

Asynchrony Deep Dive

* PoolC 양제성

Source:  2024/11/22 (Fri)

Table of Contents

Application	
Library	library calls
Kernel	system calls
Hardware	interrupts

Warm up!

Application

library calls

Library

system calls

Kernel

interrupts

Hardware

Misconceptions on Promise

```
const p = new Promise((resolve, reject) => {  
  setTimeout(() => resolve("done"), 1000)  
})
```

```
p.then(console.log)  
console.log("I'm gonna be printed first!")
```

Misconceptions on Promise

```
const p = new Promise((resolve, reject) => {  
  for (let i = 0; i < 10000000000; i++) {}  
  console.log("Hmm...")  
  resolve("done")  
});
```

```
p.then(console.log);  
console.log("Hi!");
```

Misconceptions on Promise

```
async function foo() {  
    for (let i = 0; i < 1000000000000; i++) {}  
    console.log("foo")  
}  
  
async function bar() {  
    await foo()  
    console.log("bar")  
}
```

```
bar()  
console.log("Hi!")
```

Contract of Promise

Promise doesn't make something non-blocking

“Don't run compute-heavy codes w/ Promise, rather use inherently non-blocking APIs by the runtime”

Misconceptions on Promise

```
async function foo() {  
    for (let i = 0; i < 1000000000000; i++) {}  
    console.log("foo");  
}  
  
async function bar() {  
    let a = 3; console.log("bar")  
    await foo()  
    a = 5; console.log("a =", a) // Where did `a` stored?  
}  
  
bar()  
console.log("Hi")
```


Mental model of async/await

“Cooperative (\leftrightarrow preemptive) multi-tasking”

...in a very high-level, abstracted view

Terminology Time!

Parallel vs. Concurrent

Parallel

1. A parallel system can run multiple tasks **simultaneously**
2. Tasks **necessarily** run at the same time
3. ex. Multi-core CPU

Concurrent

1. A concurrent system can **handle** multiple tasks
2. Tasks **not necessarily** run at the same time
3. ex. Time sharing system

async/sync & block/non-block

Synchronous blocking

- disk I/O (not really after Linux 2.5.23) TODO: see man page.
- Promise w/o await

Asynchronous non-blocking

- setTimeout, fetch, Promise

Synchronous blocking

- polling

Async blocking?

	Blocking	Non-blocking
Synchronous	Read/write	Read/write (O_NONBLOCK)
Asynchronous	I/O multiplexing (select/poll)	AIO

