Futures I

Ryan Eberhardt and Armin Namavari
May 26, 2020

Logistics

- Congrats on making it to week 8! 🔥
  - I can't believe it's week 8 😳
- It's exciting to see people saying they're starting to appreciate Rust more!
  - Thanks for sharing your thoughts in #reflections!

Today

- The Plan
  - Threads — the perfect solution to scalable I/O?
    - This is a rhetorical question, the answer is no.
  - Nonblocking I/O
  - Rust Futures
- These concepts are really tricky so **please ask questions!**
- It’s OK if futures don’t make sense today, we’ll review them and practice them on Thursday as well.

But first… what do you think this code does?

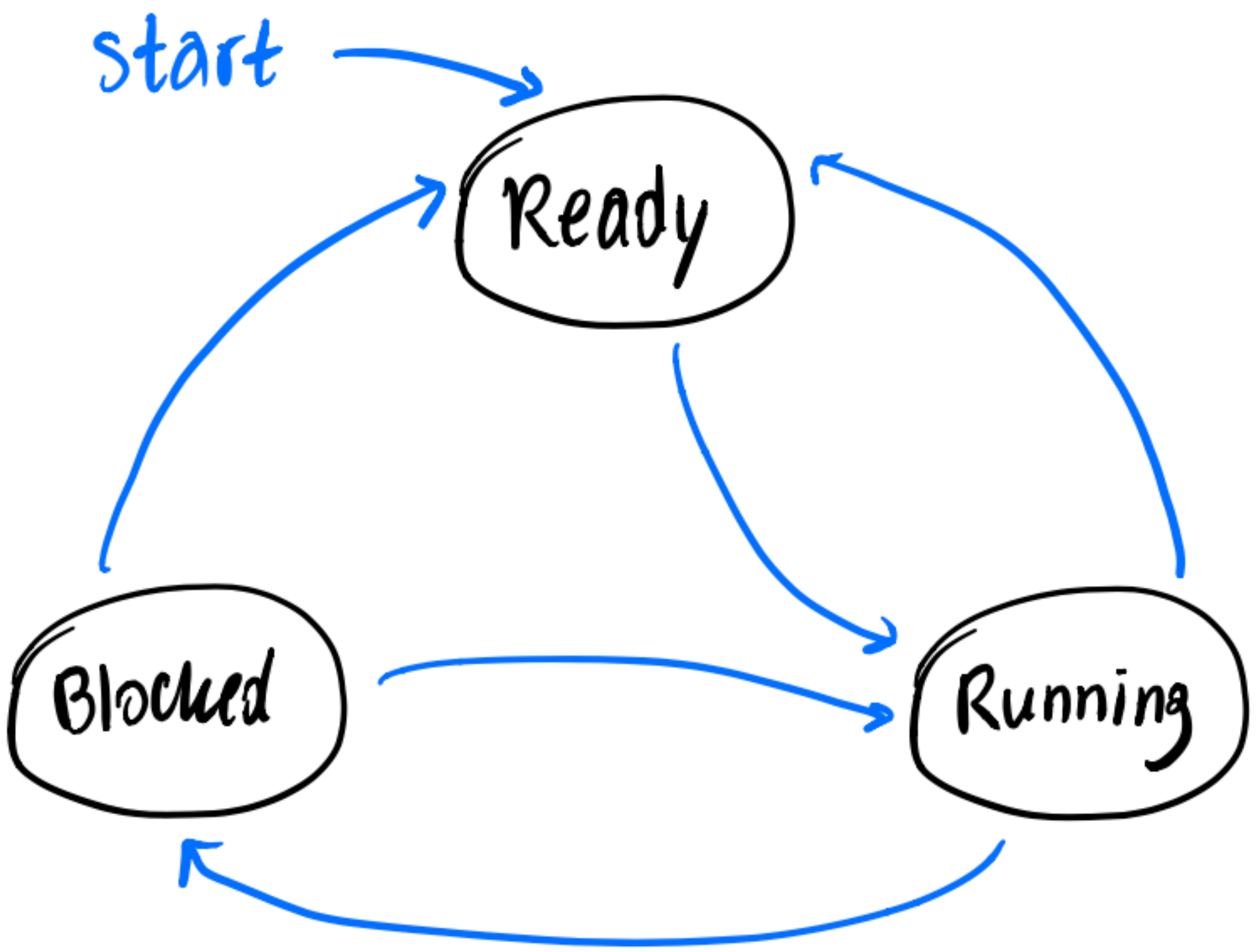
```
// Pretend you don’t see the unfamiliar syntax! (i.e. async/await)
tokio::spawn(async move { // example from the Tokio docs
    let mut buf = [0; 1024];
    loop {
        let n = match socket.read(&mut buf).await {
            Ok(n) if n == 0 => return,
            Ok(n) => n,
            Err(e) => {
                eprintln!("failed to read from socket; err = {:?}", e);
                return;
            }
        };
        if let Err(e) = socket.write_all(&buf[0..n]).await {
            eprintln!("failed to write to socket; err = {:?}", e);
            return;
        }
    }
});
```

