

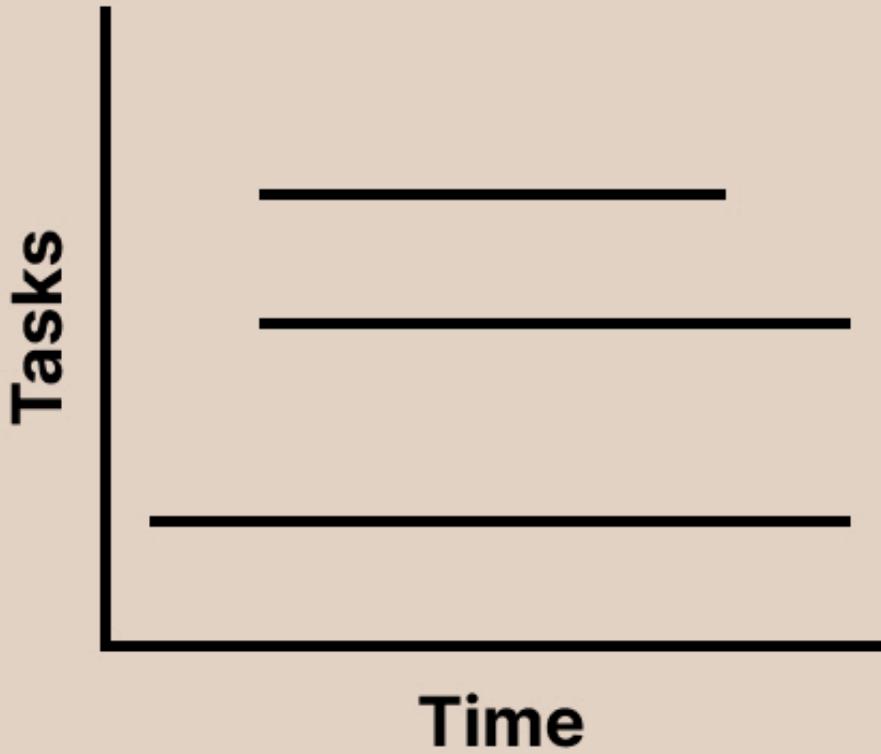


Intro to Rust Lang Concurrency

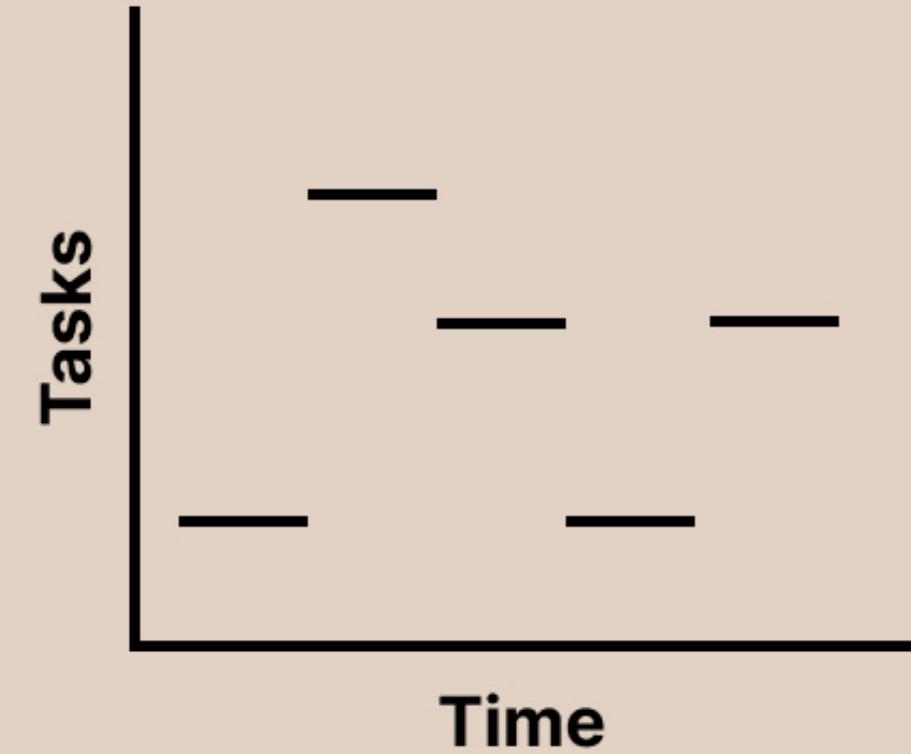
Recap: Parallelism vs Concurrency

- **Concurrency**
 - A **problem** of handling many tasks at once
- **Parallelism**
 - A **solution** to working on multiple tasks at the same time
- Today we're talking about how Rust handles concurrency with `async`

Parallelism



Concurrency



Concurrency

Today, we'll be talking about asynchronous programming in Rust.

