Chapter 8. Operating System Primitives

So far, we've mostly focused on non-blocking operations. If we want to implement something like a mutex or condition variable, something that can wait for another thread to unlock or notify it, we need a way to efficiently block the current thread.

As we saw in Chapter 4, we can do this ourselves without the help of the operating system by *spinning*, repeatedly trying something over and over again, which can easily waste a lot of processor time. If we want to block efficiently, however, we need the help of the operating system's kernel.

The kernel, or more specifically the *scheduler* part of it, is responsible for deciding which process or thread gets to run when, for how long, and on which processor core. While a thread is waiting for something to happen, the kernel can stop giving it any processor time, prioritizing other threads that can make better use of this scarce resource.

We'll need a way to inform the kernel that we're waiting for something and ask it to put our thread to sleep until something relevant happens.

Interfacing with the Kernel

The way things are communicated with the kernel depends heavily on the operating system, and often even its version. Usually, the details of how this works are hidden behind one or more libraries that handle this for us. For example, using the Rust standard library, we can just call `File::open()` to open a file, without having to know any details about the operating system's kernel interface. Similarly, using the C standard library, `libc`, one can call the standard `fopen()` function to open a file. Calling such a function will eventually result in a call into the operating system's kernel, also known as a *syscall*, which is usually done through a specialized processor instruction. (On some architectures, that instruction is literally called `syscall`.)

Programs are generally expected, sometimes even required, to not make any syscalls directly, but to make use of higher level libraries that shipped with the operating system. On Unix systems, such as those based on Linux, `libc` takes this special role of providing the standard interface to the kernel.

The "Portable Operating System Interface" standard, more commonly known as the POSIX standard, includes additional requirements for `libc` on Unix systems. For example, next to the `fopen()` function from the C standard, POSIX additionally requires the

existence of the lower-level `open()` and `openat()` functions for opening files, which often correspond directly with a syscall. Because of the special status of `libc` on Unix systems, programs written in languages other than C usually still use `libc` for all their interactions with the kernel.

Rust software, including the standard library, often makes use of `libc` through the identically named `libc` crate.

For Linux specifically, the syscall interface is guaranteed to be stable, allowing us to make syscalls directly, without using `libc`. While that's not the most common or most advised route, it is slowly becoming more popular.

However, on macOS, also a Unix operating system that follows the POSIX standard, the syscall interface of the kernel is not stable, and we're not supposed to use it directly. The only stable interfaces that programs are allowed to use are provided through the libraries that ship with the system, such as `libc`, `libc++`, and various other libraries for C, C++, Objective-C, and Swift, Apple's programming languages of choice.

Windows does not follow the POSIX standard. It does not ship with an extended `libc` that serves as the main interface to the kernel, but instead ships with a separate set of libraries, such as `kernel32.dll`, that provide Windows-specific functions, such as `CreateFileW` for opening files. Just like on macOS, we're not supposed to make use of undocumented lower-level functions or make syscalls directly.

Through their libraries, operating systems provide us with synchronization primitives that need to interact with the kernel, such as mutexes and condition variables. Which part of their implementation is part of such a library or part of the kernel varies heavily per operating system. For example, sometimes the mutex lock and unlock operations correspond directly with a kernel syscall, whereas on other systems, the library handles most of the operations and will only perform a syscall when a thread needs to be blocked or woken up. (The latter tends to be more efficient, as making a syscall can be slow.)

POSIX

As part of the POSIX Threads extensions, better known as pthreads, POSIX specifies data types and functions for concurrency. While technically specified as part of a separate system library, `libpthread`, this functionality is nowadays often included in `libc` directly.

Next to functionality like spawning and joining threads (`pthread_create` and `pthread_join`), pthread provides the most common synchronization primitives: mutexes (`pthread_mutex_t`), reader-writer locks (`pthread_rwlock_t`), and condition variables (`pthread_cond_t`).

`pthread_mutex_t`

Pthread's mutex must be initialized by calling `pthread_mutex_init()` and destroyed with `pthread_mutex_destroy()`. The initialization function takes an argument of type `pthread_mutexattr_t` that can be used to configure some of the properties of the mutex.

One of those properties is its behavior on *recursive locking*, which happens when the same thread that already holds a lock, attempts to lock it again. This results in undefined behavior when using the default setting (`PTHREAD_MUTEX_DEFAULT`), but it can also be configured to result in an error (`PTHREAD_MUTEX_ERRORCHECK`), in a deadlock (`PTHREAD_MUTEX_NORMAL`), or in a successful second lock (`PTHREAD_MUTEX_RECURSIVE`).