Review: Threads

- A “virtual process”
  - Control: the routine (i.e. function) running inside of the thread
  - State: a stack, CPU registers, status (ready/running/blocked), etc.
- **The OS manages threads**
  - The dispatcher is responsible for assigning threads to run on cores, swapping them on and off as appropriate.
    - These context switches aren’t the cheapest thing e.g. the overhead of copying stuff, cache evictions etc.
  - The scheduler is responsible for deciding what thread to run next.

The Dispatcher

- What sorts of things can move us from “running” to “blocked”?
    - I/O: reading and writing
    - Waiting: waitpid, sigsuspend, join, cv.wait(…) etc.
    - lock()
    - sleep()
- If a thread is blocked, it can’t waste CPU resources
    - This is why threading lets us overlap wait times for I/O bound operations.



Building a High Performance Server with Threads

- Great, so if we want to build a server that can handle many requests at once, we just declare a big thread pool with ~4000 threads, right?
    - Each thread needs its own stack…
    - 4000 lil' stacks adds up to a LOT of memory!
    - This ends up being very cache unfriendly
    - The OS also has to manage resources on behalf of these 4000 threads
- Upshot: if you use **blocking** operations, you are fundamentally limited by the number of threads you can run at once 😕
- Also, threads are often hard to get right
    - Race conditions, deadlock, etc.

Non-blocking I/O

- Traditionally, the read sys call would block if there is more data to be read but not available
- Instead, we could have read return a special error value instead of blocking so that we can do other useful work on this thread e.g. reading from other descriptors we're managing.
  - This is especially relevant for I/O intensive pieces of software like servers.
  - Often times you'd call these nonblocking I/O operations in a loop and use something like epoll to keep track of which are ready
- **This allows us to have concurrent I/O with one thread!**

Non-blocking I/O visualized

- Epoll is a kernel-provided mechanism that notifies us of what fds are ready for I/O.
    - Why should we attempt to do I/O on fds that aren't even ready?
- We perform I/O only on descriptors that are ready until they are no longer ready.

```
while(true){
    "Hey epoll what's ready for reading?"
    epoll => [7, 12, 15]
    "Thanks epoll"
    read(7) => 0110011000101...
    read(12) => 1001001101011 ...
    read(15) => 01101011100101
}
```

More data to read, but we return

No more data to read from fd 15

State management

- Epoll is nifty, but it forces us to manage state in tricky ways
  - If you have one thread per connection, all the state for each connection is stored in each thread's stack
  - If you're trying to use epoll, you have to store the state yourself and somehow associate each file descriptor with state

```
while(true) {
    "Hey epoll what's ready for reading?"
    epoll => [7, 12, 15]
    "Thanks epoll"
    read(7) => 0110011000101...
    read(12) => 1001001101011 ...
    read(15) => 01101011100101
}
```

More data to read, but we return

No more data to read from fd 15

State management

- Rust (and a handful of other languages) us in two ways:
    - *Futures* allow us to keep track of in-progress operations along with associated state, in one package
    - async/await syntax allows us to easily chain futures together, creating “threads” of futures

