

Async Rust

A short overview of threading, channels, and async.

Not an expert, just a rough look at how things are.

Good resources:

- Async Book
- The State of Async Rust: Runtimes
- Tokio docs - tutorial: Async in depth
 - Building a mini tokio

Terminology

- Serial, Sequential: Finish **A**, then finish **B**, then finish **C**
- Concurrent: Do **A**, **B**, and **C** alternately
- Parallel: Do **A**, **B**, and **C** simultaneously
- Synchronous, Blocking: **B** must wait for **A** to finish
(**A** = blocking, inherently serial)
- Asynchronous, Non-blocking: **B** can continue while **A** executes
(**A** = non-blocking, potential for parallelism)

Threads	Async
Allows using multiple cores to run stuff that has potential for parallelism	Allow things to run in the background while doing other stuff
Requires an OS that supports threads, and a threading implementation, such as <code>std::thread</code>	Requires <code>Futures</code> implementation such as <code>std::future</code> , and requires an <i>async runtime</i> (i.e. <i>executor</i>) to run <code>async</code> blocks
OS managed	Runtime managed
Context switches when switching between threads (~2 μ s)	Runtime switching between tasks (~500 ns).

Threads

(std)

```
use std::thread;
use std::time::Duration;

fn main() {
    thread::spawn(|| {
        for i in 1..3 {
            println!("hi number {i} from the spawned thread!");
            thread::sleep(Duration::from_millis(1));
        }
    });
    for i in 1..5 {
        println!("hi number {i} from the main thread!");
        thread::sleep(Duration::from_millis(1));
    }
}

// hi number 1 from the main thread!
// hi number 1 from the spawned thread!
// hi number 2 from the main thread!
// ...
```

Go - goroutines and channels

"Don't communicate by sharing memory; share memory by communicating"
(R.Pike)

```
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel:::<std::string::String>();
    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
    });

    loop {
        let received_nb = rx.try_recv();
        match received_nb {
            Ok(val) => {
                println!("Got: {val}");
                break;
            }
            Err(_) => println!("Not received yet, doing other work..."),
        }
    }
}

// Not received yet, doing other work...
// Not received yet, doing other work...
// Got: hi
```

```
// std::thread::spawn

pub fn spawn<F, T>(f: F) -> JoinHandle<T>
where
    F: FnOnce() -> T,
    F: Send + 'static,
    T: Send + 'static,
```

Send

Any type that implements `Send` can transfer ownership between threads safely.

e.g. `Rc<T>` does not implement `Send`, but `Arc<T>` does.

Sync

Any type that implements `Sync` can be referenced from multiple threads safely.

e.g. `RefCell<T>` and `Cell<T>` do not implement `Sync`, but `Mutex<T>` does.

Relationship between Send and Sync

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

-- also --

Types composed entirely of other types that implement the `Send` and `Sync` traits also automatically implement `Send` and `Sync`.

'static as trait bound

Mandates that the type does not contain any non-static references. This means the receiver can hold on to the type indefinitely without it becoming invalid until they decide to drop it.

Async

Generally

The idea of `async/await` and `Task/Promise/Future` semantics in programming languages is to have better control over asynchronous operations while keeping code simple.

- Do `x` while waiting for `y`
- Wait for all/any of `n` things to finish
- Code is significantly easier to read
- Error handling becomes a breeze

In Rust

```
async fn foo() { ... }
```

→

```
fn foo() { async { ... } }
```

`foo()` returns an anonymously typed object that implements `Future`

A `Future` will not run until it is `.await` ed.

```
foo().await
```

```
pub trait Future {  
    type Output;  
  
    // Required method  
    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;  
}
```

Calling `.await` on a `Future` schedules it for the async executor, which then calls `poll()` whenever there are resources available.

If the `Future` is finished, it will return `Poll::Ready(val)`.

If the `Future` is not finished, it will return `Poll::Pending` AND later call `.wake()` from `cx`, which indicates that this function is ready to be `poll()` ed again by the scheduler.

