Chapter 5. Building Our Own Channels

Channels can be used to send data between threads, and they come in many variants. Some channels can only be used between exactly one sender and one receiver, while others can send from any number of threads, or even allow multiple receivers. Some channels are blocking, meaning that receiving (and sometimes sending) is a blocking operation, making your thread sleep until the operation can be completed. Some channels are optimized for throughput, while others are optimized for low latency.

The variations are endless, and there is no one-size-fits-all version that fits all use cases.

In this chapter, we'll implement a few relatively simple channels to not only explore some more applications of atomics, but also to learn more about how our requirements and assumptions can be captured in Rust's type system.

A Simple Mutex-Based Channel

A basic channel implementation does not require any knowledge of atomics. We can take a `VecDeque`, which is basically a `Vec` that allows for efficient adding and removing of elements on both ends, and protect it with a `Mutex` to allow multiple threads to access it. We then use the `VecDeque` as a queue of data, often called *messages*, that's been sent but not yet received. Any thread that wants to send a message can simply add it to the back of the queue, and any thread that wants to receive a message just has to remove one from the front of the queue.

There's just one more thing to add, which is used to make the receive operation blocking: a `Condvar` (see "Condition Variables" in Chapter 1) to notify waiting receivers of a new messsage.

An implementation of this can be quite short and relatively straightforward, as shown below:

```
pub struct Channel<T> {
    queue: Mutex<VecDeque<T>>,
    item_ready: Condvar,
}

impl<T> Channel<T> {
    pub fn new() -> Self {
        Self {
            queue: Mutex::new(VecDeque::new()),
            item_ready: Condvar::new(),
        }
    }
```

```
    pub fn send(&self, message: T) {
        self.queue.lock().unwrap().push_back(message);
        self.item_ready.notify_one();
    }

    pub fn receive(&self) -> T {
        let mut b = self.queue.lock().unwrap();
        loop {
            if let Some(message) = b.pop_front() {
                return message;
            }
            b = self.item_ready.wait(b).unwrap();
        }
    }
}
```

Note how we didn't have to use any atomics or unsafe code and didn't have to think about the `Send` or `Sync` traits. The compiler understands the interface of `Mutex` and what guarantees that type provides, and will implicitly understand that if both `Mutex<T>` and `Condvar` can safely be shared between threads, so can our `Channel<T>`.

Our `send` function locks the mutex to push the new message to the back of the queue, and directly notifies one potentially waiting receiver after unlocking the queue, by using the condition variable.

The `receive` function also locks the mutex to pop the next message from the front of the queue, but will use the condition variable to wait if there's no message available

yet.

> *Remember that the `Condvar::wait` method will unlock the `Mutex` while waiting and relock it before returning. So, our `receive` function will not keep the mutex locked while waiting.*

While this channel is very flexible in usage, as it allows any number of sending and receiving threads, its implementation can be far from optimal in many situations. Even if there are plenty of messages ready to be received, any send or receive operation will briefly block any other send or receive operation, since they all have to lock the same mutex. If `VecDeque::push` has to grow the capacity of the `VecDeque`, all sending and receiving threads will have to wait for that one thread to finish the reallocation, which might be unacceptable in some situations.

Another property which might be undesirable is that this channel's queue might grow without bounds. Nothing is stopping senders from continuously sending new messages at a higher rate than receivers are processing them.

An Unsafe One-Shot Channel

The variety of use cases for channels is virtually endless. However, in the rest of this chapter, we'll focus on a specific type of use case: sending exactly one message from one thread to another. A channel designed for such a use case is often called a *one-shot channel*.

We could take our `Mutex<VecDeque>` based implementation from above and substitute the `VecDeque` for an `Option`, effectively reducing the capacity of the queue to exactly one message. It would avoid allocation, but would still have some of the same downsides of using a `Mutex`. We can avoid this by building our own one-shot channel from scratch using atomics.

First, let's build a minimal implementation of a one-shot channel without putting much thought into its interface. Later in this chapter, we'll explore ways to improve its interface, and how to team up with Rust's type system to provide users of our channel a pleasant experience.

The tools we need to start with are basically the same as we used for our `SpinLock<T>` (from Chapter 4): an `UnsafeCell` for storage and an `AtomicBool` to indicate its state. In this case, we use the atomic boolean to indicate whether the message is ready for consumption.