- Asynchronous programming is a *solution* to concurrency
- When we say something is **asynchronous**, we generally also mean it is **concurrent**
- Rust approaches asynchronicity differently than other languages
- Rust provides the keywords `async` and `.await` to help write and reason about asynchronous code

Concurrency is Complicated

- Asynchronous execution in *any* language is complicated
- Asynchronicity and parallelism are not mutually exclusive
 - This can make reasoning about concurrency **even harder!**
 - *You can have concurrency and parallelism at the same time in Rust, and we'll see an example of this soon*

Rust's Concurrency is Even More Complicated!

Due to the high complexity of Rust's rules and features, `async` is *even harder* to use in Rust compared to other languages.

- Asynchronous execution is still evolving both as a feature in Rust and as a programming paradigm in general
- The Rust team has **prioritized** bringing the Async Rust experience closer to parity with synchronous Rust
 - *Lots of exciting work incoming soon!*

Today

- It would be quite challenging to teach you what you need to know about `async` / `.await` in Rust in one lecture
- Instead, we will use `tokio` as a practical medium to learn how to use `async`
- There are **many** online resources dedicated to Async Rust (*see speaker note*)
- We will give you a sneak peek of `async`, and hopefully in the future you will be able to teach yourself how to use it!

Tokio

What is Tokio?

What is Tokio?

Tokio is an asynchronous runtime for the Rust programming language. It provides the building blocks needed for writing network applications. It gives the flexibility to target a wide range of systems, from large servers with dozens of cores to small embedded devices.

- <https://tokio.rs/>

What is Tokio?

At a high level, Tokio provides a few major components:

- A **multi-threaded runtime** for **executing asynchronous code**
- An asynchronous version of the standard library
- A large ecosystem of libraries

Why do we need Tokio?

- Making your program asynchronous allows it to scale much better
 - Reduces the cost of doing many things at the same time
- However, asynchronous Rust code does not run on its own
 - You must choose an *asynchronous runtime* to execute it
- The Tokio library is the most widely used runtime

Tokio and the Rust Ecosystem

Tokio is arguably one of the most important libraries in the Rust ecosystem.

- Many *major* Rust libraries built on top of Tokio
- There are large companies relying on Tokio in production

When not to use Tokio

Tokio is useful for many projects, but there are some cases where this isn't true.

- Tokio is designed for IO-bound applications, not CPU-bound
- It is important to note that Tokio is **NOT** the only asynchronous runtime
- We'll come back to this point later

Mini-Redis

We are going to create a miniature [Redis](#) client and server.

- We'll start with the basics of asynchronous programming in Rust
- Implement a subset of Redis commands (`GET` and `SET` key-value pairs)
- If you want to do the actual tutorial, follow along [here](#)

Starter Code

Set up a new crate called `my-redis` and add some dependencies.

```
$ cargo new my-redis  
$ cd my-redis
```

`Cargo.toml`

```
[dependencies]  
tokio = { version = "1", features = ["full"] }  
mini-redis = "0.4"
```

Starter Code

src/main.rs

```
use mini_redis::{client, Result};

#[tokio::main]
async fn main() -> Result<()> {
    // Open a connection to the mini-redis address.
    let mut client = client::connect("127.0.0.1:6379").await?;

    // Set the key "hello" with value "world".
    client.set("hello", "world".into()).await?;

    // Get key "hello".
    let result = client.get("hello").await?;

    println!("got value from the server; result={:?}", result);
    Ok(())
}
```



Running our Client

Start the server:

```
$ cargo install mini-redis  
  
# Run the mini server  
$ mini-redis-server
```

Run the client:

```
$ cargo run  
got value from the server; result=Some(b"world")
```

Lecture Done!

Thanks for coming!

Slides created by:

Connor Tsui, Benjamin Owad, David Rudo,
Jessica Ruan, Fiona Fisher, Terrance Chen



Breaking it down

Okay, let's actually break this down. There isn't much code, but a lot is happening.

```
use mini_redis::{client, Result};

#[tokio::main]
async fn main() -> Result<()> {
    let mut client = client::connect("127.0.0.1:6379").await?;

    client.set("hello", "world".into()).await?;

    let result = client.get("hello").await?;

    println!("got value from the server; result={:?}", result);

    Ok(())
}
```



What's the Difference?

