The Wild West of post-POSIX IO

Anil Madhavapeddy, University of Cambridge

with many thanks to Thomas Leonard, Patrick Ferris, Sadiq Jaffer, Ryan Gibb, David Allsopp, KC Sivaramakrishnan, Thomas Gazagnaire, Christiano Haesbart, Stephen Dolan and lots of help from others in the OCaml and Linux community

VMIL Keynote, 15th October 2025
at ICFP/SPLASH, Singapore

Are IO libraries still really a research topic??

Xen hypervisor stack
(~2004-present)
Unix+threads

OCaml in 1996: Single core
Spinning harddrives
Unix mattered

OCaml in 2004: SMP (2-4 sockets)
Harddrives -> SSDs
Linux / Windows dominate

Are IO libraries still really a research topic??

Xen hypervisor stack
(~2004-present)
Unix+threads

OCaml in 1996: Single core
Spinning harddrives
Unix mattered

Docker
(~2015-present)
MirageOS+virtualisation

OCaml in 2004: SMP (2-4 sockets)
Harddrives -> SSDs
Linux / Windows dominate

Are IO libraries still really a research topic??

OCaml in 1996:
- Single core
- Spinning harddrives
- Unix mattered

Xen hypervisor stack
(~2004-present)
Unix+threads

OCaml in 2004:
- SMP (2-4 sockets)
- Harddrives -> SSDs
- Linux / Windows dominate

Docker
(~2015-present)
MirageOS+virtualisation

OCaml in 2023:
- Many core (128+ cores)
- NVMe / flash
- Linux / Windows / macOS / JavaScript / wasm / unikernel
- Enclaves for encrypted memory
- GPGPU/FPGAs everywhere

Planetary computing
(~2022-present)
Heterogenous hardware
+ browser interfaces

Are IO libraries still really a research topic??

OCaml in 1996: Single core
Spinning harddrives
Unix mattered

POSIX must have

OCaml in 2004: SMP (2-4 sockets)
Harddrives -> SSDs
Linux / Windows dominate

POSIX must try

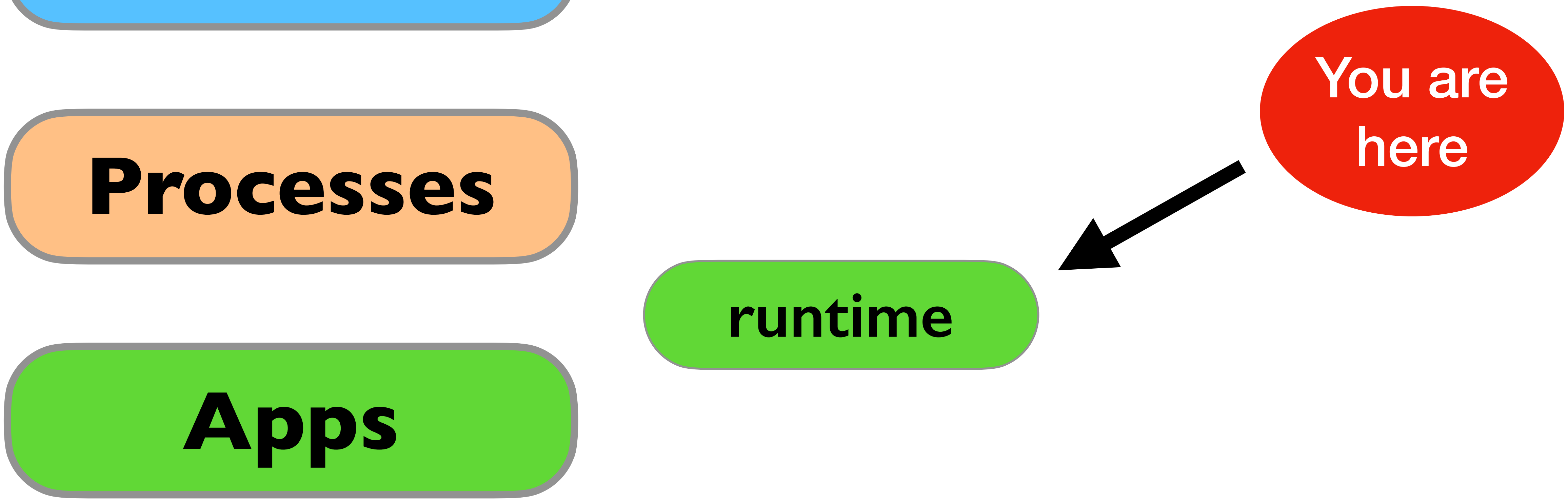
OCaml in 2023: Many core (128+ cores)
NVMe / flash
Linux / Windows / macOS / JavaScript / wasm / unikernel
Enclaves for encrypted memory
GPGPU/FPGAs everywhere

POSIX is an illusion

Deploying software on modern hardware uses shared memory parallelism all over

What are some of these post-POSIX interfaces and do they share anything in common?

We're having a go at supporting this in OCaml 5 with our Eio library using the new effect handlers feature


Hardware
Kernel
Processes
Apps
runtime
You are here

Hardware

Kernel

Processes

Apps

drivers

services

scheduling

isolation

runtime

An OS kernel drives the hardware and builds software services over it

Kernel implements services such as storage / networking

It schedules multiple userspace *processes* that are isolated from each other, but share kernel resources

Hardware

Kernel

Processes

Apps

interrupts

memory

cpus

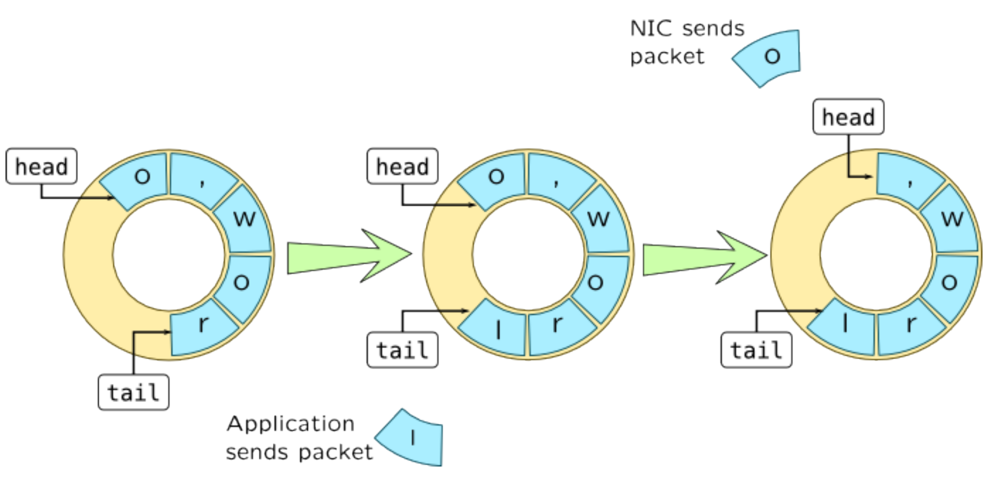
drivers

runtime

Hardware has become diverse, with thousands of device drivers in a kernel

Multicore processors and NUMA memory also make those resources complex

Ring buffers are mapped into kernel memory or via DMA

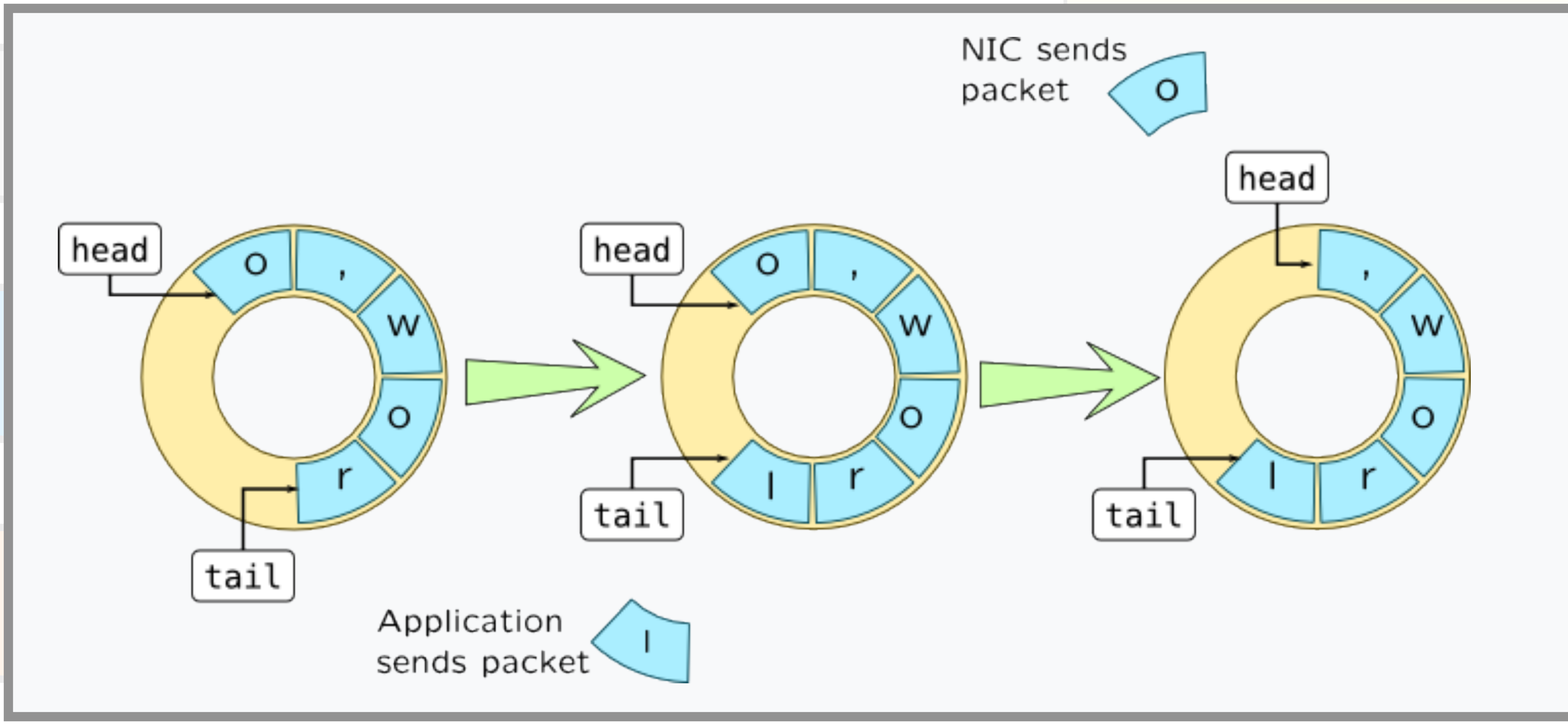


source: dtrace book

interrupts

memory

Hardware has become diverse, with thousands of device drivers in a kernel



Apps

runtime

source: dtrace book

Hardware

Kernel

Processes

Apps

isolation

syscalls

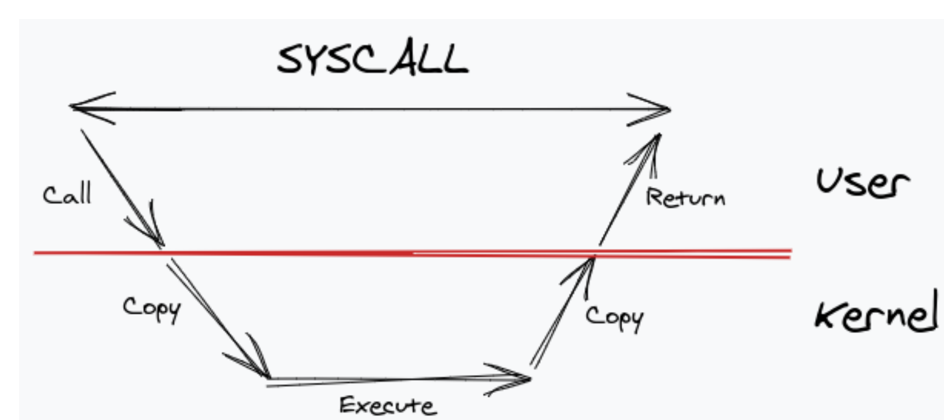
ioctls

runtime

Kernel runs processes at lower privilege level with own address spaces

Interface from process to kernel is typically via a *syscall* or *ioctl* interface

