Lecture 5 — Asynchronous I/O

Patrick Lam & Jeff Zarnett
patrick.lam@uwaterloo.ca, jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

September 9, 2024

Dreams of Neo-Tokyo Tonight

`https://www.youtube.com/watch?v=Wu5TDEpAqwQ`

Why Non-Blocking I/O?

```
fn main() -> io::Result<()> {
    let mut file = File::open("hello.txt")?;
    let mut s = String::new();
    file.read_to_string(&mut s)?;
    Ok(())
}
```

(The ? operator is an alternative to `try!` and unwrap.)

The problem is that the `read` call will block.

You don’t get to use the CPU cycles while waiting.

Threads Insteads?

Threads can be fine if you have some other code running to do work.

But maybe you would rather not use threads. Why not?

- potential race conditions;
- overhead due to per-thread stacks; or
- limitations due to maximum numbers of threads.

Non-Blocking I/O

We're going to focus on low-level I/O from sockets in this part of the lecture, using the `mio` library from `tokio`.

(Sadly, not Neo-Tokyo. This is from *Akira*.)

Gratuitous travel shot

photo credit: Patrick Lam

Non-Blocking I/O

Async file I/O is also possible via `tokio::fs` and the ideas will carry over.

One might often want to wrap the low-level I/O using higher-level abstractions, and the larger project `tokio.rs` is one way of doing that.

There are two ways to find out whether I/O is ready to be queried:
(1) Polling and (2) Interrupts.

`mio` uses polling.

Using `mio`

The key idea is to give `mio` a bunch of event sources and wait for events to happen.

- create a `Poll` instance;
- populate it with event sources e.g. `TCPListeners`; and,
- wait for events in an event loop (`Poll::poll()`).

```
let poll = Poll::new()?;
let events = Events::with_capacity(128);
```

We're going to proactively create `events`; this data structure is used by `Poll::poll` to stash the relevant `Event` objects.

Name a more iconic duo

Populating the `Poll` instance: “registering event source”.

This is a socket or file descriptor.

```
let mut listener = TcpListener::bind(address)?;
const SERVER: Token = Token(0);
poll.registry().register(&mut listener, SERVER, Interest::READABLE)?;
```

You’re telling it to check for when the `listener` indicates that something is available to read.

Waiting for Events

We're ready to wait for events on any registered listener.

```
loop {
    poll.poll(&mut events, Some(Duration::from_millis(100)))?;

    for event in events.iter() {
        match event.token() {
            SERVER => loop {
                match listener.accept() {
                    Ok((connection, address)) => {
                        println!("Got a connection from: {}", address);
                    },
                    Err(ref err) if would_block(err) => break,
                    Err(err) => return Err(err),
                }
            }
        }
    }
}

fn would_block(err: &io::Error) -> bool {
    err.kind() == io::ErrorKind::WouldBlock
}
```

`poll.poll` will populate `events`, and waits for at most 100 ms.

If all you want to do is request a web page in Rust, use the `reqwest` library.

```
let body = reqwest::get("https://www.rust-lang.org")
    .await?
    .text()
    .await?;

println!("body = {:?}", body);
```

(If you are doing multiple requests, you should create your own `Client` and `get` from it instead of `reqwest::get`).

Back to the Futures

Back to the Futures

The use of `await` is a bit tricky.

The `get` function returns a *future*.

What's that?

It's an object that will, at some point in the future, return a second object.

Plug-In Executors

There are many possible definitions of `async/await`, and the appropriate one depends on your context.

Rust allows you to specify a runtime which defines the meaning of `async/await` for your program.

The simplest `await` just blocks and waits on the current thread for the result to be ready.

I’ll Just Wait


```rust
use futures::executor::block_on;

async fn hello_world() {
    println!("hello");
}

fn main() {
    let future = hello_world();
    block_on(future);
}
```

More Options

`tokio` includes a more sophisticated executor as well.

When there are multiple active `awaits`, `tokio` can multiplex them onto different threads.

You can specify the `tokio` executor (or others) with a tag above `main()` and by declaring `main()` to be `async`.

```
#[tokio::main]
async fn main() {
    // do async stuff
}
```

Using libcurl

libcurl is a C library for transferring files.

First we'll start with the easy interface.

This is a synchronous interface that uses callbacks.

Your HTML. Give it to me.

```rust
use std::io::{stdout, Write};

use curl::easy::Easy;

// Write the contents of rust-lang.org to stdout
let mut easy = Easy::new();
easy.url("https://www.rust-lang.org/").unwrap();
easy.write_function(|data| { // callback function
    stdout().write_all(data).unwrap();
    Ok(data.len())
}).unwrap();
easy.perform().unwrap();
```

Note that we provide a lambda as a callback function.

