hepsiburada

Emre YALVAÇ

Senior Software Engineer - Hepsiburada

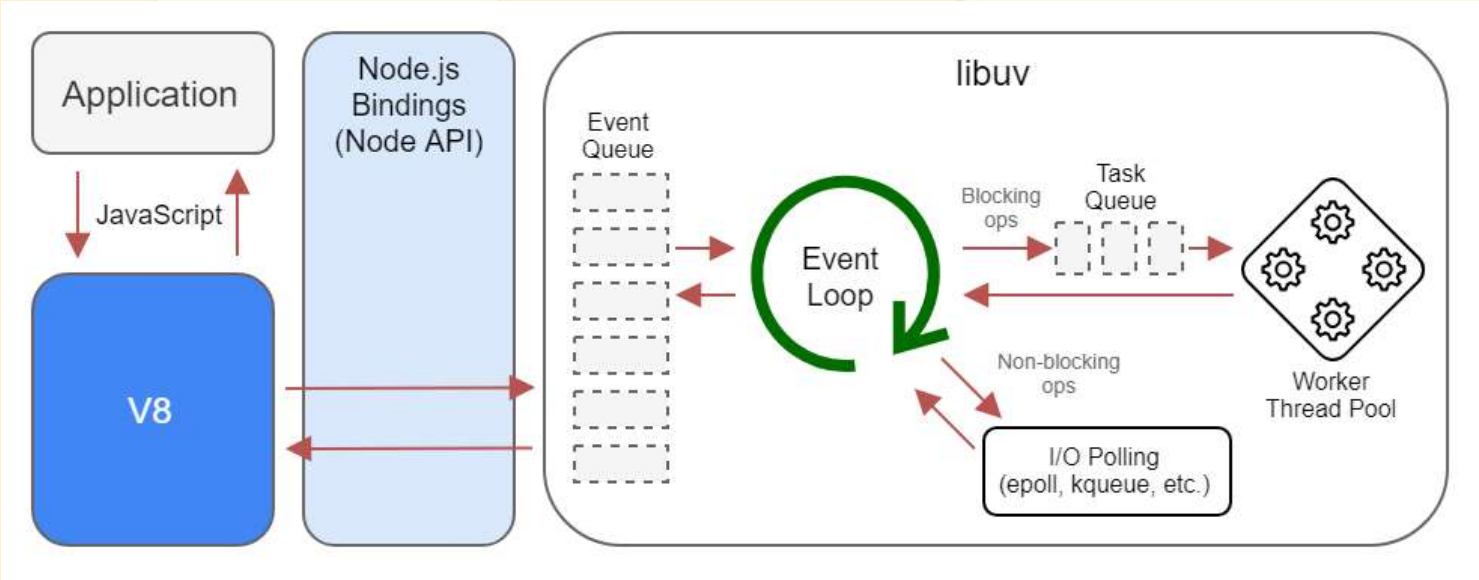
Core of Node.JS: Libuv

github/emreyalvac

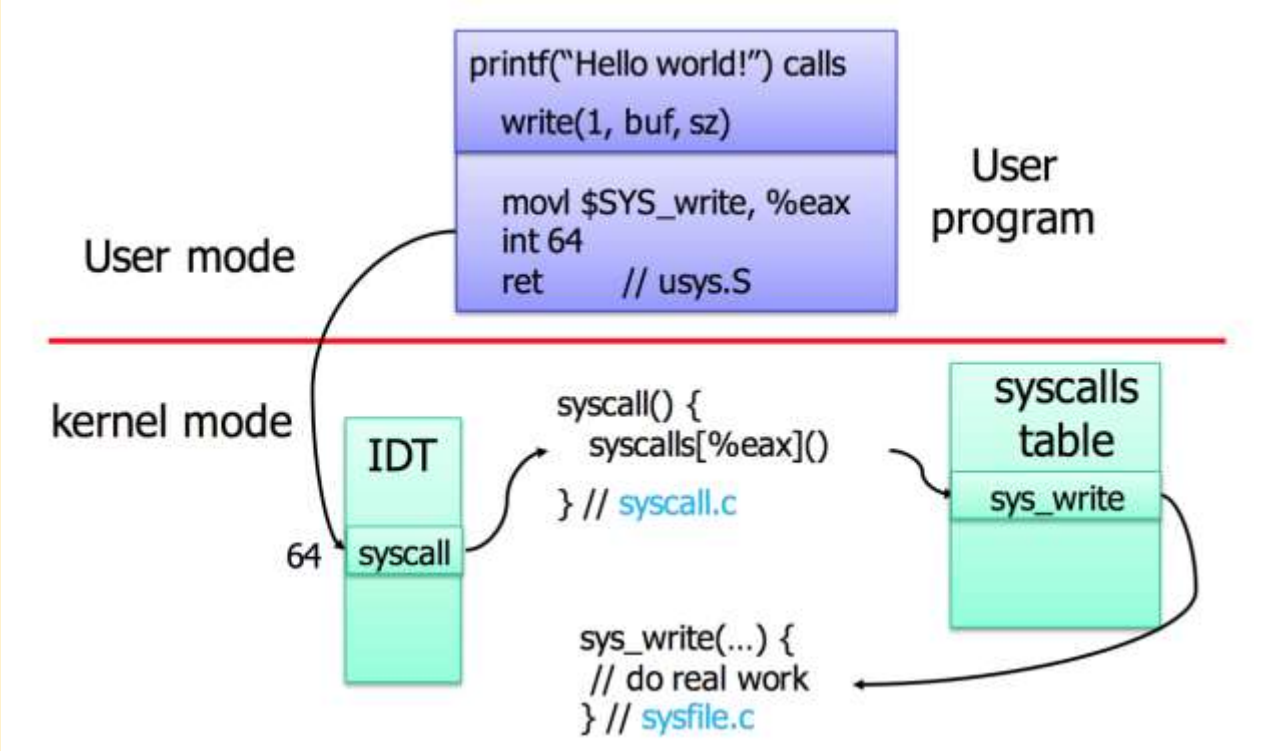
JSKonf

hepsiburada

Runtime


Application
JavaScript
V8
Node.js
Bindings
(Node API)
libuv
Event
Queue
Event
Loop
Blocking
ops
Task
Queue
Worker
Thread Pool
Non-blocking
ops
I/O Polling
(epoll, kqueue, etc.)

