threads. Instead, they are often used as a tool to make it possible to share other—often bigger—things between threads. When atomics are used to say something about other data, things can get surprisingly complicated.

UnsafeCell

An `UnsafeCell` is the primitive building block for interior mutability.

An `UnsafeCell<T>` wraps a `T`, but does not come with any conditions or restrictions to avoid undefined behavior. Instead, its `get` method just gives a raw pointer to the value it wraps, which can only be meaningfully used in `unsafe` blocks. It leaves it up to the user to use it in a way that does cause any undefined behavior.

Most commonly, an `UnsafeCell` is not used directly, but wrapped in another type that provides safety through a limited interface, such as `Cell` or `Mutex`. All types with interior mutability—including all types discussed above—are built on top of `UnsafeCell`.

Thread Safety: Send and Sync

In this chapter, we've seen several types that are not *thread safe*, types that can only be used on a single thread, such as `Rc`, `Cell`, and others. Since that restriction is needed to avoid undefined behavior, it's something the compiler needs to understand and check for you, so you can use these types without having to use `unsafe` blocks.

The language uses two special traits to keep track of which types can be safely used across threads:

Send

> A type is `Send` if it can be sent to another thread. That is, if ownership of a value of that type can be transferred to another thread. For example, `Arc<i32>` is `Send`, but `Rc<i32>` is not.

Sync

> A type is `Sync` if it can be shared with another thread. In other words, a type `T` is `Sync` if and only if a shared reference to that type, `&T`, is `Send`. For example, a `i32` is `Sync`, but a `Cell<i32>` is not.

All primitive types such as `i32`, `bool`, and `str` are all both `Send` and `Sync`.

Both of these traits are *auto traits*, which means that they are automatically implemented for your types based on their fields. A `struct` with fields that are all `Send` and `Sync`, is itself also `Send` and `Sync`.

The way to opt-out of either of these is to add a field to your type that does not implement the trait. For that purpose, the special `PhantomData<T>` type often comes in handy. That type is treated by the compiler as a `T`, except it doesn't actually exist at runtime. It's a zero sized type, taking no space.

Let's take a look at this struct:

```
struct X {
    handle: i32,
    _not_sync: PhantomData<Cell<()>>,
}
```

In this example, the struct `X` would be both `Send` and `Sync` if `handle` was its only field. However, we added a zero-sized `PhantomData<Cell<()>>` field, which is treated as if it were a `Cell<()>`. Since a `Cell<()>` is not `Sync`, neither is `X`. It is still `Send`, however, since all its fields implement `Send`.

Raw pointers (`*const T` and `*mut T`) are neither `Send` nor `Sync`, since the compiler doesn't know much about what they represent.

The way to opt-in to either of the traits is the same as with any other trait: an `impl` block to implement the trait for you type:

```
struct X {
    p: *mut i32,
}

unsafe impl Send for X {}
unsafe impl Sync for X {}
```

Note how implementing these traits requires the `unsafe` keyword, since the compiler cannot check for you if it's correct. It's a promise you make to the compiler, which it will just have to trust.

If you try to move something into another thread which is not `Send`, the compiler will politely stop you from doing that. Here is a small example to demonstrate that:

```
fn main() {
    let a = Rc::new(123);
    thread::spawn(move || {
        dbg!(a);
    });
}
```

Here, we try to send an `Rc<i32>` to a new thread, but `Rc<i32>`, unlike `Arc<i32>`, does not implement `Send`.

If we try to compile this, we're faced with the following error:

```
error[E0277]: `Rc<i32>` cannot be sent between threads safely
   --> src/main.rs:3:5
    |
3   |       thread::spawn(move || {
    |       ^^^^^^^^^^^^^ `Rc<i32>` cannot be sent between
threads safely
    |
    = help: within `[closure@src/main.rs:3:19: 5:6]`, the trait
`Send` is not implemented for `Rc<i32>`
    = note: required because it appears within the type
`[closure@src/main.rs:5:19: 5:6]`
note: required by a bound in `spawn`
```

The `thread::spawn` function requires its argument to be `Send`, and a closure is only `Send` if all of its captures are. If we try to capture something that's not `Send` our mistake is caught, protecting us from undefined behavior.

Locking: Mutexes and RwLocks

The most commonly used tool for sharing (mutable) data between threads is called a *mutex*, which is short for "`mutual exclusion`“. The job of a mutex is to ensure threads have exclusive access to some data by temporarily blocking other threads that try to access it at the same time.

Conceptually, a mutex has only two operations: lock and unlock, and two states: locked and unlocked. When a thread locks an unlocked mutex, the mutex is marked as locked and the thread can immediately continue. When a thread then attempts to lock an already locked mutex, that operation will *block*. The thread is put to sleep while it waits for the mutex to be unlocked. Unlocking is only possible on a locked mutex, and should be done by the same thread that locked it. If other threads are waiting to lock the mutex, unlocking will cause one of those threads to be woken up, so it can try to lock the mutex again and continue its course.

Protecting data with a mutex is simply the agreement between all threads that they will only access the data when they have the mutex locked. That way, no two threads can ever access that data concurrently and cause a data race.

Rust’s Mutex

The Rust standard library provides this functionality through `std::sync::Mutex<T>`. It is a generic type over a type `T`, where `T` is the type of the data the mutex is protecting mutual exclusive access to. By making this `T` part of the mutex, the data can only be accessed through the mutex, allowing for a safe interface that can guarantee all threads will uphold the agreement.

To ensure a locked mutex can only be unlocked by the thread that locked it, it does not have an `unlock()` method. Instead, its `lock()` method returns a special type called a `MutexGuard`. This guard represents the guarantee that we have locked the mutex. It behaves like an exclusive reference through the `DerefMut` trait, giving us exclusive access to the data the mutex protects. Unlocking the mutex is done by dropping the guard. When we drop the guard, we give up our ability to accesss the data, and the `Drop` implementation of the guard will unlock the mutex.