These mutexes are locked through `pthread_mutex_lock()` or `pthread_mutex_trylock()`, and unlocked through `pthread_mutex_unlock()`. Additionally, unlike Rust's standard mutex, they also support locking with a time limit, through `pthread_mutex_timedlock()`.

A `pthread_mutex_t` can be statically initialized without a call to `pthread_mutex_init()` by assigning it the value `PTHREAD_MUTEX_INITIALIZER`. However, this is only possible for a mutex with default settings.

`pthread_rwlock_t`

Pthread's reader-writer lock is initialized and destroyed through `pthread_rwlock_init()` and `pthread_rwlock_destroy()`. Similar to a mutex, a default

`pthread_rwlock_t` can also be initialized statically with `PTHREAD_RWLOCK_INITIALIZER`.

A pthread reader-writer lock has significantly fewer properties that can be configured through its initialization function, compared to a pthread mutex. Most notably, attempting to recursively write-lock it will always result in a deadlock.

Attempts to recursively acquire additional read locks, however, are guaranteed to succeed, even if there are writers waiting. This effectively rules out any efficient implementation that prioritizes writers over readers, which is why most pthread implementations prioritize readers.

Its interface is nearly identical to that of `pthread_mutex_t`, including support for time limits, except each locking function comes in two variants: one for readers (`pthread_rwlock_rdlock`), and one for writers (`pthread_rwlock_wrlock`). Perhaps surprisingly, there is only one unlock function (`pthread_rwlock_unlock`) that's used for unlocking a lock of either kind.

`pthread_cond_t`

A pthread condition variable is used together with a pthread mutex. It is initialized and destroyed through `pthread_cond_init` and `pthread_cond_destroy`, and has a few attributes that can be configured. Most notably, we can configure whether time limits will use a monotonic clock (like Rust's `Instant`) or real-time clock (like Rust's `SystemTime`, somtimes called "wall-clock time"). A condition

variable with default settings, such as one statically initialized by `PTHREAD_COND_INITIALIZER`, uses the real-time clock.

Waiting for such a condition variable, optionally with a time limit, is done through `pthread_cond_timedwait()`. Waking up a waiting thread is done by calling `pthread_cond_signal()`, or, to wake all waiting threads at once, `pthread_cond_broadcast()`.

The remaining synchronization primitives provided by pthread are barriers (`pthread_barrier_t`), spin locks (`pthread_spinlock_t`), and one-time initialization (`pthread_once_t`), which we will not discuss.

Wrapping in Rust

It might seem like we could easily expose these pthread synchronization primitives to Rust by conveniently wrapping their C type (through the `libc` crate) in a Rust struct, like this:

```
pub struct Mutex {
    m: libc::pthread_mutex_t,
}
```

However, there are a few issues with that, as this pthread type was designed for C, not for Rust.

First of all, Rust has rules about mutability and borrowing, which normally don't allow for mutations to something when shared. Since functions like `pthread_mutex_lock` will most likely mutate the mutex, we'll need interior mutability to make sure that's acceptable. So, we'll have to wrap it in an `UnsafeCell`:

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

A much bigger problem is related to *moving*.

In Rust, we move objects around all the time. For example, by returning an object from a function, passing it as an argument, or simply assigning it to a new place. Everything that we own (and that isn't borrowed by anything else), we can freely move into a new place.

In C, however, that's not universally true. It's quite common for a type in C to depend on its memory address staying constant. For example, it might contain a pointer that points into itself, or store a pointer to itself in some global data structure. In that case, moving it to a new place could result in undefined behavior.

The pthread types we discussed do not guarantee they are *movable*, which becomes quite a problem in Rust. Even a simple idiomatic `Mutex::new()` function is a problem: it would return a mutex object, which would move it into a new place in memory.

Since a user could always move around any mutex object they own, we either need to make them promise they won't do that, by making the interface `unsafe`; or we need to take away their ownership and hide everything behind a wrapper (something that `std::pin::Pin` can be used for). Neither of these are good solutions, as they impact the interface of our mutex type, making it very error-prone and/or inconvenient to use.