Syscall


printf("Hello world!") calls
write(1, buf, sz)
movl $SYS_write, %eax
int 64
ret // usys.S
User program
User mode
kernel mode
IDT
64
syscall
syscall() {
syscalls[%eax]()
} // syscall.c
syscalls table
sys_write
sys_write(...) {
// do real work
} // sysfile.c

epoll(7) — Linux manual page

NAME | SYNOPSIS | DESCRIPTION | VERSIONS | STANDARDS | HISTORY | NOTES | SEE ALSO

Search online pages

NAME top

epoll - I/O event notification facility

SYNOPSIS top

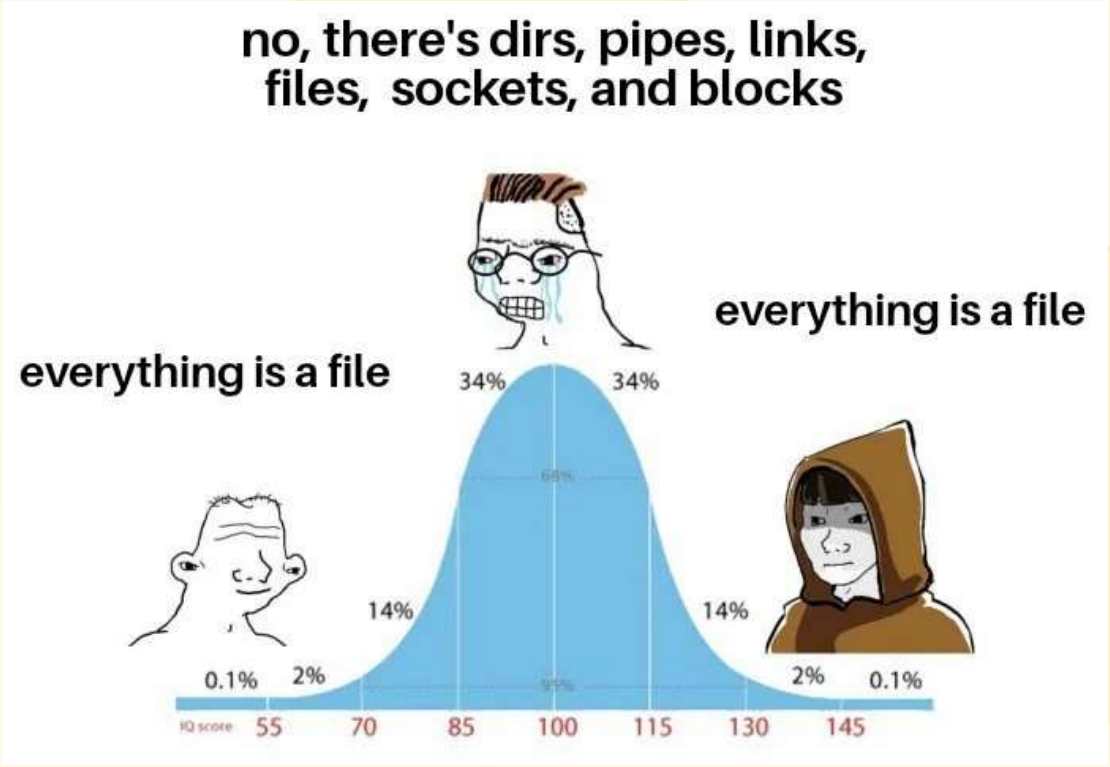
```
#include <sys/epoll.h>
```

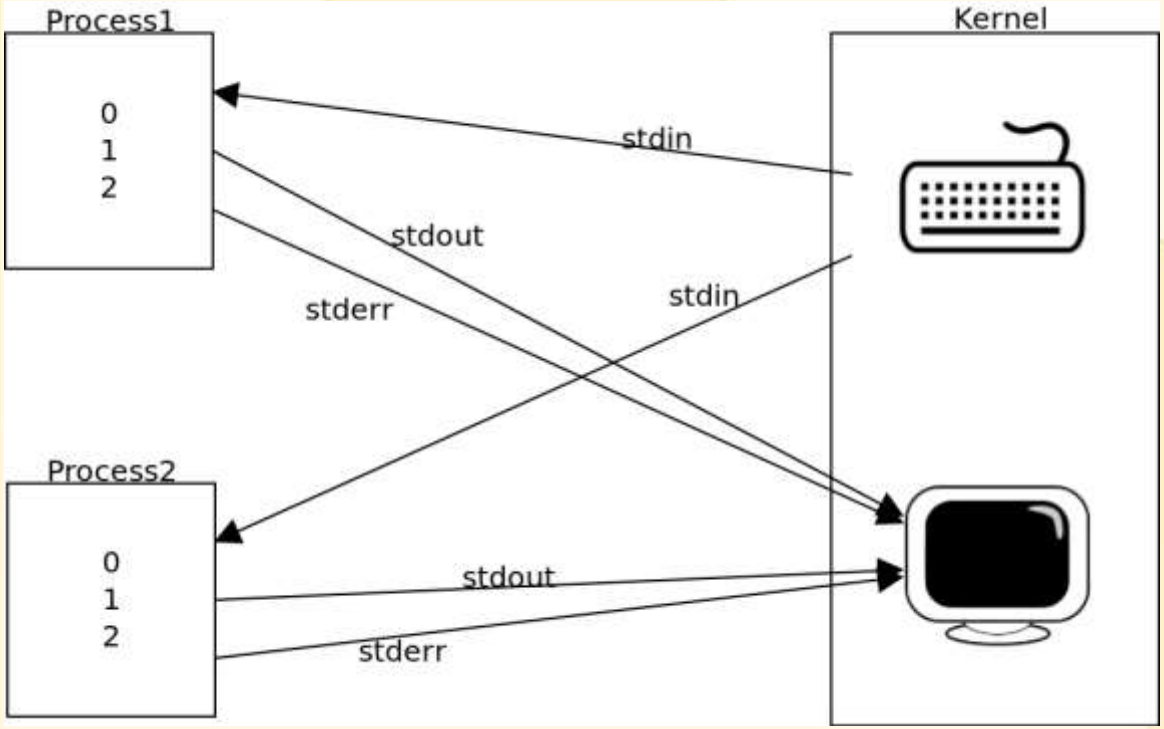
DESCRIPTION top

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.