What's going on here?

```
let mut client = client::connect("127.0.0.1:6379").await?;
```

- The `client::connect` function is provided by the `mini-redis` crate
- Establishes a TCP connection with the remote address and returns a handle
- The only indication that this is asynchronous is the `.await` operator



What is Synchronous Programming?

- Most programs are executed in the same order in which they are written
 - *Execute first line of code, then second line, etc.*
- We call this **synchronous** programming
- When a program encounters an operation that cannot be completed immediately, it will **block** until the operation completes
 - For example, establishing a TCP connection over the network

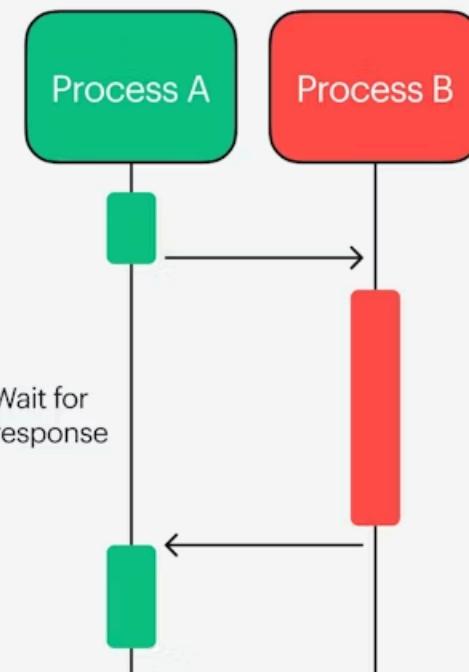
What is Asynchronous Programming?

- In asynchronous programming, operations that cannot complete immediately are *suspended*
- The thread executing is **not blocked** and can instead run other things
- When the operation completes, it becomes *unsuspended* and the thread continues processing it where it left off
- *Note that something will need to "remember" the task's state after pausing so it can resume later*