Async Executors

Embassy

Is an embedded systems framework, which has a built-in async executor.

```

#[embassy::task]
async fn button_waiter(
    mut button: ExtiInput<'static, PC13>,
    button_pressed: &'static AtomicBool,
    mut button_processed: Output<'static, PG1>,
) {
    let mut trigger_count = 0;

    loop {
        button_processed.set_low().unwrap();
        button.wait_for_rising_edge().await;
        button_processed.set_high().unwrap();
        trigger_count += 1;

        ...
    }
}

```

smol

Wrapper for `std` function implementations with `async` and `await` semantics.

```
use smol::{io, net, prelude::*, Unblock};

fn main() -> io::Result<()> {
    smol::block_on(async {
        let mut stream = net::TcpStream::connect("example.com:80").await?;
        let req = b"GET / HTTP/1.1\r\nHost: example.com\r\nConnection: close\r\n\r\n";
        stream.write_all(req).await?;

        let mut stdout = Unblock::new(std::io::stdout());
        io::copy(stream, &mut stdout).await?;
        Ok(())
    })
}
```

Tokio

The most widely used async runtime in the Rust ecosystem.

```

use tokio::io::{AsyncReadExt, AsyncWriteExt};
use tokio::net::TcpListener;

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    let listener = TcpListener::bind("127.0.0.1:8080").await?;
    loop {
        let (mut socket, _) = listener.accept().await?;
        tokio::spawn(async move {
            let mut buf = [0; 1024];
            loop {
                let n = match socket.read(&mut buf).await {
                    Ok(0) => return, // socket closed, return from loop
                    Ok(n) => n,
                    Err(e) => {
                        eprintln!("failed to read from socket; err = {:?}", e);
                        return;
                    }
                };
                // write the data back
                if let Err(e) = socket.write_all(&buf[0..n]).await {
                    eprintln!("failed to write to socket; err = {:?}", e);
                    return;
                }
            }
        });
    }
}

```

```
// tokio::task::spawn

pub fn spawn<F>(future: F) -> JoinHandle<F::Output>
where
    F: Future + Send + 'static,
    F::Output: Send + 'static,
```

Multi-threaded by default

Tokio is a multi-threaded work-stealing approach to async.

- Async tasks are executed on all cores of the system across multiple threads.
- Whenever `Futures` are polled, if the workload between threads is not in balance, idle threads might 'steal' the task from another.
- In between `poll()` calls, the `Future` might need to switch to another thread, which is why the `Send` trait bound is needed to guarantee safety.

'static and async

Tasks in async runtimes may outlive the scope they were created in, so they often have `'static` as a trait bound.

- > Borrowed data must live as long as the task
- > In synchronous Rust, borrowing data across function calls is common, but in async Rust it is difficult
- > Async Rust is fundamentally different

Tokio can be configured to be single-threaded only, but `tokio::spawn` requires `Futures` to implement `Send`. This can be avoided with `LocalSet` and `spawn_local` but they are not the main focus in Tokio.

"The Original Sin of Rust async programming is making it multi-threaded by default. If premature optimization is the root of all evil, this is the mother of all premature optimizations, and it curses all your code with the unholy `Send + 'static`, or worse yet `Send + Sync + 'static`, which just kills all the joy of actually writing Rust."

Maciej Hirszt

The effects

- You get great performance OotB
- You have more complexity
- Conflict -- sometimes you might not want to work with threads and the trait bounds associated with all that, but Tokio mandates it

My takes

- Async Rust is still maturing
- Compatibility issues between sync and async code, and between async runtimes
 - `async_compat` crate might help
- `(Sync +) Send + 'static` might increase code complexity
- Advanced language features like `Pin`

-> Using async Rust likely increases maintenance burden

Thank you! Questions?

Extras

FnOnce

```
fn consume<F>(func: F)
  where F: FnOnce() -> String
{
  println!("Consumed: {}", func());
  // Invoking `func()` again -> compile error
}

let x = String::from("x");
let move_and_return_x_closure = move || x;
consume(move_and_return_x_closure);
// Invoking `move_and_return_x_closure` again -> compile error
// Using `x` again -> compile error
```

Rayon

```
use rayon::prelude::*;
fn sum_of_squares(input: &[i32]) -> i32 {
    input.par_iter() // <-- just change that!
        .map(|&i| i * i)
        .sum()
}
```