hepsiburada

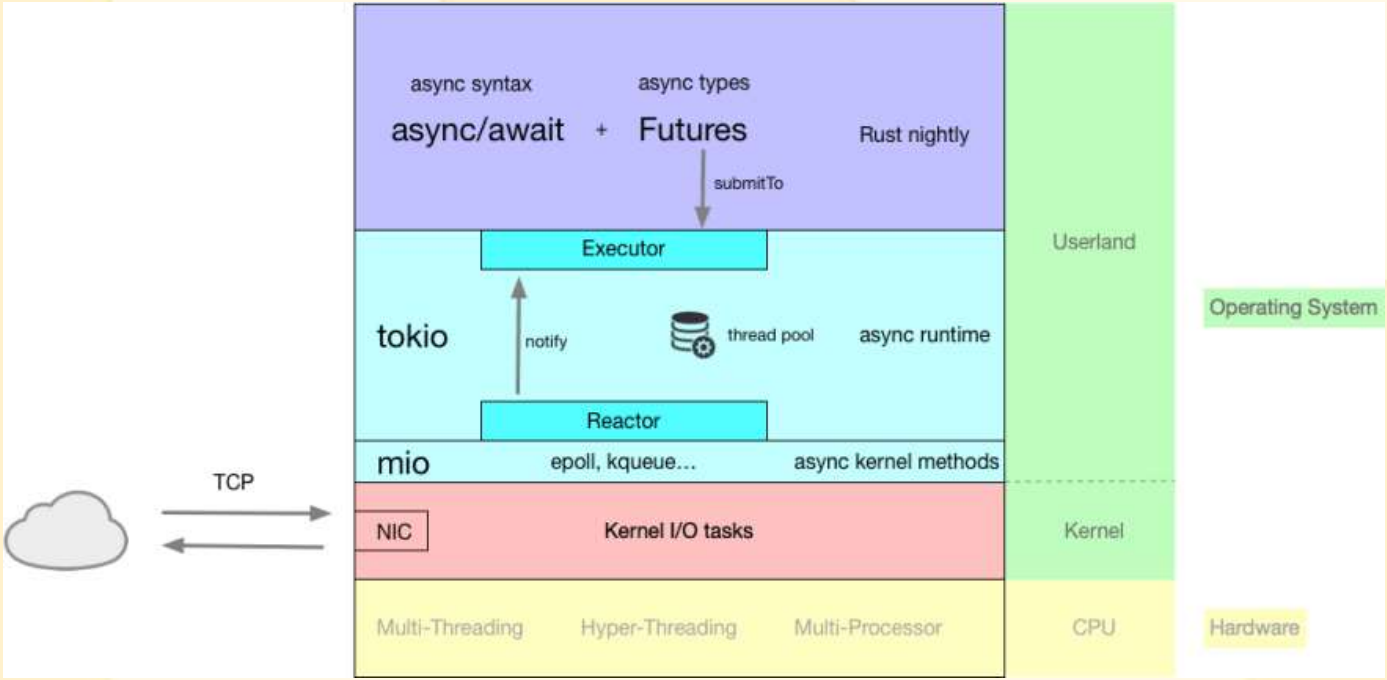
File Descriptor


no, there's dirs, pipes, links,
files, sockets, and blocks
everything is a file
everything is a file
34%
34%
14%
14%
2%
2%
0.1%
0.1%
IQ score
55
70
85
100
115
130
145


Process1
0
1
2
Kernel
stdin
stdout
stderr
stdin
Process2
0
1
2
stdout
stderr

hepsiburada

Asynchronous Programming


async syntax
async types
async/await
+
Futures
Rust nightly
submitTo
Executor
tokio
notify
thread pool
async runtime
Reactor
mio
epoll, kqueue…
async kernel methods
TCP
NIC
Kernel I/O tasks
Multi-Threading
Hyper-Threading
Multi-Processor
Userland
Kernel
CPU
Operating System
Hardware

hepsiburada

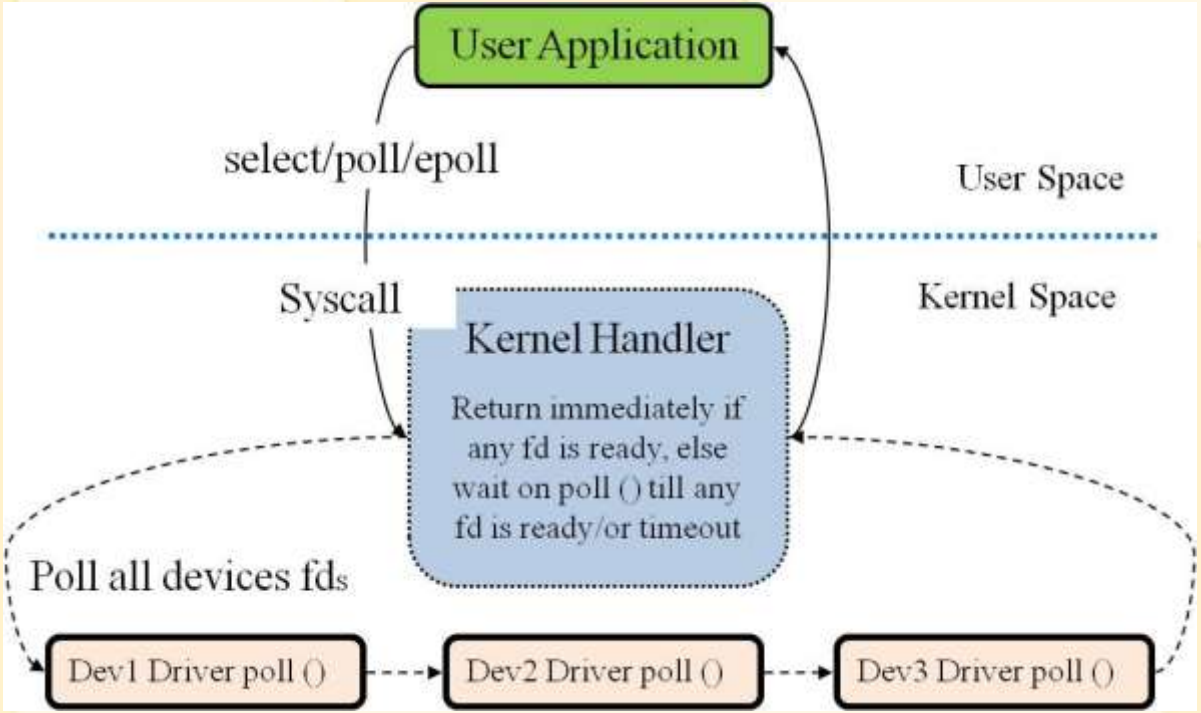
C10K Problem

http://www.kegel.com/c10k.html

And computers are big, too. You can buy a 1000MHz machine with 2 gigabytes of RAM and an 1000Mbit/sec Ethernet card for $1200 or so. Let's see - at 20000 clients, that's 50KHz, 100Kbytes, and 50Kbits/sec per client. It shouldn't take any more horsepower than that to take four kilobytes from the disk and send them to the network once a second for each of twenty thousand clients. (That works out to $0.08 per client, by the way. Those $100/client licensing fees some operating systems charge are starting to look a little heavy!) So hardware is no longer the bottleneck.