Before a message is sent, the channel is "empty" and does not contain any message of type `T` yet. We could use an `Option<T>` inside the cell to allow for the absence of a `T`. However, that could waste valuable space in memory, since our atomic boolean already tells us whether there is a message or not. Instead, we can use a `std::mem::MaybeUninit<T>`, which is essentially the bare bones unsafe version of `Option<T>`: it requires its user to manually keep track of whether it has been initialized or not, and almost its entire interface is unsafe, as it can't perform its own checks.

Putting that all together, we start our first attempt with this struct definition:

```
use std::mem::MaybeUninit;

pub struct Channel<T> {
    message: UnsafeCell<MaybeUninit<T>>,
    ready: AtomicBool,
}
```

Just like for our `SpinLock<T>`, we need to tell the compiler that our channel is safe to share between threads, or at least as long as `T` is `Send`:

```
unsafe impl<T> Sync for Channel<T> where T: Send {}
```

A new channel is empty, with `ready` set to `false`, and `message` left uninitialized:

```
impl<T> Channel<T> {
    pub const fn new() -> Self {
        Self {
            message: UnsafeCell::new(MaybeUninit::uninit()),
            ready: AtomicBool::new(false),
        }
    }

    …
}
```

To send a message, it first needs to be stored in the cell, after which we can release it to the receiver by setting the `ready` flag to `true`. Attempting to do this more than once would be dangerous, since after setting the `ready` flag, the receiver might read the message at any point, which could race with a second attempt to send a message. For now, we make this the responsibility of the user by making the method `unsafe` and leaving a note for them:

```
/// Safety: Only call this once!
pub unsafe fn send(&self, message: T) {
    (*self.message.get()).write(message);
    self.ready.store(true, Release);
}
```

In the snippet above, we use the `UnsafeCell::get` method to obtain a pointer to the `MaybeUninit<T>`, and unsafely dereference that to call `MaybeUninit::write` to initialize it. This could result in undefined behavior when misused, but we've punted that responsibility over to the caller.

For the memory ordering, we need to use release ordering, since the atomic store effectively releases the message to the receiver. This makes sure the initialization of the message will be finished from the perspective of the receiving thread if it loads `true` from `self.ready` with acquire ordering.

For receiving, we'll not bother with providing a blocking interface for now. Instead, we'll provide two methods: one to check whether a message is available, and another to receive it. We'll leave it to the user of our channel to use something like thread parking ("Thread Parking" in Chapter 1) if they want to block.

These are the last two methods to complete this version of our channel:

```
pub fn is_ready(&self) -> bool {
    self.ready.load(Acquire)
}

/// Safety: Only call this once,
/// and only after is_ready() returns true!
pub unsafe fn receive(&self) -> T {
    (*self.message.get()).assume_init_read()
}
```

While the `is_ready` method can always be called safely, the `receive` method uses `MaybeUninit::assume_init_read()`, which unsafely assumes it has already been initialized and that it isn't being used to produce multiple copies of non-`Copy` objects. Just like for `send`, we simply make that our user's problem by making the function itself `unsafe`.

The result is a technically usable channel, but one that is unwieldy and generally disappointing. If held right, it does exactly what it is supposed to do, but there are many subtle ways to misuse it.

Calling `send` more than once might result in a data race, since the second sender will be overwriting the data while the receiver might be trying to read the first message. Even if receiving was properly synchronized, calling `send` from multiple threads might result in two threads attempting to concurrently write to the cell, again resulting in a data race. Also, calling `receive` more than once results in two copies of the message, even if `T` does not implement `Copy` and thus cannot safely be copied.

A more subtle issue is the lack of a `Drop` implementation for our channel. The `MaybeUninit` type does not track whether it has been initialized or not, and will therefore not automatically drop its contents when dropped. This means that if a message is sent but never received, the message will never be dropped. This is not unsound, but it's still something to avoid. While leaking is universally considered safe in Rust, it's generally only acceptable as a consequence of another leak. For example, leaking a `Vec` also leaks its contents, but regular usage of a `Vec` does not result in any leaks.