Let’s take a look at an example to see a mutex in practice:

```
fn main() {
    let n = Mutex::new(0);
    thread::scope(|s| {
        for _ in 0..10 {
            s.spawn(|| {
                let mut guard = n.lock().unwrap();
                for _ in 0..100 {
                  *guard += 1;
                }
            });
        }
    });
    assert_eq!(n.into_inner().unwrap(), 1000);
}
```

Here, we have a `Mutex<i32>`, a mutex protecting an integer, and spawn ten threads to each increment the integer one hundred times. Each thread will first lock the mutex to obtain a mutex guard through `lock()`, and then use that guard to access the integer and modify it. The `guard` is implicitly dropped right after, when that variable goes out of scope.

After the threads are done, we can safely remove the protection from the integer through `into_inner()`. The `into_inner` method takes an exclusive reference to the mutex, which guarantees nothing else can have a reference to the mutex anymore, making locking unnecessary.

Even though the increments happen in steps of one, a thread observing the integer would only ever see multiples of 100, since it can only look at the integer when the mutex is unlocked. Effectively, thanks to the mutex, the one hundred increments together are now a single undividable, *atomic*, operation.

To clearly see the effect of the mutex, we can make each thread wait a second before unlocking the mutex:

```
    ..
            s.spawn(|| {
                let mut guard = n.lock().unwrap();
                for _ in 0..100 {
                  *guard += 1;
                }
                thread::sleep(Duration::from_secs(1)); // Sleep
before unlocking the mutex.
            });
    ..
```

When you run the program now, you will see it will take about ten seconds to complete. Each thread only waits for one second, but the mutex will ensure only one thread at a time can do so.

If we drop the guard, and therefore unlock the mutex, before sleeping one second, we will see it happen in parallel instead.

```
    ..
            s.spawn(|| {
                    let mut guard = n.lock().unwrap();
                    for _ in 0..100 {
                        *guard += 1;
                    }
                    drop(guard); // Drop the guard before sleeping
one second.
                    thread::sleep(Duration::from_secs(1));
                });
    ..
```

With this change, this program will only take about one second, since now the ten threads can execute their one-second sleep at the same time. This shows the importance of keeping the amount of time a mutex is locked as short as possible. Keeping a mutex locked longer than necessary can completely nullify any benefits of parallelism, effectively forcing everything to happen in series instead.

Lock Poisoning

The `unwrap()` calls in the examples above relate to *lock poisoning*.

A `Mutex` in Rust gets marked as *poisoned* when a thread panics while it was holding the lock. The `lock` method returns an `Err` if the lock was poisoned.

This is a mechanism to protect against leaving the data that's protected by a mutex in an inconsistent state. If in our example above a thread would panic after incrementing the integer fewer than one hundred times, the mutex unlocks and the integer is left in an unexpected state where it is not a multiple of 100 anymore, possibly breaking assumptions made by other threads. Automatically marking the mutex as poisoned in that case forces the user to handle this possibility.

Calling `lock()` on a poisoned mutex still locks the mutex. The `Err` returned by `lock()` contains the mutex guard, allowing us to correct an inconsistent state or simply ignore the poison.

While lock poisoning might seem like a good idea, it is not often used in practice. Most code either ignores poison, or uses `unwrap()` to panic if the lock was poisoned, effectively propagating panics to all users of the mutex.

LIFETIME OF THE MUTEX GUARD

While it’s convenient and important for correctness that implicitly dropping a guard unlocks the mutex, it can sometimes lead to subtle surprises. If we assign the guard a name with a `let` statement (as in our examples above), it’s relatively straight forward to see when it will be dropped, since local variables are dropped at the end of the scope they are defined in. Still, not explicitly dropping a guard might lead to keeping the mutex locked for longer than necessary, as demonstrated in the examples above.

Using a guard *without* assigning it a name is also possible, and can be very convenient at times. Since a `MutexGuard` behaves like an exclusive reference to the protected data, we can directly use it without assigning a name to the guard first. For example, if you have a `Mutex<Vec<i32>>`, you can lock the mutex, push an item into the `Vec`, and unlock the mutex again, in a single statement:

```
list.lock().unwrap().push(1);
```

Any temporaries produced within a larger expression, such as the guard returned by `lock()`, will be dropped at the end of the statement. While this might seem obvious and reasonable, it leads to a common pitfall that usually involves a `match`, `if let` or `while let` statement. Here is an example that runs into this pitfall:

```
if let Some(item) = list.lock().unwrap().pop() {
    process_item(item);
}
```

If our intention was to lock the list, pop an item, unlock the list, and *then* process the item after the list is unlocked, we made a subtle but important mistake here. The temporary guard is not dropped until the end of the entire `if let` statement, meaning we needlessly hold on to the lock while processing the item.

Perhaps surprisingly, this does not happen for a similar `if` statement, such as in this example:

```
if list.lock().unwrap().pop() == Some(1) {
    do_something();
}
```

Here, the temporary guard does get dropped before the body of the `if` statement is executed. The reason is that the condition of a regular `if` statement is always a plain boolean, which cannot borrow anything. There is no reason to extend the lifetime of temporaries from the condition to the end of the statement. For an `if let` statement, however, that might not be the case. If we had used `front()` rather than `pop()` for example, `item` would be borrowing from the list, making it necessary to keep the guard around. Since the borrow checker is only really a check and does *not* influence when or in what order things are dropped, the same happens when we use `pop()`, even though that wouldn't have been necessary.

We can avoid this by moving the pop operation to a separate `let` statement. Then the `guard` is dropped at the end of that statement, before the `if let`:

```
let item = list.lock().unwrap().pop();
if let Some(item) = item {
    process_item(item);
}
```

Reader-Writer Lock

A mutex is only concerned with exclusive access. The `MutexGuard` will provide us an exclusive reference (`&mut T`) to the protected data, even if we only wanted to look at the data and a shared reference (`&T`) would have sufficed.

A reader-writer lock is a slightly more complicated version of a mutex that does understand the difference between exclusive and shared access, and can provide either. It has three states, unlocked, locked by a single *writer* (for exclusive access), or locked by any number of *readers* (for shared access). It is commonly used for data that is often read by multiple threads, but only updated once in a while.

The Rust standard library provides this lock through the `std::sync::RwLock<T>` type. It works very similar to the standard `Mutex`, except its interface is mostly split in two parts. Instead of a single `lock()` method, it has a `read()` and `write()` method for locking as either a reader or a writer. It comes with two guard types, one for readers and one for writers: `RwLockReadGuard` and `RwLockWriteGuard`. The former only implements `Deref` to behave like a shared reference to the protected data, while the latter also implements `DerefMut` to behave like an exclusive reference.