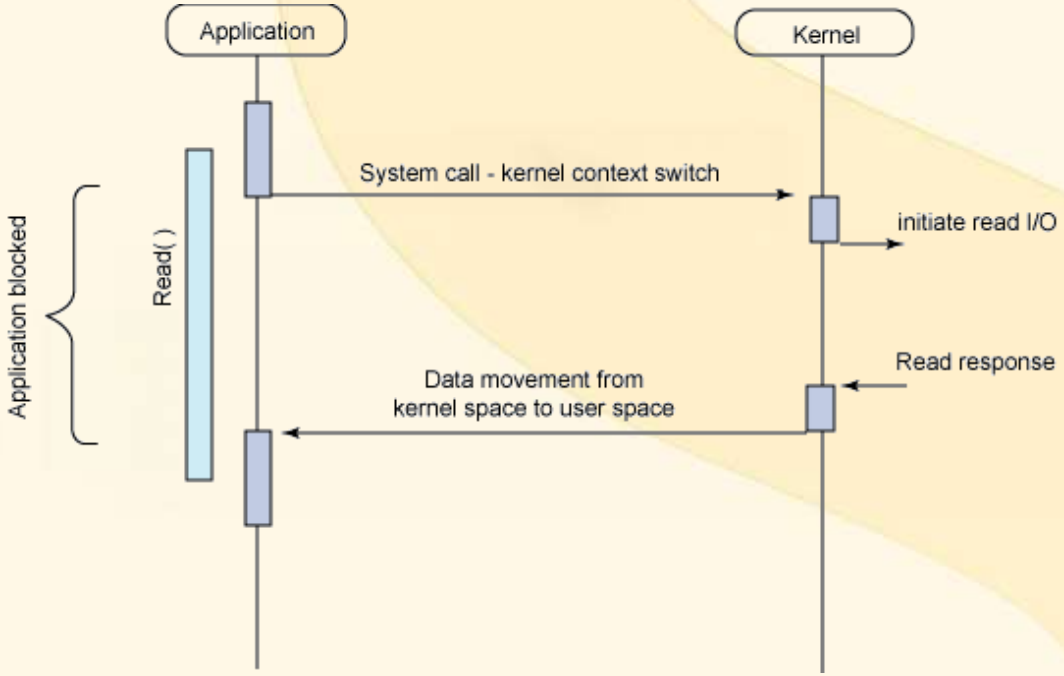
hepsiburada

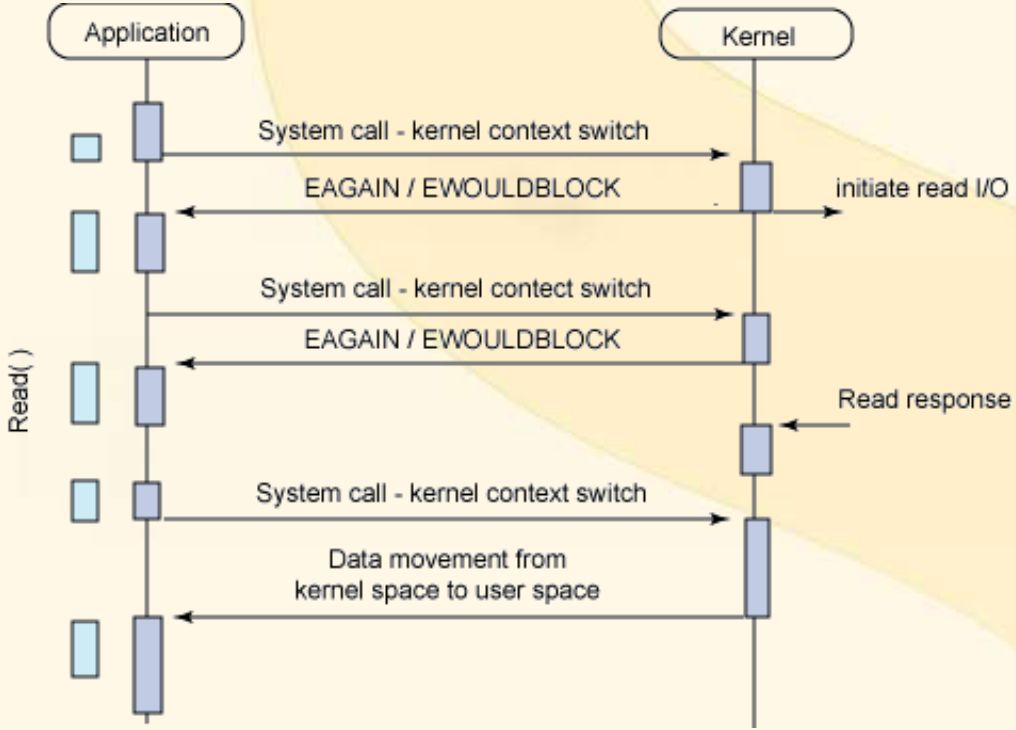
I/O Multiplexing


User Application
select/poll/epoll
User Space
Syscall
Kernel Space
Kernel Handler
Return immediately if any fd is ready, else wait on poll () till any fd is ready/or timeout
Poll all devices fds
Dev1 Driver poll ()
Dev2 Driver poll ()
Dev3 Driver poll ()

hepsiburada

Blocking & Non-Blocking I/O


Application
Kernel
System call - kernel context switch
initiate read I/O
Application blocked
Read( )
Read response
Data movement from
kernel space to user space


Application
Kernel
System call - kernel context switch
EAGAIN / EWOULDBLOCK
initiate read I/O
System call - kernel contect switch
EAGAIN / EWOULDBLOCK
Read( )
Read response
System call - kernel context switch
Data movement from
kernel space to user space

```c
int fd = open("file.txt", O_RDONLY);
char buffer[128];
int bytes_read = read(fd, buffer, sizeof(buffer));

close(fd);
```

```c
int fd = open("file.txt", O_RDONLY | O_NONBLOCK);
char buffer[128];
int bytes_read;
while ((bytes_read = read(fd, buffer, sizeof(buffer))) == -1 && errno == EAGAIN) {
  //
}
close(fd);
```

hepsiburada

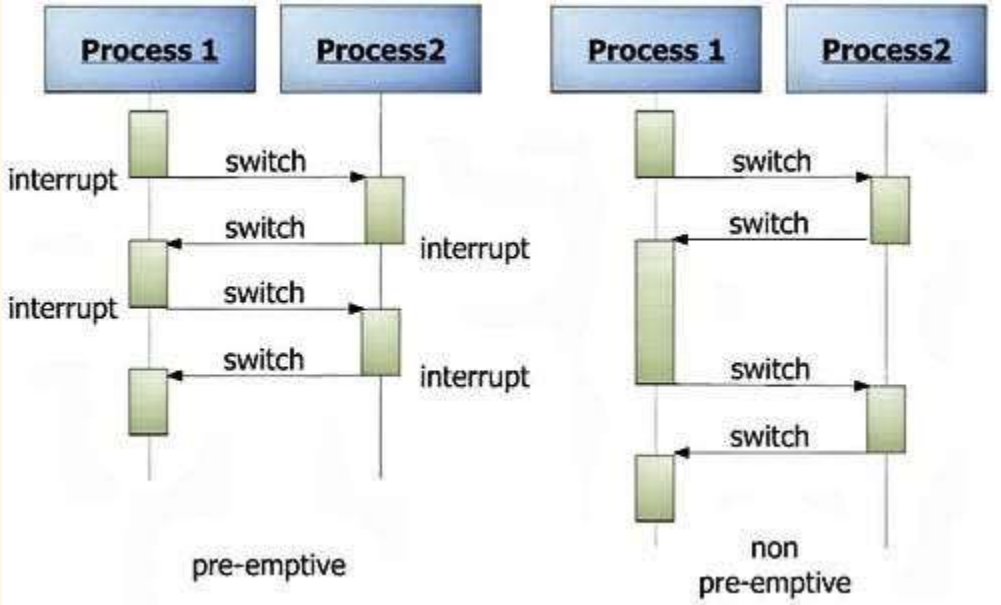
Cooperative & Preemptive Multitasking

Cooperative

scheduled by processes
can use only 1 core
may have user interface

Preemptive

scheduled by os
can use all cores available
cannot have user interface


Process 1
Process2
interrupt
switch
switch
interrupt
interrupt
switch
switch
interrupt
pre-emptive
Process 1
Process2
switch
switch
switch
switch
non
pre-emptive

hepsiburada

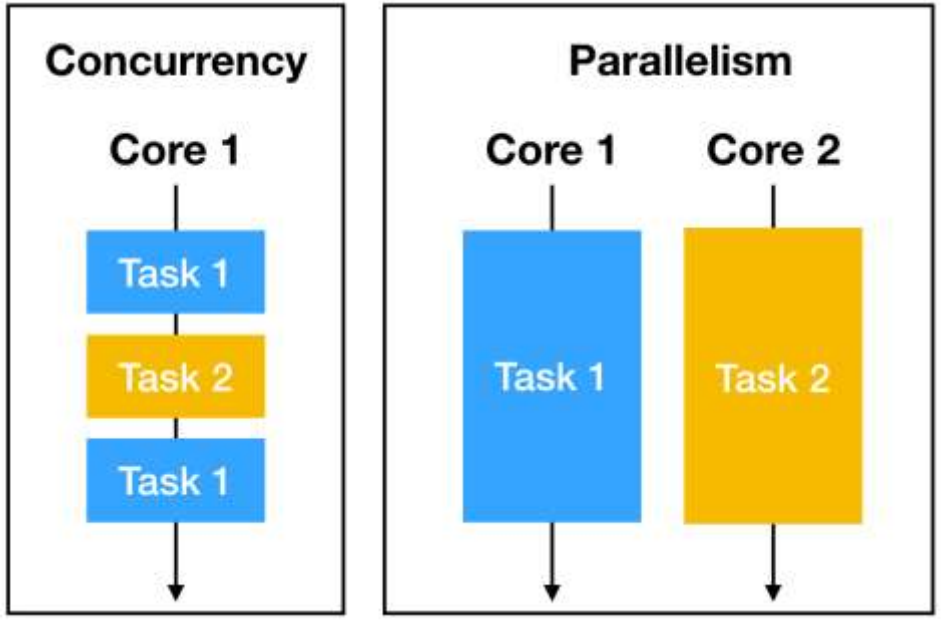
Concurrency & Parallelism

Parallelism

do a lot of things at once
deterministic
cannot be done on a single CPU

Concurrency

trying to do a lot of things at once
non-deterministic
can be done on a single CPU