<https://developer.ibm.com/articles/l-async/>

Again, Table of Contents

Application

library calls

Library

system calls

Kernel

interrupts

Hardware

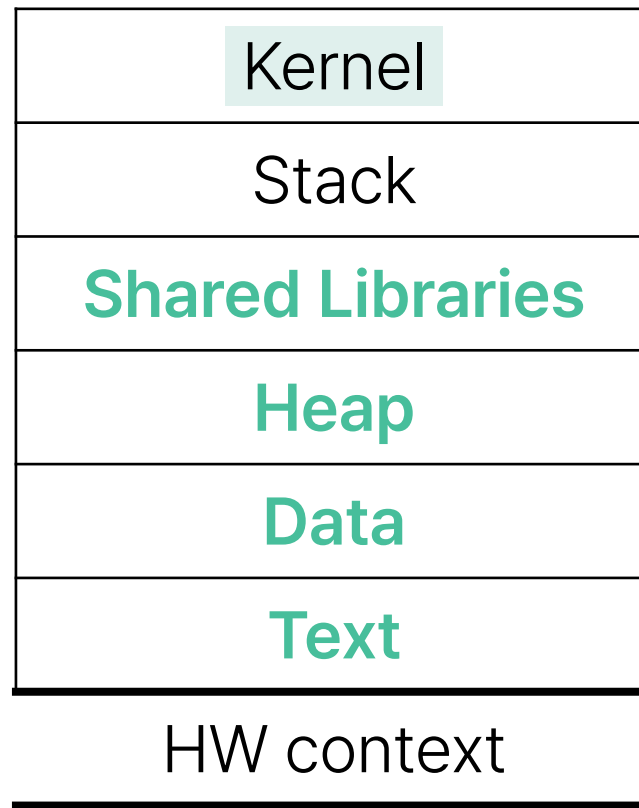
A bit of history: web servers

I/O by multi-processing

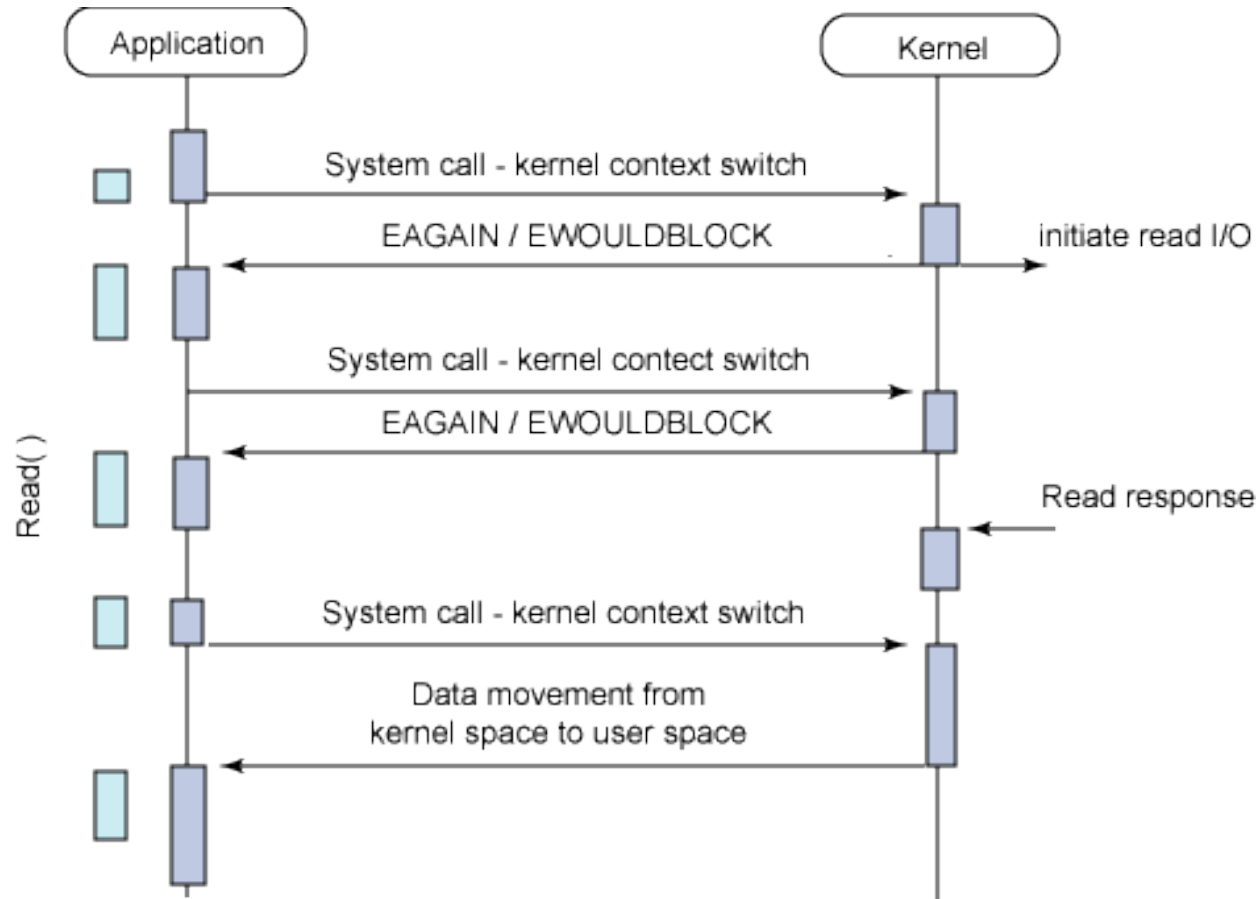
- fork + exec
- even w/ COW... too heavy!

I/O by multi-threading

- pthread_create + pthread_join
- share as much as possible



I/O multiplexing



<https://developer.ibm.com/articles/l-async/>

I/O multiplexing

“Do works only when it is possible to proceed”

UNIX I/O multiplexing: select

```
$ man 2 select
```

- allows a program to **monitor multiple file descriptors**, waiting until one or more of the file descriptors become "ready" for some class of I/O operation
- A file descriptor is considered ready if it is possible to perform a corresponding I/O operation
- WARNING: can **monitor at most FD_SETSIZE (1024) file descriptors...** and this limitation will not change. All modern applications should instead use poll(2) or epoll(7)...

UNIX I/O multiplexing: select

CAUTION: NOT 100% CORRECT!

```
fd_set reads; FD_SET(server_socket, &reads);

while (true) {
    select(server_socket + 1, &reads, ...) // block
    for (auto fd : reads) {
        if (FD_ISSET(fd, &reads)) {
            if (fd == server_socket) {
                client_socket = accept(server_socket, ...);
                FD_SET(client_socket, &reads);
            } else
                read(fd, buffer, ...);
        }
    }
}
```

Linux I/O multiplexing: epoll

\$ man 7 epoll (since Linux 2.5.45)

- The epoll API performs a similar task to poll(2): **monitoring multiple file descriptors** to see if I/O is possible on any of them
- The epoll API can be used either as an **edge-triggered** or a **level-triggered** interface and **scales well** to large numbers of watched file descriptors

Linux I/O multiplexing: epoll

CAUTION: NOT 100% CORRECT!

```
epfd = epoll_create(EPOLL_SIZE);  
ep_events = (epoll_event*) malloc(sizeof(struct epoll_event) * EPOLL_SIZE);  
struct epoll_event event;  
epoll_ctl(epfd, EPOLL_CTL_ADD, server_socket, &event);
```

Linux I/O multiplexing: epoll

CAUTION: NOT 100% CORRECT!

```
while (true) {
    event_count = epoll_wait(epfd, ep_events, EPOLL_SIZE, ...); // block
    for (int i = 0; i < event_count; i++) {
        if (ep_events[i].data.fd == server_socket) {
            client_socket = accept(server_socket, ...);
            ...
            event.data.fd = client_socket;
            epoll_ctl(epfd, EPOLL_CTL_ADD, client_socket, &event);
        } else
            read(ep_events[i].data.fd, buffer, ...);
    }
}
```

Linux I/O multiplexing: `epoll`

level-triggered vs edge-triggered

Level-triggered

- fd is considered ready if it is possible to perform a corresponding I/O operation
- A partially read socket is also read in the next loop (same with `select`)

Edge-triggered

- fd is considered ready if the I/O event has occurred since the last `epoll_wait` call
- partially read socket won't be read in the next loop

Linux I/O multiplexing: epoll

level-triggered vs edge-triggered

Caution!