A solution to this problem is to wrap the mutex in a `Box`. By putting the pthread mutex in its own allocation, it stays in the same location in memory, even if its owner is moved around.

```
pub struct Mutex {
    m: Box<UnsafeCell<libc::pthread_mutex_t>>,
}
```

> *This is how `std::sync::Mutex` was implemented on all Unix platforms before Rust 1.62.*

The downside of this approach is the overhead: every mutex now gets its own allocation, adding significant overhead to creating, destroying, and using the mutex. Another downside is that it prevents the `new` function from being `const`, which gets in the way of having a `static` mutex.

Even if `pthread_mutex_t` was movable, a `const fn new` could only initialize it with default settings, which results in undefined behavior when locking recursively. There is

no way to design a safe interface that prevents locking recursively, so this means we'd need to make the `lock` function `unsafe` to make the user promise they won't do that.

A problem that remains with our `Box` approach occurs when dropping a locked mutex. It might seem that, with the right design, it would not be possible to drop a `Mutex` while locked, since it's impossible to drop it while it's still borrowed by a `MutexGuard`. The `MutexGuard` would have to be dropped first, unlocking the `Mutex`. However, in Rust, it's safe to forget (or leak) an object, without dropping it. This means one could write something like this:

```
fn main() {
    let m = Mutex::new(..);

    let guard = m.lock(); // Lock it ..
    std::mem::forget(guard); // .. but don't unlock it.
}
```

In the example above, `m` will be dropped at the end of the scope while it is still locked. This is fine, according to the Rust compiler, because the guard has been leaked and can't be used anymore.

However, pthread specifies that calling `pthread_mutex_destroy()` on a locked mutex is not guaranteed to work and might result in undefined behavior. One work-around is to first attempt to lock (and unlock) the pthread mutex when dropping our `Mutex`, and panic (or leak the `Box`) when it is already locked, but that adds even more overhead.

These issues don't just apply to `pthread_mutex_t`, but to the other types we discussed as well. Overall, the design of the pthread synchronization primitives is fine for C, but just not a great fit for Rust.

Linux

On Linux systems, the pthread synchronization primitives are all implemented using the *futex* syscall. Its name comes from "fast user-space mutex," as the original motivation for adding this syscall was to allow libraries (like pthread implementations) to include a fast and efficient mutex implementation. It's much more flexible than that though and can be used to build many different synchronization tools.

The futex syscall was added to the Linux kernel in 2003 and has seen several improvements and extensions since then. Some other operating systems have since also added similar functionality, most notably Windows 8 in 2012 with the addition of `WaitOnAddress` (which we'll discuss a bit later, in "Windows"). In 2020, the C++ language even added support for basic futex-like operations to its standard library, with the addition of the `atomic_wait` and `atomic_notify` functions.

Futex

On Linux, `SYS_futex` is a syscall that implements various operations that all operate on a 32-bit atomic integer. The main two operations are `FUTEX_WAIT` and `FUTEX_WAKE`.

The wait operation puts a thread to sleep, and a wake operation on the same atomic variable wakes the thread up again.

These operations don't store anything in the atomic integer. Instead, the kernel remembers which threads are waiting on which memory address to allow a wake operation to wake up the right threads.

In "Waiting: Parking and Condition Variables" in Chapter 1, we saw how other mechanisms for blocking and waking up threads need a way to make sure that wake operations don't get lost in a race. For thread parking, that issue is solved by making the `unpark()` operation also apply to future `park()` operations as well. And for condition variables, that's taken care of by the mutex that is used together with the condition variable.

For the futex wait and wake operations, another mechanism is used. The wait operation takes an argument which specifies the value we expect the atomic variable to have and will refuse to block if it doesn't match. The wait operation behaves atomically with respect to the wake operation, meaning that no wake signal can get lost between the check of the expected value and the moment it actually goes to sleep.

If we make sure that the value of the atomic variable is changed right before a wake operation, we can be sure that a thread that's about to start waiting will not go to sleep, such that potentially missing the futex wake operation no longer matters.

Let's go over a minimal example to see this in practice.

