Basic non-blocking IO using epoll in Rust

I've been using async, or non-blocking IO systems for years in multiple languages such as JavaScript (Node.js), Java/Kotlin and Rust.

I couldn't, however, explain very well how it works under the hood and while I heard about `poll`, `select`, `epoll`, event queues and the like before, I wouldn't have been confident explaining how non-blocking IO actually works to anyone.

The benefits and drawbacks of using it - sure, but not how any of it works. In this post we will implement a very, very simple non-blocking HTTP server in Rust, which will serve multiple requests at a time while still running only one thread.

In this example, the focus will be on Linux, so we will be using the `epoll` family of system calls to achieve the goal of non-blocking IO.

In the resources section at the end of the post, you'll find links to some of the tutorials and libraries I read in preparation of this and these resources all deal with Linux, Windows and MacOS, so if you're interested in other platforms, or in a deeper understanding of the topic, I can highly recommend you check them out.

Let's get started!

Setup

The only dependency we're going to use is `libc` and we're not going to deal with errors...at all really. We'll propagate all errors to the top to keep it simple and if anything goes wrong, the program will panic and stop.

First, let's define a helper macro, which I copied from mio, to call `libc` system calls and wrap them in a `Result`:

```
#[allow(unused_macros)]
macro_rules! syscall {
    ($fn: ident ( $($arg: expr),* $(,)* ) ) => {{
        let res = unsafe { libc::$fn($($arg, )*) };
        if res == -1 {
            Err(std::io::Error::last_os_error())
        } else {
            Ok(res)
        }
    }};
}
```

There isn't much to this macro, it just calls the given libc syscall in an `unsafe` block and, if it returns an error (-1), that error is wrapped in a Result, and otherwise the returned value is Ok'ed back.

Next, we need to start a TCP server on `127.0.0.1:8000`, so we can test our setup later on:

```
use std::os::unix::io::{AsRawFd, RawFd};
use std::io;
use std::io::prelude::*;
use std::net::{TcpListener, TcpStream};
```

```
const HTTP_RESP: &[u8] = b"HTTP/1.1 200 OK
content-type: text/html
content-length: 5

Hello";

fn main() -> io::Result<()> {
    let listener = TcpListener::bind("127.0.0.1:8000")?;
    listener.set_nonblocking(true)?;
    let listener_fd = listener.as_raw_fd();

    ...

    Ok(())
}
```

This snippet creates a `TcpListener` on the given socket address. We set it to `nonblocking` mode, which means the `accept` operation, which enables us to accept new connections, becomes non-blocking. That means, if we tried to `accept` now and the socket wouldn't be ready yet, we would receive an `io::ErrorKind::WouldBlock` error, telling us to try again later. But `accept` would return immediately and not block.

Finally, we remember the actual file descriptor, which is just an integer on Linux, as we'll need it later on. We also define `HTTP_RESP`, which will be our hard-coded response to all incoming HTTP requests.

Next, let's set up our epoll event queue.

```
fn main() -> io::Result<()> {
    ...

    let epoll_fd = epoll_create().expect("can create epoll queue");
    ...
}


fn epoll_create() -> io::Result<RawFd> {
    let fd = syscall!(epoll_create1(0))?;
    if let Ok(flags) = syscall!(fcntl(fd, libc::F_GETFD)) {
        let _ = syscall!(fcntl(fd, libc::F_SETFD, flags | libc::FD_CLOEXEC));
    }

    Ok(fd)
}
```

And that's it. the `epoll_create` function simply calls the epoll_create1 system call and returns the file descriptor of our epoll event notification facility.

Using this file descriptor, we can now add, modify and remove interests in certain events on certain file descriptors and when we call epoll_wait later on, it will block until such events occur, notifying us, that we can, for example, read on one of the given file descriptors (such as an incoming connection).

But let's progress step-by-step. Now that we have our `epoll`, we want to add an interest in incoming connections for our server to handle:

```
const READ_FLAGS: i32 = libc::EPOLLONESHOT | libc::EPOLLIN;
const WRITE_FLAGS: i32 = libc::EPOLLONESHOT | libc::EPOLLOUT;
```

```
fn main() -> io::Result<()> {
    ...
    let mut key = 100;
    add_interest(epoll_fd, listener_fd, listener_read_event(key))?;
    ...
}

fn add_interest(epoll_fd: RawFd, fd: RawFd, mut event: libc::epoll_event) -> io::Result<()> {
    syscall!(epoll_ctl(epoll_fd, libc::EPOLL_CTL_ADD, fd, &mut event))?;
    Ok(())
}

fn listener_read_event(key: u64) -> libc::epoll_event {
    libc::epoll_event {
        events: READ_FLAGS as u32,
        u64: key,
    }
}
```

The `add_interest` function takes the file descriptor of the `epoll` queue, the file descriptor of whatever we want to add an interest in and an `epoll_event` which defines what kind of events we want to be notified about.

