Chapter 2. 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" in Chapter 1, 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 Chapter 7, 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 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 ones 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" in Chapter 1.

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 the 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*:

Every atomic operation takes an argument of type `std::sync::atomic::Ordering`, which determines what guarantees we get about the relative ordering of operations. The simplest variant with the fewest 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 this 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 the 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 Chapter 3.

Atomic Load and Store Operations

The first two atomic operations we'll look at are the most basic ones: `load` and `store`. Their function signatures are as follows, using `AtomicI32` as an example:

```
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.

```
use std::sync::atomic::AtomicBool;
use std::sync::atomic::Ordering::Relaxed;

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 gets stuck in `some_work()` for a long time, that can result in an unacceptable delay 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:

```
use std::sync::atomic::AtomicUsize;

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); // Assuming this takes some time.
                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" in Chapter 1), 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 that all 100 items have been processed, it exits the scope, which implicitly joins the background thread, and informs the user that everything is done.

Synchronization

Once the last item is processed, it might take up to one whole second for the main thread to know, introducing an unnecessary delay at the end. To solve this, we can use thread parking ("Thread Parking" in Chapter 1) 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); // Assuming this takes some time.
                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()`, which is 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 that 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 during a run of the program. 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, so that we can use zero as a placeholder before it has been calculated.

```
use std::sync::atomic::AtomicU64;

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 that 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 behavior 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`, to avoid unnecessarily wasting processor time. You could implement this using a condition variable or thread parking ("Waiting: Parking and Condition Variables" in Chapter 1), but that quickly gets too complicated for a small example. The Rust standard library provides exactly this functionality through `std::sync::Once` and `std::sync::OnceLock`, 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 subtraction, 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, using `AtomicI32` as an example:

```
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:

```
use std::sync::atomic::AtomicI32;

let a = AtomicI32::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* behavior 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 the work over, for example, four threads with 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); // Assuming this takes some time.
                    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 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 had tried to move one into more than one thread.

Closure capture subtleties 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 = &AtomicUsize::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); // Assuming this takes some time.
                    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}/100 done, {:?} average, {:?} peak",
                    total_time / n as u32,
                    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 number 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, which it then reports 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 that an inaccurate average is briefly reported 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`:

```
use std::sync::atomic::AtomicU32;

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 4,294,967,296th call will overflow the 32-bit integer, such that the next call will return 0 again.

Whether 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 4,294,966,296 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, but it is limited by the number of active threads. It's reasonable to assume there will never be billions of active threads at once, especially not all simultaneously 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 library'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, which we'll explore next.

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 does 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. Using `AtomicI32` as an example, it looks like this:

```
impl AtomicI32 {
    pub fn compare_exchange(
        &self,
        expected: i32,
        new: i32,
        success_order: Ordering,
        failure_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, indivisible atomic operation:

```
impl AtomicI32 {
    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)
        }
    }
}
```

Using this, we can load a value from an atomic variable, perform any calculation we like, and then only store the newly calculated value if the atomic variable didn't change 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 general 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); ①
    loop {
        let new = current + 1; ②
        match a.compare_exchange(current, new, Relaxed, Relaxed) { ③
            Ok(_) => return, ④
            Err(v) => current = v, ⑤
        }
    }
}
```

① First, we load the current value of `a`.

② We calculate the new value we want to store in `a`, not taking into account potential concurrent modifications of `a` by other threads.

③ We use `compare_exchange` to update the value of `a`, but *only* if its value is still the same value we loaded before.

④ If `a` was indeed still the same as before, it is now replaced by our new value and we are done.

⑤ 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.

🔥 *If the atomic variable changes from some value `A` to `B` and then back to `A` after the `load` operation, but before the `compare_exchange` operation, it would still succeed, even though the atomic variable was changed (and changed back) in the meantime. In many cases, as with our `increment` example, this is not a problem. However, there are certain algorithms, often involving atomic pointers, for which this can be a problem. This is known as the ABA problem.*

Next to `compare_exchange`, there is a similar method named `compare_exchange_weak`. The difference is that the weak version may still sometimes leave the value untouched and return an `Err`, even though the atomic value matched the expected value. On some platforms, this method can be implemented more efficiently and should be preferred in cases where the consequence of a spurious compare-and-exchange failure are insignificant, such as in our `increment` function above. In Chapter 7, 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 `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 practically impossible situations:

```
fn allocate_new_id() -> u32 {
    static NEXT_ID: AtomicU32 = AtomicU32::new(0);
    let mut id = NEXT_ID.load(Relaxed);
    loop {
        assert!(id < 1000, "too many IDs!");
        match NEXT_ID.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.

Fetch-Update

The atomic types have a convenience method called `fetch_update` for the compare-and-exchange loop pattern. It's equivalent to a `load` operation followed by a loop that repeats a calculation and `compare_exchange_weak`, just like what we did above.

Using it, we could implement our `allocate_new_id` function with a one-liner:

```
    NEXT_ID.fetch_update(Relaxed, Relaxed,
        |n| n.checked_add(1)).expect("too many IDs!")
```

Check out the method's documentation for details.

We'll not use the `fetch_update` method in this book, so we can focus on the individual atomic operations.

Example: Lazy One-Time 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 reuses 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 each others' 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, which needs to be unique every time the program is run, but stays 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 no other thread has already done so, 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(); (1)
        match KEY.compare_exchange(0, new_key, Relaxed, Relaxed) { (2)
            Ok(_) => new_key, (3)
            Err(k) => k, (4)
        }
    } else {
        key
    }
}
```

(1) We only generate a new key if `KEY` was not yet initialized.

(2) We replace `KEY` with our newly generated key, but only if it is *still* zero.

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

(4) If we lost the race to another thread that initialized `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` is more appropriate than its weak variant. We don't run our compare-and-exchange operation in a loop, and we don't want to return zero if the operation spuriously fails.

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::sync::OnceLock`.

~

Summary

- Atomic operations are indivisible; they have 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 Chapter 3.
- 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.

Next: Chapter 3. Memory Ordering