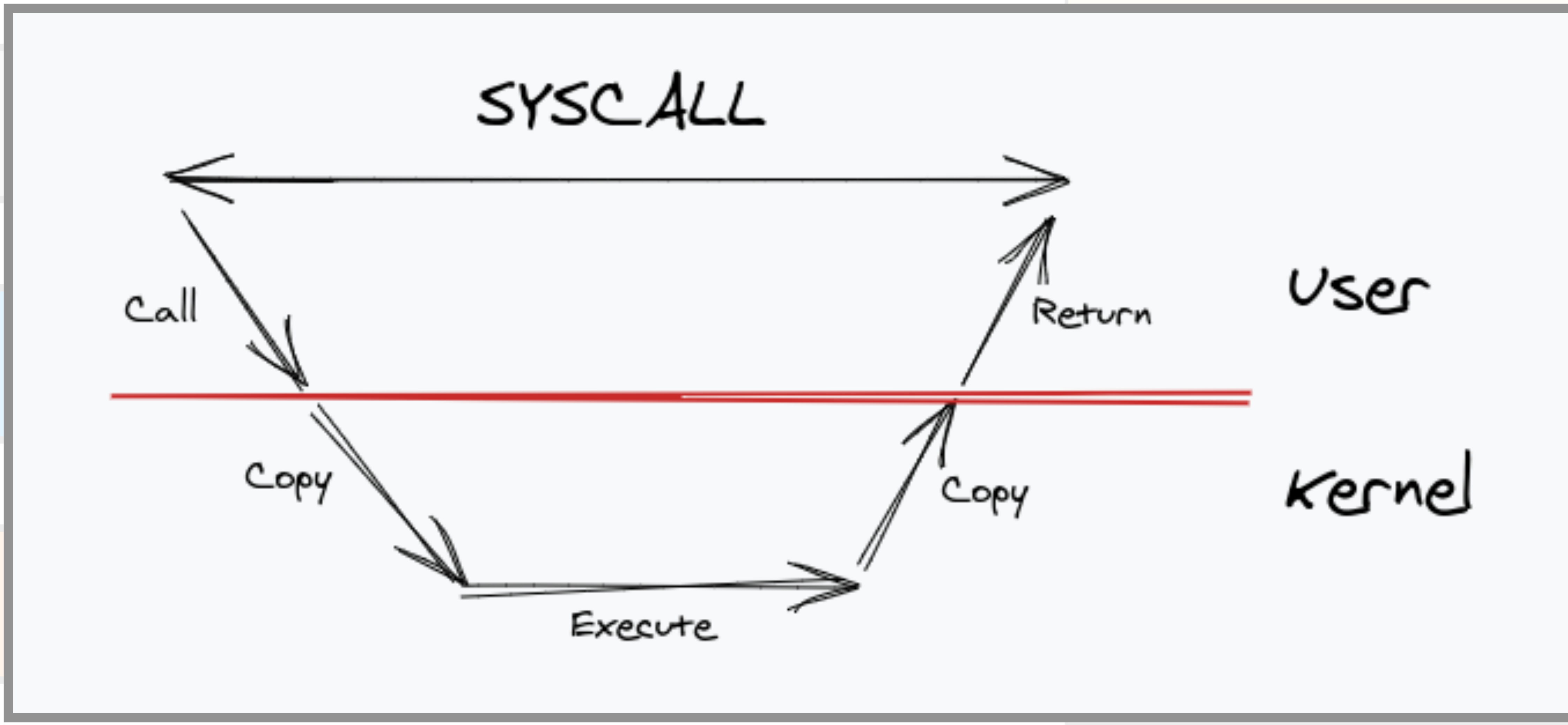
Syscalls also force a *context switch* since they switch privilege levels into the kernel



Kernel runs processes at lower privilege level with own address spaces

is
call

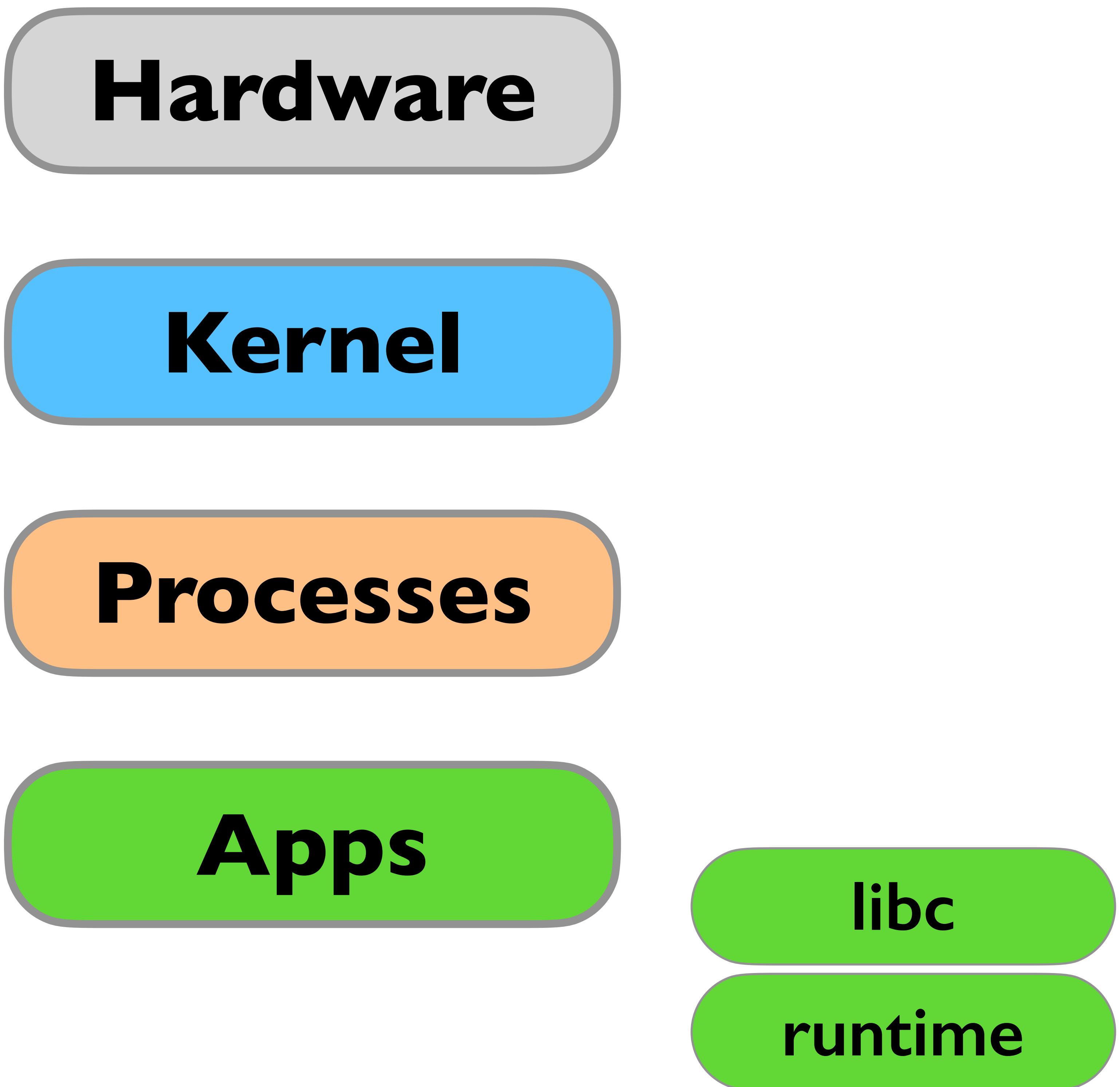
e a
ce
ge
nel



ioctls

Apps

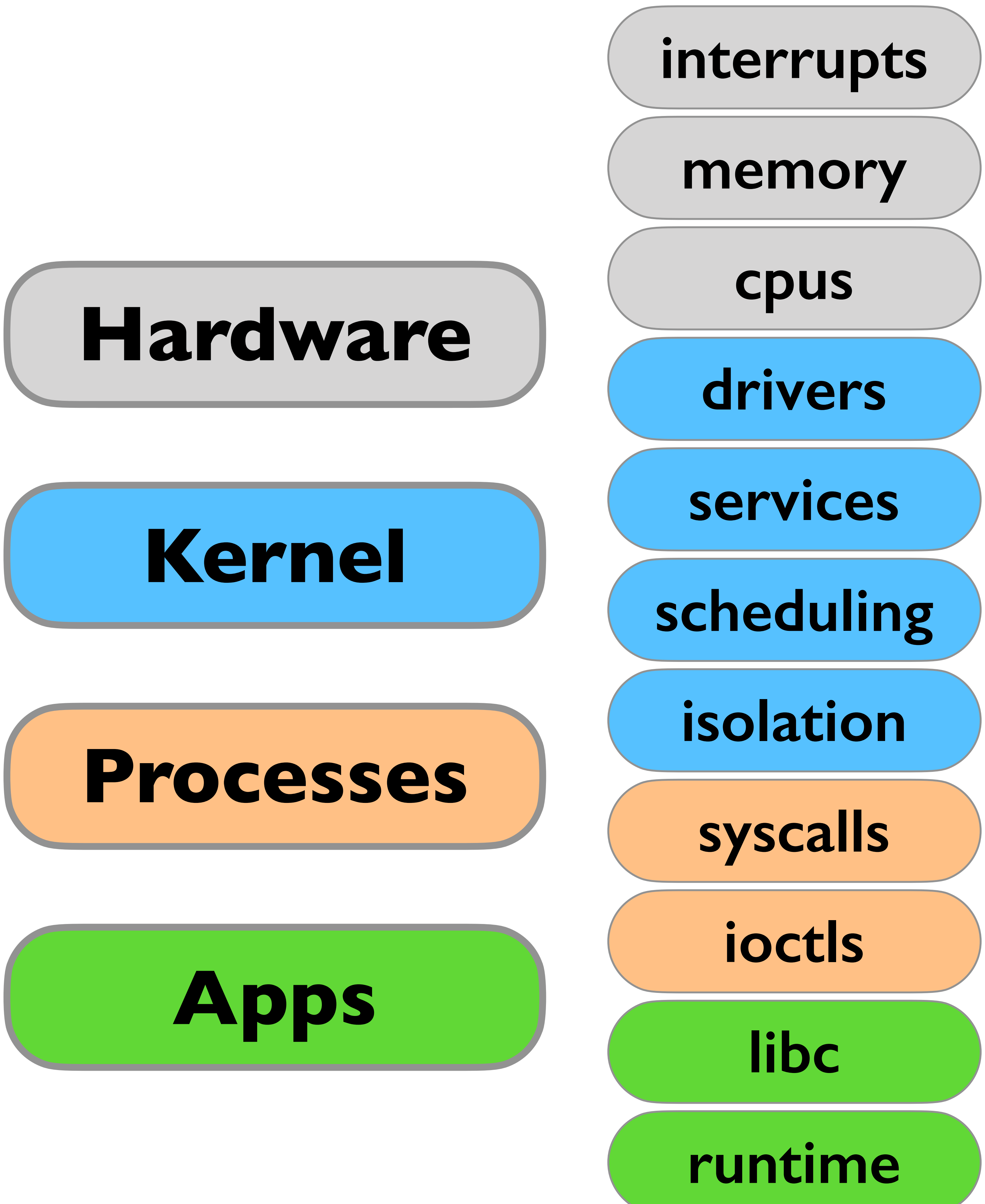
runtime



System libraries like *libc* often abstract the low-level syscalls with a C interface

Standards like POSIX define function calls that try to be portable across operating systems

Applications link with system libraries and runtimes (VMIL!) and call them to interface with outside world



Kernel schedules all the processes to try to make best use of the hardware

But the scheduling is difficult without some cooperation from the application logic

Utilising all of the hardware from a language runtime is quite tough! But it gets tougher...