Concurrency
Core 1
Task 1
Task 2
Task 1
Parallelism
Core 1
Core 2
Task 1
Task 2

hepsiburada

Native & Green Threads

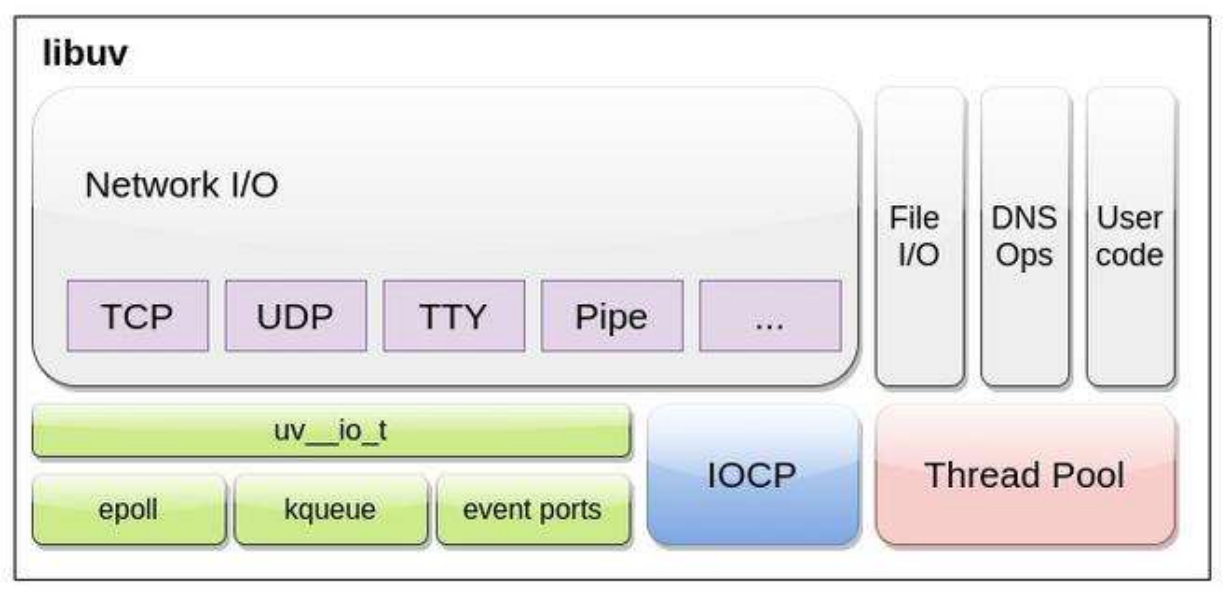
Native

os managed
direct access to hardware resources
true concurrency and parallelism
high resource usage
Expensive due to context-switch

Green

managed by user level
simulated in a single os thread
lower resource usage
more efficient in context switching

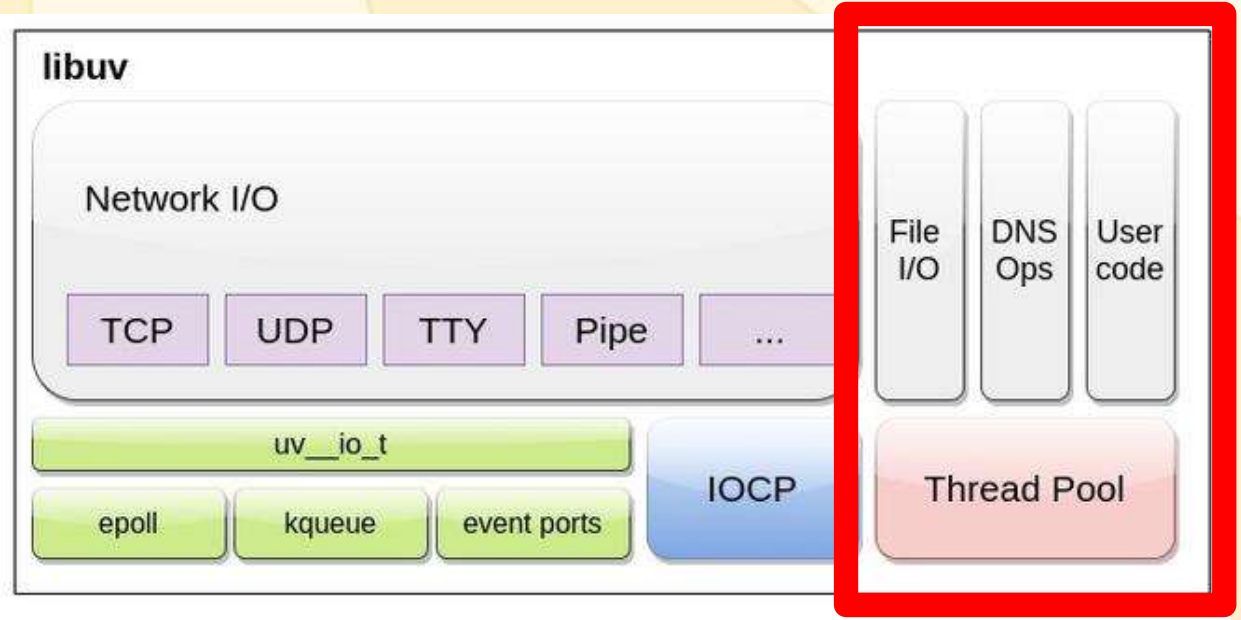
libuv


libuv
Network I/O
TCP
UDP
TTY
Pipe
...
File I/O
DNS Ops
User code
uv__io_t
epoll
kqueue
event ports
IOCP
Thread Pool

```
const OS = require('os')
process.env.UV_THREADPOOL_SIZE = OS.cpus().length
```

hepsiburada

Thread Pool


libuv
Network I/O
TCP
UDP
TTY
Pipe
...
uv__io_t
epoll
kqueue
event ports
IOCP
File I/O
DNS Ops
User code
Thread Pool

