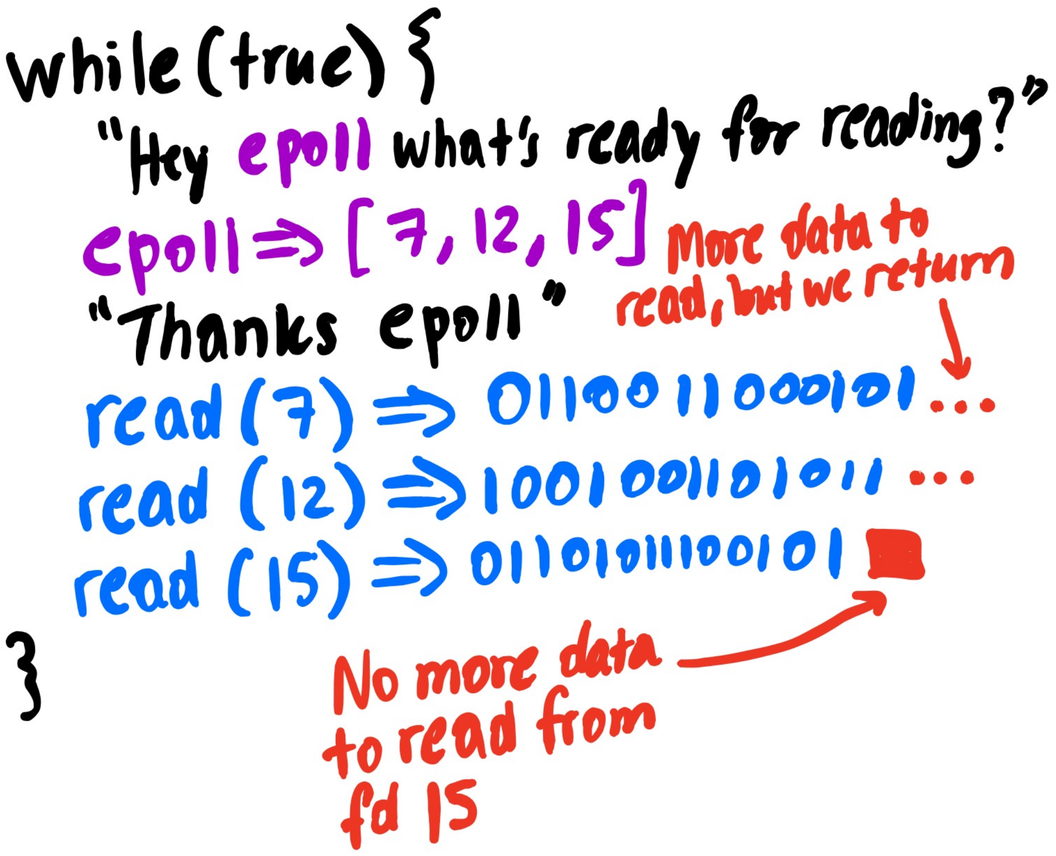
Futures II