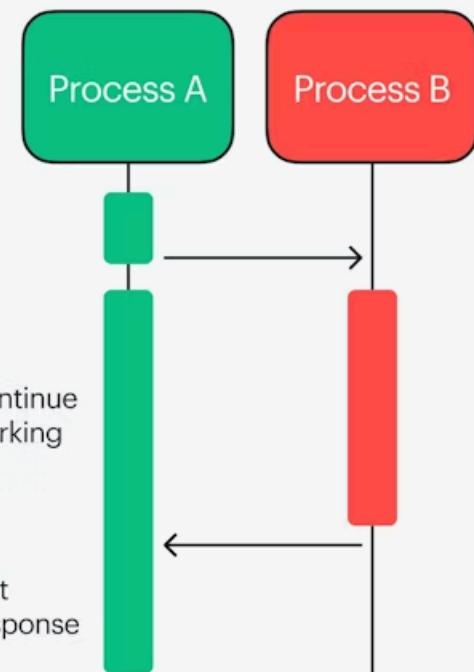
Sync vs Async

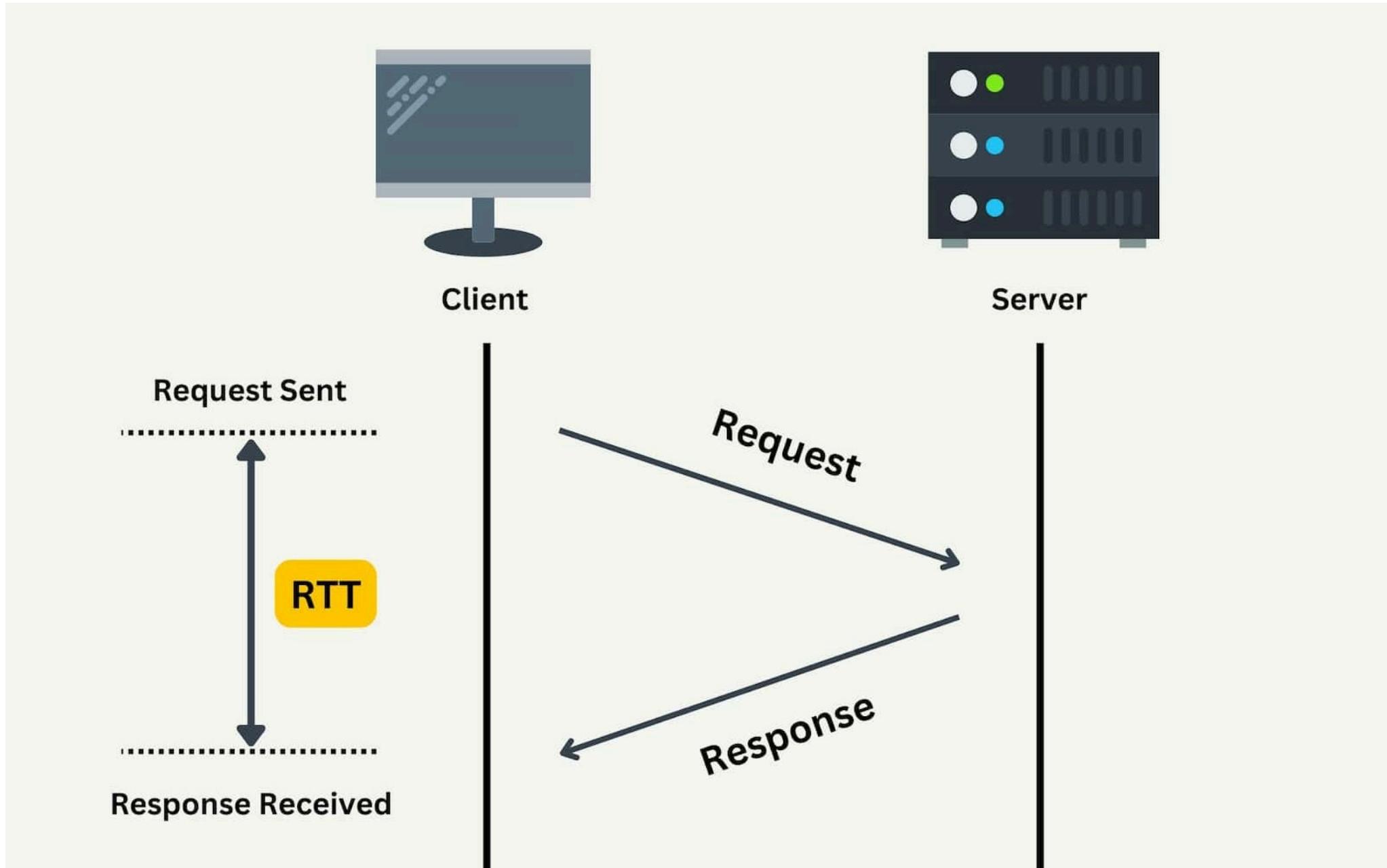
- In synchronous programming, we **wait/block** on the response
- In asynchronous programming, we can do other stuff *while* we are waiting

Synchronous



Asynchronous





Asynchronous Programming in Practice

- In our example, we only have one task (a single `client::connect`)
- Usually, asynchronous programs have *many* tasks running at the same time
- This results in much more complicated programs
 - Track all of the state necessary to suspend/resume work

async

In Rust, functions that perform asynchronous operations are labeled with the `async` keyword.

```
use mini_redis::Result;
use mini_redis::client::Client;
use tokio::net::ToSocketAddrs;

// vvvvv
pub async fn connect<T: ToSocketAddrs>(addr: T) -> Result<Client> {
    // <-- snip -->
}
```

async

```
// vvvvvv
pub async fn connect<T: ToSocketAddrs>(addr: T) -> Result<Client> { ... }
```

- The `async fn` definition looks like a regular synchronous function!
- The compiler transforms the `async fn` at **compile-time** into a routine
 - You can think of this routine as a low-level state machine



.await

```
//                                     vvvvvv
let mut client = client::connect("127.0.0.1:6379").await?;
```

- When we call `.await` on an `async fn`, we *yield* control back to the thread
- The executing thread is then allowed to go work on other tasks as the current `connect` task is processed in the background
- Once the connection has been established in `connect`, the thread can resume working on this task

Calling `async` Functions

Async functions are similar to normal Rust functions. However, calling these functions does not result in the function body executing.

```
async fn say_world() { // An asynchronous function that prints "world".  
    println!("world");  
}  
  
#[tokio::main]  
async fn main() {  
    let op = say_world();  
  
    println!("hello");  
  
    op.await;  
}
```



Calling `async` Functions

```
async fn say_world() { println!("world"); }

#[tokio::main]
async fn main() {
    // Calling `say_world()` does not execute the body of `say_world()`.
    let op = say_world();

    println!("hello"); // This `println!` comes first.

    op.await; // Calling `await` on `op` starts executing `say_world`.
}
```

```
hello
world
```

async is Lazy

- Other languages implement `async/await` (notably JavaScript and C#)
- However, Rust async operators are **lazy**
 - Async code will not run without being `.await`ed
- This results in dramatically different runtime semantics

Asynchronous `main` Function

You may have noticed something different about our `main` function.

```
#[tokio::main]
async fn main() { ... }
```

- It is labeled `async fn`
- It is annotated with `#[tokio::main]`

Asynchronous Runtimes

Asynchronous functions (`async fn`) must be **executed by a runtime**.

