Send and Sync

Send

Types that can be transferred across thread boundaries

What types are `Send`

Well that is kind of a wrong question here

What types are `!Send`

Most of `!Send` are weird and not really relevant.
There are notable mentionts though:

- Raw pointers
- `Rc` !!!

`Rc` ??? how could this be

If both threads clone `Rc` at the same time that could cause a race condition, because `Rc` doesn't use atomic operations. We just use `Arc` instead.

Sync

Types for which it is safe to share references between threads

`T` is `Sync` if and only if `&T` is `Send`

Quite funny because `&mut T` is `Sync`

So which types are `!Sync`

Interior mutability family: `RefCell` and `Cell` and others

Also `Rc` but yout already knew that

Deconstruct cells

`T` is `Sync` if and only if `&T` is `Send`

`Cell<T>` is `Send` though, so why `&Cell<T>` is not???

Well, `Cell<T>` is just `!Sync`

Iff is not implication ://

How to work with `Send` and `Sync`

`Send` is used to send:

- channels (`std::sync::mpsc`)
- `async move` blocks
- `async` functions
- `Mutex` and derivatives

What about referencing data in threads (`Sync`)

Any `&` we pass must be `'static`. This can be achieved by `Box::leak` and probably some other smart ways

```
let x = 10;
let ref_x: &'static u64 = Box::leak(Box::new(x));
let _res = tokio::join! {
    tokio::spawn(show_x(ref_x)),
};

async fn show_x(x: &u64) {
    println!("{x}");
}
```

We don't really want to do that though - we have no way of deallocating that data in a clean way, we would have to call `drop` manually which could cause other issues.

`Arc` and `Box` (?) to the rescue

Box

`Box` is great for moving but it is only single ownership. Trying to do `Sync` operations is a pain in the ass as before. Turns out its use is the same as in sync context. Should've seen that coming.

`Arc`

The real big boy - coming is clutch as a `&` replacement.

So, what is the difference?

```rust
let x = 10;
let box_x = Box::new(x);
let arc_x = Arc::new(x);

let _res = tokio::join! {
    tokio::spawn(show_x_box(box_x.clone())),
    tokio::spawn(show_x_box(box_x)),
    tokio::spawn(show_x_arc(arc_x.clone())),
    tokio::spawn(show_x_arc(arc_x)),
};

async fn show_x_box<T: std::fmt::Debug>(x: Box<T>) {
    println!("{x:?}");
}

async fn show_x_arc<T: std::fmt::Debug>(x: Arc<T>) {
    println!("{x:?}");
}
```

```rust
struct Bonk {}
let box_x = Box::new(Bonk{});
let arc_x = Arc::new(Bonk{});

let _res = tokio::join! {
    tokio::spawn(show_x_box(box_x.clone())),
    tokio::spawn(show_x_box(box_x)),
    tokio::spawn(show_x_arc(arc_x.clone())),
    tokio::spawn(show_x_arc(arc_x)),
};
```

The pain point

"Shared references in Rust disallow mutation by default, and `Arc` is no exception: you cannot generally obtain a mutable reference to something inside an `Arc`"

We are doomed

But what if we use `Box` or `RefCell` with `Arc`?

`Box` allows immutable or mutable borrows checked at compile time

`RefCell` allows immutable or mutable borrows checked at runtime

Sounds perfect right?

After all, `Rc<RefCell<T>>` rings a bell

WRONG

`RefCell<T>: !Sync`, turns out `Arc` wants `Sync` (and `Send`)!

I kind of get `Sync`, we need to read the same value in many places, but why `Send` ???

Assume we have threads `T1` and `T2`. We have some `let x = Arc::new(whatever)` in `T1` and we clone it to `T2`. Now `T1` finishes and drops its `Arc` instance. `T2` is now responsible for dropping the data!

But we drop reference rather than the data that is on the heap anyway?

This data doesn't have to be on the heap though. Turns out `Arc` will move the underlying data `T` so that it will always be in some working thread!

`Box` ?

We can get to the thing inside `Arc<T>` by calling `get_mut`, but that returns an `Option<&mut T>`.

But wait, then we don't really need `Box` at all, we can just do whatever.

`Arc` lied

We CAN get a mutable reference! It is just not very convenient and we need to handle the case when we don't.

`Arc` didn't lie

If there is any other thread that has a clone of that `Arc` we get `None` - it is basically useless

Can we finally get to the `Mutex` pls

yea ok

Turns out, the sane way of doing shared mutability is by using `Arc<Mutex<T>>` or `Arc<RwLock<T>>`.

Rw what now

`RwLock` is a read-write lock which can serve mutliple readers at the same time.

Why `Mutex` ever?

It only requires `Send`, where `RwLock` requires both `Send` and `Sync`. Note that `Mutex<T>` is `Send` and `Sync` even if `T` is `!Sync`.

Tbh only relevant with `Cell` and derivatives.

Sad takeaway is that we use one `RwLock` in our **entire** codebase and I actually ~~copied it with no understanding from some forum~~ wrote it!

Next time: `Cells`