- When using edge-triggered mode, always make sure to use non-blocking I/O (for sockets, use `O_NONBLOCK`)
- You also need to read/write until `EWouldBlock` is returned
- With this contract, it gives you more efficiency ($O(1)$ multiplexing where N is the number of file descriptors)

Linux Async I/O: aio

`$ man 7 aio (since Linux 2.5.23)`

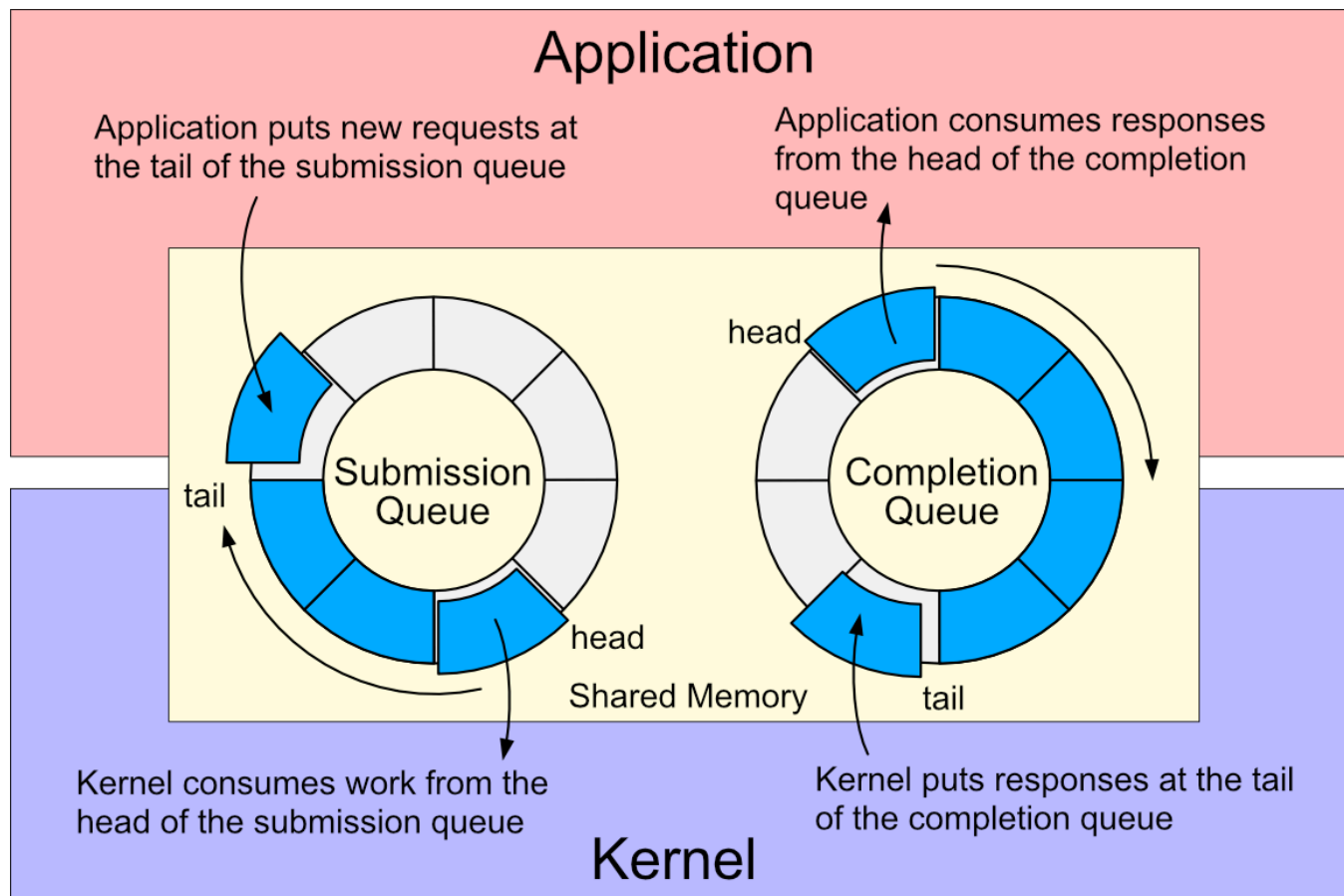
- allows applications to initiate one or more I/O operations that are performed asynchronously (i.e., **in the background**)
- The application can elect to be notified of completion of the I/O operation in a variety of ways:
 - by delivery of a signal
 - by instantiation of a thread
 - or no notification at all

Linux Async I/O: `io_uring`

\$ man 7 `io_uring` (since Linux 5.1)

- allows the user to submit one or more I/O requests, which are processed asynchronously **without blocking the calling process**
- uses 2 buffers called "ring buffer" which are **shared between user and kernel space**; avoiding the overhead of copying data between them, where possible
- "The biggest limitation of `aio` is that it only supports async IO for `O_DIRECT` access, which bypasses cache and has size/alignment restraints"
[Axboe(2019). "Efficient IO with `io_uring`"]

