

Explaining Atomics in Rust

:

Understanding atomics and the memory ordering options when dealing with them can help us better understand multithreaded programming and why Rust helps us write safe and performant multithreaded code.

Trying to understand atomics by just reading random articles and the documentation in Rust (or C++ for that matter) feels like trying to learn physics by reverse engineering $E=MC^2$.

I'll give it my best try to explain this for myself and for you in this article. If I succeed, the ratio should be WTF?/AHA! < 1 . Let me know in the [issue tracker of the repository for this article](#) how we did.

Multiprocessor programming

When writing code for multiple CPUs, there are several subtle things we need to consider. You see, both compilers and CPU's reorder the code we write if they think it will lead to faster execution. In single threaded programs, this is not something we need to consider, but once we start writing multithreaded programs, the compiler reordering can get us into trouble.

However, while the compiler ordering is possible to check by looking at the disassembled code, things get much more difficult on systems with multiple CPUs.

When threads are run on different CPUs, the internal reordering of instructions on the CPU can lead to some very hard to debug problems since we mostly observe the side effects of CPU reordering, speculative execution, pipelining and caching.

I don't think the CPU knows in advance exactly how it's going to run your code either.

- ✓ The problem atomics solve are related to memory loads and stores. Any reordering of instructions which does not operate on shared memory has no impact that we're concerned about here.

I'll be using one main reference here unless I state otherwise: [Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A](#). So when I refer to Chapter x in the Intel Developer manual, I refer to this document.

Let's start at the bottom and work our way up to a better understanding.

Strong vs Weak Memory Ordering

So to start off, we need to get some concepts right. CPUs give different guarantees when it comes to how they treat memory. We can classify them from Weak to Strong. However, it's not a precise specification so there are models which are somewhere in between.

To abstract over these differences, Rust has the concept of an [abstract machine](#). It borrows this

model from C++. This abstract machine needs to be an abstraction which makes it possible to program against weak and strong CPUs (and everything in between). You see, the [C++ abstract machine](#) specifies many ways to access memory. It kind of has to if we're supposed to use the same semantics for weak and strong processors in the same language.

A CPU with a strong memory model gives some important guarantees that make much of the semantics we use in the abstract machine no-ops. It does nothing fancy at all, and is only a hint to the compiler to not change the order of memory operations from what we as programmers wrote.

On a weak system however, it might need to set up memory fences or use special instructions to prevent synchronization issues. The best way to learn this abstract machine using experiments would be using a CPU with a weak ordering. However, since most programmers program on a CPU with a strong model, we'll just have to point out the differences so we understand why the semantics are the way they are.

Most current desktop CPUs from AMD and Intel uses strong ordering. That means that the CPU itself gives some guarantees about not reordering certain operations. Some examples of this guarantee can be found in [Intels Developer Manual](#), chapter 8.2.2:

- Reads are not reordered with other reads.
- Writes are not reordered with older reads.
- Writes to memory are not reordered with other writes (with some exceptions)

Critically, it also includes one non-guarantee:

- Reads may be reordered with older writes to different locations but not with older writes to the same location.

This last one will be important to understand `SeqCst` memory ordering later on, just make a note of it for now.

Now, in the following chapters, I'll try to be clear on these differences. I'll still use the weak model of the abstract machine as the basis for the explanations however...

❗ ...I'll point out the differences on a strongly ordered CPU using these boxes.

CPU Caches

Normally, a CPU has three levels of caching: L1, L2, and L3. While L2, and L3 are shared between cores, L1 is a per core cache. Our challenges start here.

The L1 cache uses a sort of [MESI caching protocol](#). While the name might sound mysterious, it's actually pretty simple. It is an acronym for the different states items in the cache can find themselves in:

These states apply to each cache line in the L1 cache:

- (M) Modified - modified (dirty). Need to write back data to main memory.
- (E) Exclusive - only exists in this cache. Doesn't need to be synced (clean).
- (S) Shared - might exist in other caches. Is current with main memory (clean).
- (I) Invalid - cache line is invalid. Another cache has modified it.

❗ See chapter 11.4 in [Intel's Developer Manual](#) for more information about the cache states.

Ok, so we can model this for ourselves by thinking that every cache line in the CPU has an enum with four states attached to them.

❗ Does this sound familiar?