It is effectively the multi-threaded version of `RefCell`: dynamically tracking the number of references to ensure the borrow rules are upheld.

Both `Mutex<T>` and `RwLock<T>` require `T` to be `Send`, because they can be used to send a `T` to another thread. An `RwLock<T>` additionally requires `T` to also implement `Sync`, because it allows multiple threads to hold a shared reference (`&T`) to the protected data. (Strictly speaking, you can create a lock for a `T` that doesn't fulfill these requirements, but you wouldn't be able to share it between threads as the lock itself won't implement `Sync`.)

There are many subtle variations between reader-writer lock implementations. Most importantly, some prioritize readers and some prioritize writers. In some implementations, when there is a writer waiting, new readers are blocked and will have to wait, even when the lock is already read-locked. This is done to prevent *writer starvation*, a situation where many readers collectively keep the lock from ever unlocking, never allowing any writer to update the data.

MUTEXES IN OTHER LANGUAGES

Rust standard library's `Mutex` and `RwLock` look a bit different than those you find in other languages like C or C++.

The biggest difference is that Rust's `Mutex<T>` *contains* the data it is protecting. In C++, for example, `std::mutex` does not contain the data it protects, nor does it even know what it is protecting at all. This means that it is the responsibility of the user to remember which data is protected and by which mutex, and ensure the right mutex is locked every time *`protected`* data is accessed. This is useful to keep in mind when reading code involving mutexes in other languages, or when communicating with programmers who are not familiar with Rust. A Rust programmer might talk about "the data inside the mutex", or say things like "wrap it in a mutex", which can be confusing to those only familiar with mutexes in other languages.

If you really need a stand-alone mutex that doesn't contain anything, for example to protect some external hardware, you can use `Mutex<()>`. But even in those cases, you are probably better off defining a (possibly zero-sized) type to interface with that hardware and wrapping that in a `Mutex` instead. That way, you are still forced to lock the mutex before you can interact with the hardware.

Waiting: Parking and Condition Variables

When data is mutated by multiple threads, there are many situations where they would need to *wait* for some event; for some condition about the data to be come true. For example, if we have a mutex protecting a `Vec`, we might want to wait until it contains any elements.

While a mutex does allow threads to wait until it becomes unlocked, it does not provide functionality for waiting for any other conditions. If a mutex was all we had, we'd have to keep locking the mutex to repeatedly check if there's anything in the `Vec` yet.

Thread parking

One way to wait for a notification from another thread is called *thread parking*. A thread can *park* itself, which puts it to sleep, stopping it from consuming any CPU cycles. Another thread can then *unpark* the parked thread, waking it up from its nap.

Thread parking is available through the `std::thread::park()` function. For unparking, you call the `unpark()` method on a `Thread` object representing the thread that you want to unpark. Such an object can be obtained from the join handle returned by `spawn`, or by the thread itself through `std::thread::current()`.

Let's dive into an example that uses a mutex to share a queue between two threads. In the following example, a newly spawned thread will consume items from the queue, while the main thread will insert a new item into the queue every second. Thread parking is used to make the consuming thread wait when the queue is empty.

```
let queue = Mutex::new(VecDeque::new());

thread::scope(|s| {
    // Consuming thread
    let t = s.spawn(|| loop {
        let item = queue.lock().unwrap().pop_front();
        if let Some(item) = item {
            dbg!(item);
        } else {
            thread::park();
        }
    });

    // Producing thread
    for i in 0.. {
        queue.lock().unwrap().push_back(i);
        t.thread().unpark();
        thread::sleep(Duration::from_secs(1));
        }
    });
```

The consuming thread runs an infinite loop in which it pops items out of the queue to display them using the `dbg` macro. When the queue is empty, it stops and sleep using the `park` function. If it gets unparked, it starts over and starts popping items from the queue again, until it is empty. And so on.

The producing thread produces a new number every second by pushing it into the queue. Every time it adds an item, it uses the `unpark` method on the `Thread` object that refers to the consuming thread to unpark it. That way, the consuming thread gets woken up to process the new element.

An important observation to make here is that this program would still have been theoretically correct, although inefficient, if we remove parking. This is important, because `park()` does not guarantee that it will only return because of a matching `unpark()`. While somewhat rare, it might have *spurious wake-ups*. Our example deals with that just fine, because the consuming thread will lock the queue, see that it is empty, and directly unlock it and park itself again.

An important property of thread parking is that a call to `unpark()` *before* the thread parks itself does not get lost. The request to unpark is still recorded, and the next time the thread tries to park itself, it clears that request and directly continues without actually going to sleep. To see why that is critical for correct operation, let's go through a possible ordering of the steps executed by both threads:

1. The consuming thread, let's call that C, locks the queue.
2. C tries to pop an item from the queue, but it is empty, resulting in `None`.
3. C unlocks the queue.
4. The producing thread, which we'll call P, locks the queue.
5. P pushes a new item on to the queue.

6. P unlocks the queue again.
7. P calls `unpark()` to notify C that there's new items.
8. C calls `park()` to go to sleep, to wait for more items.

While there is only a very brief moment between releasing the queue in step 3 and parking in step 8, steps 4 through 7 could potentially happen in that brief moment before the thread parks itself. If `unpark()` would be ignored if the thread wasn't parked, the notification would be lost. The consuming thread would still be waiting, even though there is an item in the queue. Thanks to unpark requests getting saved for a future call to `park()`, we don't have to worry about this.

However, unpark requests don't stack up. Calling `unpark()` two times and then calling `park()` two times afterwards still results in the thread going to sleep. The first `park()` clears the request and returns directly, but the second one goes to sleep as usual.

This means that in our example above it's important that we only park the thread if we saw the queue is empty, rather than parking after every processed item. While it's extremely unlikely to happen in this example because of the huge (one second) sleep, it's possible for multiple `unpark()` calls to end up waking up only a single `park()` call.

Unfortunately, this all does mean that if `unpark()` is called right after `park()` returns but before the queue gets locked and emptied out, the `unpark()` call was unnecessary but still causes the next `park()` call to instantly return. This results in the (empty) queue getting locked and unlocked again an extra time. While this doesn't affect the correctness of the program, it does affect its efficiency and performance.