Since we made the user responsible for everything, it's only a matter of time before this results in an unfortunate accident.

Safety Through Runtime Checks

To allow for a safer interface, we can add some checks to make misuse result in a panic with a clear message, which is much preferable to undefined behavior.

Let's start with the issue of calling `receive` before a message is ready. This one is simple to handle, as all we have to do is make the `receive` method validate the `ready` flag before attempting to read a message:

```
/// Panics if no message is available yet.
///
/// Tip: Use `is_ready` to check first.
///
/// Safety: Only call this once!
pub unsafe fn receive(&self) -> T {
    if !self.ready.load(Acquire) {
        panic!("no message available!");
    }
    (*self.message.get()).assume_init_read()
}
```

The function is still unsafe, as the user is still responsible for not calling this function more than once, but failing to check `is_ready()` first no longer results in undefined behavior.

Since we now have an acquire-load of the ready flag inside the `receive` method providing the necessary synchronization, we can lower the memory ordering of the load in `is_ready` to `Relaxed`, since that one is now only used for indicative purposes:

```
pub fn is_ready(&self) -> bool {
    self.ready.load(Relaxed)
```

```
}
```

> *Remember that the total modification order (see "Relaxed Ordering" in Chapter 3) on `ready` guarantees that after `is_ready` loads `true` from it, `receive` will also see `true`. There is no possibility of `is_ready` returning `true` and `receive()` still panicking, regardless of the memory ordering used in `is_ready`.*

The next issue to address is what happens when calling `receive` more than once. We can easily make that result in a panic as well by setting the `ready` flag back to `false` in our `receive` method, like this:

```
/// Panics if no message is available yet,
/// or if the message was already consumed.
///
/// Tip: Use `is_ready` to check first.
pub fn receive(&self) -> T {
    if !self.ready.swap(false, Acquire) {
        panic!("no message available!");
    }
    // Safety: We've just checked (and reset) the ready flag.
    unsafe { (*self.message.get()).assume_init_read() }
}
```

We've simply changed the `load` for a `swap` to `false`, and suddenly the `receive` method is fully safe to call in any condition. The function is no longer marked as `unsafe`. Instead of making the user responsible for everything, we now take responsibility for the unsafe code, resulting in less stress for our user.

For `send`, things are slightly more complicated. To prevent multiple `send` calls from accessing the cell at the same time, we need to know whether another `send` call has already started. The `ready` flag only tells us whether another `send` call has already finished, so that won't suffice.

Let's add a second flag, named `in_use`, to indicate whether the channel has been taken in use:

```
pub struct Channel<T> {
    message: UnsafeCell<MaybeUninit<T>>,
    in_use: AtomicBool, // New!
    ready: AtomicBool,
}

impl<T> Channel<T> {
    pub const fn new() -> Self {
        Self {
            message: UnsafeCell::new(MaybeUninit::uninit()),
            in_use: AtomicBool::new(false), // New!
            ready: AtomicBool::new(false),
        }
    }

    …
}
```

Now all we need to do is set `in_use` to `true` in the `send` method before accessing the cell and panic if it was already set by another call:

```
/// Panics when trying to send more than one message.
pub fn send(&self, message: T) {
    if self.in_use.swap(true, Relaxed) {
        panic!("can't send more than one message!");
    }
    unsafe { (*self.message.get()).write(message) };
    self.ready.store(true, Release);
}
```

We can use relaxed memory ordering for the atomic swap operation, because the *total modification order* (see "Relaxed Ordering" in Chapter 3) of `in_use` guarantees that there will only be a single swap operation on `in_use` that will return `false`, which is the only case in which `send` will attempt to access the cell.

We now have a fully safe interface, though there is still one problem left. The last remaining issue occurs when sending a message that's never received: it will never be dropped. While this does not result in undefined behavior and is allowed in safe code, it's definitely something to avoid.

Since we reset the `ready` flag in the `receive` method, fixing this is easy: the `ready` flag indicates whether there's a not-yet-received message in the cell that needs to be dropped.

In the `Drop` implementation of our `Channel`, we don't need to use an atomic operation to check the atomic `ready` flag, because an object can only be dropped if it is fully

owned by whichever thread is dropping it, with no outstanding borrows. This means we can use the `AtomicBool::get_mut` method, which takes an exclusive reference (`&mut self`), proving that atomic access is unnecessary. The same holds for `UnsafeCell`, through `UnsafeCell::get_mut`.