In Rust we have two kind of references & shared references, and `&mut` exclusive references.

It will help you a lot if you stop thinking of them as `mutable` and `immutable` references, since this is not true all the time. `Atomics` and types which allow for interior mutability do indeed break this mental model. Think of them as `exclusive` and `shared` instead.

This does indeed map very well to the `E` and `S`, two of the possible states data can have in the L1 Cache. Modelling this in the language can (and does) provide the possibility of optimizations which languages without these semantics can't do.

In Rust, only memory which is `Exclusive` can be modified by default.

This means, as long as we don't break the rule and mutate `Shared` references, all Rust programs can assume that the L1 cache on the core they're running on is up to date and does not need any synchronization.

Of course, many programs needs to share memory between cores to work, but doing so explicitly and with care can lead to better code for running on multiple processors.

Inter-processor Communication

So, if we really do need to access and change memory which is `Shared` how do the other cores know that their L1 cache is invalid?

Well, a cache line is invalidated if the data exists in the L1 cache on a different core (remember, it's in the `Shared` state) and is modified there. To actually inform the other cores that their cached data is invalid, there must be some way for the cores to communicate with each other, right?

Yes there is, however, it's pretty hard to find documentation about the exact details. Each core has what we can think of as a *mailbox*.

This mailbox can buffer a certain number of messages. Each message is buffered here to avoid interrupting the CPU all the time and force it to handle every message sent from other cores immediately.

Now, at some point the CPU checks this mailbox and updates its cache accordingly.

Let's take an example of a cache line which is marked as `Shared`.

If a CPU modifies this cache line, it is invalidated in the other caches. The core that modified the data sends a message to rest of the CPU's. When the other cores check their *mailbox*, they see that this cache line is now invalid and the state is updated accordingly in each cache on the other cores.

The L1 cache on each core then fetches the correct value from main memory (or L2/L3 cache) and sets its state to `Shared` again.

❗ On a strongly ordered CPU this model works in a slightly different way. If a core modifies a `Shared` cache line, it forces the other cores to invalidate their corresponding cache line before the value is actually modified. such CPUs often have a [cache coherency mechanism](#) which changes the state of the caches on each core.

Memory ordering

Now that we have some idea of how the CPUs are designed to coordinate between them, we can talk about different memory orderings and what they mean.

In Rust, the memory ordering is represented by the `std::sync::atomic::Ordering` enum, which has 5 possible values.

A mental model

While in practice this is pretty hard to do in Rust due to its type system (we have no way to get a pointer to an `Atomic` for example), I find it useful to imagine an observer-core. This observer core is interested in the same memory as we perform atomic operations on for some reason, so we'll divide each model in two: how it looks on the core it's running and how it looks from an observer-core.

✅ Remember that Rust inherits its memory model for atomics from C++ 20, and C copies its model from C++ as well. In these languages, the type system doesn't constrain you in the same way as it does in safe Rust, so accessing an `Atomic`, using a pointer is easier.

Relaxed

On the current CPU:

Relaxed memory ordering on atomics will prevent the compiler from reordering these instructions themselves, but, on weakly ordered CPUs, it might reorder all other memory accesses. It's OK if

you only increment a counter, but it might get you into trouble if you use a flag to implement a spin-lock for example, since you can't trust that "normal" memory access before and after the flag is set is not reordered.

On the observer CPU:

Both the compiler and the CPU are free to reorder any other memory access except from switching two `Relaxed` load/stores with each other. The observer core might observe operations in a different order than the "program order" (as we wrote them). They will however, always see `Relaxed` operation-A before `Relaxed` operation-B if we wrote them in that order.

`Relaxed` is therefore the weakest of the possible memory orderings. It implies that the operation does not do any specific synchronization with the other CPUs.

❗ On a strongly ordered CPU all memory operations are said to have `Acquire/Release` semantics by default. Therefore, this only serves as a hint to the compiler that these operations can't be reordered amongst themselves. The reason for using these operations on a strongly ordered system is that they allow for the compiler to reorder all other memory accesses as it sees fit.

If you wonder why you seem to be able to use `Relaxed` and get the same result as you do with `Acquire/Release` this is the reason. However, it's important to try to understand the "abstract machine" model and not only rely on experience you get by running experiments on a strongly ordered CPU. Your code might break on a different CPU.