- A runtime provides components needed for running asynchronous programs
 - Task scheduler
 - I/O event handlers
 - Timers
- A runtime must be started by an actual `fn main` function

#[tokio::main]

The `#[tokio::main]` function is a macro that transforms the `async fn main()` into a synchronous `fn main()`.

```
#[tokio::main]
async fn main() {
    println!("hello");
}

/// The above gets transformed into:
fn main() {
    let mut rt = tokio::runtime::Runtime::new().unwrap();
    rt.block_on(async {
        println!("hello");
    })
}
```



Back to the Server

Let's go back to implementing the Redis server. We want to:

- Accept inbound TCP sockets in a loop
- Process each socket
 - Read the command
 - Print to `stdout`
 - For now, respond with an "unimplemented" error

Accept TCP Connections

```
use tokio::net::{TcpListener, TcpStream};
use mini_redis::{Connection, Frame};

#[tokio::main]
async fn main() {
    // Bind the listener to the address.
    let listener = TcpListener::bind("127.0.0.1:6379").await.unwrap();

    loop {
        // The second item contains the IP and port of the new connection.
        let (socket, _) = listener.accept().await.unwrap();
        process(socket).await;
    }
}
```

Process Sockets

Now we can write the asynchronous `process` function:

```
async fn process(socket: TcpStream) {
    let mut connection = Connection::new(socket);

    // Read the request.
    if let Some(frame) = connection.read_frame().await.unwrap() {
        println!("GOT: {:?}", frame);

        // Respond with an error.
        let response = Frame::Error("unimplemented".to_string());
        connection.write_frame(&response).await.unwrap();
    }
}
```



Server and Client

If we run both the server and client programs:

Server:

```
GOT: Array([Bulk(b"set"), Bulk(b"hello"), Bulk(b"world")])
```

Client:

```
Error: "unimplemented"
```

Server Problem

Our server has a small problem...

```
#[tokio::main]
async fn main() {
    let listener = TcpListener::bind("127.0.0.1:6379").await.unwrap();

    loop {
        let (socket, _) = listener.accept().await.unwrap();
        process(socket).await;
    }
}
```

- *Other than the fact that it only returns "unimplemented" errors*
- Sockets are processed one at a time!

Limited Concurrency

```
loop {  
    let (socket, _) = listener.accept().await.unwrap();  
    process(socket).await;  
}
```

- When a connection is accepted:
 - The server `process` es the `socket` until completion
 - Goes back to listening
- We are `process` ing sequentially
- Ideally, we want to process many requests concurrently

More Concurrency

If we want to process connections concurrently, we will need to `spawn` new tasks.

- We can spawn a new concurrent task for each socket
- This allows the executor to process each connection concurrently!

tokio::spawn

`tokio::spawn` spawns a new task on the `tokio` runtime.

```
loop {
    let (socket, _) = listener.accept().await.unwrap();

    // A new task is spawned for each inbound socket. The socket is
    // moved to the new task and processed there.
    tokio::spawn(async move {
        process(socket).await;
    });
}
```

- Now our server can accept many concurrent requests!

Tasks

You may have noticed we have been using the word "Task".

- A Tokio task is an asynchronous *green thread*
- We create Tokio tasks by passing an `async` block/scope to `tokio::spawn`
- Tokio tasks are *very* lightweight
 - Can spawn millions of tasks without much overhead!

Tasks vs Threads

When we talk about Tokio:

- **Tasks** are lightweight units of execution managed by the Tokio scheduler
 - Tasks can be paused and resumed
- **Threads** are heavyweight units of execution managed by the operating system's scheduler
 - You *cannot* spawn millions of threads without any overhead
- The tokio runtime runs on top of OS threads
 - Many tasks can exist on one thread



Tasks and Threads

Recall that Tokio is a **multi-threaded runtime** for **executing asynchronous code**.