Hardware

Hypervisor

Kernel

Processes

Apps

interrupts
memory
cpus
drivers
services
scheduling
isolation
syscalls
ioctls
libc
runtime

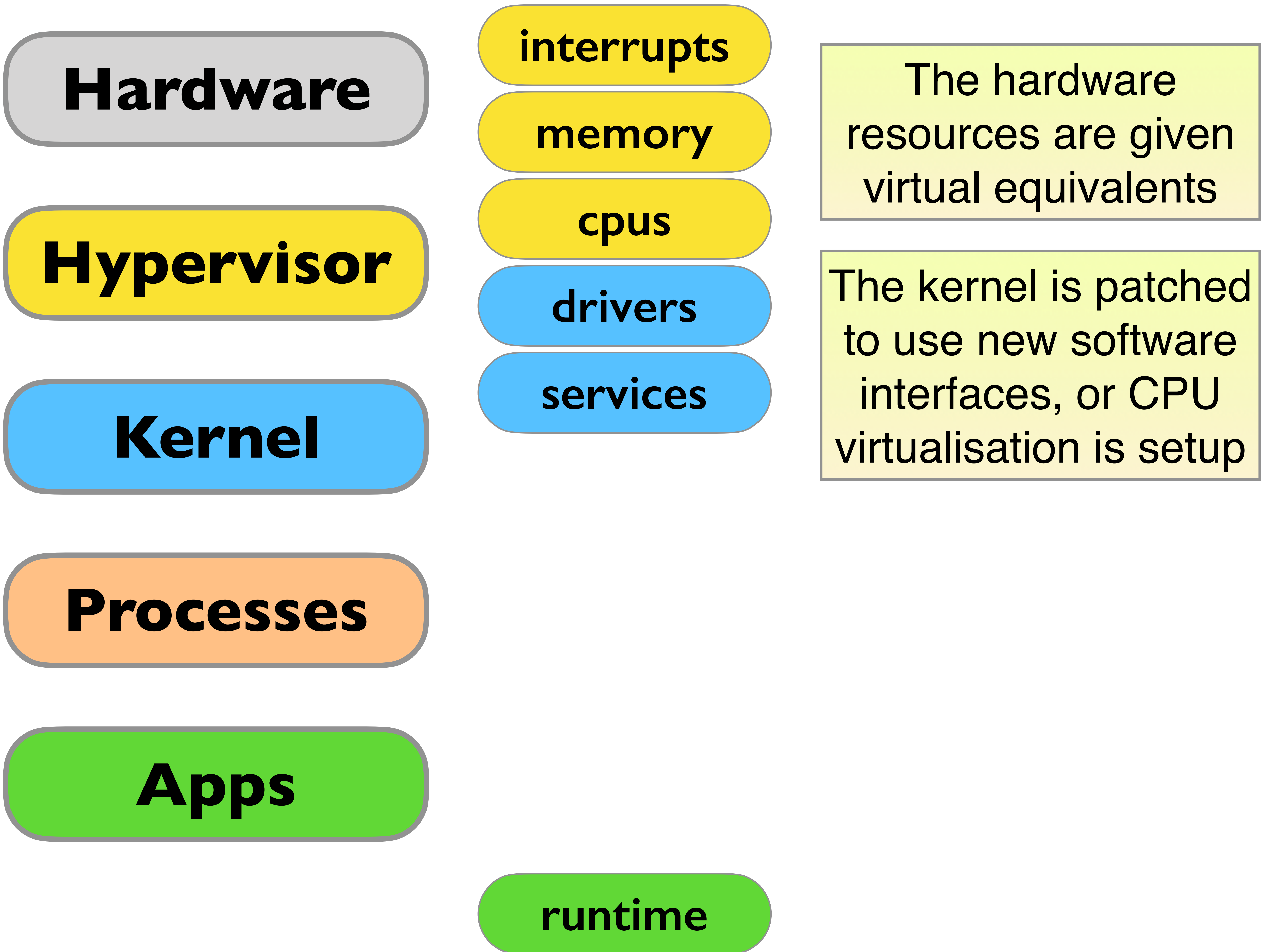
Multiplex *abstraction* of physical computer

Carve up resources so poorly utilised physical computers are consolidated

2003: Xen hypervisor 1.0 was released.

2018: tens of billions of virtual machines are “in the cloud”

2025: hypervisors run everywhere


Hardware
Hypervisor
Kernel
Processes
Apps
interrupts
memory
cpus
drivers
services
runtime
The hardware resources are given virtual equivalents
The kernel is patched to use new software interfaces, or CPU virtualisation is setup

