mio

mio is a Rust crate for non-blocking I/O API interface for building **high performance I/O apps**.

1. Creating a Poll. reads events from OS and puts them in Event
2. Register an Event source
3. Create an Event loop

At the end you'll have a very small (but quick) TCP server that accepts connections and then drops (disconnects) them.

```
// `Poll` allows for polling of readiness events.
let poll = Poll::new()?;
// `Events` is collection of readiness `Event`s and can be filled by
// calling `Poll::poll`.
let events = Events::with_capacity(128);
```

Poll

Poll is a struct allows a program to monitor a large number of `event::Source`s. It waits until one of them is ready for some class of operations e.g read or write.[1]

```
let mut poll = Poll::new()?;
poll.registry().register(<event>, <token>, <interest> )?;
```

event::Source

Token

token is a wrapper around `usize` and is used as an argument to `Registry::register` and `Registry::reregister`.

epoll

The implementation of epoll uses epoll systemcall defined at `sys/epoll.h`. According to the official documentation, it is monitoring multiple file descriptions to see if `I/O` operation is possible on any of them.

- interest list (epoll set)
- ready list (a set of references to the interest list)

Other systemcalls related to epoll are the following:

- epoll_create
- epoll_create1

[1] `event token` and `interest` are to be interpolated

- epoll_cntl
- epoll_wait

epoll_create1: if the argument is 0, it has the same functionality of epoll_create. the other argument we can pass to this syscall is **FD_CLOEXEC**.
epoll_wait: blocks the current thread if no event is already available.

A code example

```
use std::{error::Error, time::Duration};

use mio::{
    net::{TcpListener, TcpStream},
    Events, Interest, Poll, Token,
};
const SERVER: Token = Token(0);
const CLIENT: Token = Token(1);

fn main() -> Result<(), Box<dyn Error>> {
    let mut poll = Poll::new()?;
    let mut events = Events::with_capacity(1024);
    let addr = "127.0.0.1:13265".parse()?;
    let mut server = TcpListener::bind(addr)?;
    let mut client = TcpStream::connect(addr)?;

    poll.registry()
        .register(&mut server, SERVER, Interest::READABLE)?;
    poll.registry()
        .register(&mut client, CLIENT, Interest::WRITABLE | Interest::READABLE)?;
    loop {
        poll.poll(&mut events, None)?;
        println!("Event");
        for event in events.iter() {
            match event.token() {
                SERVER => {
                    let connection = server.accept()?;
                    drop(connection);
                }
                CLIENT => {
                    if event.is_writable() {
                        // We can (likely) write to the socket without blocking.
                    }

                    if event.is_readable() {
                        // We can (likely) read from the socket without blocking.
                    }

                    // Since the server just shuts down the connection, let's
                    // just exit from our event loop.
```

```
                    return Ok(());
                }
                _ => {
                    unreachable!()
                }
            }
        }
    }
}

fn main1() -> Result<(), Box<dyn Error>> {
    let mut events = Events::with_capacity(1024);
    let mut poll = Poll::new()?;

    // Register `event::Source`s with `poll`.

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

    for event in events.iter() {
        println!("Got an event for {:?}", event.token());
    }
    Ok(())
}
```