First, we need to be able to invoke these syscalls. We can use the `syscall` function from the `libc` crate to do so, and wrap each of them in a convenient Rust function, as follows:

```
#[cfg(not(target_os = "linux"))]
compile_error!("Linux only. Sorry!");

pub fn wait(a: &AtomicU32, expected: u32) {
    // Refer to the futex (2) man page for the syscall signature.
    unsafe {
        libc::syscall(
            libc::SYS_futex, // The futex syscall.
            a as *const AtomicU32, // The atomic to operate on.
            libc::FUTEX_WAIT, // The futex operation.
            expected, // The expected value.
            std::ptr::null::<libc::timespec>(), // No timeout.
        );
    }
}

pub fn wake_one(a: &AtomicU32) {
    // Refer to the futex (2) man page for the syscall signature.
    unsafe {
        libc::syscall(
            libc::SYS_futex, // The futex syscall.
            a as *const AtomicU32, // The atomic to operate on.
            libc::FUTEX_WAKE, // The futex operation.
            1, // The number of threads to wake up.
        );
```

```
    }
}
```

Now, as a usage example, let's use these to make a thread wait for another. We'll use an atomic variable that we initialize at zero, which the main thread will futex-wait on. A second thread will change the variable to one, and then run a futex wake operation on it to wake up the main thread.

Just like thread parking and waiting on a condition variable, a futex wait operation can *spuriously* wake up, even when nothing happened. Therefore, it is most commonly used in a loop, repeating it if the condition we're waiting for isn't met yet.

Let's take a look at the example below:

```
fn main() {
    let a = AtomicU32::new(0);

    thread::scope(|s| {
        s.spawn(|| {
            thread::sleep(Duration::from_secs(3));
            a.store(1, Relaxed);  1
            wake_one(&a);  2
        });

        println!("Waiting...");
        while a.load(Relaxed) == 0 {  3
            wait(&a, 0);  4
        }
```

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

1 The spawned thread will set the atomic variable to one after a few seconds.

2 It then executes a futex wake operation to wake up the main thread, in case it was sleeping, so it can see that the variable has changed.

3 The main thread waits as long as the variable is zero, before continuing to print its final message.

4 The futex wait operation is used to put the thread to sleep. Very importantly, this operation will check whether `a` is still zero before going to sleep, which is the reason that the signal from the spawned thread cannot get lost between `3` and `4`. Either `1` (and therefore `2`) did not happen yet and it goes to sleep, or `1` (and maybe `2`) already happened, and it immediately continues.

An important observation to make here is that the `wait` call is avoided entirely if `a` was already set to one before the `while` loop. In a similar fashion, if the main thread had also stored in the atomic variable whether it started waiting for the signal (by setting it to a value other than zero or one), the signaling thread could skip the futex wake operation if the main thread hadn't started waiting yet. This is what makes futex-based synchronization primitives so fast: since we manage the state ourselves, we don't need to rely on the kernel, except when we actually need to block.

Since Rust 1.48, the standard library's thread parking functions on Linux are implemented like this. They use one atomic variable per thread, with three possible states: zero for the idle and initial state, one for "unparked but not yet parked," and minus one for "parked but not yet unparked."

In Chapter 9, we'll implement mutexes, condition variables, and reader-writer locks using these operations.

Futex Operations

Next to the wait and wake operations, the futex syscall supports several other operations. In this section, we'll briefly discuss every operation supported by this syscall.

The first argument to the futex is always a pointer to the 32-bit atomic variable to operate on. The second argument is a constant representing the operation, such as `FUTEX_WAIT`, to which up to two flags can be added: `FUTEX_PRIVATE_FLAG` and/or `FUTEX_CLOCK_REALTIME`, which we will discuss below. The remaining arguments depend on the operation and are described for each of the operations below.

`FUTEX_WAIT`

: This operation takes two additional arguments: the value the atomic variable is expected to have and a pointer to a `timespec` representing the maximum time to wait.

If the value of the atomic variable matches the expected value, the wait operation blocks until woken up by one of the wake operations, or until the duration specified by the `timespec` has passed. If the pointer to the `timespec` is null, there is no time limit. Additionally, the wait operation might spuriously wake up and return without a corresponding wake operation, before the time limit is reached.

The check and blocking operation happens as a single atomic operation with respect to other futex operations, meaning that no wake signals can get lost between them.

The duration specified by the `timespec` represents a duration on the monotonic clock (like Rust's `Instant`) by default. By adding the `FUTEX_CLOCK_REALTIME` flag, the real-time clock (like Rust's `SystemTime`) is used instead.

The return value indicates whether the expected value matched and whether the timeout was reached or not.

`FUTEX_WAKE`

This operation takes one additional argument: the number of threads to wake up, as an `i32`.

This wakes up as many threads as specified that are blocked in a wait operation on the same atomic variable. (Or fewer if there are not that many waiting threads.) Most commonly, this argument is either one to wake up just one thread, or `i32::MAX` to wake up all threads.

The number of awoken threads is returned.