Linux Async I/O: `io_uring`



<https://developers.redhat.com/articles/2023/04/12/why-you-should-use-iouring-network-io>

Linux Async I/O: liburing

```
struct io_uring_sqe sqe;  
struct io_uring_cqe cqe;  
/* get an sqe and fill in a READV operation */  
sqe = io_uring_get_sqe(&ring);  
io_uring_prep_readv(sqe, fd, &iovec, 1, offset);  
/* tell the kernel we have an sqe ready for consumption */  
io_uring_submit(&ring);  
/* wait for the sqe to complete */  
io_uring_wait_cqe(&ring, &cqe);  
/* read and process cqe event */  
app_handle_cqe(cqe);  
io_uring_cqe_seen(&ring, cqe);
```

Linux Async I/O: `io_uring`

Reduced system calls

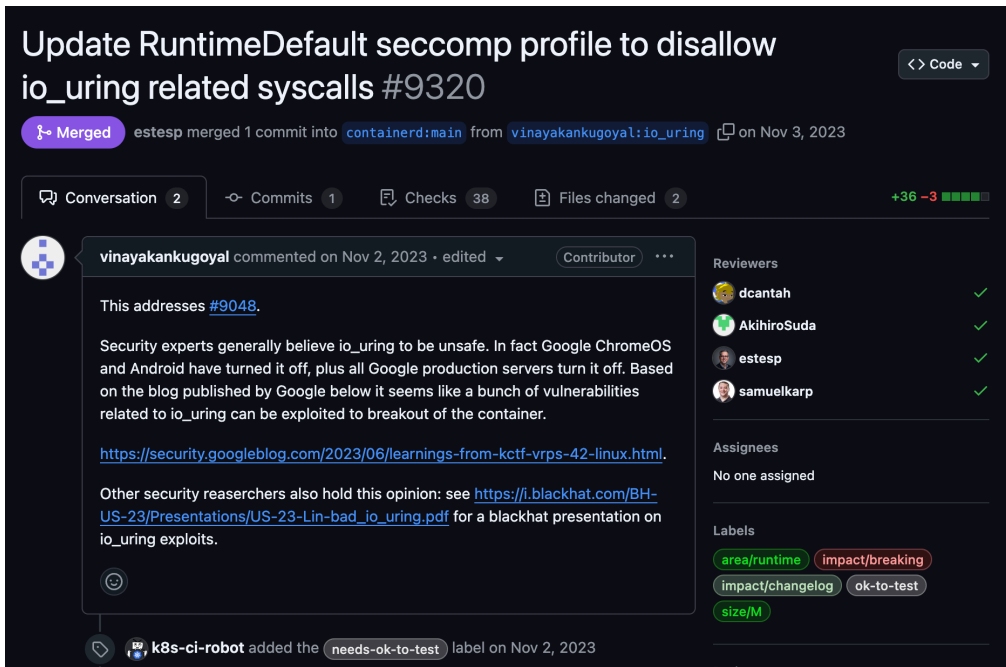
- system calls(e.g. `read`, `write`, ...) are packed(not an official term) into **a single system call**, `io_uring_enter`

Even more reduced system calls

- supports **"Kernel side polling"**...
 - # of `io_uring_enter` calls are even reduced
- the kernel spawns a dedicated kernel thread to poll the submission queue

Linux Async I/O: `io_uring`

Sharing always causes some problems



Limiting `io_uring`

To protect our users, we decided to limit the usage of `io_uring` in Google products:

- **ChromeOS:** We [disabled](#) `io_uring` (while we explore new ways to sandbox it).
- **Android:** Our [seccomp-bpf filter](#) ensures that `io_uring` is unreachable to apps. Future Android releases will use SELinux to [limit `io_uring` access to a select few system processes](#).
- **GKE AutoPilot:** We are investigating disabling `io_uring` by default.
- It is disabled on production Google servers.

**So... why did we go through all the way down here?
Let's take a breath and think about the original goal...**

Async runtime implementation

"Asynchronous systems are implemented
with the help of the kernel!"