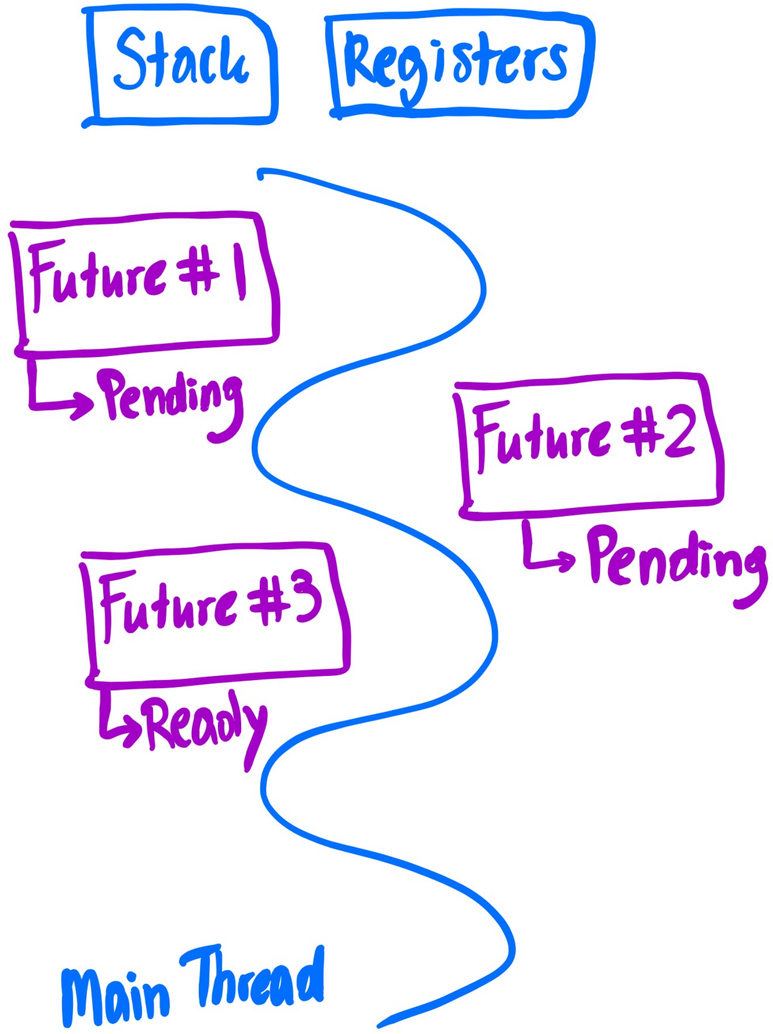
Ryan Eberhardt and Armin Namavari
May 28, 2020