This event has two elements, `events` and `u64`. `events` is a set of flags, set depending on what kind of events we're interested in. In this case, we are only interested in read events, so `EPOLLIN`. However, in both the read and write case, we additionally set the `EPOLLONESHOT` flag, which means that when we get notified based on this interest, the interest is removed.

So if we want to, for example, read and keep reading (because we expect there to be more data), we'll need to register this interest again.

These parameters are passed on to epoll_ctl using the `EPOLL_CTL_ADD` flag, which indicates we're adding an interest.

Alright, our server is set up, our epoll queue exists, interest in new connections has been shown and our next step takes us already to our event loop!

```
fn main() -> io::Result<()> {
    ...
    let mut events: Vec<libc::epoll_event> = Vec::with_capacity(1024);
    ...
    loop {
        events.clear();
        let res = match syscall!(epoll_wait(
            epoll_fd,
            events.as_mut_ptr() as *mut libc::epoll_event,
            1024,
            1000 as libc::c_int,
        )) {
            Ok(v) => v,
            Err(e) => panic!("error during epoll wait: {}", e),
        };

        // safe  as long as the kernel does nothing wrong - copied from mio
        unsafe { events.set_len(res as usize) };
        ...
    }
}
```

We start an infinite `loop`. In this loop, the first thing we do is to `clear` our events vector. This `events` vector is a mutable vector of size 1024, which we'll pass to `epoll_wait`, and where new

events will be set.

This is also why we clear it, because we don't want any old events in here.

Then, we call `epoll_wait` with the above mentioned `epoll_fd`, the `events`, the maximum amount of events it should return and a timeout in milliseconds. This timeout will be hit, if no events happen.

Once called, `epoll_wait` will block until one of three things happens:

- A file descriptor, which we registered interest in has an event
- A signal handler interrupts the call
- The timeout expires

We could set the timeout to `-1`, which will make `epoll_wait` block infinitely, or until any of the other 2 things happens.

When we stop waiting, we get the number of events back as `res` here. Then we set the length of `events` to this number. We use `unsafe` here, because `set_len` doesn't check any of the safety mechanisms for us.

Usually, we'd need to make sure, that `res` is smaller, or equal to the capacity and that all values from the old length (`0` in this case), to the new length are set. This is safe here, since the syscall will set exactly the amount of values it returns to us. I copied this snippet from `mio` and have seen it in other tutorials as well, so this should be fine.

Next, let's look at the events we get back and start accepting and handling requests!

Handling requests

First, we simply iterate over `events` and match on the `u64` value of the events, which is the unique key we provided for each of the file descriptors we're interested in:

```
#[derive(Debug)]
pub struct RequestContext {
    pub stream: TcpStream,
    pub content_length: usize,
    pub buf: Vec<u8>,
}

impl RequestContext {
    fn new(stream: TcpStream) -> Self {
        Self {
            stream,
            buf: Vec::new(),
            content_length: 0,
        }
    }
}

fn main() -> io::Result<()> {
    ...
    let mut request_contexts: HashMap<u64, RequestContext> = HashMap::new();
    ...
        println!("requests in flight: {}", request_contexts.len());
        for ev in &events {
            match ev.u64 {
```

```
                100 => {
                    match listener.accept() {
                        Ok((stream, addr)) => {
                            stream.set_nonblocking(true)?;
                            println!("new client: {}", addr);
                            key += 1;
                            add_interest(epoll_fd, stream.as_raw_fd(), listener_read_event(key))?;
                            request_contexts.insert(key, RequestContext::new(stream));
                        }
                        Err(e) => eprintln!("couldn't accept: {}", e),
                    };
                    modify_interest(epoll_fd, listener_fd, listener_read_event(100))?;
                }
                ...
}

fn modify_interest(epoll_fd: RawFd, fd: RawFd, mut event: libc::epoll_event) -> io::Result<()> {
    syscall!(epoll_ctl(epoll_fd, libc::EPOLL_CTL_MOD, fd, &mut event))?;
    Ok(())
}
```

If we encounter `100`, we know it's our server, since that's the hard-coded key we set for it. Since we only registered interest in `read` events, this means that we can now `accept` an incoming connection without blocking, which is what we do.