Hardware

Hypervisor

Kernel

Processes

Apps

services

scheduling

isolation

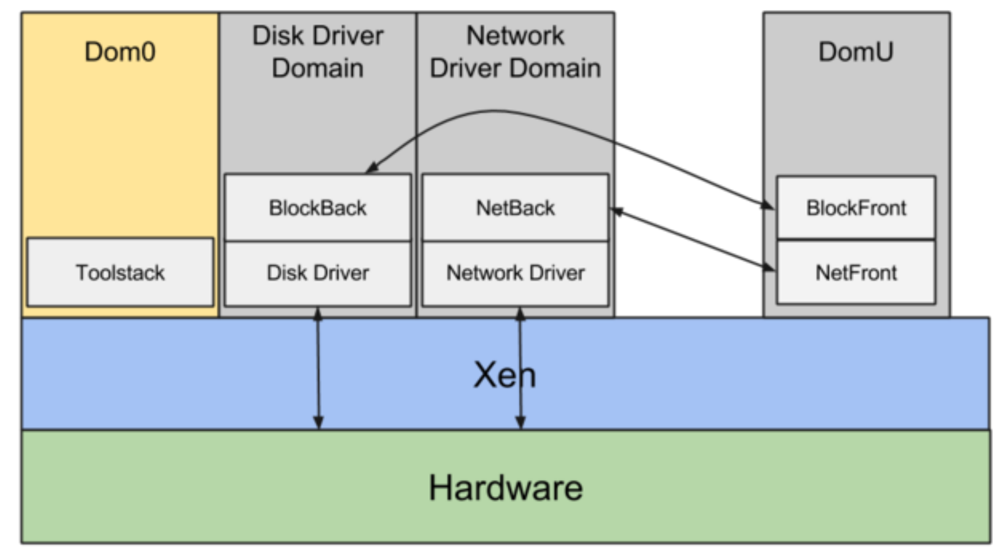
syscalls

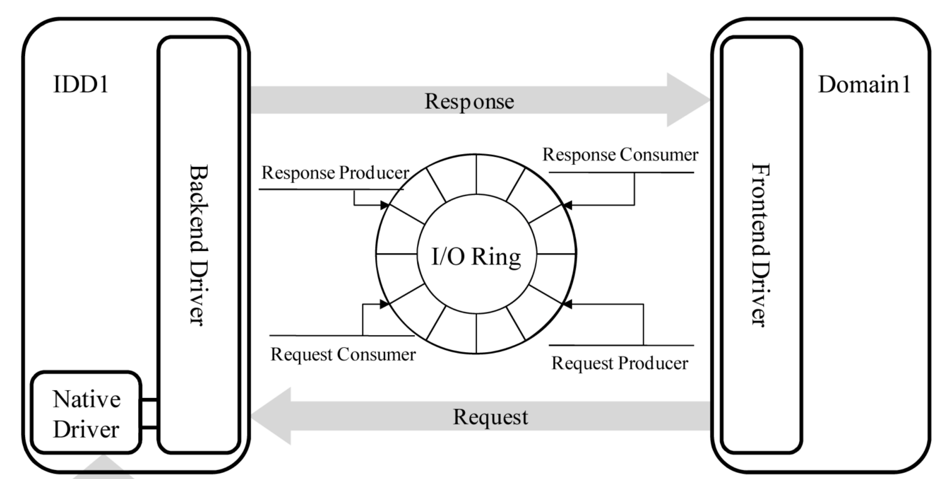
ioctls

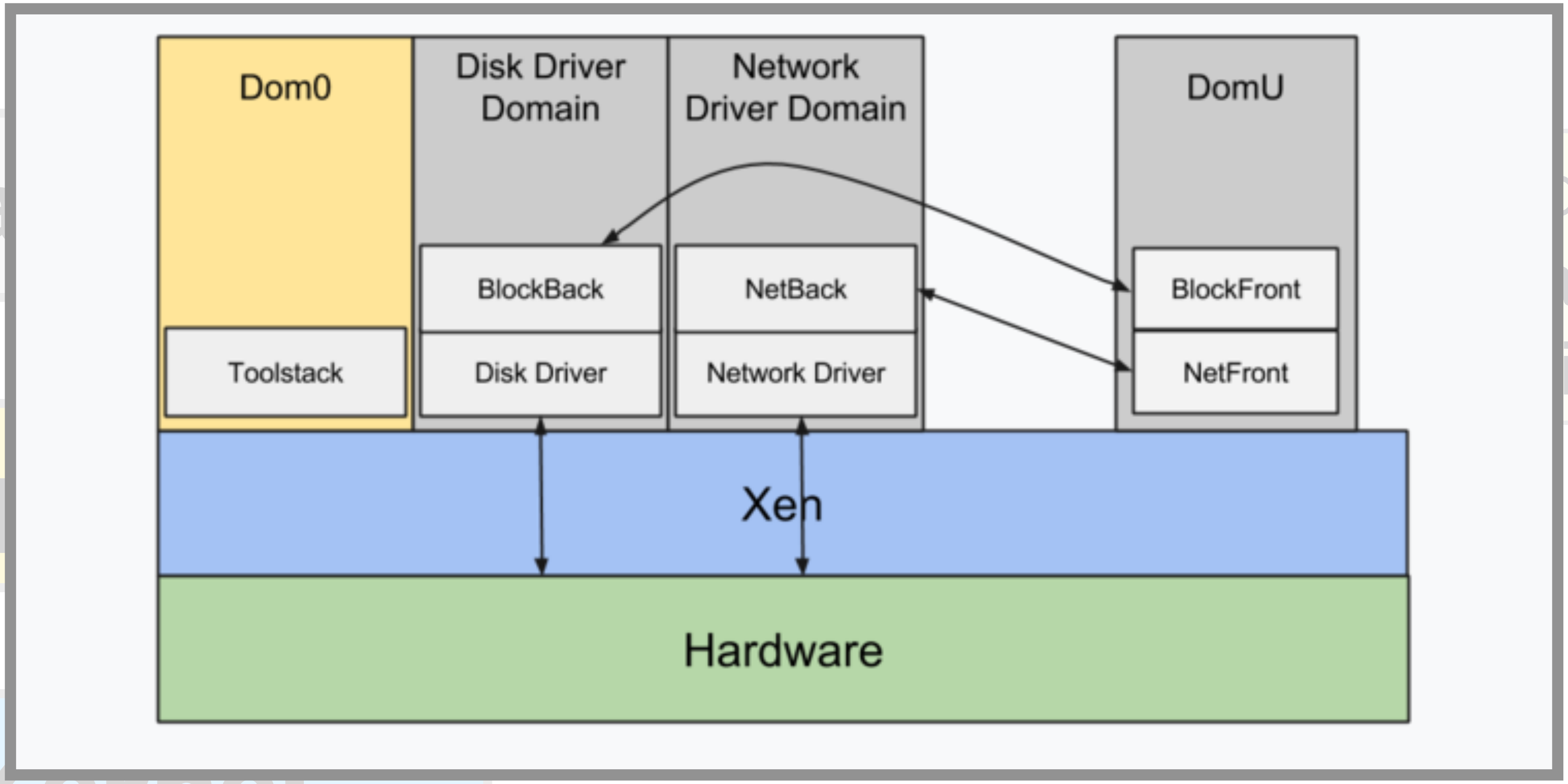
libc

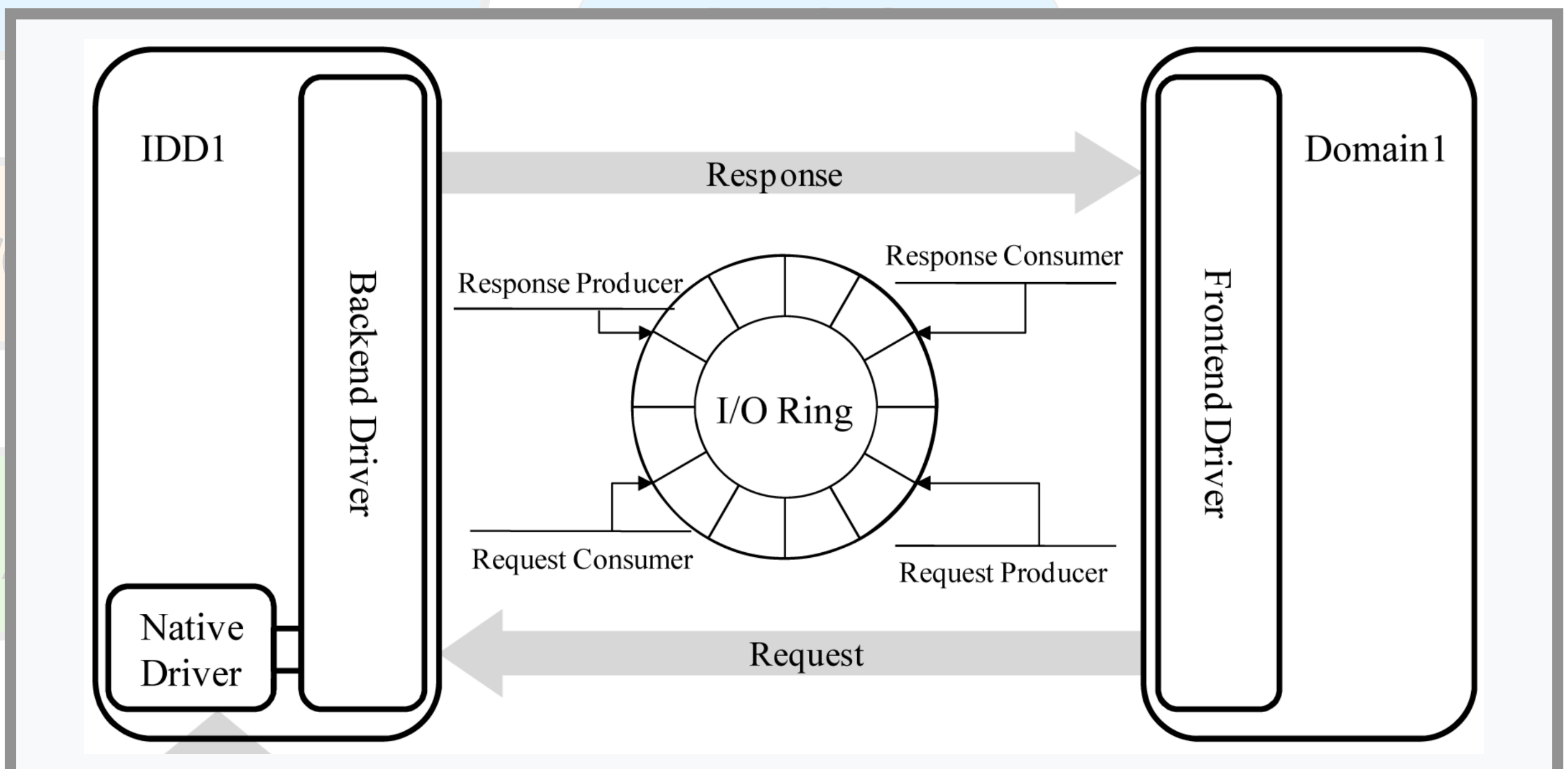
runtime

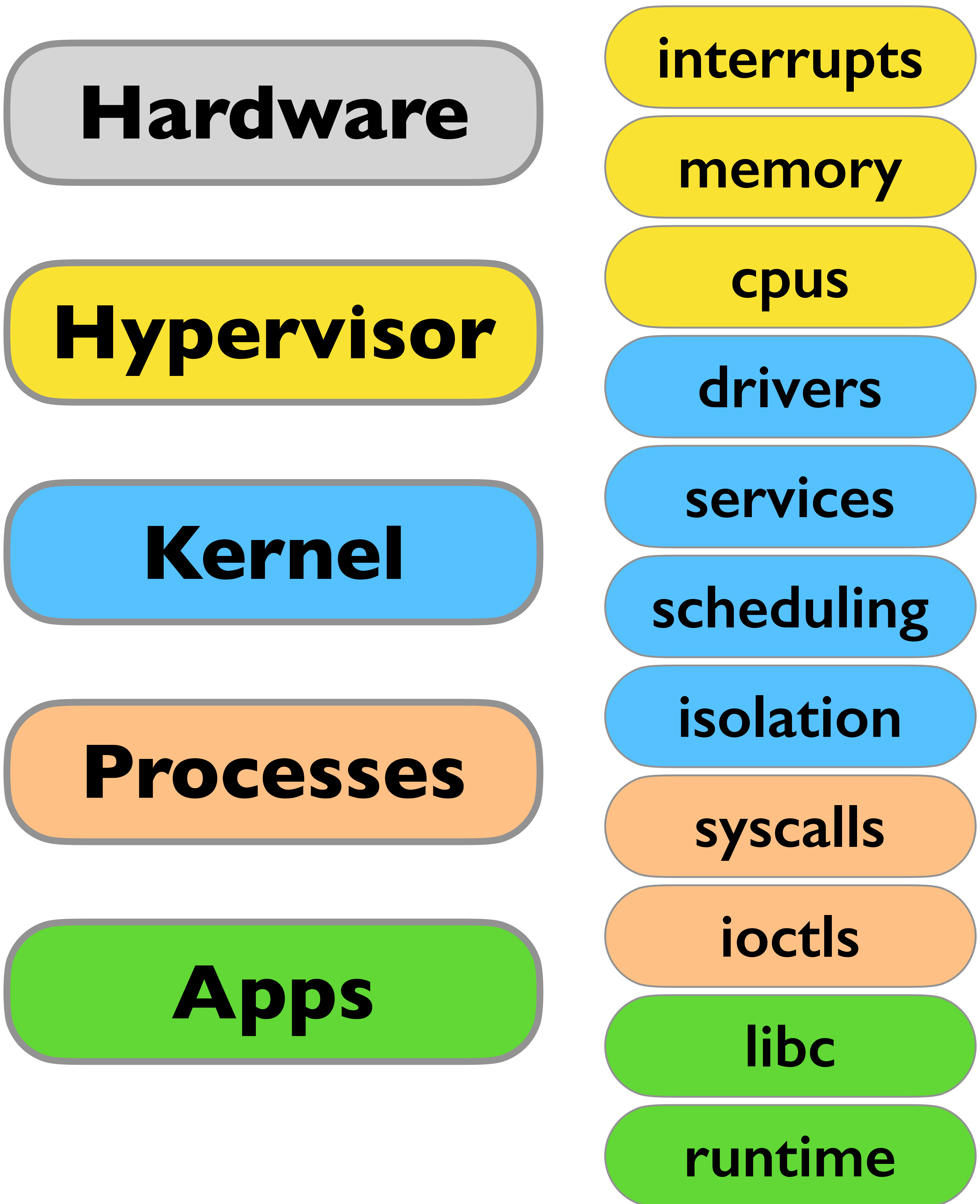
The hardware resources are given virtual equivalents






Dom0
Disk Driver Domain
Network Driver Domain
DomU
BlockBack
NetBack
BlockFront
Toolstack
Disk Driver
Network Driver
NetFront
Xen
Hardware


IDD1
Backend Driver
Native Driver
Response
Response Producer
Response Consumer
I/O Ring
Request Consumer
Request Producer
Request
Frontend Driver
Domain1



The hypervisor starts fighting the kernel for scheduling resources

Since *multiple* guest kernels can run, each thinks it owns the underlying hardware