Using that, here's the final piece of our fully safe and non-leaking channel:

```
impl<T> Drop for Channel<T> {
    fn drop(&mut self) {
        if *self.ready.get_mut() {
            unsafe { self.message.get_mut().assume_init_drop() }
        }
    }
}
```

Let's try it out!

Since our `Channel` doesn't provide a blocking interface (yet), we'll manually use thread parking to wait on a message. The receiving thread will `park()` itself as long as there's no message ready, and the sending thread will `unpark()` the receiver once it has sent something.

Here's a complete test program, sending the string literal `"hello world!"` through our `Channel` from a second thread back to the main thread:

```
fn main() {
    let channel = Channel::new();
    let t = thread::current();
    thread::scope(|s| {
        s.spawn(|| {
            channel.send("hello world!");
            t.unpark();
        });
        while !channel.is_ready() {
            thread::park();
        }
        assert_eq!(channel.receive(), "hello world!");
    });
}
```

This program compiles, runs, and exits cleanly, showing that our `Channel` works as it should.

If we duplicate the `send` line, we can also see one of our safety checks in action, producing the following panic message when the program is run:

```
thread '<unnamed>' panicked at 'can't send more than one message!', src/main.rs
```

While a panicking program isn't great, it's far better for a program to reliably panic than to get anywhere near the potential horrors of undefined behavior.

Using a Single Atomic for the Channel State

In case you can't get enough of implementing channels, here's a subtle variation that can save one byte of memory.

Instead of using two separate atomic booleans to represent the state of the channel, we instead use a single `AtomicU8` to represent all four states. Rather than atomically swapping booleans, we'll have to use `compare_exchange` to atomically check if the channel is in the expected state and change it to another state.

```
const EMPTY: u8 = 0;
const WRITING: u8 = 1;
const READY: u8 = 2;
const READING: u8 = 3;

pub struct Channel<T> {
    message: UnsafeCell<MaybeUninit<T>>,
    state: AtomicU8,
}

unsafe impl<T: Send> Sync for Channel<T> {}

impl<T> Channel<T> {
    pub const fn new() -> Self {
        Self {
            message: UnsafeCell::new(MaybeUninit::uninit()),
            state: AtomicU8::new(EMPTY),
        }
```

```
    }

    pub fn send(&self, message: T) {
        if self.state.compare_exchange(
            EMPTY, WRITING, Relaxed, Relaxed
        ).is_err() {
            panic!("can't send more than one message!");
        }
        unsafe { (*self.message.get()).write(message) };
        self.state.store(READY, Release);
    }

    pub fn is_ready(&self) -> bool {
        self.state.load(Relaxed) == READY
    }

    pub fn receive(&self) -> T {
        if self.state.compare_exchange(
            READY, READING, Acquire, Relaxed
        ).is_err() {
            panic!("no message available!");
        }
        unsafe { (*self.message.get()).assume_init_read() }
    }
}

impl<T> Drop for Channel<T> {
    fn drop(&mut self) {
        if *self.state.get_mut() == READY {
            unsafe { self.message.get_mut().assume_init_drop() }
        }
```

```
        }
    }
```

Safety Through Types

While we've successfully protected users of our `Channel` from undefined behavior, they still risk a panic if they accidentally use it incorrectly. Ideally, the compiler would check correct usage and point out misuse before the program is even run.

Let's take a look at the issue of calling `send` or `receive` more than once.

To prevent a function from being called more than once, we can let it take an argument *by value*, which—for non-`Copy` types—will consume the object. After an object is consumed, or moved, it's gone from the caller, preventing it from being used another time.

By representing the ability to call `send` or `receive` each as a separate (non-`Copy`) type, and consuming that object when performing the operation, we can make sure each can only happen once.