Take a [look at this article](#) where they run the same code on both a strongly ordered CPU and a weakly ordered CPU to see the differences.

Acquire

On the current CPU:

Any memory operation written after the `Acquire` access stays after it. It's meant to be paired with a `Release` memory ordering flag, forming a sort of a "memory sandwich". All memory access between the load and store will be synchronized with the other CPUs.

On weakly ordered systems, this might result in using special CPU instructions before the `Acquire` operation, which forces the current core to process all messages in its mailbox (many CPUs have both serializing and memory ordering instructions). A *memory fence* will most likely also be implemented to prevent the CPU from reordering memory access before the `Acquire` load. The `Acquire` operation will therefore be synchronized with modifications on memory done on the other CPUs.

Memory fences

Since this is the first time we encounter this term, let's not leave it alone assuming everybody knows what it is. A Memory fence is a **hardware** concept. The memory fence prevents the CPU from reordering instructions by forcing it to finish loads/stores to and from memory before the fence, thereby making sure no such operation *before* the fence is reordered with any such operation *after* it.

To be able to distinguish between instructions only regarding reads, writes, or both, they're called different names. A fence preventing both from being reordered is called a full fence.

Let's take a quick look at some documentation from [Intel's Developer Manual](#) for one such full fence. (it's abbreviated, and the emphasis is mine)

Program synchronization can also be carried out with serializing instructions (see Section 8.3). These instructions are typically used at critical procedure or task boundaries to force completion of all previous instructions before a jump to a new section of code or a context switch occurs. Like the I/O and locking instructions, the processor **waits until all previous instructions have been completed and all buffered writes have been drained to memory** before executing the serializing instruction. The SFENCE, LFENCE, and MFENCE instructions provide a performance-efficient way of **ensuring load and store memory ordering** between routines that produce weakly-ordered results and routines that consume that data.

MFENCE — Serializes all store and load operations that occurred prior to the MFENCE instruction in the program instruction stream.

MFENCE is such an instruction. You'll find documentation for it throughout chapter 8.2 and 8.3 of [Intel's Developer Manual](#). You won't see these instructions too often on strongly ordered processors since they're rarely needed, but on weakly ordered systems they'll be critical to be able to implement the Acquire/Release model used in the abstract machine.


On the observer CPU:

Since Acquire is a load operation, it doesn't modify memory, there is nothing to observe.

However, and there is one caveat, if the observer core itself does an Acquire load, it will be able to see all memory operations happening from the Acquire load, and to the Release store (including the store). This means there must be some global synchronization going on. Let's discuss this a bit more in when we talk about Release.

Acquire is often used to write locks, where some operations need to stay after the successful acquisition of the lock. For this exact reason, Acquire only makes sense in load operations.

Most atomic methods in Rust that involve stores will panic if you pass in Acquire as the memory ordering of a store operation.

 On strongly ordered CPUs, this will be a no-op and will therefore not incur any cost in terms of performance. It will however prevent the compiler from reordering any memory operations we write *after* the Acquire operation to happen *before* the Acquire operation.

Release

On the current CPU:

In contrast to `Acquire`, any memory operation written before the `Release` memory ordering flag stays before it. It's meant to be paired with an `Acquire` memory ordering flag.

On weakly ordered CPUs the compiler might insert a `memory fence` to ensure that the CPU doesn't reorder the memory operations before the `Release` operation so that they happen after it. There is also a guarantee that all other cores which do an `Acquire` must see all memory operations after the `Acquire` and before this `Release`.

So not only do the operations need to be correctly ordered locally, there is also a guarantee that the changes must be visible to an observing core at this point.

That means some global synchronization must happen either before each `Acquire` access, or after a `Release` store. This basically leaves us with two choices:

1. An `Acquire` load must ensure that it processes all messages and if any other core has invalidated any memory we load, it must fetch the correct value.
2. A `Release` store must be atomic and invalidate all other caches holding that value before it modifies it.

Only doing one of these is enough though. This is partially what makes it weaker than `SeqCst` but also more performant.

On the observer CPU:

An observing CPU might not see these changes in any specific order, unless it itself uses an `Acquire` load of the memory. If it does, it will see all memory which has been modified between the `Acquire` and the `Release`, including the `Release` store itself.