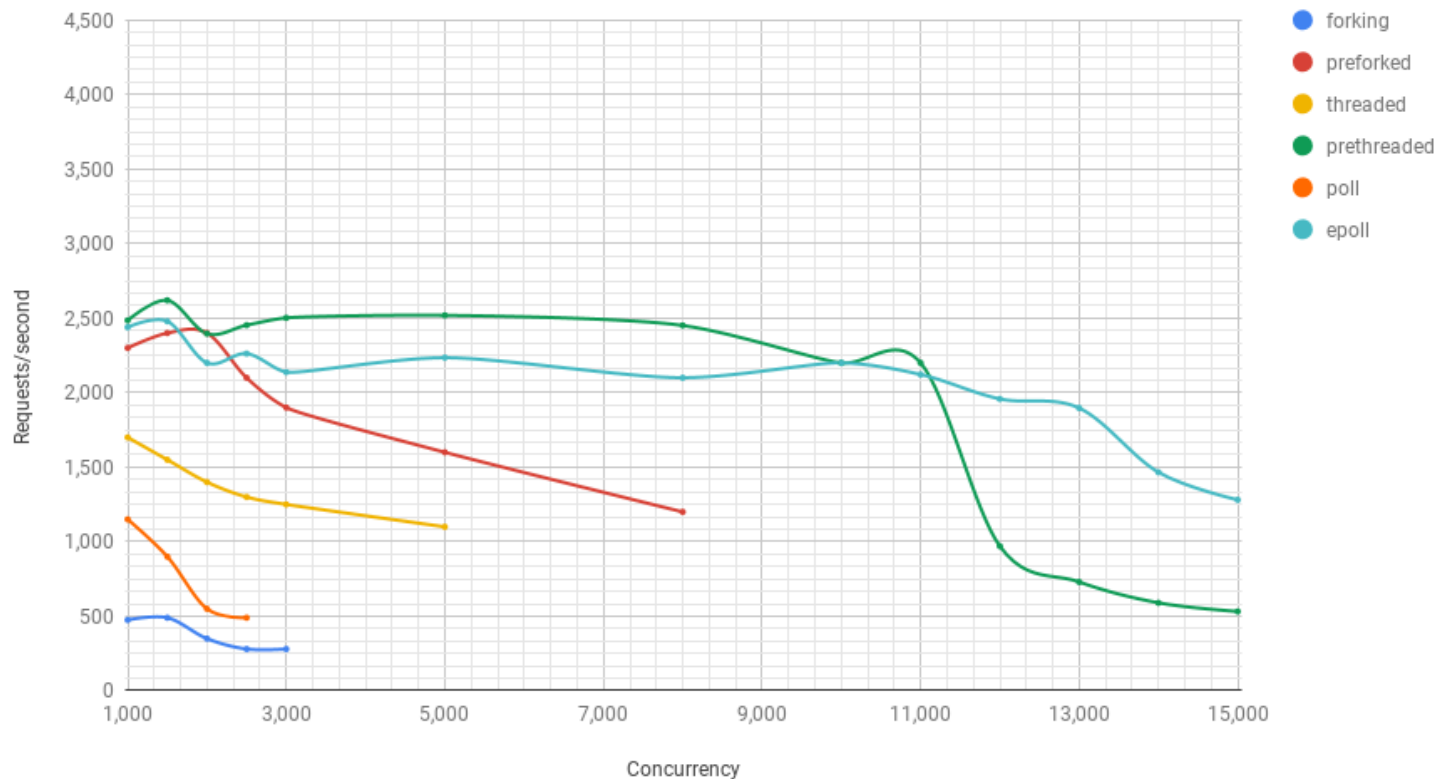
Async runtime implementation

Various ways to implement async runtime

- forking
- pre-forked
- threaded
- pre-threaded (a.k.a. thread pool)
- I/O multiplexing

Async runtime implementation

Requests/sec Vs. Concurrency (1,000+ concurrent connections)



https://unixism.net/loti/async_intro.html

Async runtime implementation

1. "As you can see, prethreaded, or **the thread pool based web server gives the epoll(7) based server a run for its money up until a concurrency of 11,000 users** in this particular benchmark.
2. And that is a lot of concurrent users. ...
3. This is very significant, given that in terms of complexity, **thread pool based programs are way easier to code** compared to their asynchronous counterparts.
4. This also means **they are way easier to maintain** as well, since they are naturally a lot easier to understand."

Async runtime implementation

Node.js

- use a **cross-platform library** libuv as its event loop
- libuv uses OS-specific I/O multiplexing system calls to handle multiple I/O operations
 - Linux: `epoll`, `io_uring` (since Node.js 20.3.0)
 - BSD: `kqueue`
 - Windows: `IOPCP`
- libuv also maintains a thread pool(!) for DNS operations and **file system operations**

Async runtime implementation

Let's dig in and see how `libuv` gives us the asynchronous capability!