This brings us to the following interface design, where instead of a single `Channel` type, a channel is represented by a pair of a `Sender` and a `Receiver`, which each have a method that takes `self` by value:

```
pub fn channel<T>() -> (Sender<T>, Receiver<T>) { … }

pub struct Sender<T> { … }
pub struct Receiver<T> { … }

impl<T> Sender<T> {
    pub fn send(self, message: T) { … }
}

impl<T> Receiver<T> {
    pub fn is_ready(&self) -> bool { … }
    pub fn receive(self) -> T { … }
}
```

The user can create a channel by calling `channel()`, which will give them one `Sender` and one `Receiver`. They can freely pass each of these objects around, move them to another thread, and so on. However, they cannot end up with multiple copies of either of them, guaranteeing that `send` and `receive` can each only be called once.

To implement this, we need to find a place for our `UnsafeCell` and `AtomicBool`. Previously, we just had a single struct with those fields, but now we have two separate structs, each of which could outlive the other.

Since the sender and receiver will need to share the ownership of those variables, we'll use an `Arc` ("Reference Counting" in Chapter 1) to provide us with a reference-counted shared allocation, in which we store the shared `Channel` object. As shown be-

low, the `Channel` type does not have to be public, as its existence is just an implementation detail irrelevant to the user.

```
pub struct Sender<T> {
    channel: Arc<Channel<T>>,
}

pub struct Receiver<T> {
    channel: Arc<Channel<T>>,
}

struct Channel<T> { // no longer `pub`
    message: UnsafeCell<MaybeUninit<T>>,
    ready: AtomicBool,
}

unsafe impl<T> Sync for Channel<T> where T: Send {}
```

Just like before, we implement `Sync` for `Channel<T>` on the condition that `T` is `Send`, to allow it to be used across threads.

Note how we no longer need the `in_use` atomic boolean like we did in our previous channel implementation. It was only used by `send` to check that it hadn't been called more than once, which is now statically guaranteed through the type system.

The `channel` function to create a channel and sender-receiver pair looks similar to the `Channel::new` function we had previously, except it wraps the `Channel` in an `Arc`, and

wraps that `Arc` and a clone of it in the `Sender` and `Receiver` types:

```
pub fn channel<T>() -> (Sender<T>, Receiver<T>) {
    let a = Arc::new(Channel {
        message: UnsafeCell::new(MaybeUninit::uninit()),
        ready: AtomicBool::new(false),
    });
    (Sender { channel: a.clone() }, Receiver { channel: a })
}
```

The `send`, `is_ready`, and `receive` methods are basically identical to the ones we implemented before, with a few differences:

- They are now moved to their respective type, such that only the (one single) sender can send, and only the (one single) receiver can receive.
- `send` and `receive` now take `self` by value rather than by reference, to make sure they can each only be called once.
- `send` can no longer panic, as its precondition (only being called once) is now statically guaranteed.

So, they now look like this:

```
impl<T> Sender<T> {
    /// This never panics. :)
    pub fn send(self, message: T) {
```

```
        unsafe { (*self.channel.message.get()).write(message) };
        self.channel.ready.store(true, Release);
    }
}

impl<T> Receiver<T> {
    pub fn is_ready(&self) -> bool {
        self.channel.ready.load(Relaxed)
    }

    pub fn receive(self) -> T {
        if !self.channel.ready.swap(false, Acquire) {
            panic!("no message available!");
        }
        unsafe { (*self.channel.message.get()).assume_init_read() }
    }
}
```

The `receive` function can still panic, since the user might still call it before `is_ready()` returns `true`. It also still uses `swap` to set the `ready` flag back to `false` (instead of just `load`), so that the `Drop` implementation of `Channel` knows whether there's an unread message that needs to be dropped.

That `Drop` implementation is exactly the same as the one we implemented before:

```
impl<T> Drop for Channel<T> {
    fn drop(&mut self) {
        if *self.ready.get_mut() {
            unsafe { self.message.get_mut().assume_init_drop() }
```

```
        }
    }
}
```

The `Drop` implementation of `Arc<Channel<T>>` will decrement the reference counter of the allocation when either the `Sender<T>` or `Receiver<T>` is dropped. When dropping the second one, that counter reaches zero, and the `Channel<T>` itself is dropped. That will invoke our `Drop` implementation above, where we get to drop the message if one was sent but not received.