Userspace schedules some resources too

Multiplexed scheduling is considered harmful

Hardware

Hypervisor

Kernel

Container

Processes

Apps

interrupts

memory

cpus

drivers

services

scheduling

isolation

namespace

syscalls

ioctls

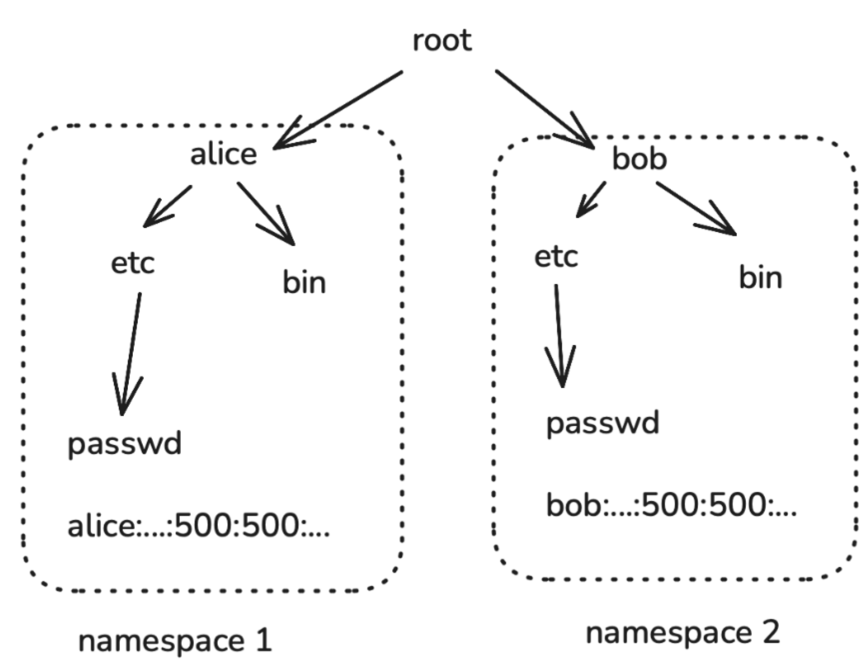
libc

runtime

VMs cannot share kernel resources and so can be isolated

Linux introduces process namespaces to virtualise syscalls

A *container* is a process set and filesystem isolated but sharing kernel



Hardware

Hypervisor

Kernel

Contain

Processes

Apps

interrupts

memory

cpus

syscalls

ioctls

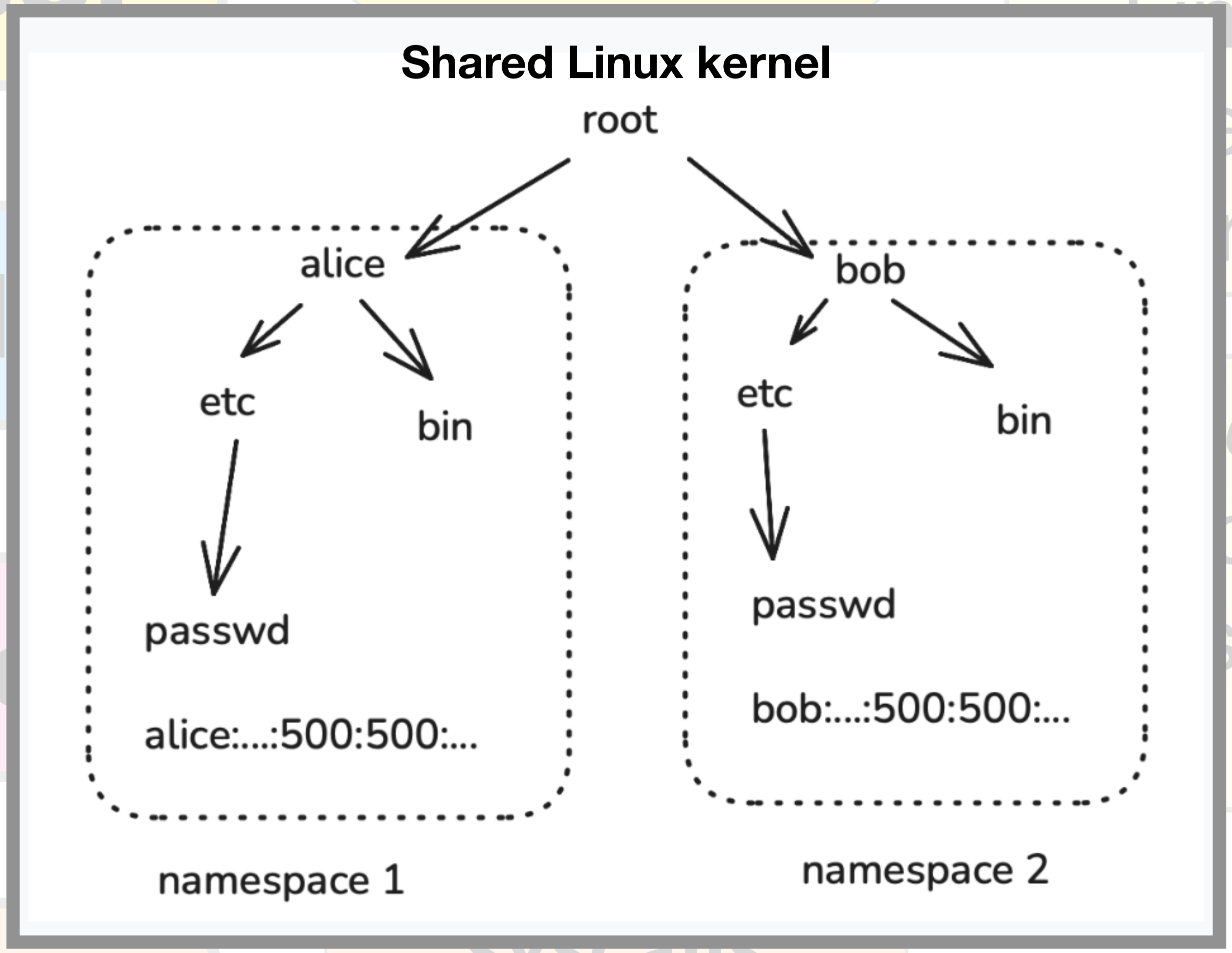
libc

runtime

VMs cannot share kernel resources and so can be isolated

inux introduces
ess namespaces
rtualise syscalls

container is a
ocess set and
system isolated
sharing kernel



Hardware

Hypervisor

Kernel

Container

Processes

Apps

interrupts

memory

cpus

drivers

services

scheduling

isolation

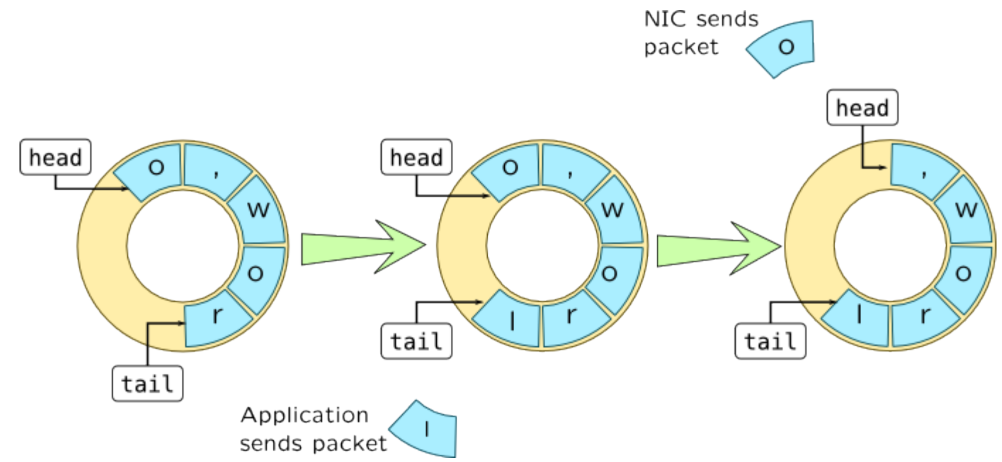
namespace

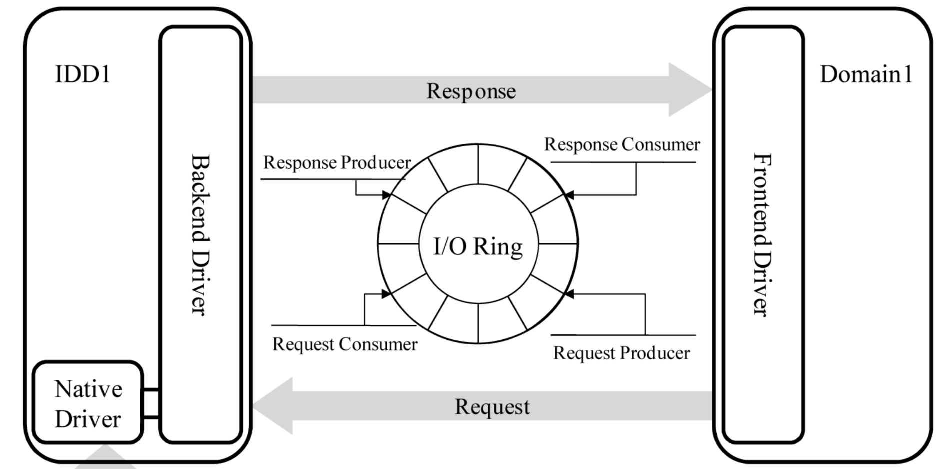
syscalls

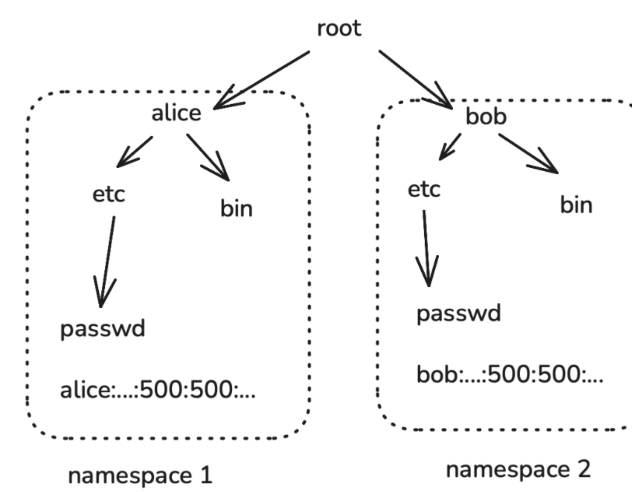
ioctls

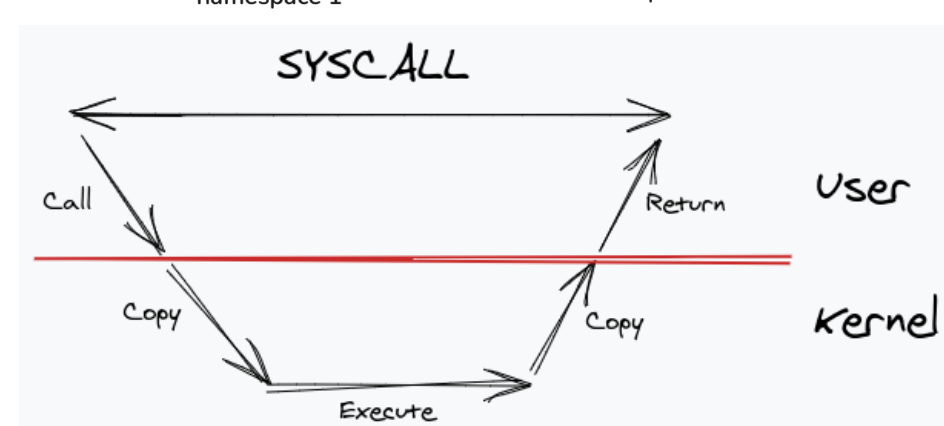
libc

runtime





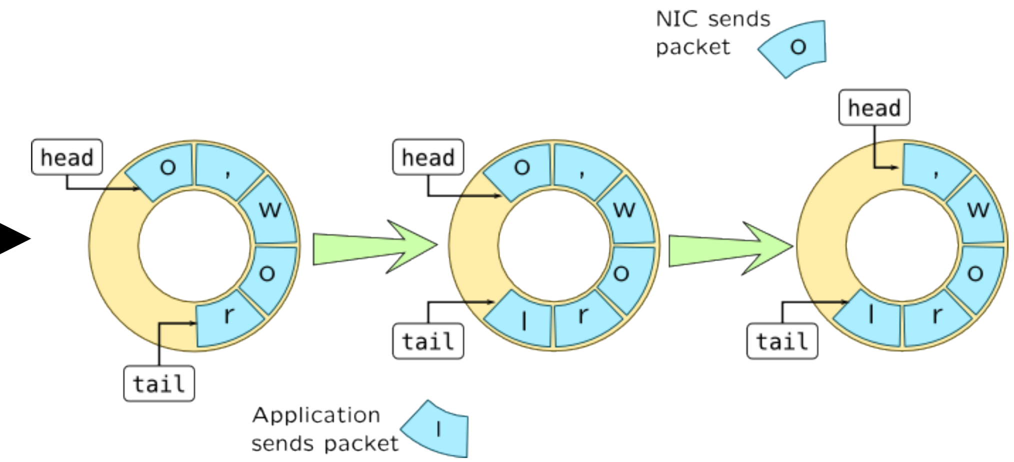




Hardware

interrupts

SHARED MEMORY RING

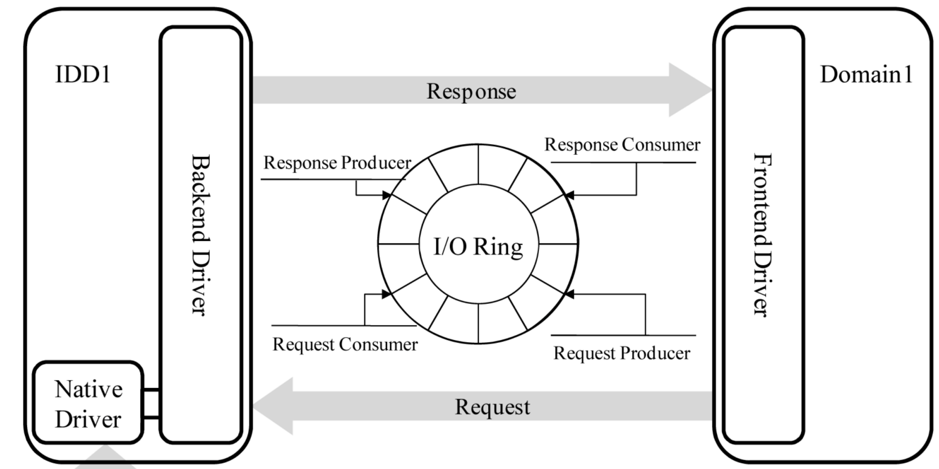


Hypervisor

cpus

drivers

SHARED MEMORY RING

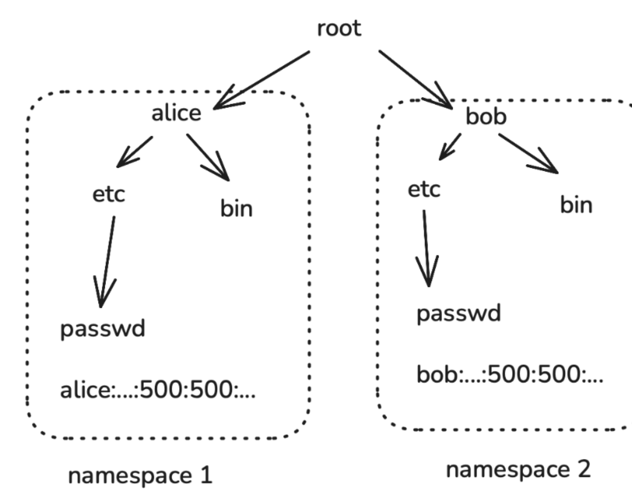