hepsiburada

Event Loop

Kernel Queue

kqueue

epoll

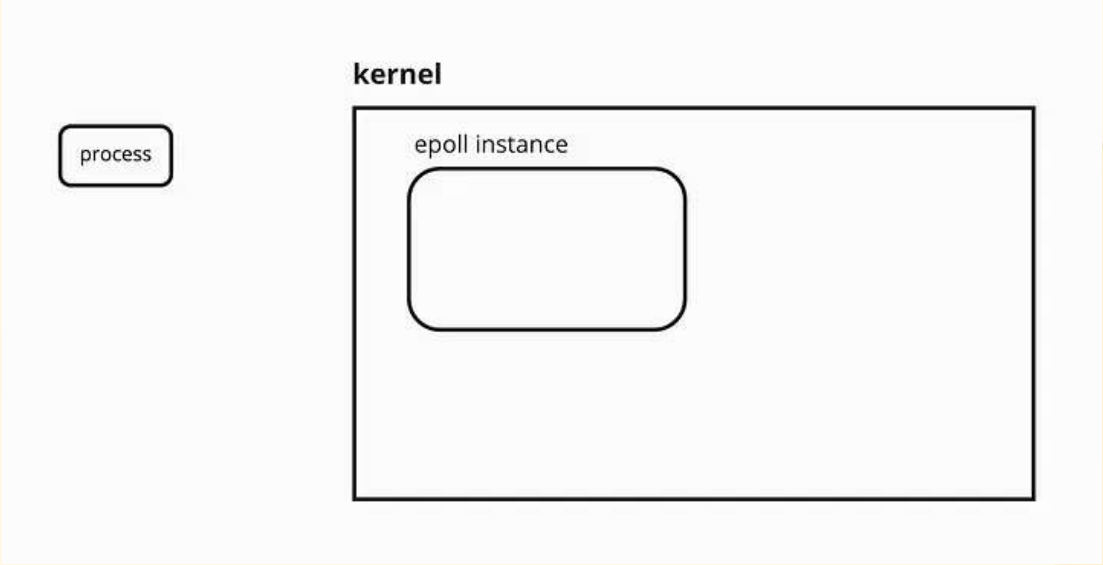
IOCP

uv_run

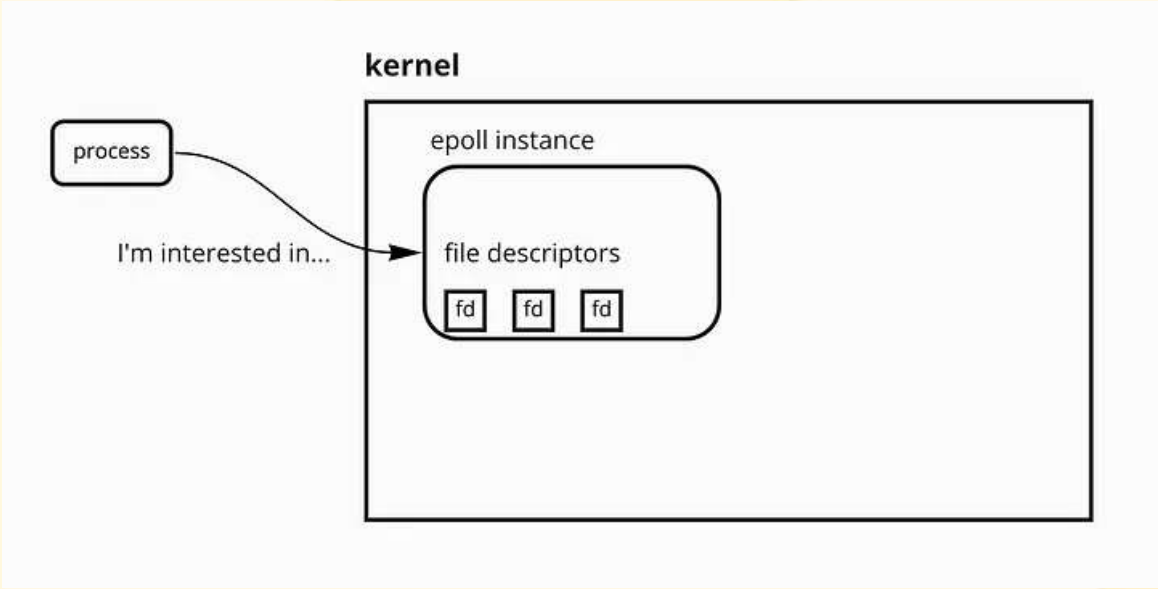
```
while there are still events to process:
    e = get the next event
    if there is a callback associated with e:
        call the callback
```