FUTEX_WAIT_BITSET

This operation takes four additional arguments: the value the atomic variable is expected to have, a pointer to a `timespec` representing the maximum time to wait, a pointer that's ignored, and a 32-bit "bitset" (a `u32`).

This operation behaves the same as `FUTEX_WAIT`, with two differences.

The first difference is that it takes a `bitset` argument that can be used to wait for only specific wake operations, rather than all wake operations on the same atomic variable. A `FUTEX_WAKE` operation is never ignored, but a signal from a `FUTEX_WAKE_BITSET` operation is ignored if the wait bitset and the wake bitset do not have any 1-bit in common.

For example, a `FUTEX_WAKE_BITSET` operation with a bitset of `0b0101` will wake up a `FUTEX_WAIT_BITSET` operation with a bitset of `0b1100`, but not one with a bitset of `0b0010`.

This might be useful when implementing something like a reader-writer lock, to wake up a writer without waking up any readers. However, note that using two separate atomic variables can be more efficient than using a single one for two different kinds of waiters, since the kernel will keep a single list of waiters per atomic variable.

The other difference with `FUTEX_WAIT` is that the `timespec` is used as an absolute timestamp, rather than a duration. Because of this, `FUTEX_WAIT_BITSET` is often used with a bitset of `u32::MAX` (all bits set), effectively turning it into a regular `FUTEX_WAIT` operation, but with an absolute timestamp for the time limit.

FUTEX_WAKE_BITSET

This operation takes four additional arguments: the number of threads to wake up, two pointers that are ignored, and a 32-bit "bitset" (a `u32`).

This operation is identical to `FUTEX_WAKE`, except it does not wake up `FUTEX_WAIT_BITSET` operations with a bitset that does not overlap. (See `FUTEX_WAIT_BITSET` above.)

With a bitset of `u32::MAX` (all bits set), this is identical to `FUTEX_WAKE`.

FUTEX_REQUEUE

This operation takes three additional arguments: the number of threads to wake up (an `i32`), the number of threads to requeue (an `i32`), and the address of a secondary atomic variable.

This operation wakes up a given number of waiting threads, and then *requeues* a given number of remaining waiting threads to instead wait on another atomic variable.

Waiting threads that are requeued continue to wait, but are no longer affected by wake operations on the primary atomic variable. Instead, they are now woken up by wake operations on the secondary atomic variable.

This can be useful for implementing something like the "notify all" operation of a condition variable. Instead of waking up all threads, which will subsequently try to lock a mutex, most likely making all but one thread wait for that mutex right afterwards, we could only wake up a single thread and requeue all the others to directly wait for the mutex without waking them up first.

Just like with the `FUTEX_WAKE` operation, the value of `i32::MAX` can be used to requeue all waiting threads. (Specifying a value of `i32::MAX` for the number of threads to wake up is not very useful, since that will make this operation equivalent to `FUTEX_WAKE`.)

The number of awoken threads is returned.

`FUTEX_CMP_REQUEUE`

This operation takes four additional arguments: the number of threads to wake up (an `i32`), the number of threads to requeue (an `i32`), the address of a secondary atomic variable, and the value the primary atomic variable is expected to have.

This operation is nearly identical to `FUTEX_REQUEUE`, except it refuses to operate if the value of the primary atomic variable does not match the expected value. The

check of the value and the requeueing operation happens atomically with respect to other futex operations.

Unlike `FUTEX_REQUEUE`, this returns the sum of the number of awoken and requeued threads.

`FUTEX_WAKE_OP`

This operation takes four additional arguments: the number of threads to wake up on the primary atomic variable (an `i32`), the number of threads to potentially wake up on the second atomic variable (an `i32`), the address of a secondary atomic variable, and a 32-bit value encoding both an operation and a comparison to be made.

This is a very specialized operation that modifies the secondary atomic variable, wakes a number of threads waiting on the primary atomic variable, checks if the previous value of the atomic variable matches a given condition, and if so, also wakes a number of threads on a secondary atomic variable.

In other words, it is identical to the following code, except the entire operation behaves atomically with respect to other futex operations:

```
let old = atomic2.fetch_update(Relaxed, Relaxed, some_operation);
wake(atomic1, N);
if some_condition(old) {
```

```
        wake(atomic2, M);
    }
```

The modifying operation to perform and the condition to check are both specified by the last argument to the syscall, encoded in its 32 bits. The operation can be one of the following: assignment, addition, binary `or`, binary `and-not`, and binary `xor`, with either a 12-bit argument or a 32-bit argument that's a power of two. The comparison can be chosen from `==`, `!=`, `<`, `<=`, `>`, and `>=`, with a 12-bit argument.