scheduling

isolation

Container

SHARED MEMORY RING



syscalls

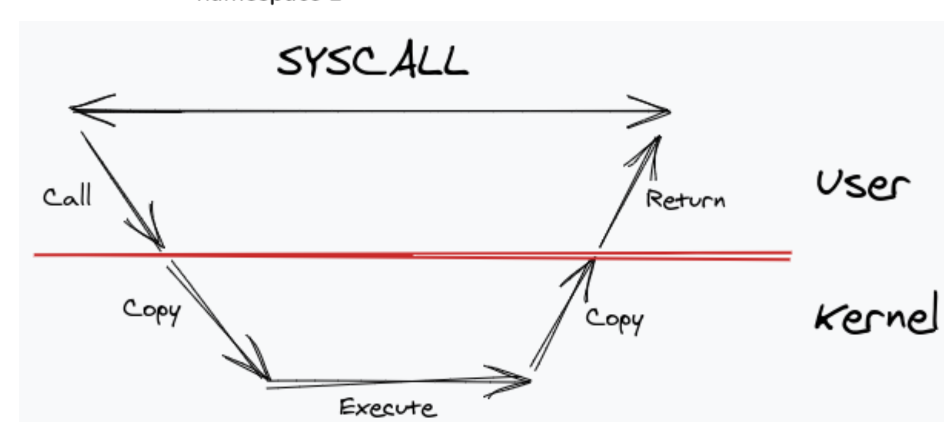
Processes

ioctls

Apps

libc

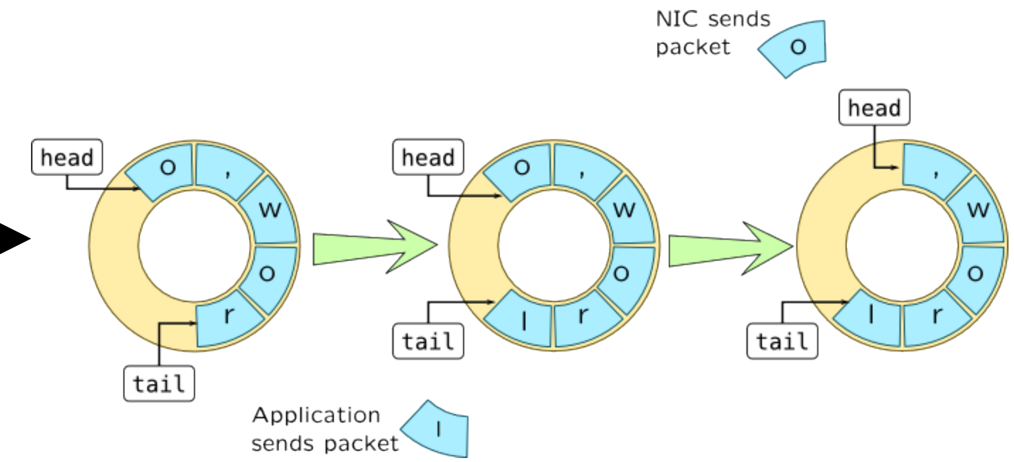
runtime



Hardware

interrupts

SHARED MEMORY RING

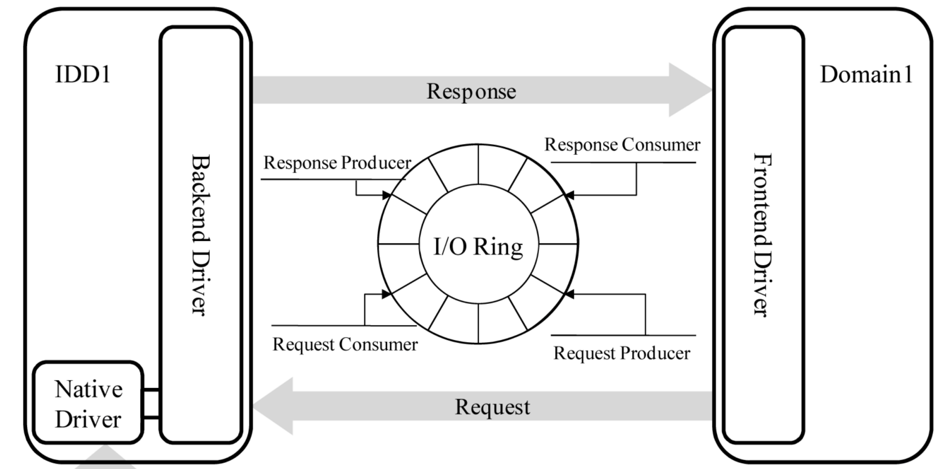


Hypervisor

cpus

drivers

SHARED MEMORY RING

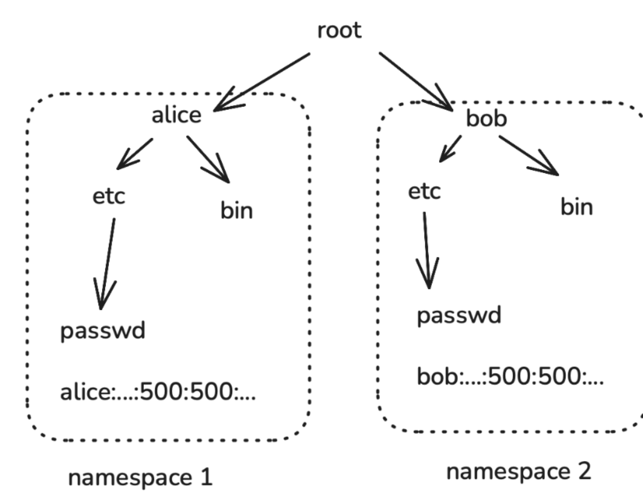


scheduling

Container

isolation

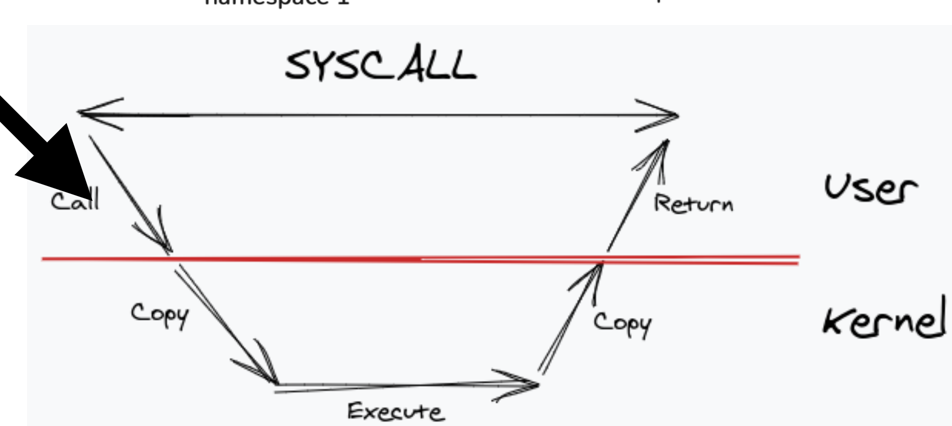
SHARED MEMORY RING



Processes

syscalls

NOT SHARED MEMORY RING



Apps

libc

runtime

Hardware

Hypervisor

Kernel

Container

Processes

Apps

interrupts

memory

cpus

drivers

services

scheduling

isolation

namespace

syscalls

ioctls

libc

runtime

Deploying software on modern hardware uses shared memory parallelism all over

What are some of these post-POSIX interfaces and do they share anything in common?

We're having a go at supporting this in OCaml 5 with our Eio library using the new effect handlers feature

Hardware

Hypervisor

Kernel

Container

Processes

Apps

interrupts

memory

cpus

drivers

services

scheduling

isolation

namespace

syscalls

ioctls

libc

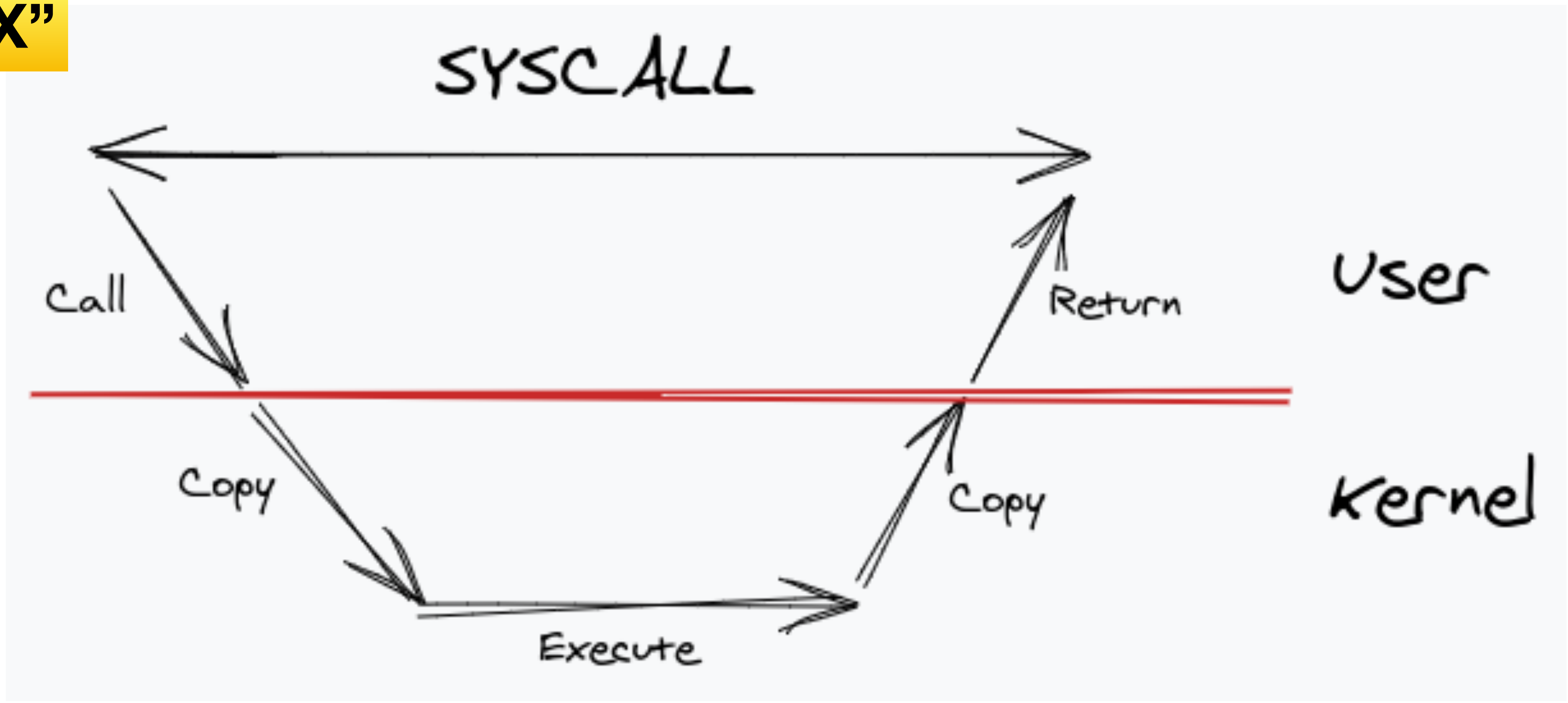
runtime

Deploying software on modern infrastructure needs shared memory parallelism everywhere

What are some of these post-POSIX interfaces and do they share anything in common?