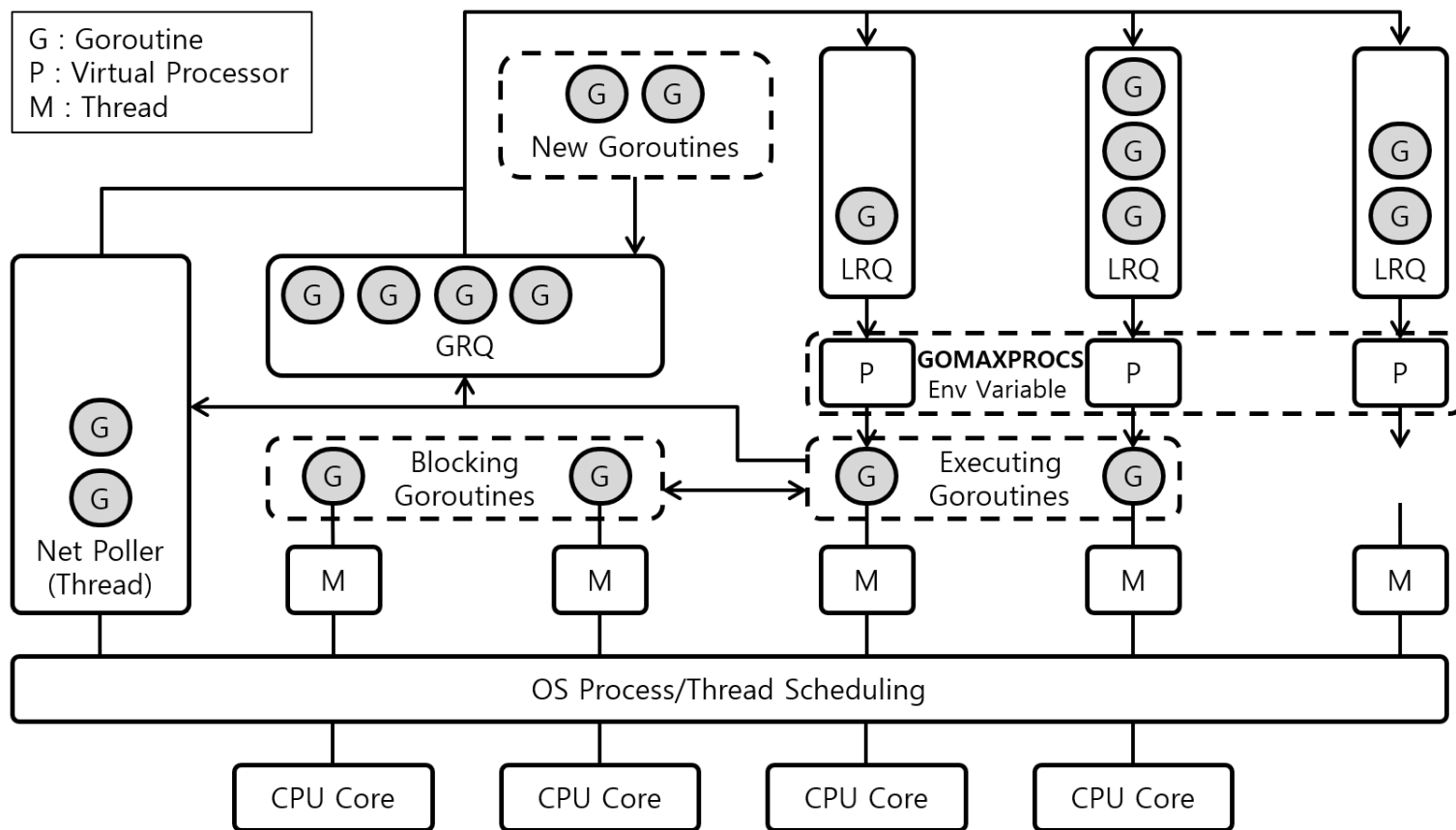
<https://github.com/J3m3/node-experiment/>

Async runtime implementation

Golang

- use **green thread** model to provide async capability
- **M:N threading model** (M user threads → N kernel threads)
- goroutines, **stackful coroutines!**
- delegates inherently blocking system calls to different OS threads
- **net poller** thread for I/O multiplexing

Async runtime implementation



<https://velog.io/@khsb2012/go-goroutine>

Async runtime implementation

Rust

- green thread(!) model (until Rust 0.9)
- **stackless coroutine** approach
- Future trait (something like Promise but **inert**)
- Rust **standard library does not provide async runtime**;
they are provided by external crates like tokio, async-std, etc.

Async runtime implementation

Rust (tokio crate)

- one of the **executor** of Future (again, Future is inert)
- uses mio crate for **I/O multiplexing**
- multiple executor instances can run in **parallel**
- **Async mutexes?**

“The spawned task may be executed on the same thread as where it was spawned, or it may execute on a different runtime thread. The task can also be moved between threads after being spawned.”

Async runtime implementation

Async mutexes?

```
// Let's assume we use a single-threaded runtime
async fn main() {
    let x = Arc::new(Mutex::new(0));

    let x1 = Arc::clone(&x);
    tokio::spawn(async move {
        let mut x = x1.lock(); // mutex locked
        some_async_fn().await; // yield
    });

    let x2 = Arc::clone(&x);
    tokio::spawn(async move {
        let mut x = x2.lock(); // deadlock!
    });
}
```

Async runtime implementation

Async mutexes?

```
// Let's assume we use a single-threaded runtime
async fn main() {
    let x = Arc::new(Mutex::new(0));

    let x1 = Arc::clone(&x);
    tokio::spawn(async move {
        *x1.lock() += 1;
    });

    let x2 = Arc::clone(&x);
    tokio::spawn(async move {
        *x2.lock() -= 1; // What about now?
    });
}
```