Today

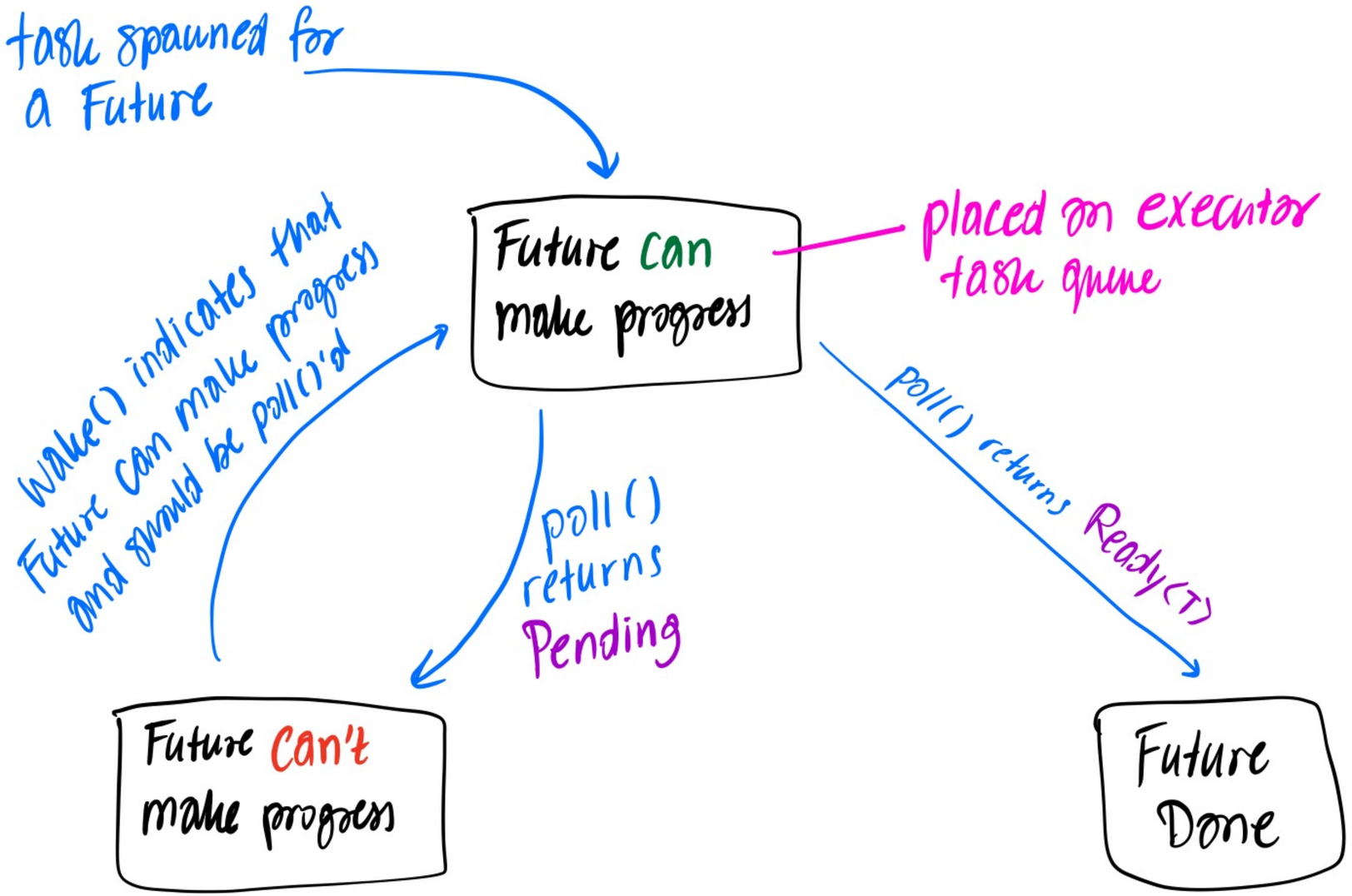
- The Plan
  - Review futures from last time
  - Talk about how futures can be combined together
  - Live coding example
  - Parting thoughts on async/await
- These concepts are really tricky so **please ask questions!**
  - You will get practice with these concepts in project 2!

Non-blocking I/O and Futures





What is an executor really doing?