hepsiburada

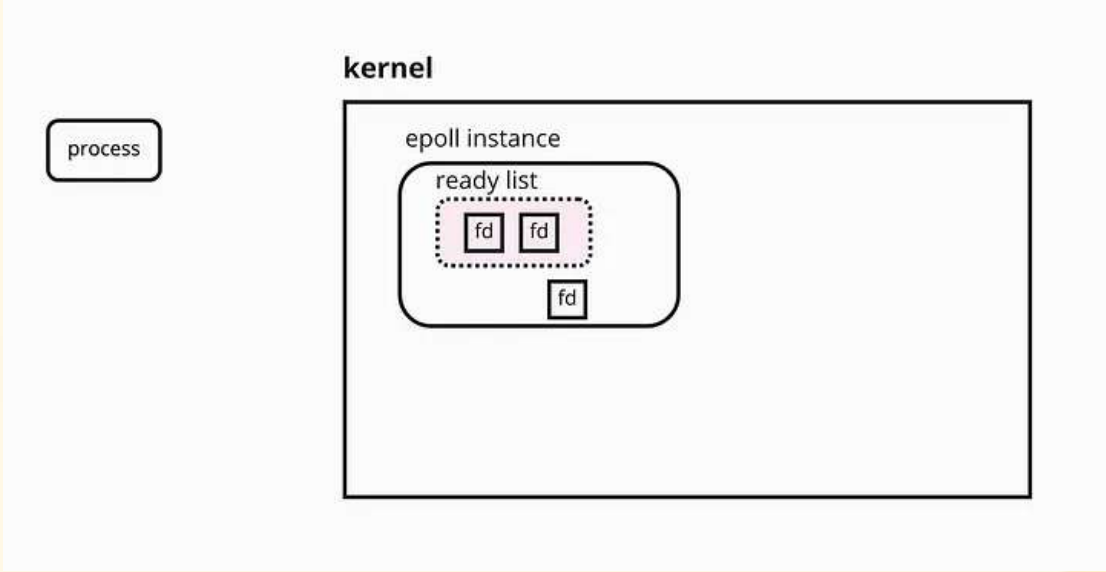
epoll

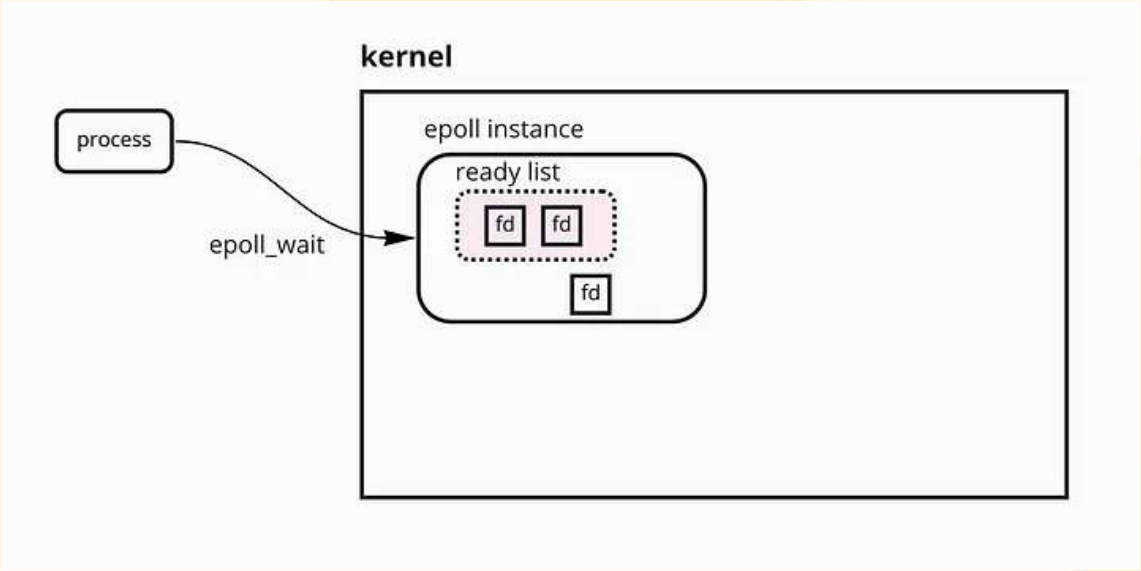


https://man7.org/linux/man-pages/man2/epoll_create.2.html

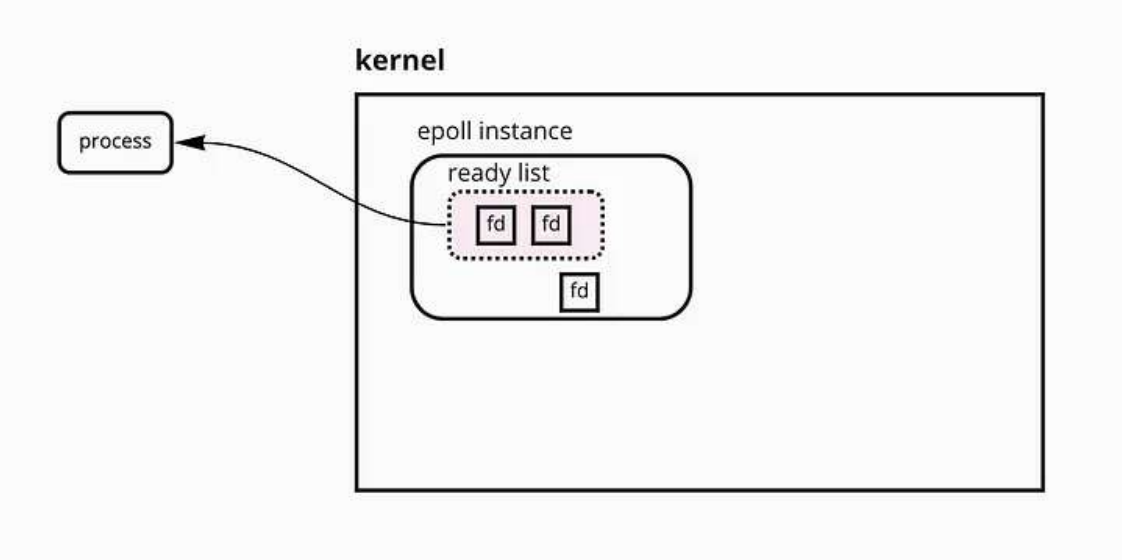


https://man7.org/linux/man-pages/man2/epoll_ctl.2.html

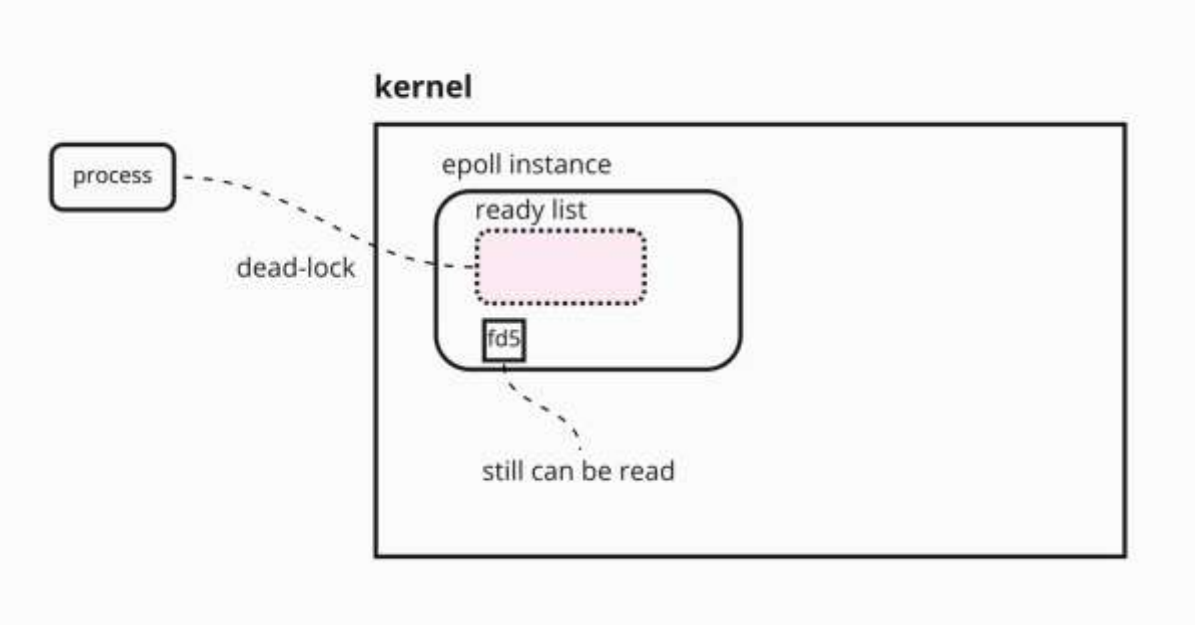

kernel
process
epoll instance
ready list
fd
fd
fd



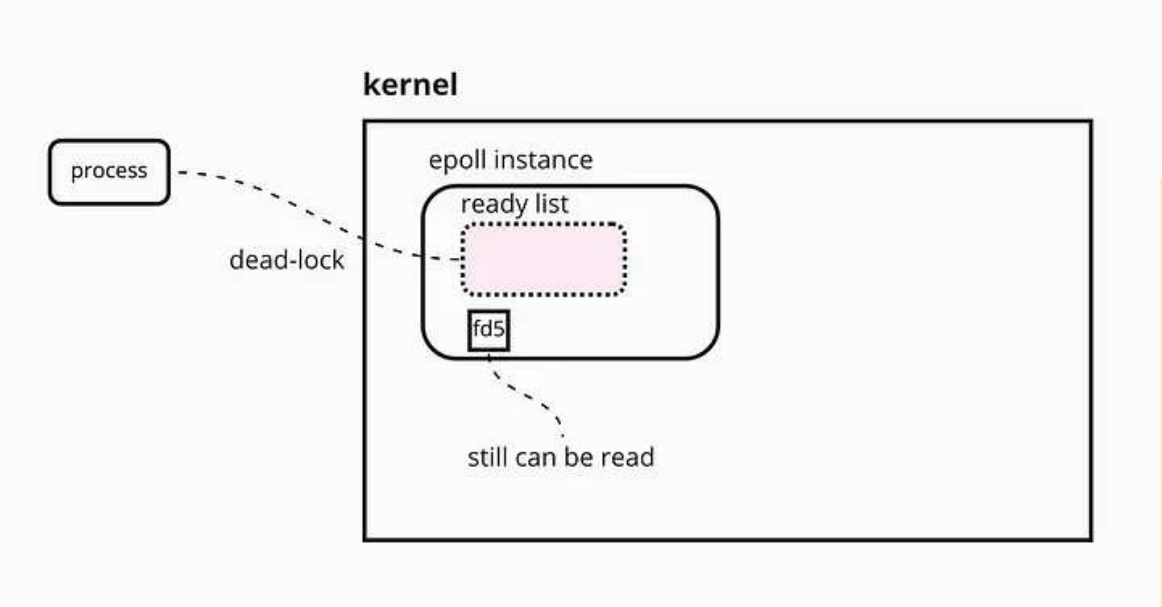
https://man7.org/linux/man-pages/man2/epoll_wait.2.html