Choosing the right I/O strategy

CPU-bound tasks

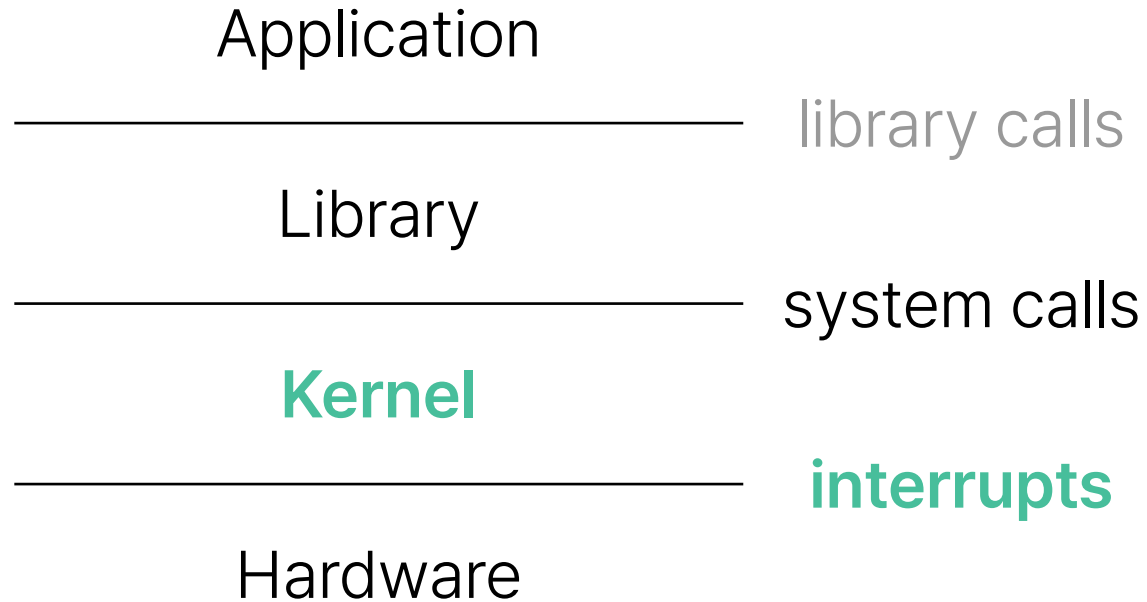
- thread pool is suitable
- what about single-threaded I/O multiplexing (e.g. Node.js)?

I/O-bound tasks

- single-threaded I/O multiplexing is suitable
- what about thread pools?

Actually, we can mix and match different strategies!

Again * 2, Table of Contents

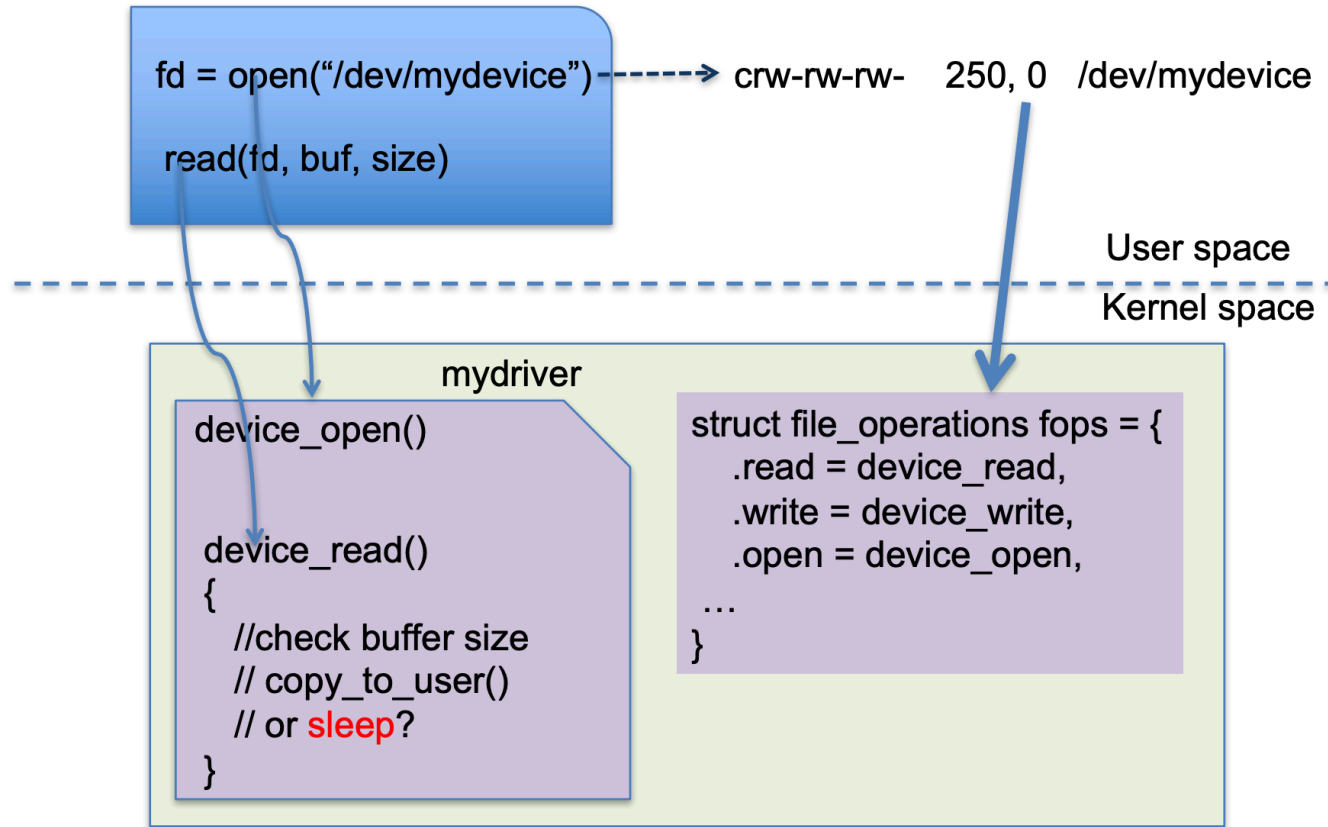


Interrupts

How do we communicate with hardware?