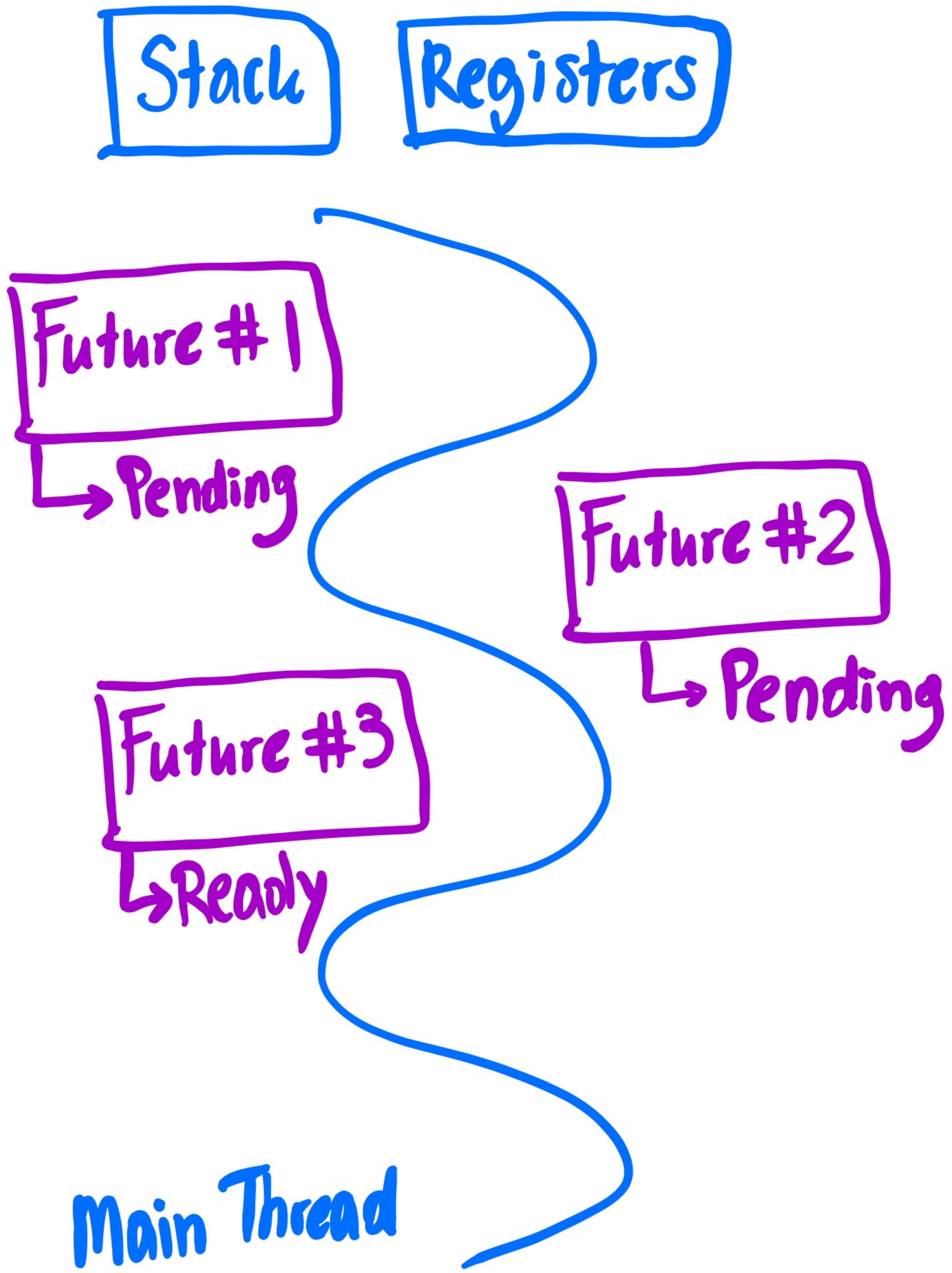
while (true) {
"Hey epoll what's ready for reading?"
epoll ⇒ [7, 12, 15]
"Thanks epoll"
read (7) ⇒ 0110011000101...
read (12) ⇒ 1001001101011...
read (15) ⇒ 01101011100101
}

More data to read, but we return

No more data to read from fd 15

Intro to Futures

- Future: the result of a computation that may or may not have completed.
    - A “computation in progress”
    - Very similar to promises in Javascript (if you’re familiar with those)
    - A single thread can run multiple futures =>
- In Rust, futures are structs that implement the Future trait
    - These structs could represent, for instance, a nonblocking I/O operation.