This mechanism works well for simple situations like in our example, but quickly breaks down when things get more complicated. For example, if we had multiple consumer threads taking items from the same queue, the producer thread would have no way of knowing which of the consumers is actually waiting and should be woken up. The producer will have to know exactly when a consumer is waiting, and what condition it is waiting for.

Condition Variables

Condition variables are a more commonly used option for waiting for something to happen to data protected by a mutex. They have two basic operations: *wait* and *notify*. Threads can wait on a condition variable, and will be woken up when someone notifies that same condition variable. Multiple threads can wait on the same condition variable, and notifications can either be sent to one waiting thread, or to all of them.

This means that we can create a condition variable for specific events or conditions we're interested in, such as the queue being non-empty, and wait on that condition. Any thread that causes that event or condition to happen then notifies the condition variable, without having to know which or how many threads are interested in that notification.

To avoid the issue of missing notifications in the brief moment between unlocking a mutex and waiting for a condition variable, condition variables provide a way to *atomically* unlock the mutex and start waiting. This means there is simply no possible moment for notifications to get lost.

The Rust standard library provides a condition variable as `std::sync::Condvar`. Its `wait` method takes a `MutexGuard` that proves we've locked the mutex. It unlocks the mutex and goes to sleep, and when woken up it re-locks the mutex and returns a new `MutexGuard`, proving that the mutex is locked again.

It has two notify functions: `notify_one` to wake up just one waiting thread (if any), and `notify_all` to wake them all up.

Let's modify the example we used for thread parking to use `Condvar` instead:

```
let queue = Mutex::new(VecDeque::new());
let not_empty = Condvar::new();
```

```
thread::scope(|s| {
    s.spawn(|| {
        loop {
            let mut q = queue.lock().unwrap();
            let item = loop {
                if let Some(item) = q.pop_front() {
                    break item;
                } else {
                    q = not_empty.wait(q).unwrap();
                }
            };
            drop(q);
            dbg!(item);
        }
    });

    for i in 0.. {
        queue.lock().unwrap().push_back(i);
        not_empty.notify_one();
        thread::sleep(Duration::from_secs(1));
    }
});
```

We had to change a few things:

- We now not only have a `Mutex` containing the queue, but also a `Condvar` to communicate the *`not empty`* condition through.
- We no longer need to know which thread to wake up, so we don’t store the return value from `spawn` anymore. Instead, we notify the consumer through the condition variable with the `notifiy_one` method.
- Unlocking, waiting, and re-locking is all done by the `wait` method. We had to restructure the control flow a bit to be able to pass the guard to the `wait` method, while still dropping it before processing an item.

Now we can spawn as many consuming threads as we like, and even spawn more later, without having to change anything. The condition variable takes care of delivering the notifications to whichever thread is interested.

If we had a more complicated system with threads that are interested in different conditions, we could define a `Condvar` for each condition. For example, one to indicate the queue is non-empty, and another one to indicate it is empty. Then each thread can wait for whichever condition is relevant to what they are doing.

Normally, a `Condvar` is only ever used together with one single `Mutex`. If two threads try to concurrently `wait` on a condition variable using two different mutexes, it might cause a panic.

A downside of a `Condvar` is that it only works when used together with a `Mutex`, but for almost most use cases that is perfectly fine, as that's exactly what's already used anyway.

Summary

- Multiple threads can run concurrently within the same program, and can be spawned at any time.
- When the main thread ends, the entire program ends.
- Data races are undefined behavior, and fully prevented (in safe code) by Rust's type system.
- Data that is `Send` can be sent to other threads, and data that is `Sync` can be shared between threads.
- Regular threads might run as long as the program does, and thus can only borrow `'static` data such as statics and leaked allocations.
- Reference counting (`Arc`) can be used to share ownership to make sure data lives as long as any thread is using it.
- Scoped threads are useful to limit the lifetime of a thread to alllow it to borrow non-`'static` data, such as local variables.
- `&T` is a *shared* reference. `&mut T` is an *exclusive* reference. Regular types do not allow mutation through a shared reference.

- Some types have interior mutability, which allows for mutation through shared references, and is always based on `UnsafeCell`.
- `Cell` and `RefCell` are the standard types for single-threaded interior mutability. Atomics, `Mutex`, and `RwLock` are their multi-threaded equivalents.
- `Cell` and atomics only allow replacing the value as a whole, while `RefCell`, `Mutex`, and `RwLock` allow you to mutate the value directly by dynamically enforcing access rules.
- Thread parking can be a convenient way to wait for some condition.
- When a condition is about data protected by a `Mutex`, using a `Condvar` is more convenient than thread parking.

Chapter 2. Introduction to Atomics

The word *atomic* comes from the Greek word *ἄτομος*, meaning *indivisible*, something that cannot be cut into smaller pieces. In computer science, it is used to describe an operation that is indivisible: it is either fully completed, or it didn't happen yet.

As mentioned in "Borrowing and Data Races", multiple threads concurrently reading and modifying the same variable normally results in undefined behavior. However, atomic operations do allow for different threads to safely read and modify the same variable. Since such an operation is indivisible, it either happens completely before or completely after another operation, avoiding undefined behavior. Later, in X REF HERE FOR CH04, we'll see how this works at the hardware level.

Atomic operations are the main building block for anything involving multiple threads. All the other concurrency primitives, such as mutexes and condition variables, are implemented using atomic operations.

In Rust, atomic operations are available through as methods on the standard atomic types that live in `std::sync::atomic`. They all have names starting with `Atomic`, such as `AtomicI32` or `AtomicUsize`. Which of them are available depends on the hardware architecture and sometimes operating system, but almost all platforms provide at least all atomic types up to the size of a pointer.

Unlike most types, they allow modification through a shared reference (e.g. `&AtomicU8`). This is possible thanks to interior mutability, as discussed in "Interior Mutability".

Each of the available atomic types has the same interface with methods for storing and loading, methods for atomic *fetch and modify* operations, and some more advanced *compare and exchange* methods. We'll discuss them in detail in rest of this chapter.

But, before we can dive into the different atomic operations, we briefly need to touch upon a concept called *memory ordering*.