`Release` is often used together with `Acquire` to write locks. For a lock to function, some operations need to stay after the successful acquisition of the lock, but before the lock is released. **For this reason, and opposite of the `Acquire` ordering, most load methods in Rust will panic if you pass in a `Release` ordering.**

❗ On a strongly ordered CPU, all instances of a `Shared` value are invalidated in all L1 caches where the value is present before it is modified. That means that the `Acquire` load will already have an updated view of all relevant memory, and a `Release` store will instantly invalidate any cache lines which contain the data on other cores.

That's why these semantics have no performance cost on such a system.

AcqRel

This is intended to be used on operations which both load and store a value.

`AtomicBool::compare_and_swap` is one such operation. Since this operation both loads and stores a value, this could matter on weakly ordered systems in contrast to `Relaxed` operations.

We can think of this operation more or less like a fence. Memory operations that are written before it will not be reordered across this boundary and memory operations that are written after it will not be reordered before it.

! Read the `Acquire` and `Release` paragraphs, the same applies here.

SeqCst

! In this part of this article, I'll talk about this using a strongly ordered CPU as the basis for discussion. You'll not see these "strongly ordered" sections here.

`SeqCst` stands for Sequential Consistency and gives the same guarantee that `Acquire/Release` does but also promises to establish a single total modification order.

`SeqCst` has been [critiqued](#) for being promoted as the recommended ordering to use and I've seen several [objections](#) about using it since it seems to be hard to prove that you have a **good** reason for using it. It has also been criticized for being [slightly broken](#).

This figure should explain where `SeqCst` might fail in upholding it's guarantees:

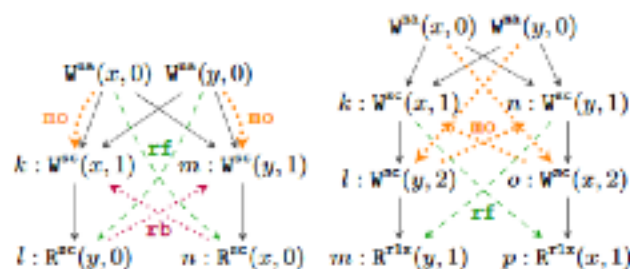


Figure 2. Inconsistent C11 executions of SB and 2+2W.

<https://plv.mpi-sws.org/scfix/paper.pdf>

Got it? Good! We'll focus on the practical sides of `SeqCst` and not the theoretical foundations of it from now on.

Just know that there is a bit of discussion around it, and that most likely `Acquire/Release` will cover most of your challenges, at least on a strongly ordered CPU.

Let's consider `SeqCst` in contrast to an `Acquire/Release` operation.

I'll use this Godbolt example to explain:



Compiler Explorer - Rust (rustc 1.40.0)
CompileExplore

The code looks like this:


```

use std::sync::atomic::{AtomicBool, Ordering};
static X: AtomicBool = AtomicBool::new(true);
static Y: AtomicBool = AtomicBool::new(true);

pub fn example(val: bool) -> bool {
    let x = X.load(Ordering::Acquire);
    X.store(val | x, Ordering::Release);
    let y = Y.load(Ordering::Acquire);
    x || y
}

```

The outputted assembly using `Acquire/Release` will look like this:

```

movb    example::X.0.0(%rip), %al # load(Acquire)
testb   %al, %al
setne    %al
orb      %dil, %al
movb     %al, example::X.0.0(%rip) # store(Release)
movb     $1, %al                    # load(Acquire)
retq

```

Now, on a weakly ordered CPU, the instructions might differ, but the result must be the same.

The store operation which used `Release` memory ordering is `movb %al, example::X.0.0(%rip)`. We know that on a strongly ordered system, this is enough to make sure that this is immediately set as `Invalid` in other caches if they contain this data.

So what's the problem?

Well, to actually point that out, we need to take a look at the relevant part of the C++ specification for `Release-Acquire`:

The synchronization is established only between the threads releasing and acquiring the same atomic variable. Other threads can see different order of memory accesses than either or both of the synchronized threads.

Rusts documentation for `Release` re-iterates on that and states:

... In particular, all **previous** writes become visible to all threads that perform an `Acquire` (or stronger) load of this value.