The Future Trait

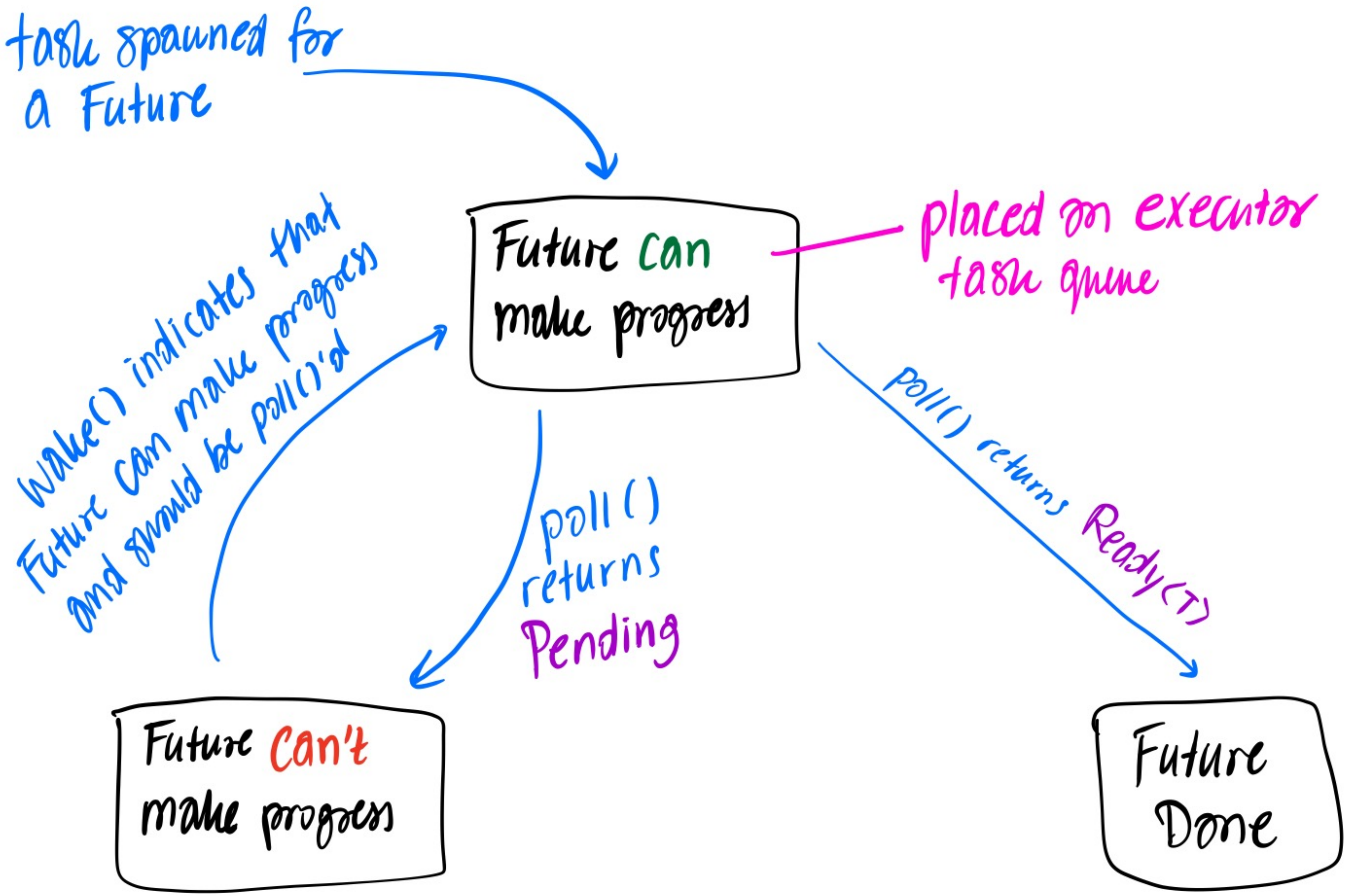
```
trait Future { // This is a simplified version of the Future definition
    type Output;
    fn poll(&mut self, cx: &mut Context) -> Poll<Self::Output>;
    // cx contains a “waker” that provides a notification mechanism
    // to indicate that the Future is ready to make more progress
    // e.g. data becomes available to read
}

enum Poll<T> {
    Ready(T),
    Pending,
}
```