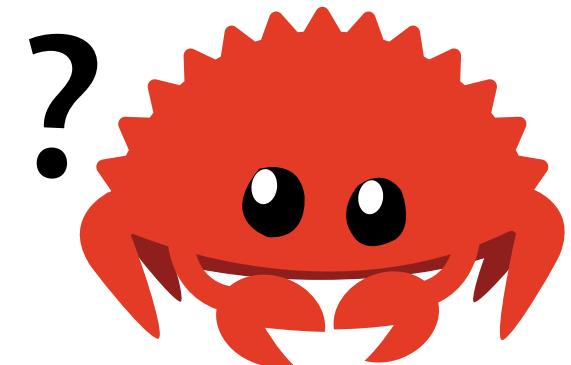
- When tasks are spawned, the Tokio scheduler will ensure that the task executes once it has work to do (when it needs to resume)
- The scheduler might decide to execute the task on a different thread than the one it was originally spawned on
- Tokio implements a **work-stealing** scheduler
- **Important: Not all asynchronous runtimes do this!**

`std::thread::spawn`

Recall that we usually had to `move` variables and values into closures for `std::thread::spawn`.

```
let v = vec![1, 2, 3];

thread::spawn(|| {
    println!("Here's a vector: {:?}", v);
});
```



```
note: function requires argument type to outlive ``static``
--> src/main.rs:6:5
   | /     thread::spawn(|| {
   | |         println!("Here's a vector: {:?}", v);
   | |     });
   | |____^
```

T: '**static**' Bound

More precisely:

- The closure that we pass to `thread::spawn` needs a '**static**' bound
- The closure cannot contain references to data owned elsewhere
- Tokio tasks require the exact same T: '**static**' bound for the same reason

'static Bound

```
#[tokio::main]
async fn main() {
    let v = vec![1, 2, 3];

    tokio::task::spawn(async {
        println!("Here's a vec: {:?}", v);
    });
}
```



- Here is the asynchronous version of the same code, and we will get a similar error

```
error[E0373]: async block may outlive the current function, but it borrows `v`, which is owned by the current function
--> src/main.rs:7:23
7 |     task::spawn(async {
8 |         ^_____
|         |         println!("Here's a vec: {:?}", v);
|         |         - `v` is borrowed here
9 |     });
|     ^_____| may outlive borrowed value `v`
```



```
note: function requires argument type to outlive ``static``
--> src/main.rs:7:17
7 |     task::spawn(async {
8 |         ^_____
|         |         println!("Here's a vector: {:?}", v);
9 |     });
|     ^_____|
```



```
help: to force the async block to take ownership of `v` (and any other
referenced variables), use the `move` keyword
7 |     task::spawn(async move {
8 |         println!("Here's a vec: {:?}", v);
9 |     });
```

async move

The solution is similar to the synchronous `thread::spawn` example: Add `move!`

```
#[tokio::main]
async fn main() {
    let v = vec![1, 2, 3];

    //                                     vvvv
    tokio::task::spawn(async move {
        println!("Here's a vec: {:?}", v);
    });
}
```

Send Bound

Since Tokio is a multi-threaded runtime, it also requires a `Send` bound.

- Because tasks can execute on multiple threads, the runtime needs to be able to `Send` the task between threads
- This is a huge (and almost controversial) topic that we won't get into

The Send Bound Error

If you see this error (or something similar):

```
error: future cannot be sent between threads safely
--> src/main.rs:6:5
6 |     tokio::spawn(async {
|     ^^^^^^^^^^^^^^ future created by async block is not `Send`
```

- RUN AWAY!!!
- If you can't run away, then that is a good time to dive deeper into how `async` and `Future`'s work under the hood
 - *See the speaker notes for a high-level overview*



Shared State

We still have a major flaw...

```
#[tokio::main]
async fn main() {
    let listener = TcpListener::bind("127.0.0.1:6379").await.unwrap();
    let mut db = HashMap::new(); // <-- What do we do with this?

    loop {
        let (socket, _) = listener.accept().await.unwrap();

        tokio::spawn(async move {
            process(socket).await; // How do we pass `db` to each task?
        });
    }
}
```

Shared State

```
let listener = TcpListener::bind("127.0.0.1:6379").await.unwrap();
let mut db = HashMap::new(); // <-- What do we do with this?

loop {
    let (socket, _) = listener.accept().await.unwrap();

    tokio::spawn(async move {
        process(socket).await; // How do we pass `db` to each task?
    });
}
```

- When we process a request, we want to *share* the database among tasks
 - It's not super useful to reset the database for every connection

Shared State: Arc<Mutex<T>>

We can use an `Arc<Mutex<T>>` to allow shared ownership and mutual exclusion!

```
let db = Arc::new(Mutex::new(HashMap::new()));

loop {
    let (socket, _) = listener.accept().await.unwrap();
    // Clone the handle to the hash map.
    let db = db.clone();

    tokio::spawn(async move { process(socket, db).await; });
}
```

- This `Mutex` is the standard library synchronous (blocking) mutex
- See the [tutorial](#) for the `process` code



Server Complete!