Now is the time where we go and take a closer look at one *non-guarantee* I mentioned in the start. The [Intel Developers Manual](#) goes in to a little more detail in chapter 8.2.3.4:

8.2.3.4 Loads May Be Reordered with Earlier Stores to Different Locations

The Intel-64 memory-ordering model allows a load to be reordered with an earlier store to a different location. However, loads are not reordered with stores to the same location.

So if we put this information together, we could have the following situation:

```
let x = X.load(Ordering::Acquire);
X.store(val | x, Ordering::Release); # earlier store to different location
let y = Y.load(Ordering::Acquire); # load

// *Could* get reordered on the CPU to this:

let x = X.load(Ordering::Acquire);
let y = Y.load(Ordering::Acquire);
X.store(val | x, Ordering::Release);
```

Now, I've tried my best on my Intel CPU to provoke a situation where this causes a problem, but I have not managed to get a simple example going to reliably showing that. But in theory, judging by the specs and the description of the abstract machine, `Acquire/Release` doesn't prevent this.

If we change our code to:

```
use std::sync::atomic::{AtomicBool, Ordering};
static X: AtomicBool = AtomicBool::new(true);
static Y: AtomicBool = AtomicBool::new(true);

pub fn example(val: bool) -> bool {
    let x = X.load(Ordering::SeqCst);
    X.store(val | x, Ordering::SeqCst);
    let y = Y.load(Ordering::SeqCst);
    x || y
}
```

We get the following assembly output:

```
movb    example::X.0.0(%rip), %al
testb   %al, %al
setne   %al
orb     %dil, %al
xchgb   %al, example::X.0.0(%rip)
movb    $1, %al
retq
```

The interesting change here is the store operation which is now changed from a simple load to a special instruction `xchgb %al, example::X.0.0(%rip)`. This is an *atomic operation* (`xchg` has an [implicit lock prefix](#)).

Since the `xchg` instruction is a [locked](#) instruction (when it refers to memory) it will make sure that all cache lines on other cores referring to the same memory are locked when the memory is fetched and then invalidated after the modification. In addition it works as a full memory fence which we can derive from chapter 8.2.3.9 in the [Intel Developer Manual](#):

8.2.3.9 Loads and Stores Are Not Reordered with Locked Instructions

The memory-ordering model prevents loads and stores from being reordered with locked instructions that execute earlier or later. The examples in this section illustrate only cases in which a locked instruction is executed before a load or a store. The reader should note that reordering is prevented also if the locked instruction is executed after a load or a store.

 For our observing core this is a noticeable change.

Sequential Consistency

If we somehow relied on the fact that the value we get from a `load` happens after, for example, the release of a flag, we could observe that it actually changed before the `Release` operation if we used `Acquire/Release` semantics. At least in theory.

Using the locking instruction prevents this. So in addition to having `Acquire/Release` guarantees, it also guarantees that no other memory operations, **reads** or writes, will happen in between.

Single total modification order

On a weakly ordered CPU, `SeqCst` also gives some guarantees which we get by default on a strongly ordered CPU, most importantly a *single total modification order*.

That means that if we have two observing cores, they will see all `SeqCst` operations in the same order. `Acquire/Release` does not give this guarantee. Observer-1 could see the two changes in a different order than observer-2 (remember the mailbox analogy).


Let's say core-1 acquires a flag X using `compare_and_swap` using `Acquire` ordering, and core-2 does the same on Y. Both do the same operations and then change the flag value back using `Release` store.

There is nothing that prevents Observer-1 from seeing the flag-Y change back before flag-X, and observer-2 the opposite.

`SeqCst` prevents this from happening.

On a strongly ordered system, every `store` is immediately visible on all other cores, so the modification order is not a real issue there.

`SeqCst` is the strongest of the memory orderings. It also has a slightly higher cost than the others.

 You can see an example of why above, since every atomic instruction has an overhead of involving the CPU's [cache coherency mechanism](#) and locking the memory location in the other caches. The fewer such instructions we need while still having a correct program, the better the performance.

Atomic Operations

In addition to the memory fences discussed above, using the atomic types in the

`std::sync::atomic` module gives access to some important CPU instructions we normally don't see in Rust:

From [Implementing Scalable Atomic Locks for Multi-Core Intel® EM64T and IA32 Architectures](#):

User level locks involve utilizing the atomic instructions of processor to atomically update a memory space. The atomic instructions involve utilizing a lock prefix on the instruction and having the destination operand assigned to a memory address. The following instructions can run atomically with a lock prefix on current Intel processors: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCH8B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, and XCHG...