Executors

- In order to actually execute futures, we need some sort of runtime or "executor" that repeatedly calls the "poll" function of the Future object.
  - This is a generalization of the loop for nonblocking I/O we had earlier.
- A popular executor in the Rust ecosystem is Tokio and it's what you'll be using in Project 2!
- If you have multiple cores on your machine, you can actually execute futures truly in parallel!
  - This means that if you have multiple async tasks running, you need to protect shared data using synchronization primitives.

What is an executor really doing?



Combining futures together

- Map — apply some function to the output of the future
  - We can combine a function and a future to get a new future!
- Join — start executing a group of futures concurrently
  - We can take futures, put them together, and get a new future!
- Rust lets us ergonomically chain futures together by using the await keyword.

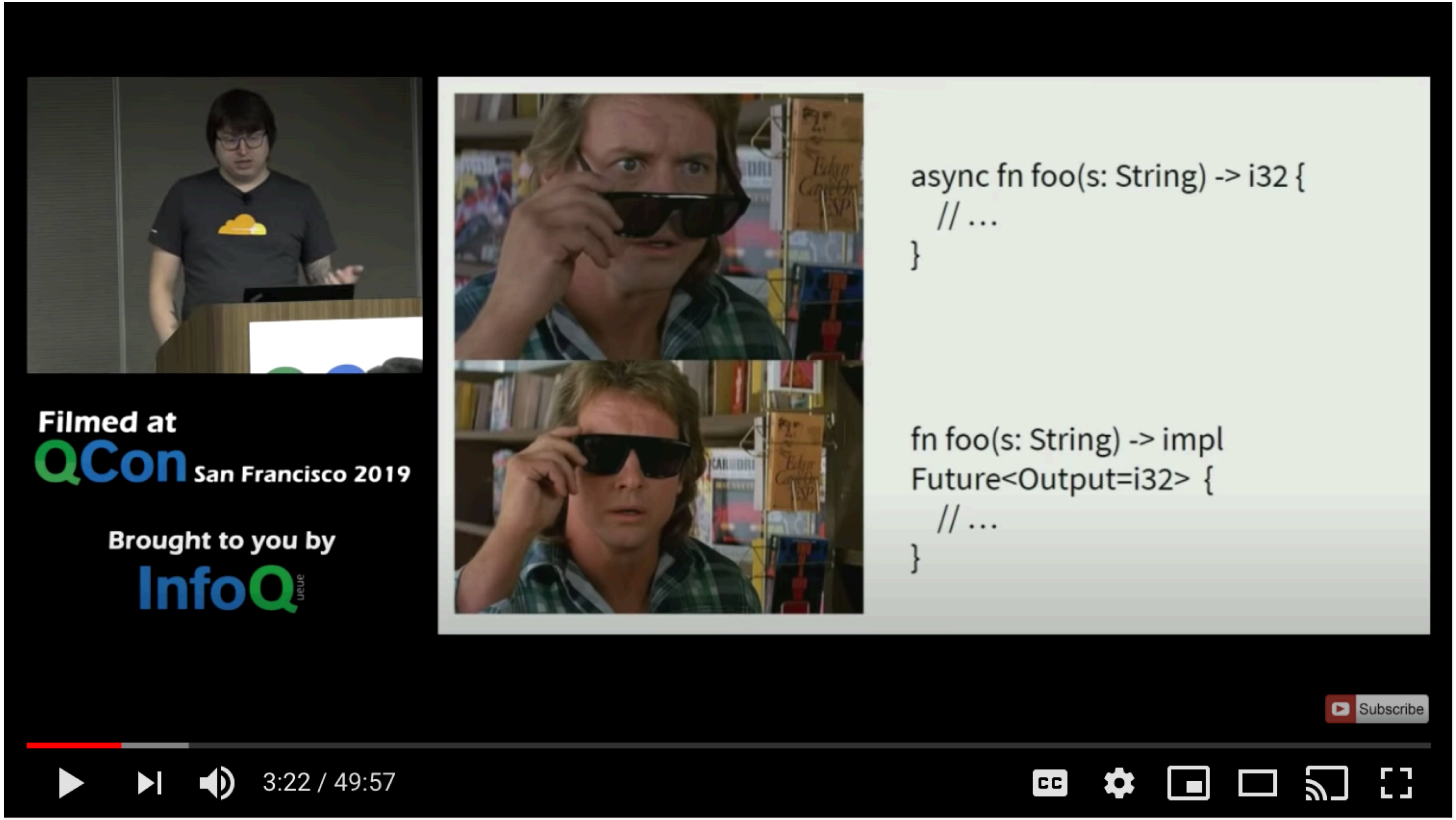
Async/Await Code Example

```
tokio::spawn(async move { // example from the Tokio docs for a TCP echo server
    let mut buf = [0; 1024];

    // In a loop, read data from the socket and write the data back.
    loop {
        let n = match socket.read(&mut buf).await { // non-blocking read!
            // socket closed
            Ok(n) if n == 0 => return, // no more data to read
            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 { // non-blocking write!
            eprintln!("failed to write to socket; err = {:?}", e);
            return;
        }
    }
});
```

