Rust Atomics And Locks

Mara Bos

June 14, 2023

Contents

1 Basics of Rust Concurrency

1.1 Threads in Rust

Rust std assigns every thread a unique identifier.

Remark (Output Locking). The `println` macro uses `std::io::Stdout::lock()` to make sure its output does not get interrupted. A `println!()` expression will wait until any concurrently running one is finished before writing any output.

Since a thread might run until the every end of the program's execution, the `spawn` function has a `'static` lifetime bound on its argument type. In other words, it only accepts functions that may kept around forever.

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

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

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

1.2 Scoped Threads

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

1.3 Shared Overship and Reference Counting

1.3.1 Statics

`static` is "owned" by the entire program.

```
static X: [i32; 3] = [1, 2, 3];
thread::spawn(|| dbg!(&X));
thread::spawn(|| dbg!(&X));
```

Every thread can borrow it, since it's guaranteed to always exist.

1.3.2 Leaking

Using `Box::leak`, one can release ownership of a `Box`, promising to never drop it.

```
let x: &'static [i32; 3] = Box::leak(Box::new([1, 2, 3]));
thread::spawn(move || dbg!(x));
thread::spawn(move || dbg!(x));
```

1.3.3 Reference Counting

To make sure that shared data gets dropped and deallocated, 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 dropped only when there are no owners left.

`std::rc::Rc` is short for "reference count". Cloning it will not allocate anything new, but instead increment a counter stored next to the contained value.

`Rc` is not **thread-safe**. Instead we can use `std::sync::Arc`, which stands for "atomically reference counted"

1.3.4 Borrowing and Data Races

1.3.5 Interior Mutability

A `std::cell::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.

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

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
}
```

A `std::cell::RefCell` does allow you to borrow its content, at a small runtime cost. A `RefCell<T>` does not only hold a `T`, but also holds a counter that keep track of any outstanding borrows. If you try to borrow it while it is already mutably borrowed (or vice-versa), it will panic, which avoids undefined behavior. A `RefCell` can only be used within a single thread.

Borrowing the contents is done by calling `borrow` or `borrow_mut`.

1.3.6 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.

1.3.7 Atomics

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<T>`.

1.3.8 UnsafeCell

`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.

1.4 Locking: Mutexes and RwLocks

1.4.1 Rust's Mutex

`std::sync::Mutex<T>`

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 access the data, and the Drop implementation of the guard will unlock the mutex.

```
use std::sync::Mutex;
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);
}
```

The `into_inner` method takes ownership of the mutex, which guarantees that nothing else can have a reference to the mutex anymore, making locking unnecessary.

1.4.2 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 holding the lock. When that happens, the `Mutex` will no longer be locked, but calling its lock method will result in an Err to indicate it has been poisoned.

This is a mechanism to protect against leaving the data that's protected by a mutex in an inconsistent state. In our example above, if a thread would panic after incrementing the integer fewer than 100 times, the mutex would unlock and the integer would be left in an unexpected state where it is no longer a multiple of 100, possibly breaking assumptions made by other threads. Automatically marking the mutex as poisoned in that case forces the user to handle this possibility.

1.4.3 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. It has `read()` and `write()` method for locking as either a reader or a writer.

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.

1.4.4 Waiting: Parking and Condition Variables

One way to wait for a notification from another thread is called **thread parking**.

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()`.

```
use std::collections::VecDeque;
fn main() {
    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));
        }
```

```
    });
}
```

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.

However, unpark requests don't stack up.

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 first unlocks the mutex and goes to sleep. Later, when woken up, it relocks the mutex and returns a new `MutexGuard`.

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

```
use std::sync::Condvar;
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));
    }
});
```

2 Atomics