See the `futex(2)` Linux man page for details on the encoding of this argument, or use the `linux-futex` crate on crates.io, which includes a convenient way to construct this argument.

This operation returns the total number of awoken threads.

At first glance this may seem like a flexible operation with many use cases. However, it was designed for just one specific use case in GNU `libc` where two threads had to be woken up from two separate atomic variables. That specific case has been replaced by a different implementation that no longer makes use of `FUTEX_WAKE_OP`.

The `FUTEX_PRIVATE_FLAG` can be added to any of these operations to enable a possible optimization if all relevant futex operations on the same atomic variable(s) come from

threads of the same process, which is usually the case. To make use of it, every relevant futex operation must include this same flag. By allowing the kernel to assume there will be no interactions with other processes, it can skip some otherwise potentially expensive steps in performing futex operations, improving performance.

In addition to Linux, NetBSD also supports all the futex operations described above. OpenBSD also has a futex syscall, but only supports the `FUTEX_WAIT`, `FUTEX_WAKE`, and `FUTEX_REQUEUE` operations. FreeBSD does not have a native futex syscall, but does include a syscall called `_umtx_op`, which includes functionality nearly identical to `FUTEX_WAIT` and `FUTEX_WAKE`: `UMTX_OP_WAIT` (for 64-bit atomics), `UMTX_OP_WAIT_UINT` (for 32-bit atomics), and `UMTX_OP_WAKE`. Windows also includes functions that behave very similarly to the futex wait and wake operations, which we will discuss later in this chapter.

New Futex Operations

As of Linux 5.16, released in 2022, there is an additional futex syscall: `futex_waitv`. This new syscall allows waiting for more than one futex at once, by providing it a list of atomic variables (and their expected values) to wait on. A thread blocked on `futex_waitv` can be woken up by a wake operation on any of the specified variables.

This new syscall also leaves space for future extensions. For example, it's possible to specify the size of the atomic variable to wait on. While the initial implementation only supports 32-bit atomics, just like the original `futex` syscall, this might be extended in the future to include 8-bit, 16-bit, and 64-bit atomics.

Priority Inheritance Futex Operations

Priority inversion is a problem that occurs when a high priority thread is blocked on a lock held by a low priority thread. The high priority thread effectively has its priority "inverted," since it now has to wait for the low priority thread to release the lock before it can make progress.

A solution to this problem is *priority inheritance*, in which the blocking thread inherits the priority of the highest priority thread that is waiting for it, temporarily increasing the priority of the low priority thread while it holds the lock.

In addition to the seven futex operations we discussed before, there are six priority inheriting futex operations specifically designed for implementing *priority inheriting* locks.

The general futex operations we discussed before do not have any requirements for the exact contents of the atomic variable. We got to choose ourselves what the 32 bits

represent. However, for a priority inheriting mutex, the kernel needs to be able to understand whether the mutex is locked, and if so, which thread has locked it.

To avoid having to make a syscall on every state change, the priority inheritance futex operations specify the exact contents of the 32-bit atomic variable, so the kernel can understand it: the highest bit represents whether there are any threads waiting to lock the mutex, and the lowest 30 bits contain the thread ID (the Linux `tid`, not the Rust `ThreadId`) of the thread that holds the lock, or zero when unlocked.

As an extra feature, the kernel will set the second highest bit if the thread that holds the lock terminates without unlocking it, but only if there are any waiters. This allows for the mutex to be *robust*: a term used to describe a mutex that can gracefully handle a situation where its "owning" thread unexpectedly terminates.

The priority inheriting futex operations have a one to one correspondence to the standard mutex operations: `FUTEX_LOCK_PI` for locking, `FUTEX_UNLOCK_PI` for unlocking, and `FUTEX_TRYLOCK_PI` for locking without blocking. Additionally, the `FUTEX_CMP_REQUEUE_PI` and `FUTEX_WAIT_REQUEUE_PI` operations can be used to implement a condition variable that pairs with a priority inheriting mutex.

We'll not discuss these operations in detail. See the `futex(2)` Linux man page or the `linux-futex` crate on crates.io for their details.

macOS

The kernel that's part of macOS supports various useful low-level concurrency related syscalls. However, just like most operating systems, the kernel interface is not considered stable, and we're not supposed to use it directly.