Async: Under the Hood



Await vs. Join

```
async fn assemble_book() -> String {
    // The request returns a future for a non-blocking read operation
    let half1 = request_first_half_server();
    let half2 = request_second_half_server();
    let first_half_str: String = half1.await;
    let second_half_str: String = half2.await;
    format!("{}{}", first_half_str, second_half_str)
}

async fn assemble_book() -> String {
    // The request returns a future for a non-blocking read operation
    let half1 = request_first_half_server();
    let half2 = request_second_half_server();
    let (first_half_str, second_half_str) = futures::join!(half1, half2);
    format!("{}{}", first_half_str, second_half_str)
}
```

Async/Await in Rust

- Rust enables us to write our code in a way that looks blocking, but actually runs asynchronously
  - Like many fancy features in Rust, we get this from the magic of the Rust compiler — async/await provide us with syntactic sugar.
  - Long story short: the Rust compiler is able to transform your chain of async computation (i.e. futures) into an efficient state machine.
- This is amazing! You get the ergonomics of writing code that looks like it's blocking but the performance benefits of nonblocking operations!

General Tips for Async Rust

- Never block in async code!
  - Asynchronous tasks are cooperative (not preemptive)
- You can only use `await` in `async` functions.
- Rust won't let you write async functions in traits (for technical reasons that have to do with lifetimes and the fact that you can't have associated type bounds *yet*)
  - You can use a crate called async-trait though!
- Be cognizant of shared state between tasks and synchronize appropriately! (e.g. you may need a Mutex<T>, but of course, one that will play well with Futures)
  - Tokio provides its own async implementations of concurrency primitives. E.g. you can replace `std::sync::mutex` with `tokio::sync::mutex` (the API is nearly identical)

Additional Resources/References

- A great talk about how Rust arrived on the design for futures
- Another great talk about futures
- Phil Levis' CS110 Lecture on Events, Threads, and Async I/O
- The Rust Docs on Futures
- An article on futures
- John Ousterhout on why threads are a bad idea
- A great (and very accessible) Medium article explaining epoll (also has great illustrations!)
- A CS242 Assignment on Implementing Futures
- Note: the syntax for futures has changed over time so some of these articles may use outdated syntax — for the most up-to-date syntax, check out the docs.