Let's try it out:

```
fn main() {
    thread::scope(|s| {
        let (sender, receiver) = channel();
        let t = thread::current();
        s.spawn(move || {
            sender.send("hello world!");
            t.unpark();
        });
        while !receiver.is_ready() {
            thread::park();
        }
        assert_eq!(receiver.receive(), "hello world!");
    });
}
```

It's a bit inconvenient that we still have to manually use thread parking to wait on a message, but we'll deal with that problem later.

Our goal, for now, was to make at least one form of misuse impossible at compile time. Unlike last time, attempting to send twice does not result in a program that panics, but instead does not result in a valid program at all. If we add another `send` call to the working program above, the compiler now catches the issue and patiently informs us of our mistake:

```
error[E0382]: use of moved value: `sender`
  --> src/main.rs
   |
   |             sender.send("hello world!");
   |                    --------------------
   |                     `sender` moved due to this method call
   |
   |             sender.send("second message");
   |             ^^^^^^ value used here after move
   |
note: this function takes ownership of the receiver `self`, which moves `sender`
  --> src/lib.rs
   |
   |     pub fn send(self, message: T) {
   |                 ^^^^
   = note: move occurs because `sender` has type `Sender<&str>`,
           which does not implement the `Copy` trait
```

Depending on the situation, it can be extremely tricky to design an interface that catches mistakes at compile time. If the situation does lend itself to such an interface, it can result not only in more convenience for the user, but also in a reduced number of runtime checks for things that are now statically guaranteed. We no longer needed the `in_use` flag, and removed the `swap` and check from the `send` method, for example.

Unfortunately, new problems may arise that could lead to more runtime overhead. In this case, the problem was the split ownership, for which we had to reach for an `Arc` and pay the cost of an allocation.

Having to make trade-offs between safety, convenience, flexibility, simplicity, and performance is unfortunate, but sometimes unavoidable. Rust generally strives to make it easy to excel at all of these, but sometimes makes you trade a bit of one to maximize another.

Borrowing to Avoid Allocation

The `Arc`-based channel implementation we just designed is very convenient to use—at the cost of some performance, since it has to allocate memory. If we want to optimize for efficiency, we can trade some convenience for performance by making the user responsible for the shared `Channel` object. Instead of taking care of the allocation and ownership of the `Channel` behind the scenes, we can force the user to create a `Channel` that can be *borrowed* by the `Sender` and `Receiver`. That way, they can choose

to simply put that `Channel` in a local variable, avoiding the overhead of allocating memory.

We will also have to trade in some simplicity, since we will now have to deal with borrowing and lifetimes.

So, the three types will now look as follows, with `Channel` public again, and `Sender` and `Receiver` borrowing it for a certain lifetime.

```
pub struct Channel<T> {
    message: UnsafeCell<MaybeUninit<T>>,
    ready: AtomicBool,
}

unsafe impl<T> Sync for Channel<T> where T: Send {}

pub struct Sender<'a, T> {
    channel: &'a Channel<T>,
}

pub struct Receiver<'a, T> {
    channel: &'a Channel<T>,
}
```

Instead of a `channel()` function to create a `(Sender, Receiver)` pair, we move back to the `Channel::new` we had earlier in this chapter, allowing the user to create such an object as a local variable.

In addition, we need a way for the user to create a `Sender` and `Receiver` object that will borrow the `Channel`. This will need to be an exclusive borrow (`&mut Channel`), to make sure there can't be multiple senders or receivers for the same channel. By providing both the `Sender` and the `Receiver` at the same time, we can *split* the exclusive borrow into two shared borrows, such that both the sender and receiver can reference the channel, while preventing anything else from touching the channel.

This leads us to the following implementation:

```
impl<T> Channel<T> {
    pub const fn new() -> Self {
        Self {
            message: UnsafeCell::new(MaybeUninit::uninit()),
            ready: AtomicBool::new(false),
        }
    }

    pub fn split<'a>(&'a mut self) -> (Sender<'a, T>, Receiver<'a, T>) {
        *self = Self::new();
        (Sender { channel: self }, Receiver { channel: self })
    }
}
```

The `split` method, with its somewhat complicated signature, warrants a closer look. It exclusively borrows `self` through an exclusive reference, but it splits that into two shared references, wrapped in the `Sender` and `Receiver` types. The `'a` lifetime makes

it clear that both of those objects borrow something with a limited lifetime; in this case, the `Channel` itself. Since the `Channel` is exclusively borrowed, the caller will not be able to borrow or move it as long as the `Sender` or `Receiver` object exists.