This lambda is to be invoked when the library receives data from the network (i.e. `write_function()`).

Processing the Output

In the body of the lambda, we simply write the received data to `stdout` and return the number of bytes we processed (all of them, in this case).

Looking at the original libcurl documentation, you'll see how the Rust bindings are a fairly straightforward translation.

We call `easy.perform()` to, well, perform the request, blocking until it finishes, and using the callback to process the received data.

M-m-m-multi-cURL

The real reason we're talking about libcurl is the asynchronous multi interface.

Network communication is a great example of asynchronous I/O.

You can start a network request and move on to creating more without waiting for the results of the first one.

For requests to different recipients, it certainly makes sense to do this.

The Multi Handle

The main tool here is the “multi handle”.

The structure for the new multi-handle type is `curl::multi::Multi`.

The multi functions may return a `MultiError` rather than the easy `Error`.

Pack ’em In

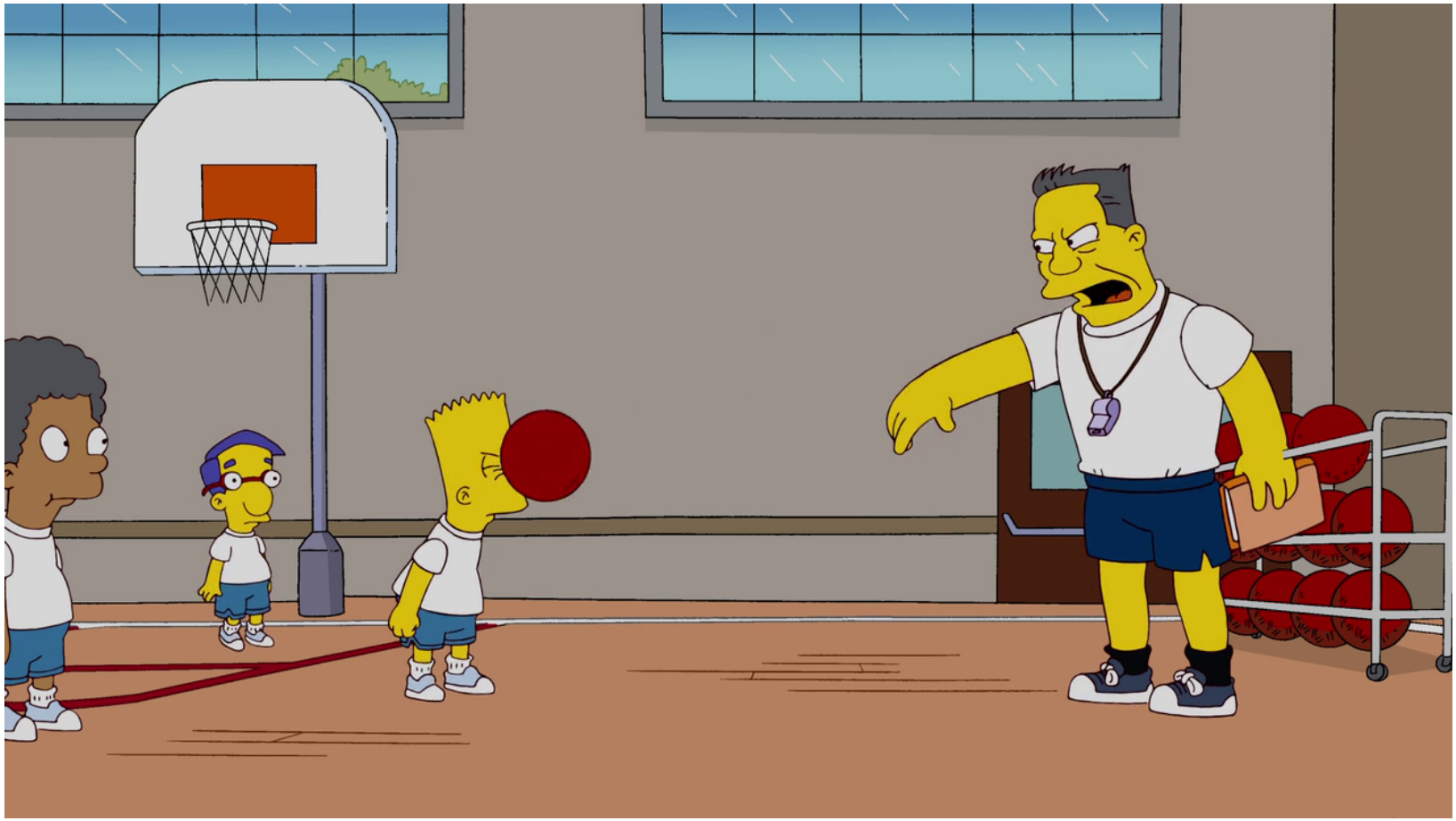
Once we have a multi handle, we can add easy objects—however many we need—to the multi handle.