Memory Ordering

Every atomic operation takes an argument of type `std::sync::atomic::Ordering`, which determines which guarantees we get about the relative ordering of operations. The simplest variant with the least guarantees is `Relaxed`. `Relaxed` still guarantees consistency on a single atomic variable, but does not promise anything about the relative order of operations between different variables.

What that means is that two threads might see operations on different variables happen in a different order. For example, if one thread writes to one variable first and then to a second variable very quickly afterwards, another thread might see that happen in opposite order.

In this chapter we'll only look at use cases where this is not a problem, and simply use `Relaxed` everywhere without going more into detail. We'll discuss all the details of memory ordering and the other available memory orderings in X REF HERE FOR CH03.

Atomic Loads and Stores

The first two atomic operations we'll look at are the most basic ones: `load` and `store`. Their function signatures are as follows:

```
impl AtomicI32 {
    pub fn load(&self, ordering: Ordering) -> i32;
    pub fn store(&self, value: i32, ordering: Ordering);
}
```

The `load` method atomically loads the value stored in the atomic variable, and the `store` method atomically stores a new value in it. Note how the `store` method takes a shared reference (`&T`) rather than an exclusive reference (`&mut T`), even though it modifies the value.

Let's take a look at some realistic use cases for these two methods.

Example: Stop Flag

The first example uses an `AtomicBool` for a *stop flag*. Such a flag is used to inform other threads to stop running.

```
fn main() {
    static STOP: AtomicBool = AtomicBool::new(false);

    // Spawn a thread to do the work.
    let background_thread = thread::spawn(|| {
        while !STOP.load(Relaxed) {
            some_work();
        }
    });

    // Use the main thread to listen for user input.
    for line in std::io::stdin().lines() {
        match line.unwrap().as_str() {
            "help" => println!("commands: help, stop"),
            "stop" => break,
            cmd => println!("unknown command: {cmd:?}"),
        }
    }

    // Inform the background thread it needs to stop.
    STOP.store(true, Relaxed);

    // Wait until the background thread finishes.
    background_thread.join().unwrap();
}
```

In this example, the background thread is repeatedly running `some_work()`, while the main thread allows the user to enter some commands to interact with the program. In this simple example, the only useful command is `stop` to make the program stop.

To make the background thread stop, the atomic `STOP` boolean is used to communicate this condition to the background thread. When the foreground thread reads the `stop` command, it sets the flag to true, which is checked by the background thread before each new iteration. The main thread waits until the background thread is finished with its current iteration using the `join` method.

This simple solution works great as long as the flag is regularly checked by the background thread. If it can get stuck in `some_work()` for a long time, this might result in an unacceptable elay between the `stop` command and the program quitting.

Example: Progress Reporting

In our next example we process 100 items one by one on a background thread, while the main thread gives the user regular updates on the progress.

```
fn main() {
    let num_done = AtomicUsize::new(0);

    thread::scope(|s| {
        // A background thread to process all 100 items.
        s.spawn(|| {
            for i in 0..100 {
                process_item(i);
                num_done.store(i + 1, Relaxed);
            }
        });
        // The main thread shows status updates, every second.
        loop {
            let n = num_done.load(Relaxed);
            if n == 100 { break; }
            println!("Working.. {n}/100 done");
            thread::sleep(Duration::from_secs(1));
        }
    });

    println!("Done!");
}
```

This time, we use a scoped thread ("Scoped Threads"), which will automatically handle the joining of the thread for us, and also allow us to borrow local variables.

Every time the background thread finishes processing an item, it stores the number of processed items in an `AtomicUsize`. Meanwhile, the main thread shows that number to the user to inform them of the progress, about once per second. Once the main thread sees all 100 items have been processed, it exits the scope, which implicitly joins the background thread, and informs the user everything is done.

Synchronization

Once the last item is processed, it might take up to one whole second for the main thread to know that, introducing an unnecessary delay at the end. To solve this, we can use thread parking ("Thread parking") to wake the main thread from its sleep whenever there is new information it might be interested in.

Here's the same example, but now using `thread::park_timeout` rather than `thread::sleep`:

```
fn main() {
    let num_done = AtomicUsize::new(0);

    let main_thread = thread::current();

    thread::scope(|s| {
        // A background thread to process all 100 items.
        s.spawn(|| {
            for i in 0..100 {
                process_item(i);
                num_done.store(i + 1, Relaxed);
                main_thread.unpark(); // Wake up the main thread.
            }
        });

        // The main thread shows status updates.
        loop {
            let n = num_done.load(Relaxed);
            if n == 100 { break; }
            println!("Working.. {n}/100 done");
            thread::park_timeout(Duration::from_secs(1));
        }
    });

    println!("Done!");
}
```

Not much has changed. We've obtained a handle to the main thread through `thread::current()`, that's now used by the background thread to unpark the main thread after every status update. The main thread now uses `park_timeout` rather than `sleep`, such that it can be interrupted.

Now, any status updates are immediately reported to the user, while still repeating the last update every second to show the program is still running.

Example: Lazy Initialization

The last example before we move on to more advanced atomic operations is about *lazy initialization*.

Imagine there is a value `x` which we are reading from a file, obtaining from the operating system, or calculating in some other way, that we expect to be constant. Maybe `x` is the version of the operating system, or the total amount of memory, or the 400th digit of tau. It doesn't really matter for this example.

Since we don't expect it to change, we can request or calculate it only the first time we need it, and remember the result. The first thread that needs it will have to calculate the value, but it can store it in an atomic `static` to make it available for all threads, including itself if it needs it again later.

Let's take a look at an example of this. To keep things simple, we'll assume `x` is never zero, such that we can use zero as a placeholder before it has been calculated.

```
fn get_x() -> u64 {
    static X: AtomicU64 = AtomicU64::new(0);
    let mut x = X.load(Relaxed);
    if x == 0 {
        x = calculate_x();
        X.store(x, Relaxed);
    }
    x
}
```

The first thread to call `get_x()` will check the static `X` and see it is still zero, calculate its value, and store the result back in the static to make it available for future use. Later, any call to `get_x()` will see the value in the static is nonzero, and return it immediately without calculating it again.