The only way software should interact with the macOS kernel is through the libraries that ship with the system. These libraries include its standard library implementations for C (libc), C++ (libc++), Objective-C, and Swift.

As a POSIX-compliant Unix system, the macOS C library includes a full pthread implementation. The standard locks in other languages tend to use pthread's primitives under the hood.

Pthread's locks tend to be relatively slow on macOS compared to the equivalent on other operating systems. One of the reasons is that the locks on macOS behave as *fair locks* by default. This means that when several threads attempt to lock the same mutex, they are served in order of arrival, like a perfect queue. While fairness can be a desirable property, it can significantly reduce performance, especially under high contention.

os_unfair_lock

Next to the pthread primitives, macOS 10.12 introduced a new lightweight platform-specific mutex, which is not fair: `os_unfair_lock`. It is just 32 bits in size, initialized statically with the `OS_UNFAIR_LOCK_INIT` constant, and does not require destruction. It

can be locked through `os_unfair_lock_lock()` (blocking) or `os_unfair_lock_trylock()` (non-blocking), and is unlocked through `os_unfair_lock_unlock()`.

Unfortunately, it does not come with a condition variable, nor does it have a reader-writer variant.

Windows

The Windows operating system ships with a set of libraries that together form the *Windows API*, often called the "Win32 API" (even on 64-bit systems). This forms a layer on top of the "Native API": the largely undocumented interface with the kernel, which we're not supposed to use directly.

The Windows API is made available to Rust programs through Microsoft's official `windows` and `windows-sys` crates, which are available on crates.io.

Heavyweight Kernel Objects

Many of the older synchronization primitives available on Windows are managed fully by the kernel, making them quite heavyweight, and giving them similar properties as other kernel-managed objects, such as files. They can be used by multiple processes, they can be named and located by their name, and they support fine-grained permis-

sions, similar to files. For example, it's possible to allow a process to wait on some object, without allowing it to send signals through it to wake others.

These heavyweight kernel-managed synchronization objects include `Mutex` (which can be locked and unlocked), `Event` (which can be signalled and waited for), and `WaitableTimer` (which can be automatically signalled after a chosen time, or periodically). Creating such an object results in a `HANDLE`, just like opening a file, that can be easily passed around and used with the regular `HANDLE` functions; most notably the family of wait functions. These functions allow us to wait for one or more objects of various types, including the heavyweight synchronization primitives, processes, threads, and various forms of I/O.

Lighter-Weight Objects

A lighter-weight synchronization primitive included in the Windows API is the "critical section."

The term *critical section* refers to a part of a program, a "section" of code, that may not be entered concurrently by more than one thread. The mechanism for protecting a critical section is often called a mutex. In this case, however, Microsoft used the name "critical section" for the mechanism, quite possibly because the name "mutex" was already taken by the heavyweight `Mutex` object discussed above.

A Windows `CRITICAL_SECTION` is effectively a *recursive mutex*, except it uses the terms "enter" and "leave" rather than "lock" and "unlock." As a recursive mutex, it is designed to only protect against other threads. It allows the same thread to lock (or "enter") it more than once, requiring it to also unlock (leave) it the same number of times.

This is something to keep in mind when wrapping this type in Rust. Successfully locking (entering) a `CRITICAL_SECTION` shouldn't result in an exclusive reference (`&mut T`) to data protected by it. Otherwise, a thread could use this to create two exclusive references to the same data, which immediately results in undefined behavior.

A `CRITICAL_SECTION` is initialized using the `InitializeCriticalSection()` function, destroyed with `DeleteCriticalSection()`, and may not be moved. It is locked through `EnterCriticalSection()` or `TryEnterCriticalSection()`, and unlocked with `LeaveCriticalSection()`.

> *Until Rust 1.51, `std::sync::Mutex` on Windows XP was based on a (`Box`-allocated) `CRITICAL_SECTION` object. (Rust 1.51 dropped support for Windows XP.)*

Slim reader-writer locks

As of Windows Vista (and Windows Server 2008), the Windows API includes a much nicer locking primitive that's very lightweight: the *slim reader-writer lock*, or *SRW lock* for short.

The `SRWLOCK` type is just one pointer in size, can be statically initialized with `SRWLOCK_INIT`, and does not require destruction. While not in use (borrowed), we're even allowed to move it, making it an excellent candidate for being wrapped in a Rust type.