Ok, so when we use the methods of an atomic, such as `fetch_add` on `AtomicUsize`, the compiler actually changes the instructions it emits for the addition of the two numbers on the CPU. The assembly would instead look something like (an AT&T dialect) `lock addq ...`, `...` instead of `addq ...`, `...` which we'd normally expect.

- ✓ An atomic operation is a set of operations executed as one indivisible unit. Any observer is prevented from seeing any of the sub operations or acquiring the same data while it's being operated on. Any conflicting operation-B from another core on the same data will have to wait until the the first atomic operation-A is finished.

Let's take the simple example of increasing a counter. There are three steps: load data, modify it and store data.

For each step, another core might change go in and load the same data, modify it and store it back before we're finished.

```
LOAD NUMBER
---- a competing core can load the same value here ----
INCREASE NUMBER
---- a competing core can increase the same value ----
---- a competing core can store its data here ----
STORE NUMBER
---- we overwrite that data here ----
```

Often we want to prevent anyone from observing or interfering from the point we load our data until we store our data. This is exactly what atomic operations solve for us.

A pretty normal use case for atomics is spin-locks. A very simple (and unsafe) one could look [like this](#):

```
static LOCKED: AtomicBool = AtomicBool::new(false);
static mut COUNTER: usize = 0;

pub fn spinlock(inc: usize) {
    while LOCKED.compare_and_swap(false, true, Ordering::Acquire) {}
    unsafe { COUNTER += inc };
    LOCKED.store(false, Ordering::Release);
}
```

This will result in the following assembly:


```

xorl    %eax, %eax
lock    cmpxchgb %cl, example::LOCKED(%rip)
jne     .LBB@_1
addq    %rdi, example::COUNTER(%rip)
movb    $0, example::LOCKED(%rip)
retq

```

`lock cmpxchgb %cl, example::LOCKED(%rip)` is the atomic operation we do in `compare_and_swap`. `lock cmpxchgb` is a [locking](#) operation; it reads a flag and changes its value if a certain condition is met.

 From chapter 8.2.5 in [Intel's Developer Manual](#):

Synchronization mechanisms in multiple-processor systems may depend upon a strong memory-ordering model. Here, a program can use a locking instruction such as the XCHG instruction or the LOCK prefix to ensure that a read-modify-write operation on memory is carried out atomically. Locking operations typically operate like I/O operations in that they wait for all previous instructions to complete and for all buffered writes to drain to memory (see Section 8.1.2, "Bus Locking").


Now what does this `lock` instruction prefix do?

It can quickly become a bit technical, but as far as I understand it, an easy way to model this is that it sets the cache line state to `Modified` already when the memory is fetched from the cache.

This way, from the moment it's fetched from a core's L1 cache it's marked as `Modified`. The processor uses its [cache coherence mechanism](#) to make sure the state is updated to `Invalid` on all other caches where it exists - even though they've not yet processed all of their messages in their mailboxes yet.

If message passing is the normal way of synchronizing changes, the `locked` instruction (and other memory-ordering or serializing instructions) involves a more expensive and more powerful mechanism which bypasses the message passing, locks the cache lines on the other caches (so no load or store operation can happen when while it's in progress) and sets them as invalid accordingly, which forces the cache to fetch an updated value from memory.

 If you're interested in reading more about this, then take a look at chapter 8 of the [Intel® 64 and IA-32 Architectures Software Developer's Manual](#).

 A cache line is most often 64 bytes on a 64 bit system. This can vary based on the exact CPU, but what is important to consider is that the locking mechanisms used if a memory crosses two cache lines is much more expensive and might involve bus locking and other hardware techniques.

Atomic operations crossing cache line boundaries have a very varying support based

on different architectures.

Conclusion

Are you still there? If so, relax now, we're done for today. Thanks for staying with me and reading through, I do sincerely hope you enjoyed it and got some value out of it.

I fundamentally believe that getting a good mental model around problems you actually work hard to deal with has numerous benefits for both your personal motivation and how you write your code, even though you never venture into the `std::sync::atomic` module at all in your daily life.

Until next time!