Our server is essentially done!

- Supports concurrent connections
 - Spawns an asynchronous task for each connection
- Supports concurrent requests and commands
 - Backing database synchronized with a `Mutex`

What else can we do?

- There are *many* more things we can do to improve our client and server:
 - Use a better database
 - Reducing contention with sharding
 - Persistent storage (file I/O)
 - Multi-node servers (distributed systems)
 - Proxy caching
 - Connection Pooling
- If you are interested in this kind of software engineering, make sure to read the rest of the [Tokio tutorial!](#)

Recap: When not to use Tokio

Tokio is useful for many projects, but there are some cases where this isn't true.

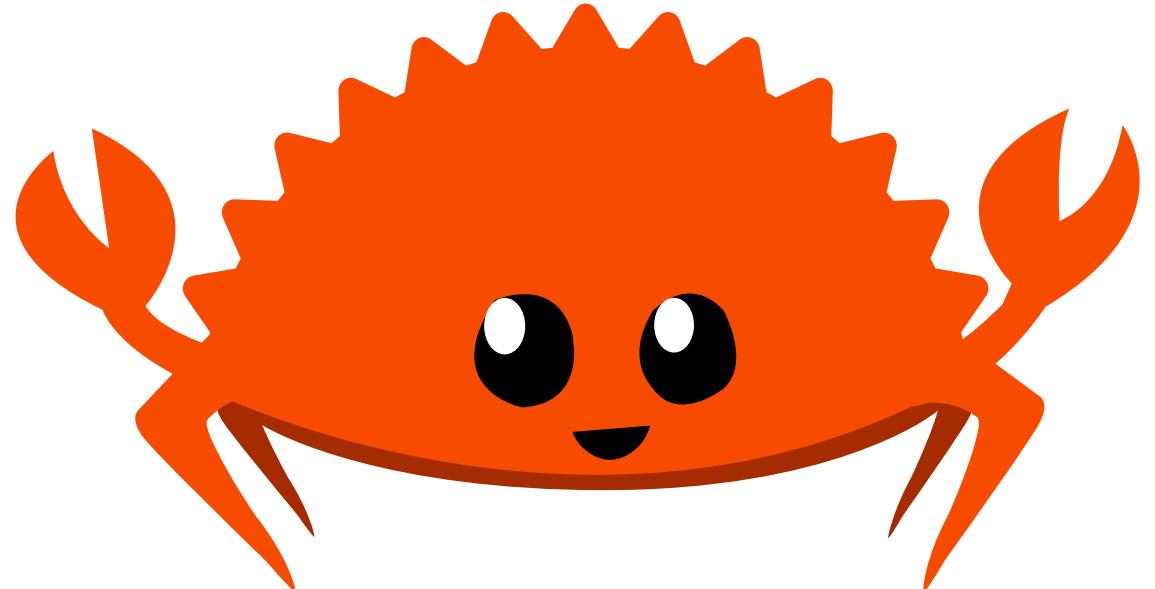
- Tokio is designed for IO-bound applications, not CPU-bound
- No benefit in sequential / low-concurrency programs
- Reading *many* files can also have degrading performance
 - *Operating systems do not provide **stable** asynchronous file APIs*
- It is important to note that Tokio is **NOT** the only asynchronous runtime

Summary: Tokio

- Rust allows us to write concurrent asynchronous code similar to how we would write synchronous code
- Asynchronous code requires an asynchronous runtime to work
- Tokio is a multi-threaded, work-stealing, high-performance `async` runtime
- We can *easily* engineer massively parallel and concurrent servers with Tokio

The End!

We've reached the end of our prepared content!



Why Rust?

- Déjà vu?

Why Rust?

Let's briefly go back to the very beginning of the semester.

- What is Rust?
- What are the biggest advantages of Rust?
- What are some issues that Rust has?
- Who is Rust for?

What is Rust?

- A language empowering everyone to build reliable and efficient software
- From the official rust [website](#), Rust is:
 - Fast
 - Reliable
 - Productive

Rust's Advantages

- Rust is fast
- Rust is memory safe
- Rust enables fearless concurrency
- Rust is modern

Rust's Pitfalls

- Rust is hard
- Rust is young

Who is Rust for?

- Rust targets complex programs while providing stability and security
- Rust is not a magic silver bullet
- Rust is *not* for everyone
- Rust is arguably the best tool for the specific problems it is trying to solve!