We're having a go at supporting this in OCaml 5 with our Eio library using the new effect handlers feature

“POSIX”

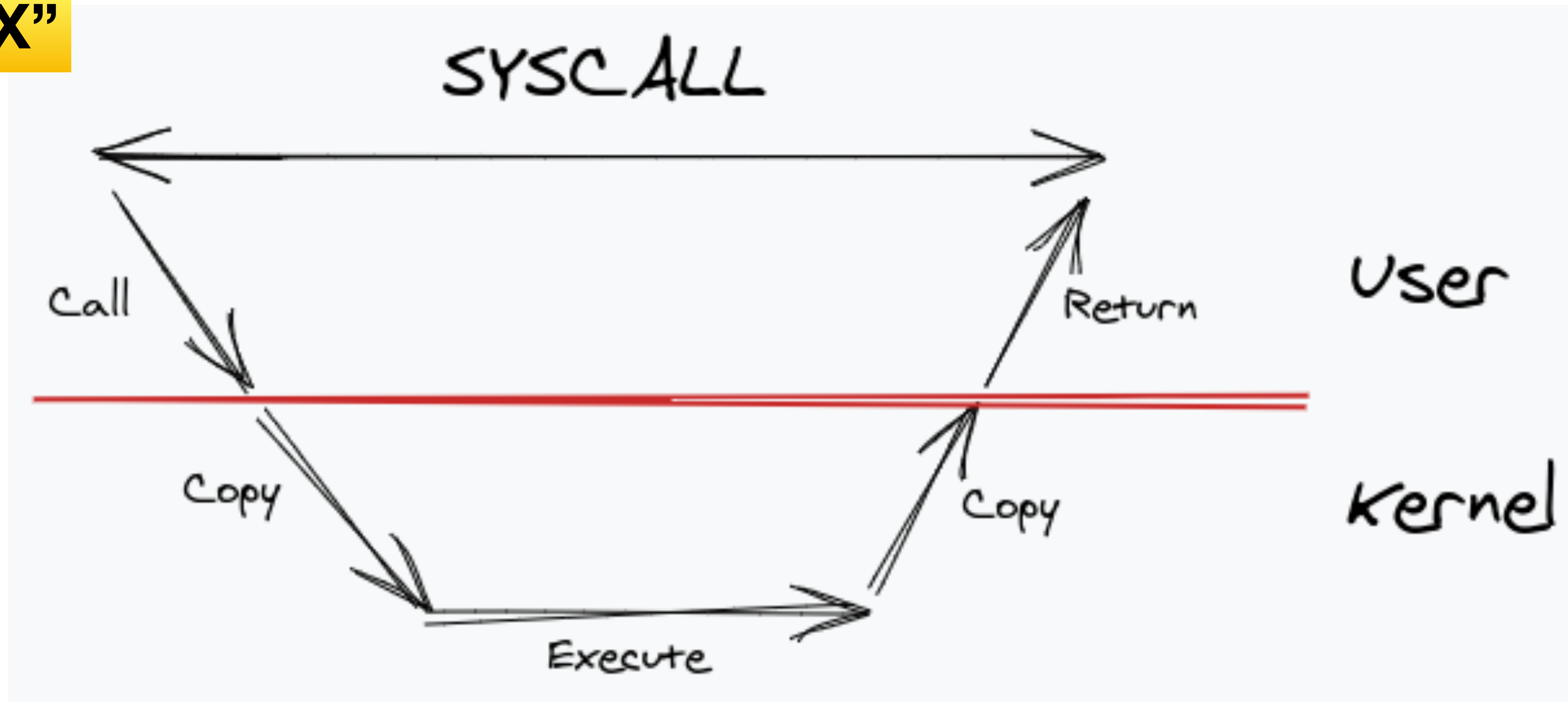


Synchronous, Async via threads

Includes a context switch from user to the kernel

image credit: https://developers.mattermost.com/blog/hands-on-iouring-go/

“POSIX”



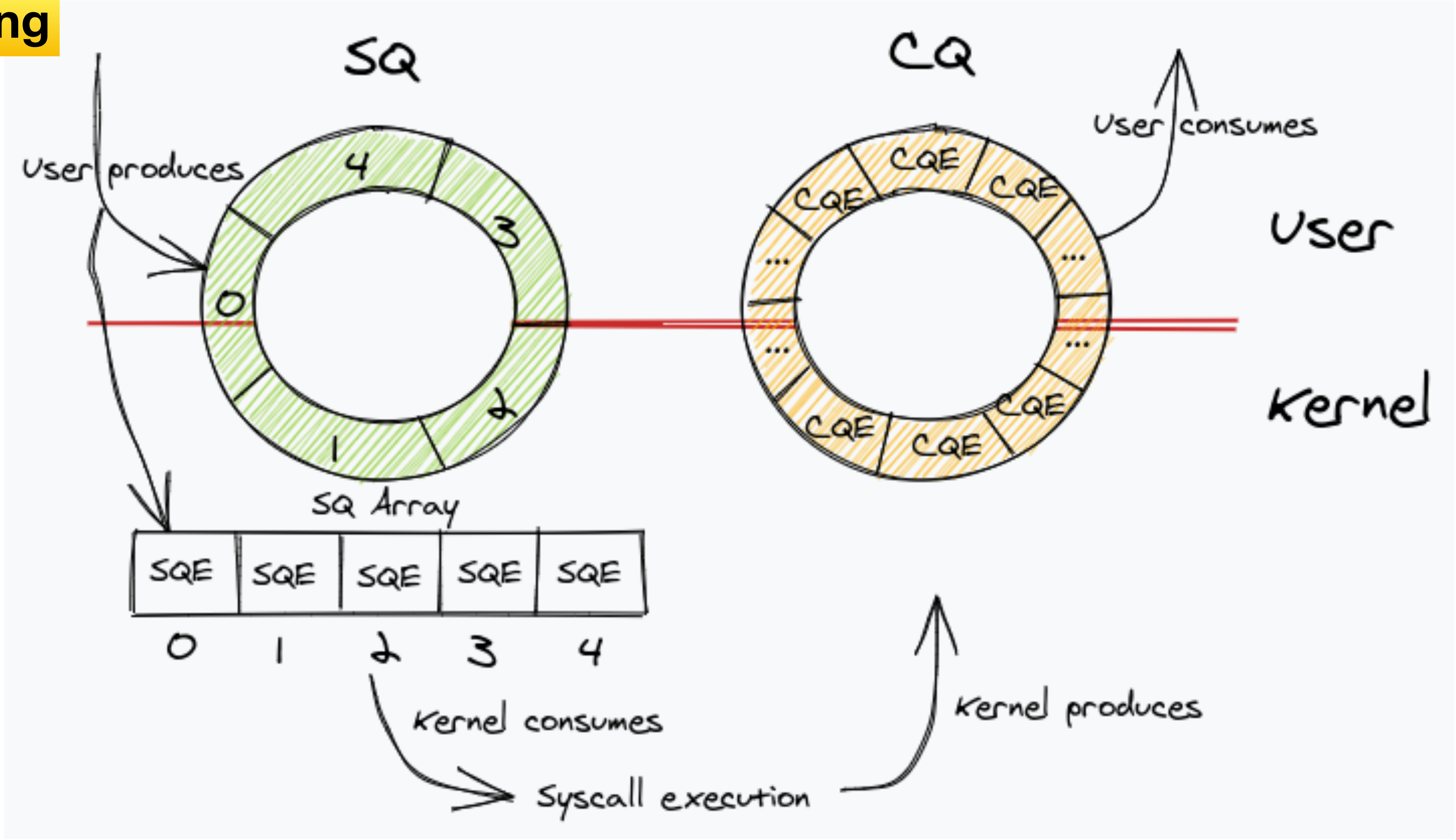
Synchronous, Async via threads

Includes a context switch from user to the kernel

```
[pid 1189143] pread64(4,
"\332-5\370`\306\223\323\374\16\326\212GD#\271Dh\325\341\362:\3234\373\314\334Ql`\374\360"...,
32768, 2064384) = 32768
[pid 1189143] futex(0x55572e6d4f48, FUTEX_WAIT_BITSET_PRIVATE|FUTEX_CLOCK_REALTIME, 0, NULL,
FUTEX_BITSET_MATCH_ANY <unfinished ...>
[pid 1189142] poll([{fd=5, events=POLLOUT}], 1, 0) = 1 ([{fd=5, revents=POLLOUT}])
[pid 1189142] futex(0x55572e6d4f4c, FUTEX_WAKE_PRIVATE, 1) = 1
[pid 1189145] <... futex resumed>)       = 0
[pid 1189142] poll([{fd=5, events=POLLOUT}], 1, 0 <unfinished ...>
[pid 1189145] futex(0x55572e6d4ee0, FUTEX_WAKE_PRIVATE, 1 <unfinished ...>
[pid 1189142] <... poll resumed>)        = 1 ([{fd=5, revents=POLLOUT}])
[pid 1189145] <... futex resumed>)       = 0
[pid 1189142] futex(0x55572e6d4f4c, FUTEX_WAKE_PRIVATE, 1 <unfinished ...>
[pid 1189145] pwrite64(5, "\270\213N\374\335\277j\32\357\233\234\222+\246`2\312\214S\230
\326e\244\n\16\310\217\325\242R\324"..., 32768, 0 <unfinished ...>
[pid 1189142] <... futex resumed>)       = 1
[pid 1189144] <... futex resumed>)       = 0
[pid 1189142] poll([{fd=5, events=POLLOUT}], 1, 0 <unfinished ...>
```

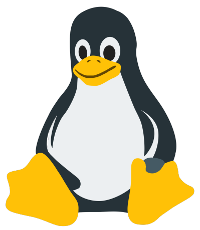
image credit: https://developers.mattermost.com/blog/hands-on-iouring-go/