However, if a second thread calls `get_x()` while the first one is still calculating `x`, the second thread will also see a zero and also calculate x in parallel. One of the threads will end up overwriting the result of the other, depending on which one finishes first. This is called a *race*. Not a *data race*, which is undefined behaviour and impossible in Rust without using `unsafe`, but still a race with an unpredictable winner.

Since we expect `x` to be constant, it doesn't matter who wins the race, as the result will be the same regardless. Depending on how much time we expect `calculate_x()` to take, this might be a very good or very bad strategy.

If `calculate_x()` is expected to take a long time, it's better if threads wait while the first thread is still initializing `X`. You could implement this using a condition variable or thread parking ("Waiting: Parking and Condition Variables"), but that quickly gets too complicated for a small example. The Rust standard library provides exactly this functionality through `std::sync::Once` and `std::lazy::SyncOnceCell`, so there's usually no need to implement these yourself.

Fetch-and-Modify Operations

Now that we've seen a few use cases for the basic `load` and `store` operations, let's move on to more interesting operations: the *fetch and modify* operations. These operations modify the atomic variable, but also load (fetch) the original value, as a single atomic operation.

The most commonly used ones are `fetch_add` and `fetch_sub`, which perform addition and substraction respectively. Some of the other available operations are `fetch_or` and `fetch_and` for bitwise operations, and `fetch_max` and `fetch_min` which can be used to keep a running maximum or minimum.

Their function signatures are as follows:

```
impl AtomicI32 {
    pub fn fetch_add(&self, v: i32, ordering: Ordering) -> i32;
    pub fn fetch_sub(&self, v: i32, ordering: Ordering) -> i32;
    pub fn fetch_or(&self, v: i32, ordering: Ordering) -> i32;
    pub fn fetch_and(&self, v: i32, ordering: Ordering) -> i32;
    pub fn fetch_nand(&self, v: i32, ordering: Ordering) -> i32;
    pub fn fetch_xor(&self, v: i32, ordering: Ordering) -> i32;
    pub fn fetch_max(&self, v: i32, ordering: Ordering) -> i32;
    pub fn fetch_min(&self, v: i32, ordering: Ordering) -> i32;
    pub fn swap(&self, v: i32, ordering: Ordering) -> i32; //
"fetch_store"
}
```

The one outlier is the operation that simply stores a new value, regardless of the old value. Instead of `fetch_store`, it has been called `swap`.

Here's a quick demonstration showing how `fetch_add` returns the value before the operation:

```
let a = AtomicU32::new(100);
let b = a.fetch_add(23, Relaxed);
let c = a.load(Relaxed);

assert_eq!(b, 100);
assert_eq!(c, 123);
```

The `fetch_add` operation incremented `a` from 100 to 123, but returned to us the old value of 100. Any next operation will see the value of 123.

The return value from these operations is not always relevant. If you only need the operation to be applied to the atomic value, but are not interested in the value itself, it's perfectly fine to simply ignore the return value.

An important thing to keep in mind is that `fetch_add` and `fetch_sub` implement *wrapping* behaviour for overflows. Incrementing a value past the maximum representable value will wrap around and result in the minimum representable value. This is different than the behavior of the plus and minus operators on regular integers, which will panic in debug mode on overflow.

In "Compare-and-Exchange Operations", we'll see how to do atomic addition with overflow checking.

But first, let's see some real world use cases of these methods.

Example: Progress Reporting from Multiple Threads

In "Example: Progress Reporting", we used an `AtomicUsize` to report the progress of a background thread. If we had split up the work over, for example, four threads each processing 25 items, we'd need to know the progress from all four threads.

We could use a separate `AtomicUsize` for each thread and load them all in the main thread and sum them up, but an easier solution is to use a single `AtomicUsize` to track the total number of processed items over all threads.

To make that work, we can no longer use the `store` method as that would overwrite the progress from other threads. Instead, we can use an atomic add operation to increment the counter after every processed item.

Let’s update the example from “Example: Progress Reporting” to split the work over four threads:

```
fn main() {
    let num_done = &AtomicUsize::new(0);

    thread::scope(|s| {
        // Four background threads to process all 100 items, 25
each.
        for t in 0..4 {
            s.spawn(move || {
                for i in 0..25 {
                    process_item(t * 25 + i);
                    num_done.fetch_add(1, Relaxed);
                }
            });
        }

        // The main thread shows status updates, every second.
        loop {
            let n = num_done.load(Relaxed);
            if n == 100 { break; }
            println!("Working.. {n}/100 done");
            thread::sleep(Duration::from_secs(1));
        }
    });

    println!("Done!");
}
```

A few things have changed. Most importantly, we now spawn four background threads rather than one, and use `fetch_add` instead of `store` to modify the `num_done` atomic variable.

More subtly, we now use a `move` closure for the background threads, and `num_done` is now a reference. This is not related to our use of `fetch_add`, but rather to how we spawn four threads in a loop. This closure captures `t` to know which of the four threads it is, and thus whether to start at item 0, 25, 50, or 75. Without the `move` keyword, the closure would try to capture `t` by reference. That isn’t allowed, as it only exists briefly during the loop.

As a `move` closure, it moves (or copies) its captures rather than borrowing them, giving it a copy of `t`. Because it also captures `num_done`, we've changed that variable to be a reference, since we do still want to borrow that same `AtomicUsize`. Note that the atomic types do not implement the `Copy` trait, so we'd have gotten an error if we tried to move one into more than one thread.

Closure capture subtilities aside, the change to use `fetch_add` here is very simple. We don't know in which order the threads will increment `num_done`, but as the addition is atomic, we don't have to worry about anything and can be sure it will be exactly 100 when all threads are done.

Example: Statistics

Continuing with this concept of reporting what other threads are doing through atomics, let's extend our example to also collect and report some statistics on the time it takes to process an item.