- interrupts!
- interrupt handlers, a part of a device driver, are executed when CPU receives an interrupt signal
- let's briefly see how interrupts are combined with I/O multiplexing implementation

Device driver side



Linux Device Driver by Prof. Yan Luo

poll syscall implementation

Some of kernel functions involved in poll syscall

- sys_poll syscall handler
- do_poll kernel function
- do_pollfd kernel function
- xxx_poll function defined in xxx device driver
- poll_wait function called by xxx_poll
- xxx device driver's wake_up function in ISR

poll syscall implementation

foreach fd (given from user space):

 find device corresponding to fd

 call device poll function to setup wait queues \

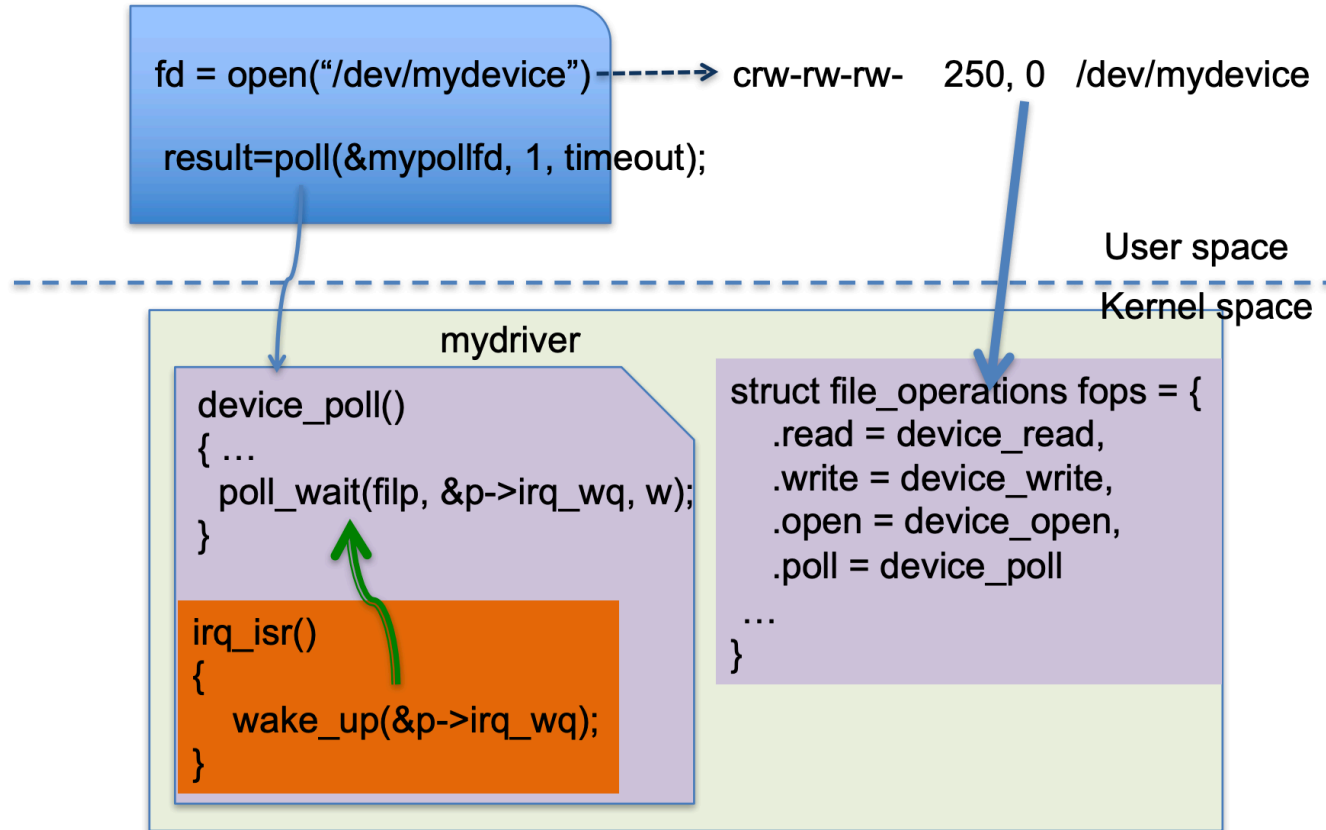
 (with poll_wait) and to collect its "ready-now" mask

while time remaining in timeout and no devices are ready:

 sleep

return from system call (either due to timeout or to ready devices)

poll in device driver



Linux Device Driver by Prof. Yan Luo

poll in device driver

```
560 static __poll_t uio_poll(struct file *filep, poll_table *wait)
561 {
562     struct uio_listener *listener = filep->private_data;
563     struct uio_device *idev = listener->dev;
564     __poll_t ret = 0;
565
566     mutex_lock(&idev->info_lock);
567     if (!idev->info || !idev->info->irq)
568         ret = -EIO;
569     mutex_unlock(&idev->info_lock);
570
571     if (ret)
572         return ret;
573
574     poll_wait(filep, &idev->wait, wait);
575     if (listener->event_count != atomic_read(&idev->event))
576         return EPOLLIN | EPOLLRDNORM;
577     return 0;
578 }
```

<https://elixir.bootlin.com/linux/v6.12.1/source/drivers/uio/uio.c>