If we wouldn't have gotten the notification and just called `listener.accept()`, that call would have blocked until there would have been a connection. In this case, we get notified that there is already a connection and that our call to `accept()` will return immediately.

Once we accept the connection, we set the `TcpStream`, which is the connection to the client, to `nonblocking` as well. We increment the `key`, to get another unique value and register interest in this `TcpStream`, using it's file descriptor. Again, we're interested in read events only.

This is, because we're an HTTP server and the next step after accepting the connection is to read the incoming request.

Finally, we add an entry into our `request_contexts` map. This is a very primitive store for keeping the state of all active connections, because we'll need it later on, when we want to read the request and write an answer.

The `RequestContext` struct has the `TcpStream` of the incoming connection and two fields we'll deal with later, when we read the request payload. We put this context into the map using the unique `key` as a key.

As mentioned earlier, because we use oneshot events, we need to re-register our interest in read events on our server, so we're notified of further connections. We use the `modify_interest` function for that, which does the same as `add_interest`, just that it uses the `EPOLL_CTL_MOD` flag, because the file descriptor is already known to the `epoll` instance.

OK, that's it for accepting connections. The next step is to read the incoming HTTP requests into a buffer and then, to write back a request - all in a non-blocking fashion.

```
fn main() -> io::Result<()> {
    ...
                key => {
                    let mut to_delete = None;
```

```
                    if let Some(context) = request_contexts.get_mut(&key) {
                        let events: u32 = ev.events;
                        match events {
                            v if v as i32 & libc::EPOLLIN == libc::EPOLLIN => {
                                context.read_cb(key, epoll_fd)?;
                            }
                            v if v as i32 & libc::EPOLLOUT == libc::EPOLLOUT => {
                                context.write_cb(key, epoll_fd)?;
                                to_delete = Some(key);
                            }
                            v => println!("unexpected events: {}", v),
                        };
                    }
                    if let Some(key) = to_delete {
                        request_contexts.remove(&key);
                    }
                }
    ...
}
```

If we get an event, that was NOT triggered for the key `100`, we know that it must have been triggered by one of the incoming connections, as those are the only other file descriptors we registered interest in.

Right off the bat, we define a `to_delete` variable, to have a mechanism to remove the context from our map, if the request has finished. We'll come to that part a bit later.

We check if there is a mapping for the given `key` in our context map and if so, we match on the returned `events`, which is the bitmap, which tells us what kind of event was triggered.

We'll handle the `read` case first, which is the case if `EPOLLIN` was set. In this case, we call the `read_cb` method on the request context:

```
impl RequestContext {
...
    fn read_cb(&mut self, key: u64, epoll_fd: RawFd) -> io::Result<()> {
        let mut buf = [0u8; 4096];
        match self.stream.read(&mut buf) {
            Ok(_) => {
                if let Ok(data) = std::str::from_utf8(&buf) {
                    self.parse_and_set_content_length(data);
                }
            }
            Err(e) if e.kind() == io::ErrorKind::WouldBlock => {}
            Err(e) => {
                return Err(e);
            }
        };
        self.buf.extend_from_slice(&buf);
        if self.buf.len() >= self.content_length {
            println!("got all data: {} bytes", self.buf.len());
            modify_interest(epoll_fd, self.stream.as_raw_fd(), listener_write_event(key))?;
        } else {
            modify_interest(epoll_fd, self.stream.as_raw_fd(), listener_read_event(key))?;
        }
        Ok(())
    }

    fn parse_and_set_content_length(&mut self, data: &str) {
        if data.contains("HTTP") {
            if let Some(content_length) = data
                .lines()
                .find(|l| l.to_lowercase().starts_with("content-length: "))
            {
                if let Some(len) = content_length
                    .to_lowercase()
```

```
                    .strip_prefix("content-length: ")
                {
                    self.content_length = len.parse::<usize>().expect("content-length is valid");
                    println!("set content length: {} bytes", self.content_length);
                }
            }
        }
    }
...
}
```

Alright, that's quite a bit of code, so let's step through it.

First, we define a temporary buffer to read into and call `.read()` on our saved `TcpStream`. This won't block, as we have been notified that this file descriptor is in fact readable.

Then, we see if we can parse the data to utf8 and, if so, we search for the `content-length` header and parse it's value.

Note: This is just some very dumb and minimal pseudo-HTTP parsing and **definitely** not how this should be done as it's wildly insecure and just plain wrong, but it'll do for what we try to achieve and test here.

We're just interested in the amount of data we can expect to read here, so we know when we can stop reading. Once we found the content length, we set it in the request context.

Then we move the data from the temporary buffer into the buffer inside request context. We won't do anything with it in this case, but it's good to know that we could.