Next to `num_done`, we're adding two atomic variables, `total_time` and `max_time`, to keep track of the amount of time spent processing items. We'll use these to report the average and peak processing times.

```
fn main() {
    let num_done = &AtomicU32::new(0);
    let total_time = &AtomicU64::new(0);
    let max_time = &AtomicU64::new(0);

    thread::scope(|s| {
        // Four background threads to process all 100 items, 25
each.
        for t in 0..4 {
            s.spawn(move || {
                for i in 0..25 {
                    let start = Instant::now();
                    process_item(t * 25 + i);
                    let time_taken = start.elapsed().as_micros()
as u64;
                    num_done.fetch_add(1, Relaxed);
                    total_time.fetch_add(time_taken, Relaxed);
                    max_time.fetch_max(time_taken, Relaxed);
                }
            });
        }

        // The main thread shows status updates, every second.
        loop {
            let total_time =
Duration::from_micros(total_time.load(Relaxed));
            let max_time =
Duration::from_micros(max_time.load(Relaxed));
            let n = num_done.load(Relaxed);
            if n == 100 { break; }
            if n == 0 {
                println!("Working.. nothing done yet.");
            } else {
                println!(
                    "Working.. {n}/1000 done, {:?} average, {:?}
peak",
                    total_time / n,
                    max_time,
                );
            }
            thread::sleep(Duration::from_secs(1));
        }
    });

    println!("Done!");
}
```

The background threads now use `Instant::now()` and `Instant::elapsed()` to measure the time they spend in `process_item()`, An atomic add operation is used to add the amount of microseconds to `total_time`, and an atomic max operation is used to keep track of the highest measurement in `max_time`.

The main thread divides the total time by the number of processed items to obtain the average processing time, and reports that together with the peak time from `max_time`.

Since the three atomic variables are updated separately, it is possible for the main thread to load the values after a thread has incremented `num_done`, but before it has updated `total_time`, resulting in an underestimate of the average. More subtly, because the `Relaxed` memory ordering gives no guarantees about the relative order of operations as seen from another thread, it might even briefly see a new updated value of `total_time` while still seeing an old value of `num_done`, resulting in an overestimate of the average.

Neither of this is a big issue in our example. The worst that can happen is briefly reporting an inaccurate average to the user.

If we want to avoid this, we can put the three statistics inside a `Mutex`. Then we'd briefly lock the mutex while updating the three numbers, which no longer have to be atomic by themselves. This effectively turns the three updates into a single atomic operation, at the cost of locking and unlocking a mutex, and potentially temporarily blocking threads.

Example: ID Allocation

Let's move on to a use case where we actually need the return value from `fetch_add`.

Suppose we need some function, `allocate_new_id()`, that gives a new unique number every time it is called. We might use these numbers to identify tasks or other things in our program; things that need to be uniquely identified by something small that can be easily stored and passed around between threads, such as an integer.

Implementing this function turns out to be trivial using `fetch_add`:

```
fn allocate_new_id() -> u32 {
    static NEXT_ID: AtomicU32 = AtomicU32::new(0);
    NEXT_ID.fetch_add(1, Relaxed)
}
```

We simply keep track of the *next* number to give out, and increment it every time we load it. The first caller will get a `0`, the second a `1`, and so on.

The only problem here is the wrapping behavior on overflow. The 4294967296th call will overflow the 32-bit integer, such that the next call will return 0 again.

If this is a problem depends on the use case: how likely is it to be called this often, and what's the worst that can happen if the numbers are not unique? While this might seeem like a huge number, modern computers can easily execute our function that many times within seconds. If memory safety is dependent on these numbers being unique, our implementation above is not acceptable.

To solve this, we can attempt to make the function panic if it is called too many times, like this:

```
// This version is problematic.
fn allocate_new_id() -> u32 {
    static NEXT_ID: AtomicU32 = AtomicU32::new(0);
    let id = NEXT_ID.fetch_add(1, Relaxed);
    assert!(id < 1000, "too many IDs!");
    id
}
```

Now, the `assert` statement will panic after a thousand calls. However, this happens *after* the atomic add operation already happened, meaning that `NEXT_ID` has already been incremented to 1001 when we panic. If another thread then calls the function, it'll increment it to 1002 before panicking, and so on. Although it might take significantly longer, we'll run into the same problem after 4294966296 panics when `NEXT_ID` will overflow to zero again.

There are three common solutions to this problem. The first one is to not `panic` but instead completely `abort` the process on overflow. The `std::process::abort` function will abort the entire process, ruling out the possibility of anything continuing to call our function. While aborting the process might take a brief moment in which the function can still be called by other threads, the chance of that happening billions of times before the program is truly aborted is negligible.

This is in fact how the overflow check in `Arc::clone()` in the standard library is implemented, in case you somehow manage to clone it `isize::MAX` times. That'd take hundreds of years on a 64-bit computer, but is achieveable in seconds if `isize` is only 32 bits.

A second way to deal with the overflow is to use `fetch_sub` to decrement the counter again before panicking, like this:

```
fn allocate_new_id() -> u32 {
    static NEXT_ID: AtomicU32 = AtomicU32::new(0);
    let id = NEXT_ID.fetch_add(1, Relaxed);
    if id >= 1000 {
        NEXT_ID.fetch_sub(1, Relaxed);
        panic!("too many IDs!");
    }
    id
}
```

It's still possible for the counter to very briefly be incremented beyond 1000 when multiple threads execute this function at the same time, it is limited by the amount of active threads. It's reasonable to assume there will never be billions of active threads at once, especially not all simultaniously executing the same function in the brief moment between `fetch_add` and `fetch_sub`.

This is how overflows are handled for the number of running threads in the standard libary's `thread::scope` implementation.

The third way of handling overflows is arguably the only truly correct one, as it prevents the addition from happening at all if it would overflow. However, we cannot implement that with the atomic operations we've seen so far. For this, we'll need compare-and-exchange operations.

Compare-and-Exchange Operations

The most advanced and flexible atomic operation is the *compare and exchange* operation. This operation checks if the atomic value is equal to a given value, and only if that is the case will it replace it with a new value, all atomically as a single operation. It will return the previous value and tell us whether it replaced it or not.

Its signature is a bit more complicated than the ones we've seen so far:

```
impl AtomicI32 {
    pub fn compare_exchange(
        &self,
        expected: i32,
        new: i32,
        store_order: Ordering,
        load_order: Ordering
    ) -> Result<i32, i32>;
}
```

Ignoring memory ordering for a moment, it is basically identical to the following implementation, except it all happens as a single undividable atomic operation:

```
pub fn compare_exchange(&self, expected: i32, new: i32) ->
Result<i32, i32> {
    // In reality, the load, comparison and store,
    // all happen as a single atomic operation.
    let v = self.load();
    if v == expected {
        // Value is as expected. Replace it and report success.
        self.store(new);
        Ok(v)
    } else {
        // The value was not as expected. Leave it untouched and
report failure.
        Err(v)
    }
}
```

What this allows us to do, is to load value, perform any calculation we like, and then update the atomic variable to the updated value *only if it hasn't changed* in the meantime. If we put this in a loop to retry if it did change, we could use this to implement all the other atomic operations, making this the most generic one.