Once those objects both cease to exist, however, the mutable borrow expires and the compiler happily lets the `Channel` object be borrowed again by a second call to `split()`. While we can assume `split()` cannot be called again while the `Sender` and `Receiver` still exist, we cannot prevent a second call to `split()` after those objects are dropped or forgotten. We need to make sure we don't accidentally create a new `Sender` or `Receiver` object for a channel that already has its `ready` flag set, since that would break the assumptions that prevent undefined behavior.

By overwriting `*self` with a new empty channel in `split()`, we make sure it's in the expected state when creating the `Sender` and `Receiver` states. This also invokes the `Drop` implementation on the old `*self`, which will take care of dropping a message that was previously sent but not received.

> *Since the lifetime in the signature of `split` comes from `self`, it can be elided. The signature of `split` in the snippet above is identical to this less verbose version:*
>
> ```
> pub fn split(&mut self) -> (Sender<T>, Receiver<T>) { … }
> ```
>
> *While this version doesn't explicitly show that the returned objects borrow `self`, the compiler still checks correct usage of the lifetime exactly the same as it does with*

> *the more verbose version.*

The remaining methods and `Drop` implementation are the same as in our `Arc`-based implementation, except for an additional `'_` lifetime argument to the `Sender` and `Receiver` types. (If you forget those, the compiler will helpfully suggest adding them.)

For completeness, here's the remaining code:

```
impl<T> Sender<'_, T> {
    pub fn send(self, message: T) {
        unsafe { (*self.channel.message.get()).write(message) };
        self.channel.ready.store(true, Release);
    }
}

impl<T> Receiver<'_, T> {
    pub fn is_ready(&self) -> bool {
        self.channel.ready.load(Relaxed)
    }

    pub fn receive(self) -> T {
        if !self.channel.ready.swap(false, Acquire) {
            panic!("no message available!");
        }
        unsafe { (*self.channel.message.get()).assume_init_read() }
    }
}

impl<T> Drop for Channel<T> {
    fn drop(&mut self) {
```

```
            if *self.ready.get_mut() {
                unsafe { self.message.get_mut().assume_init_drop() }
            }
        }
    }
```

Let's test it!

```
    fn main() {
        let mut channel = Channel::new();
        thread::scope(|s| {
            let (sender, receiver) = channel.split();
            let t = thread::current();
            s.spawn(move || {
                sender.send("hello world!");
                t.unpark();
            });
            while !receiver.is_ready() {
                thread::park();
            }
            assert_eq!(receiver.receive(), "hello world!");
        });
    }
```

The reduction in convenience compared to the `Arc`-based version is quite minimal: we only needed one more line to manually create a `Channel` object. Note, however, how the channel has to be created before the scope, to prove to the compiler that its existence will outlast both the sender and receiver.

To see the compiler's borrow checker in action, try adding a second call to `channel.split()` in various places. You'll see that calling it a second time within the thread scope results in an error, while calling it after the scope is acceptable. Even calling `split()` before the scope is fine, as long as you stop using the returned `Sender` and `Receiver` before the scope starts.

Blocking

Let's finally deal with the last remaining major inconvenience of our `Channel`, the lack of a blocking interface. We've already used thread parking every time we tested a new variant of our channel. It shouldn't be too hard to integrate that pattern into the channel itself.

To be able to unpark the receiver, the sender needs to know which thread to unpark. The `std::thread::Thread` type represents a handle to a thread and is exactly what we need for calling `unpark()`. We'll store the handle to the receiving thread inside the `Sender` object, as follows:

```
use std::thread::Thread;

pub struct Sender<'a, T> {
    channel: &'a Channel<T>,
    receiving_thread: Thread, // New!
}
```

This handle would refer to the wrong thread, however, if the `Receiver` object is sent between threads. The `Sender` would be unaware of that and would still refer to the thread that originally held the `Receiver`.