https://man7.org/linux/man-pages/man2/epoll_create.2.html



https://man7.org/linux/man-pages/man2/epoll_create.2.html



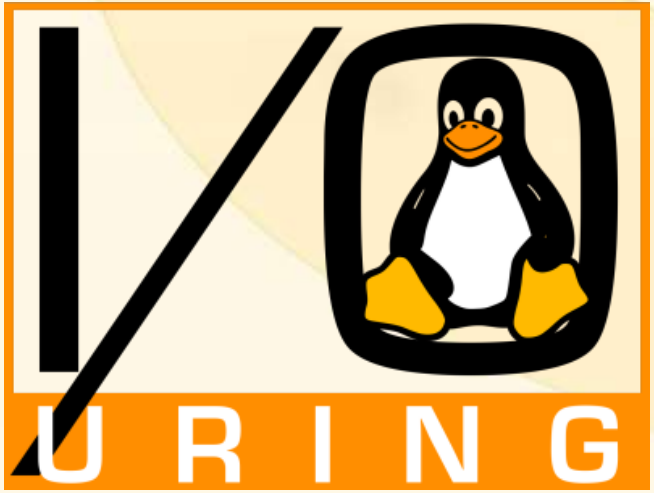
https://man7.org/linux/man-pages/man2/epoll_create.2.html

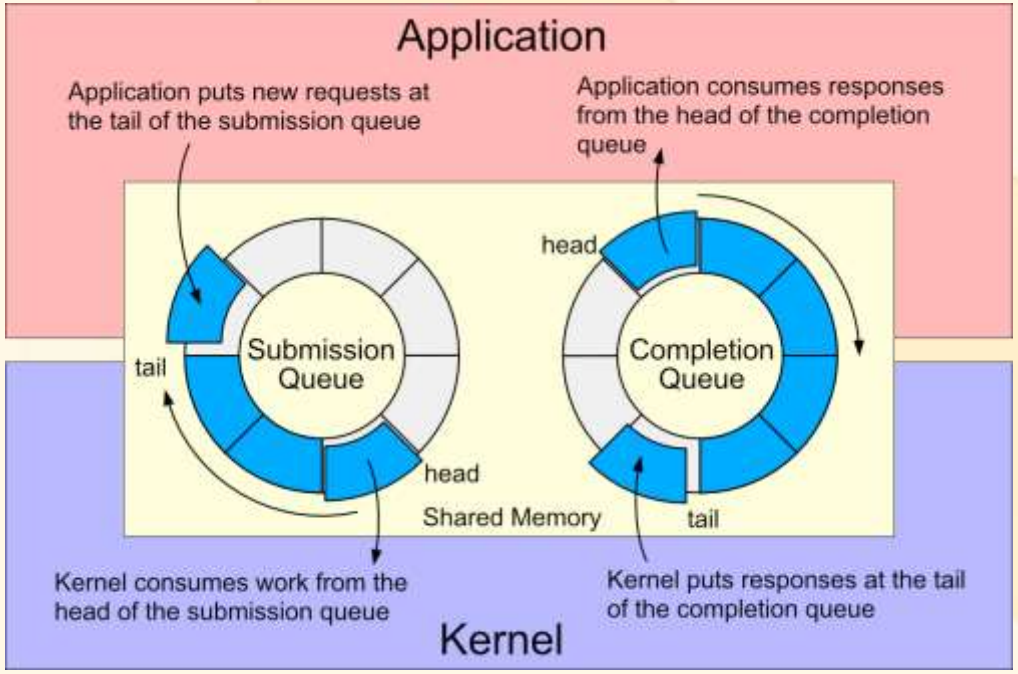
hepsiburada

Libuv Project Structure

hepsiburada

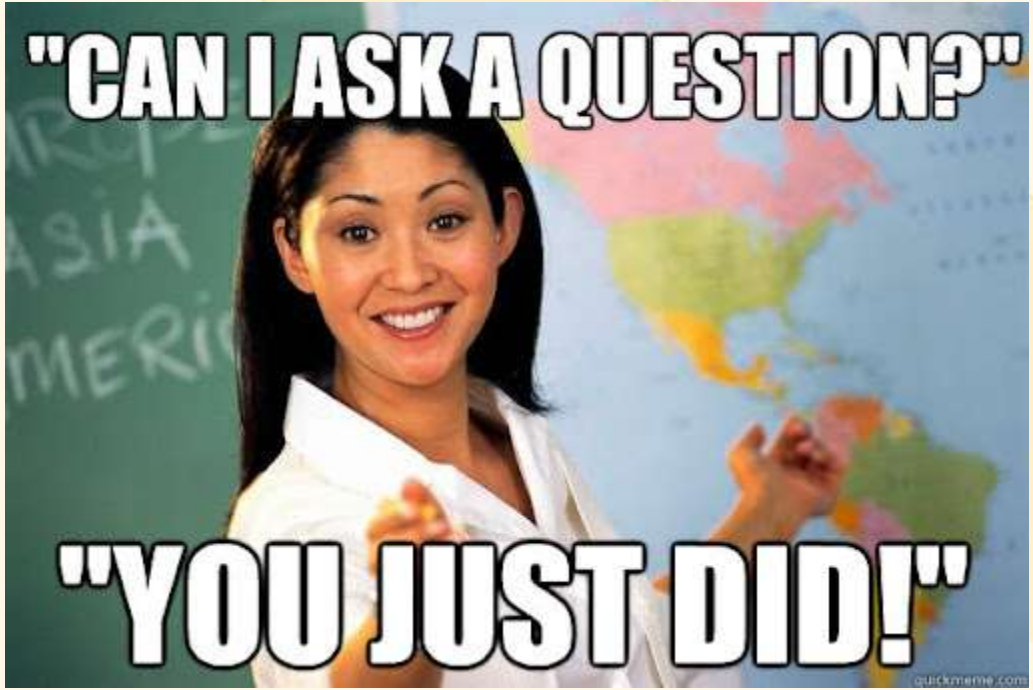
Contribution


I/O
URING


Application
Application puts new requests at the tail of the submission queue
Application consumes responses from the head of the completion queue
Submission Queue
tail
head
Completion Queue
head
tail
Shared Memory
Kernel consumes work from the head of the submission queue
Kernel puts responses at the tail of the completion queue
Kernel

io_uring vs epoll

	c: 50 bytes: 128	c: 50 bytes: 512	c: 500 bytes: 128	c: 500 bytes: 512	c: 1000 bytes: 128	c: 1000 bytes: 512
io_uring tcp-echo server	249297	252822	193452	179966	158911	163111
epoll tcp-echo server	223135	227143	173357	173772	156449	155492


"CAN I ASK A QUESTION?"
"YOU JUST DID!"

hepsiburada

THANKS