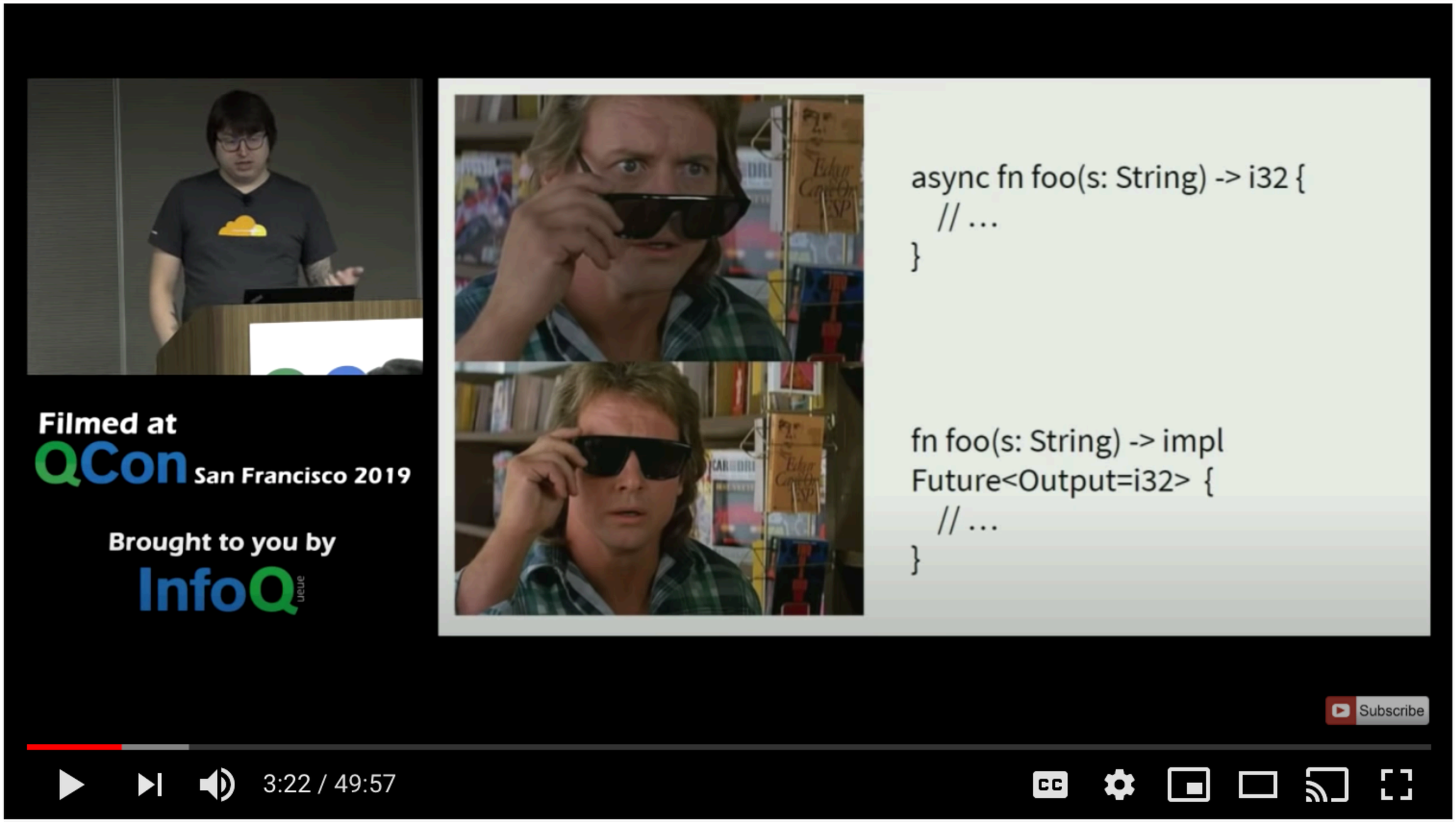
Combining futures together

- 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

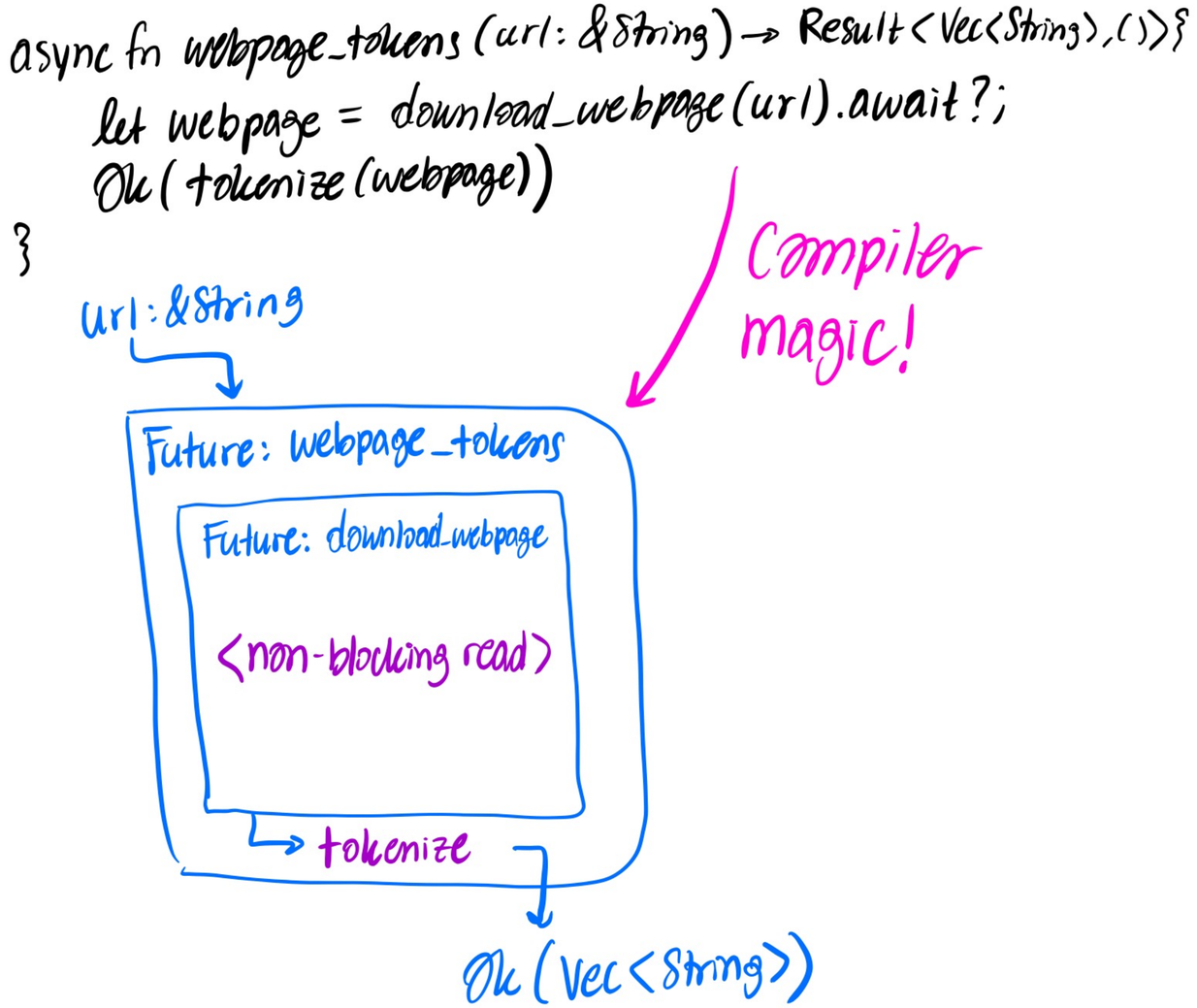
```
tokio::spawn(async move { // example from the Tokio docs for a TCP echo server
    let mut buf = [0; 1024];

    // 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



Async: Under the Hood

- The Rust compiler transforms the async function into a function that returns a future.
- This particular future will apply `tokenize` to the output of the future returned by `download_webpage`



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)
}
```

Link-Explorer Revisited with Async/Await

- Let’s revamp link-explorer link explorer example with async/await!
- Recall the version we had with threading.
  - I’ve upgraded it to work with a ThreadPool
  - Let’s see how well it does
- Now we’re going to code up the async version of it
  - And we’ll have to use async synchronization primitives to protect shared data!

Results

- Threadpool (20 threads, implicitly limits the number of files open at once)

```
Armins-MacBook-Pro-2:link_explorer_pool armin$ time cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.11s
     Running `target/debug/link_explorer_pool`
https://en.wikipedia.org/wiki/Artificial_intelligence was the longest article with length 1103513

real    0m6.584s
user    0m6.365s
sys     0m1.491s
```

- Async/await (Tokio, max 20 threads + a semaphore to restrict how many files can be open at once)

```
Armins-MacBook-Pro-2:link_explorer_async armin$ time cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.13s
     Running `target/debug/link_explorer_async`
https://en.wikipedia.org/wiki/Artificial_intelligence was the longest article with length 1103513

real    0m4.285s
user    0m6.757s
sys     0m0.698s
```

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! 🔥
- However, this also means that a lot of your code ends up having to become async — you can only call an async function in an async block
  - It also makes backtraces harder to interpret 😕

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)