It provides exclusive (writer) locking and unlocking through `AcquireSRWLockExclusive()`, `TryAcquireSRWLockExclusive()`, and `ReleaseSRWLockExclusive()`, and provides shared (reader) locking and unlocking through `AcquireSRWLockShared()`, `TryAcquireSRWLockShared()`, and `ReleaseSRWLockShared()`. It is often used as a regular mutex, simply by ignoring the shared (reader) locking functions.

An SRW lock prioritizes neither writers nor readers. While not guaranteed, it attempts to serve all lock requests in order, as far as possible without reducing performance. One must not attempt to acquire a second shared (reader) lock on a thread that already holds one. Doing so could lead to a permanent deadlock if the operation gets queued behind an exclusive (writer) lock operation of another thread, which would be blocked because of the first shared (reader) lock that the first thread already holds.

The SRW lock was introduced to the Windows API together with the condition variable. Similar to an SRW lock, a `CONDITION_VARIABLE` is just one pointer in size, can be initialized statically with `CONDITION_VARIABLE_INIT`, and does not require destruction. We're also allowed to move it, as long as it isn't in use (borrowed).

This condition variable can not only be used together with an SRW lock, through `SleepConditionVariableSRW`, but also with a critical section, through `SleepConditionVariableCS`.

Waking up waiting threads is done either through `WakeConditionVariable` to wake a single thread, or `WakeAllConditionVariable` to wake all waiting threads.

> *Originally, Windows SRW locks and condition variables used in the standard library were wrapped in a `Box` to avoid moving the objects. Microsoft didn't document the movability guarantees until we requested that in 2020. Since then, as of Rust 1.49, `std::sync::Mutex`, `std::sync::RwLock`, and `std::sync::Condvar` on Windows Vista and later directly wrap an `SRWLOCK` or `CONDITION_VARIABLE`, without any allocations.*

Address-Based Waiting

Windows 8 (and Windows Server 2012) introduced a new, more flexible, type of synchronization functionality that is very similar to the Linux `FUTEX_WAIT` and `FUTEX_WAKE` operations we discussed earlier in this chapter.

The `WaitOnAddress` function can operate on an 8-bit, 16-bit, 32-bit, or 64-bit atomic variable. It takes four arguments: the address of the atomic variable, the address of a variable that holds the expected value, the size of the atomic variable (in bytes), and the maximum number of milliseconds to wait before giving up (or `u32::MAX` for an infinite timeout).

Just like the `FUTEX_WAIT` operation, it compares the value of the atomic variable with the expected value, and goes to sleep if they match, waiting for a corresponding wake operation. The check and sleep operation happens atomically with respect to wake operations, meaning no wake signal can get lost in between.

Waking a thread that's waiting on `WaitOnAddress` is done through either `WakeByAddressSingle` to wake a single thread, or `WakeByAddressAll` to wake all waiting threads. These two functions take just a single argument: the address of the atomic variable, which was also passed to `WaitOnAddress`.

Some, but not all, of the synchronization primitives of the Windows API are implemented using these functions. More importantly, they are a great building block for building our own primitives, which we will do in Chapter 9.

~

Summary

- A *syscall* is a call into the operating system's kernel and is relatively slow compared to a regular function call.

- Usually, programs don't make syscalls directly, but instead go through the operating system's libraries (e.g., `libc`) to interface with the kernel. On many operating systems, this is the only supported way of interfacing with the kernel.
- The `libc` crate gives Rust code access to `libc`.
- On POSIX systems, `libc` includes more than what's required by the C standard to comply with the POSIX standard.
- The POSIX standard includes *pthreads*, a library with concurrency primitives such as `pthread_mutex_t`.
- Pthread types are designed for C, not for Rust. For example, they are not movable, which can be a problem.
- Linux has a *futex* syscall supporting several waiting and waking operations on an `AtomicU32`. The wait operation verifies the expected value of the atomic, which is used to avoid missing notifications.
- In addition to pthread, macOS also provides `os_unfair_lock` as a lightweight locking primitive.
- Windows heavyweight concurrency primitives always require interacting with the kernel, but can be passed between processes and used with the standard Windows waiting functions.

- Windows lightweight concurrency primitives include a "slim" reader-writer lock (*SRW lock*) and a condition variable. These are easily wrapped in Rust, as they are movable.
- Windows also provides basic futex-like functionality, through `WaitOnAddress` and `WakeByAddress`.

Next: Chapter 9. Building Our Own Locks