Creation of the easy object is the same as it is when being used alone.

Then, we add the easy (or easy2) object to the multi handle with `add()` (or `add2()`).

The `add()` or `add2()` functions return an actual easy handle.

BOMBARDMENT!



Once we have finished putting all the easy handles into the multi handle, we can dispatch them all at once with `perform()`.

Repeat Performance

This function returns, on success, the number of easy handles in that multi handle that are still running.

If it's down to 0, then we know that they are all done.

This does mean that we're going to call `perform()` more than once.

Doing so doesn't restart or interfere with anything that was already in progress.

We can check often, but the intention is to do other stuff in the meantime.

I’m booooooored

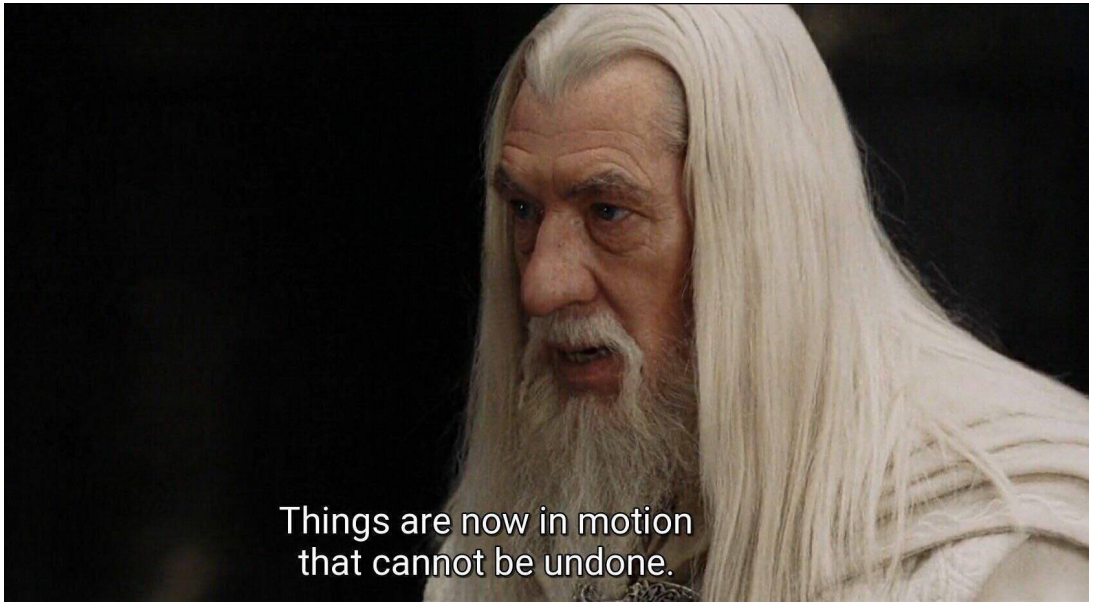
Suppose we’ve run out of things to do though.

We can wait, if we want, using `wait()`.

This function will block the current thread until something happens.

The first parameter to `wait()` is an array of extra file descriptors you can wait on (but we will always want this to be `&mut []` in this course).

After Perform



In the meantime though, the perform operations are happening, and so are whatever callbacks we have set up (if any).

As the I/O operation moves through its life cycle, the state of the easy handle is updated appropriately.

Each easy handle has an associated status message as well as a return code.

Dude, What Happened?

We pass `messages()` a callback which finds out what happened and makes sure all is well.

What we are looking for is that the callback's parameter `msg` has `result_for` including `Some`—request completed.

If not, this request is still in progress and we aren't ready to evaluate whether it was successful or not.

If there are more handles to look at, we should go on to the next. If it is done, we should look at the result.

If it is `Error` then there is an error. Else, everything succeeded.

Tidying Up

When a handle has finished, you need to remove it from the multi handle.

Remove the handle you got back from `add/2` with `remove/2`.

You don’t have to cleanup the easy handle because Rust.