Once this is done, we check if we read all the data we expected to read. If so, we call `modify_interest` again, setting our interest to `write` this time.

That's because once we're done reading the incoming request, the next step would be to do something with the payload and write something back to the client.

If we still have data to read, we just re-register the read interest.

After registering the `write` interest, we expect the `EPOLLOUT` flag to be set:

```
fn main() -> io::Result<()> {
    ...
                            v if v as i32 & libc::EPOLLOUT == libc::EPOLLOUT => {
                                context.write_cb(key, epoll_fd)?;
                                to_delete = Some(key);
                            }
                            v => println!("unexpected events: {}", v),
    ...
}
```

If that's set, we call the `write_cb` on the request context and also flag this context for deletion, since after writing, our work will be done and we have to clean up.

```
impl RequestContext {
...
    fn write_cb(&mut self, key: u64, epoll_fd: RawFd) -> io::Result<()> {
        match self.stream.write(HTTP_RESP) {
            Ok(_) => println!("answered from request {}", key),
            Err(e) => eprintln!("could not answer to request {}, {}", key, e),
```

```
        };
        self.stream.shutdown(std::net::Shutdown::Both)?;
        let fd = self.stream.as_raw_fd();
        remove_interest(epoll_fd, fd)?;
        close(fd);
        Ok(())
    }
...
}
...
fn close(fd: RawFd) {
    let _ = syscall!(close(fd));
}

fn remove_interest(epoll_fd: RawFd, fd: RawFd) -> io::Result<()> {
    syscall!(epoll_ctl(
        epoll_fd,
        libc::EPOLL_CTL_DEL,
        fd,
        std::ptr::null_mut()
    ))?;
    Ok(())
}
```

The write callback is more simple than reading. Since we got notified, that the socket is writable, we call `stream.write()` with our hard-coded HTTP response, to greet our client with a friendly `Hello`.

After that, we need to clean up the connection. First, we call `shutdown` on the `TcpStream`, closing the connection. Then, we remove interest in the file descriptor, close the file descriptor and we're done here.

Since we set `to_delete` as well, the request context will also be removed from the context map and we should have a clean slate again.

Let's see if any of this actually works and if so, if it does roughly what we set out to achieve.

Testing

We want to make sure, that the server can handle multiple requests at the same time. Not only accept multiple requests, but also process them, without getting slower if there are more than a couple of them.

One thing to note, because we completely skimped on error handling in this post, if any io errors happen (e.g. you cancel an incoming request, while we're reading), the application will panic and stop.

So let's send some longer-running requests to the server:

```
while true; do curl --location --request \
POST 'http://localhost:8000/upload' \
--form 'file=@/home/somewhere/some_image.png' \
-w ' Total: %{time_total}' && echo '\n'; done;
```

This command, sends a file (optimally a bigger one - several megabytes for example), to our server in an endless loop. We can start this script in multiple terminals and we'll have a big of IO-heavy traffic going on.

Also, with `-w %{time_total}`, we time the request and print out something like:

```
Hello Total: 1,010951
```

for every successful request. “Hello” is the response and then we print the total time the request took, which is about 1 second in this case.

One thing to notice is, that the requests are indeed being served properly and without getting slower. Also, if we check the server logs, we see something like this:

```
...
requests in flight: 6
...
```

...which tells us, that we’re indeed handling several requests concurrently with our 1 thread. Also, if you check your load-creating `while curl` scripts, the requests still take the same amount of time.

This is important, as having requests in-flight alone is meaningless, if they’re not served quickly. This proves that our solution works and we’re indeed handling many IO-heavy requests concurrently and very quickly using non-blocking IO. Great!

The full example code can be found here

Conclusion

Non-blocking IO, or asynchronous programming has been something I’ve been using on a high level for a while and in various forms, such as futures, promises, or callback-based APIs.

However, I never quite understood what’s actually happening under the hood and why this model works at all. I’m still not 100% confident on all the nitty-gritty details, but researching and writing this example have helper me a ton to get a better understanding.

I have to give a huge thanks to the people who created the tutorials and learning-friendly code bases I used to build and write this, which can be found in the resources below.

I do hope I was able to get some of my new understanding across to you, dear reader, as well and that you found this small journey enlightening, interesting, or at least entertaining. :)

My plan is to move further up the stack, exploring the depths of asynchronous programming in Rust some more.

Resources

- Code Example
- Asynchronous Programming Under Linux
- Linux Applications Performance: Part VII: epoll Servers
- Exploring Async Basics
- Epoll, Kqueue and IOCP
- Polling Rust Library
- Mio