The Future of Rust

We believe that Rust is the future of computer systems.

- We *do not* mean that people will *eventually* start using Rust
- Many companies (large and small) have *already* placed their bets on Rust
 - "Exponential growth" of Rust in big tech
 - Many startups methodically choosing Rust to build their foundation
- Rust will continue to grow, and many more people will continue to adopt it

Fish 4.0

- Completely rewritten in Rust
by Feb 27, 2025



fish-shell / fish-shell

Code Issues 482 Pull requests 47 Discussions Actions Projects Wiki Security

Rewrite it in Rust #9512

Merged ridiculousfish merged 50 commits into fish-shell:master from ridiculousfish:riir on Feb 19, 2023

Conversation 164 Commits 50 Checks 0 Files changed 126

ridiculousfish commented on Jan 28, 2023 · edited by zanchey

(Editor's note - please read [#9512 \(comment\)](#) and [#9512 \(comment\)](#) before commenting if you are new to fish or not familiar with the context - [@zanchey](#))

([Progress report November 2023](#))

(Sorry for the [meme](#); also [this is obligatory](#).)

I think we should transition to Rust and aim to have it done by the next major release:

- Nobody really likes C++ or CMake, and there's no clear path for getting off old toolchains. Every year the pain will get worse.
- C++ is becoming a legacy language and finding contributors in the future will become difficult, while Rust has an active and growing community.
- Rust is what we need to turn on concurrent function execution.
- Being written in Rust will help fish continue to be perceived as modern and relevant.

This should be thought of as a "port" instead of a "rewrite" because we would not start from scratch; instead we would translate C++ to Rust, incrementally, module by module, in the span of one release. We'll use an FFI so the Rust and C++ bits can talk to each other until C++ is gone, and tests and CI keep passing at every commit.

To prove it can work, in this PR I've ported FLOG, topic monitor, wgetopt, builtin_wait, and some others to Rust. The Rust bits live in a crate that lives inside the C++ and links with it. You can just build it the usual way:

1. [Install Rust 1.67 or later](#)
2. cmake as usual, and it should work, by way of [corrosion](#)

It works in [GH Actions CI](#) too.

The [Plan](#) has a high level description, and the [Development Guide](#) has more technical details on how to proceed. Please consider both to be part of this PR.

1238 90 93 274 5 390 332 127

Zed

- A code editor built from scratch in Rust. (not a ...
Vscode fork ...)

News: Zed Available on Windows →

Love your editor again

Zed is a minimal code editor crafted for speed and collaboration with humans and AI.

Download now

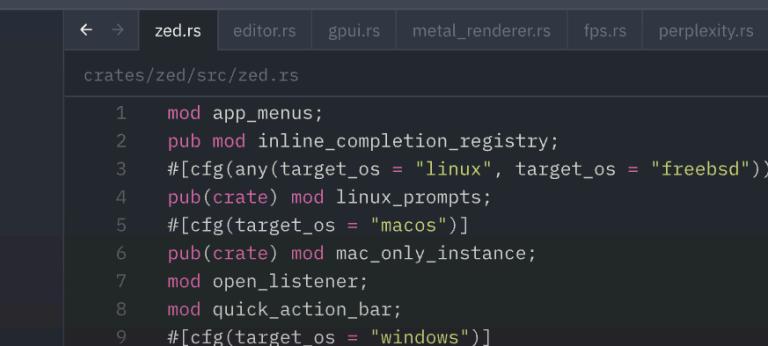
Clone source

Available for macOS, Linux, and Windows

Rust to efficiently
ores and your GPU.

Intelligent
Integrate LLMs into your workflow to
generate, transform, and analyze code.

Collaborative
Chat with teammates, w
notes together, and shar
screen and project. All in



```
1 mod app_menus;
2 pub mod inline_completion_registry;
3 #[cfg(any(target_os = "linux", target_os = "freebsd"))]
4 pub(crate) mod linux_prompts;
5 #[cfg(target_os = "macos")]
6 pub(crate) mod mac_only_instance;
7 mod open_listener;
8 mod quick_action_bar;
9 #[cfg(target_os = "windows")]
```

Course Goals

We wanted all students to:

- Be able to read, write, and reason about Rust code
- Become an intermediate to advanced Rust developer
- Be confident that you can use Rust going forward!

Why Rust?

We hope that you all can answer this question now!

Thanks for taking Rust StuCo!

Rust StuCo (98-008) has been created by:

Connor Tsui, Benjamin Owad, David Rudo,
Jessica Ruan, Fiona Fisher, Terrance Chen