Longer Example

```
const URLS:[&str; 4] = [
  "https://www.microsoft.com",
  "https://www.yahoo.com",
  "https://www.wikipedia.org",
  "https://slashdot.org" ];

use curl::Error;
use curl::easy::{Easy2, Handler, WriteError};
use curl::multi::{Easy2Handle, Multi};
use std::time::Duration;
use std::io::{stdout, Write};

struct Collector(Vec<u8>);
impl Handler for Collector {
    fn write(&mut self, data: &[u8]) -> Result<usize, WriteError> {
        self.0.extend_from_slice(data);
        stdout().write_all(data).unwrap();
        Ok(data.len())
    }
}
fn init(multi:&Multi, url:&str) -> Result<Easy2Handle<Collector>, Error> {
    let mut easy = Easy2::new(Collector(Vec::new()));
    easy.url(url)?;
    easy.verbose(false)?;
    Ok(multi.add2(easy).unwrap())
}
```

Longer Example

```
fn main() {
    let mut easys : Vec<Easy2Handle<Collector>> = Vec::new();
    let mut multi = Multi::new();

    multi.pipelining(true, true).unwrap();
    for u in URLS.iter() {
        easys.push(init(&multi, u).unwrap());
    }
    while multi.perform().unwrap() > 0 {
        // .messages() may have info for us here...
        multi.wait(&mut [], Duration::from_secs(30)).unwrap();
    }

    for eh in easys.drain(..) {
        let mut handler_after:Easy2<Collector> = multi.remove2(eh).unwrap();
        println!("got response code {}", handler_after.response_code().unwrap
            ());
    }
}
```

Recycle, Reduce, Reuse

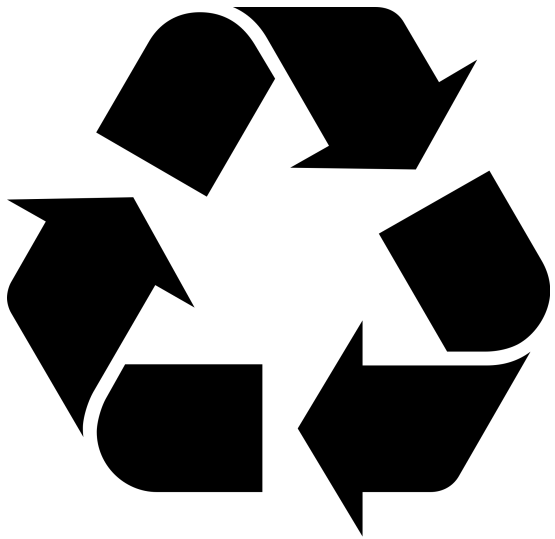


Image Credit: Wikipedia user Krdan

Can we re-use an easy handle rather than destroy and create a new one?

Recycle, Reduce, Reuse

The official docs say that you can re-use one.

But you have to remove it from the multi handle and then re-add it.

... presumably after having changed anything that you want to change about that handle.

Always Running

You could have a situation where there are constantly handles in progress.

You might never be at a situation where there are no messages left.

And that is okay.

Scaling This

The developer claims that you can have multiple thousands of connections in a single multi handle.

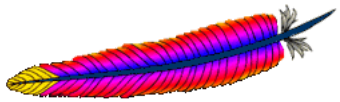
60k ought to be enough for anyone!

cURL + select

Process, Threads, AIO?! Four Choices

- Blocking I/O; 1 process per request.
- Blocking I/O; 1 thread per request.
- Asynchronous I/O, pool of threads,
  callbacks,
  each thread handles multiple connections.
- Nonblocking I/O, pool of threads,
  multiplexed with select/poll, event-driven,
  each thread handles multiple connections.

Blocking I/O; 1 process per request



Old Apache model:

- Main thread waits for connections.
- Upon connect, forks off a new process, which completely handles the connection.
- Each I/O request is blocking: e.g. reads wait until more data arrives.

Advantage:

- “Simple to understand and easy to program.”

Disadvantage:

- High overhead from starting 1000s of processes. (can somewhat mitigate with process pool).

Can handle $\sim$10 000 processes, but doesn’t generally scale.

Blocking I/O; 1 thread per request

We know that threads are more lightweight than processes.

Same as 1 process per request, but less overhead.

I/O is the same—still blocking.

Advantage:

- Still simple to understand and easy to program.

Disadvantages:

- Overhead still piles up, although less than processes.
- New complication: race conditions on shared data.

Asynchronous I/O Benefits

In 2006, perf benefits of asynchronous I/O on lighttpd[1]:

version		fetches/sec	bytes/sec	CPU idle
1.4.13	sendfile	36.45	3.73e+06	16.43%
1.5.0	sendfile	40.51	4.14e+06	12.77%
1.5.0	linux-aio-sendfile	72.70	7.44e+06	46.11%

(Workload: 2 $\times$ 7200 RPM in RAID1, 1GB RAM,
transferring 10GBytes on a 100MBit network).

[1] `http://blog.lighttpd.net/articles/2006/11/12/lighty-1-5-0-and-linux-aio/`

Using Asynchronous I/O in Linux (select/poll)

Basic workflow:

1. enqueue a request;
2. … do something else;
3. (if needed) periodically check whether request is done; and
4. read the return value.

See the ECE 252 notes if you want to learn about select/poll!