We can handle that problem by making the `Receiver` a bit more restrictive, by not allowing it to be sent between threads anymore. As discussed in "Thread Safety: Send and Sync" in Chapter 1, we can use the special `PhantomData` marker type to add this restriction to our struct. A `PhantomData<*const ()>` does the job, since a raw pointer, such as `*const ()`, does not implement `Send`:

```
pub struct Receiver<'a, T> {
    channel: &'a Channel<T>,
    _no_send: PhantomData<*const ()>, // New!
}
```

Next, we'll have to modify the `Channel::split` method to fill in the new fields, like this:

```
    pub fn split<'a>(&'a mut self) -> (Sender<'a, T>, Receiver<'a, T>) {
        *self = Self::new();
        (
            Sender {
                channel: self,
                receiving_thread: thread::current(), // New!
            },
            Receiver {
```

```
                channel: self,
                _no_send: PhantomData, // New!
            }
        )
    }
```

We use the handle to the current thread for the `receiving_thread` field, since the `Receiver` object we return will stay on the current thread.

The `send` method doesn't change much, as shown below. We only have to call `unpark()` on the `receiving_thread` to wake up the receiver in case it is waiting:

```
impl<T> Sender<'_, T> {
    pub fn send(self, message: T) {
        unsafe { (*self.channel.message.get()).write(message) };
        self.channel.ready.store(true, Release);
        self.receiving_thread.unpark(); // New!
    }
}
```

The `receive` function undergoes a slightly larger change. The new version won't panic if there's no message yet, but will instead patiently wait for a message using `thread::park()` and try again, as many times as necessary.

```
impl<T> Receiver<'_, T> {
    pub fn receive(self) -> T {
        while !self.channel.ready.swap(false, Acquire) {
```

```
            thread::park();
        }
        unsafe { (*self.channel.message.get()).assume_init_read() }
    }
}
```

> *Remember that `thread::park()` might return spuriously. (Or because something other than our `send` method called `unpark()`.) This means that we cannot assume that the `ready` flag has been set when `park()` returns. So, we need to use a loop to check the flag again after getting unparked.*

The `Channel<T>` struct, its `Sync` implementation, its `new` function, and its `Drop` implementation remain unchanged.

Let's try it out!

```
fn main() {
    let mut channel = Channel::new();
    thread::scope(|s| {
        let (sender, receiver) = channel.split();
        s.spawn(move || {
            sender.send("hello world!");
        });
        assert_eq!(receiver.receive(), "hello world!");
    });
}
```

Clearly, this `Channel` is more convenient to use than the last one, at least in this simple test program. We've had to pay for this convenience by trading in some flexibility: only the thread that calls `split()` may call `receive()`. If you swap the `send` and `receive` lines, this program will no longer compile. Depending on the use case, that might be entirely fine, useful, or very inconvenient.

There are a number of ways to address that issue, many of which will cost us some additional complexity and affect performance. In general, the number of variations and trade-offs we can continue to explore are virtually endless.

We could easily spend an unhealthy number of hours implementing another twenty variants of a one-shot channel, each with slightly different properties, for every imaginable use case and more. While that might sound like lots of fun, we should probably avoid that rabbit hole and end this chapter before things get out of hand.

~

Summary

- A *channel* is used to send *messages* between threads.

- A simple and flexible, but potentially inefficient, channel is relatively easy to implement with just a `Mutex` and a `Condvar`.
- A *one-shot channel* is a channel designed to send only one message.
- The `MaybeUninit<T>` type can be used to represent a potentially not-yet-initialized `T`. Its interface is mostly unsafe, making its user responsible for tracking whether it has been initialized, not duplicating `Copy` data, and dropping its contents if necessary.
- Not dropping objects (also called *leaking* or *forgetting*) is safe, but frowned upon when done without good reason.
- Panicking is an important tool for creating a safe interface.
- Taking a non-`Copy` object by value can be used to prevent something to be done more than once.
- Exclusively borrowing and splitting borrows can be a powerful tool for forcing correctness.
- We can make sure an object stays on the same thread by making sure its type does not implement `Send`, which can be achieved with the `PhantomData` marker type.
- Every design and implementation decision involves a trade-off and can best be made with a specific use case in mind.

- Designing something without a use case can be fun and educational, but can turn out to be an endless task.

Next: Chapter 6. Building Our Own "Arc"