To demonstrate, let's increment an `AtomicU32` by one without using `fetch_add`, just to see how `compare_exchange` is used in practice:

```
fn increment(a: &AtomicU32) {
    let mut current = a.load(Relaxed); ❶
    loop {
        let new = current + 1; ❷
        match a.compare_exchange(current, new, Relaxed, Relaxed)
{ ❸
            Ok(_) => return, ❹
            Err(v) => current = v, ❺
        }
    }
}
```

❶ First, we load the current value of a.

❷ We calculate the new value we want to store in a, not taking into account potential concurrent modifications of a by other threads.

❸ We use compare_exchange to update the value of a, but *only* if its value is still the same value we loaded before.

❹ If a was indeed still the same as before, it is now replaced by our new value and we are done.

❺ If a was not the same as before, another thread must've changed it in the brief moment since we loaded it. The compare_exchange operation gives us the changed value that a had, and we'll try again using that value instead. The brief moment between loading and updating is so short, that it's unlikely for this to loop more than a few iterations.

Next to compare_exchange, there is also a very similar compare_exchange_weak method. The difference is that the weak version might sometimes leave the value untouched and still return an Err, even though the atomic value matched the expected value. On some platforms, this method can be implemented more efficiently and sould be preferred in cases where the consequence of a spurious compare and exchange failure are insignificant, such as in our increment function above. In X REF HERE FOR CH04, we'll dive into the low level details to find out why the weak version can be more efficient.

Example: ID Allocation Without Overflow

Now, back to our overflow problem in in allocate_new_id() from "Example: ID Allocation".

To stop incrementing `NEXT_ID` beyond a certain limit to prevent overflows, we can use `compare_exchange` to implement atomic addition with an upper bound. Using that idea, let's make a version of `allocate_new_id` that always handles overflow correctly, even in the practically impossible situations:

```
fn allocate_new_id() -> u32 {
    static NEXT_ID: AtomicU32 = AtomicU32::new(0);
    let mut id = a.load(Relaxed);
    loop {
        assert!(id < 1000, "too many IDs!");
        match a.compare_exchange_weak(id, id + 1, Relaxed,
Relaxed) {
            Ok(_) => return id,
            Err(v) => id = v,
        }
    }
}
```

Now we check and panic **before** modifying `NEXT_ID`, guaranteeing it will never be incremented beyond 1000, making overflow impossible. We can now raise the upper limit from 1000 to `u32::MAX` if we want, without having to worry about edge cases in which it might get incremented beyond the limit.

Example: Racy Lazy Initialization

In "Example: Lazy Initialization", we looked at an example of lazy initialization of a constant value. We made a function that lazily initializes a value on the first call, but re-uses it on later calls. When multiple threads run the function concurrently during the first call, more than one thread might execute the initialization, and they will overwrite eachothers result in an unpredictable order.

This is fine for values that we expect to be constant, or when we don't care about changing values. However, there are also use cases where such a value gets initialized to a different value each time, even though we need every invocation of the function within a single run of the program to return the same value.

For example, imagine a function `get_key()` that returns a randomly generated key that's only generated once per run of the program. It might be an encryption key used for communication with the program, that needs to be unique every time the program is run, but stay constant within a process.

This means we cannot simply use a `store` operation after generating a key, since that might overwrite a key generated by another thread just moments ago, resulting in two threads using different keys. Instead, we can use `compare_exchange` to make sure we only store the key if we're still the first, and otherwise throw our key away and use the stored key instead.

Here's an implementation of this idea:

```
fn get_key() -> u64 {
    static KEY: AtomicU64 = AtomicU64::new(0);
    let key = KEY.load(Relaxed);
    if key == 0 {
        let new_key = generate_random_key(); ❶
        match KEY.compare_exchange(0, new_key, Relaxed, Relaxed)
{ ❷
            Ok(_) => new_key, ❸
            Err(k) => k, ❹
        }
    } else {
        key
    }
}
```

❶ We only generate a new key if `KEY` was not yet initialized.

❷ We replace `KEY` by our newly generateed key, but only if it is *still* zero.

❸ If we swapped the zero for our new key, we return our newly generated key. New invocations of `get_key()` will now return the same new key that's now stored in `KEY`.

❹ If we lost the race to another thread initializing `KEY` before we could, we forget our newly generated key and use the key from `KEY` instead.

This is a good example of a situation where `compare_exchange_weak` is not appopriate, and we want to use the strong version instead. We don't run our compare and exchange operation in a loop, and we don't want to return zero if the operation spuriously failed. Even if we did put this in a loop, generating a new key probably takes more than a few simple operations, so risking a spurious failure from `compare_exchange_weak` is simply not worth the small performance benefit it might bring us.

As mentioned in "Example: Lazy Initialization", if `generate_random_key()` takes a lot of time, it might make more sense to block threads during initialization, to avoid potentially spending time generating keys that will not be used. The Rust standard library provides such functionality through `std::sync::Once` and `std::lazy::SyncOnceCell`.

Summary

- Atomic operations are *undividable*; they either fully completed, or they haven't happened yet.
- Atomic operations in Rust are done through the atomic types in `std::sync::atomic`, such as `AtomicI32`.
- Not all atomic types are available on all platforms.
- The relative ordering of atomic operations is tricky when multiple variables are involved. More in X REF HERE FOR CH03.
- Simple loads and stores are nice for very basic inter-thread communication, like stop flags and status reporting.
- Lazy initialization can be done as a *race*, without causing a *data race*.
- Fetch-and-modify operations allow for a small set of basic atomic modifications that are especially useful when multiple threads are modifying the same atomic variable.
- Atomic addition and subtraction silently wrap around on overflow.

- Compare-and-exchange operations are the most flexible and general, and a building block for making any other atomic operation.
- A *weak* compare-and-exchange operation can be slightly more efficient.