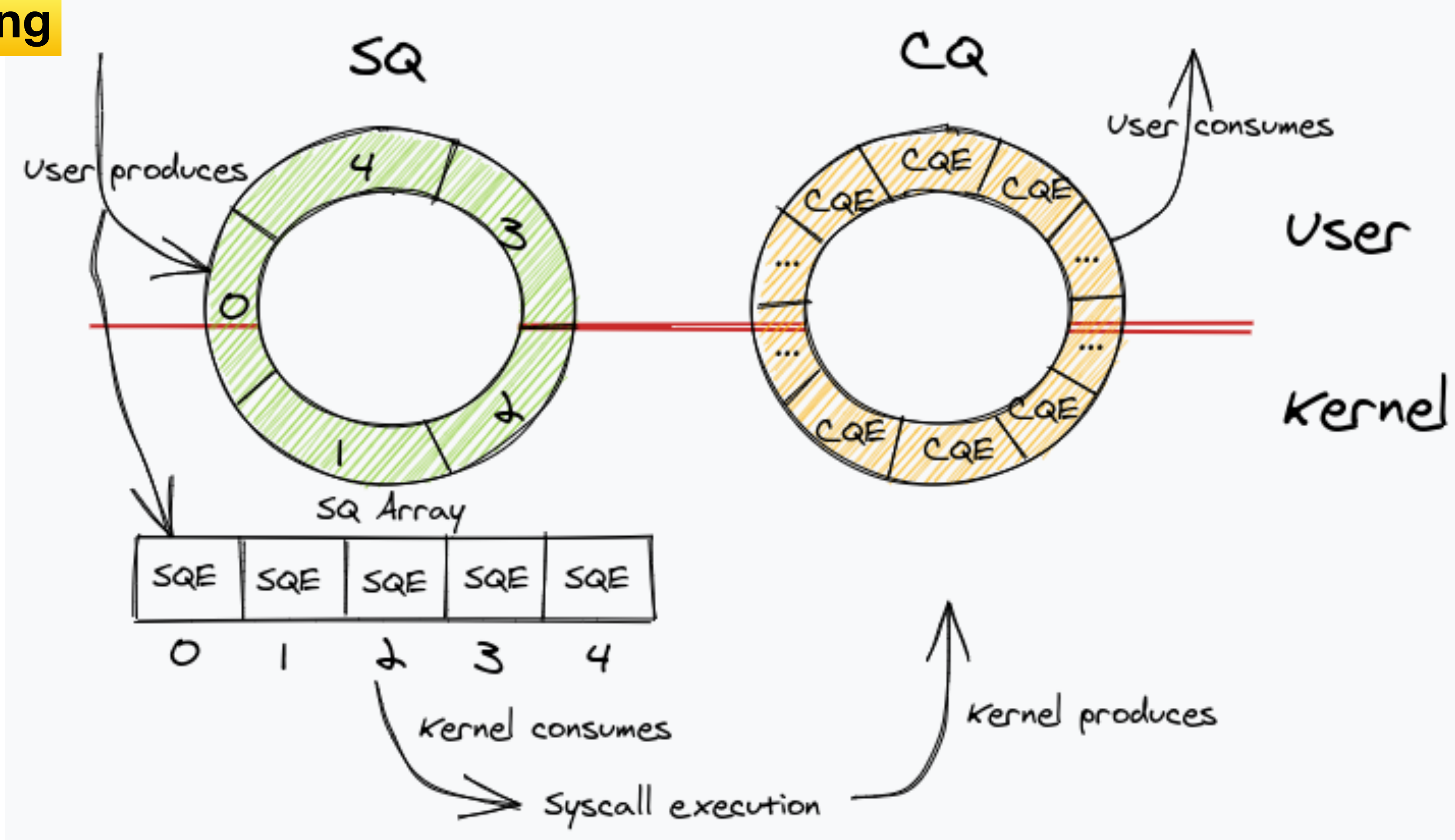
io_uring



Multiple requests on shared ring

io_uring





Multiple requests on shared ring

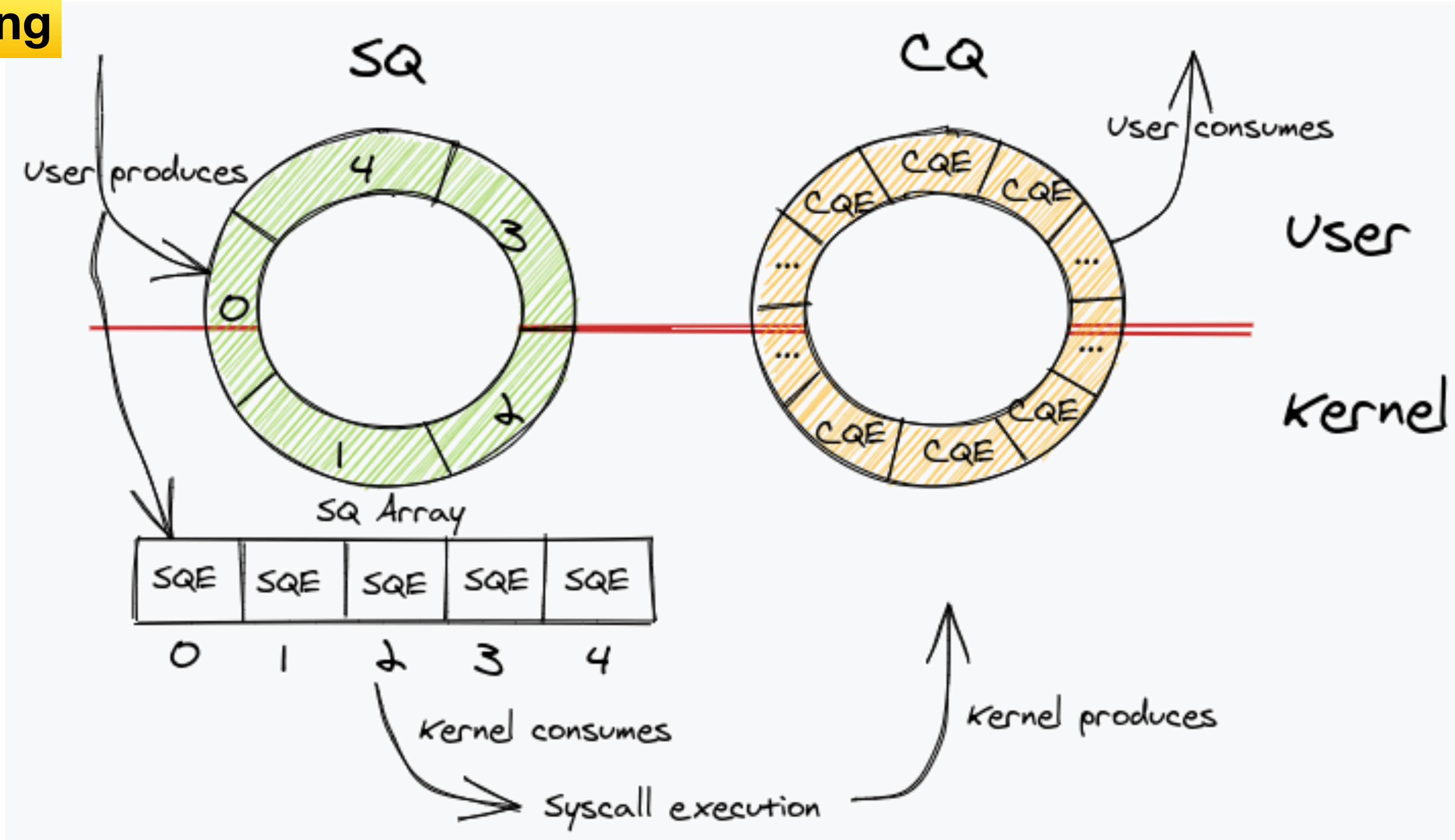
SQ: applications submit requests to a shared submission queue with user data attached

```
enum io_uring_op {
	IORING_OP_NOP,
	IORING_OP_READV,
	IORING_OP_WRITEV,
	IORING_OP_FSYNC,
	IORING_OP_READ_FIXED,
	IORING_OP_WRITE_FIXED,
	IORING_OP_POLL_ADD,
	IORING_OP_POLL_REMOVE,
	IORING_OP_SYNC_FILE_RANGE,
	IORING_OP_SENDMSG,
	IORING_OP_RECVMSG,
	IORING_OP_TIMEOUT,
	IORING_OP_TIMEOUT_REMOVE,
	IORING_OP_ACCEPT,
	IORING_OP_ASYNC_CANCEL,
	IORING_OP_LINK_TIMEOUT,
	IORING_OP_CONNECT,
	IORING_OP_FALLOCATE,
	IORING_OP_OPENAT,
	IORING_OP_CLOSE,
	IORING_OP_FILES_UPDATE,
	IORING_OP_STATX,
	IORING_OP_READ, ...
```

https://github.com/axboe/liburing/blob/master/src/include/liburing/io_uring.h#L214

io_uring

Multiple requests on shared ring



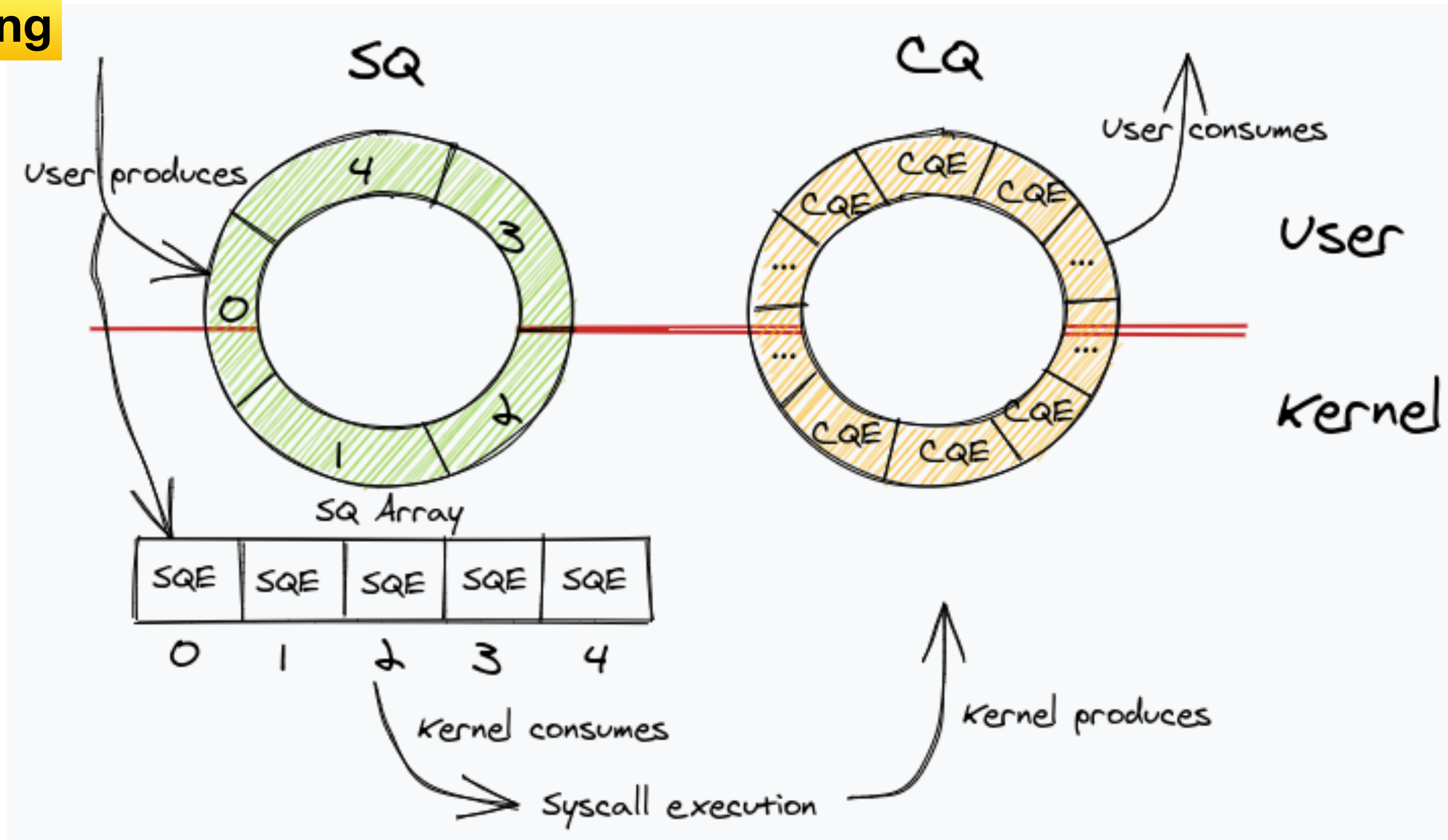
SQ: applications submit requests to a shared submission queue with user data attached

CQ: kernel asynchronously places the result into a ring with user data + status attached

```
struct io_uring_cqe {
  __u64    user_data; /* sqe->user_data value passthru */
  __s32    res;       /* result code for this event */
  __u32    flags;
```

https://github.com/axboe/liburing/blob/master/src/include/liburing/io_uring.h#L214

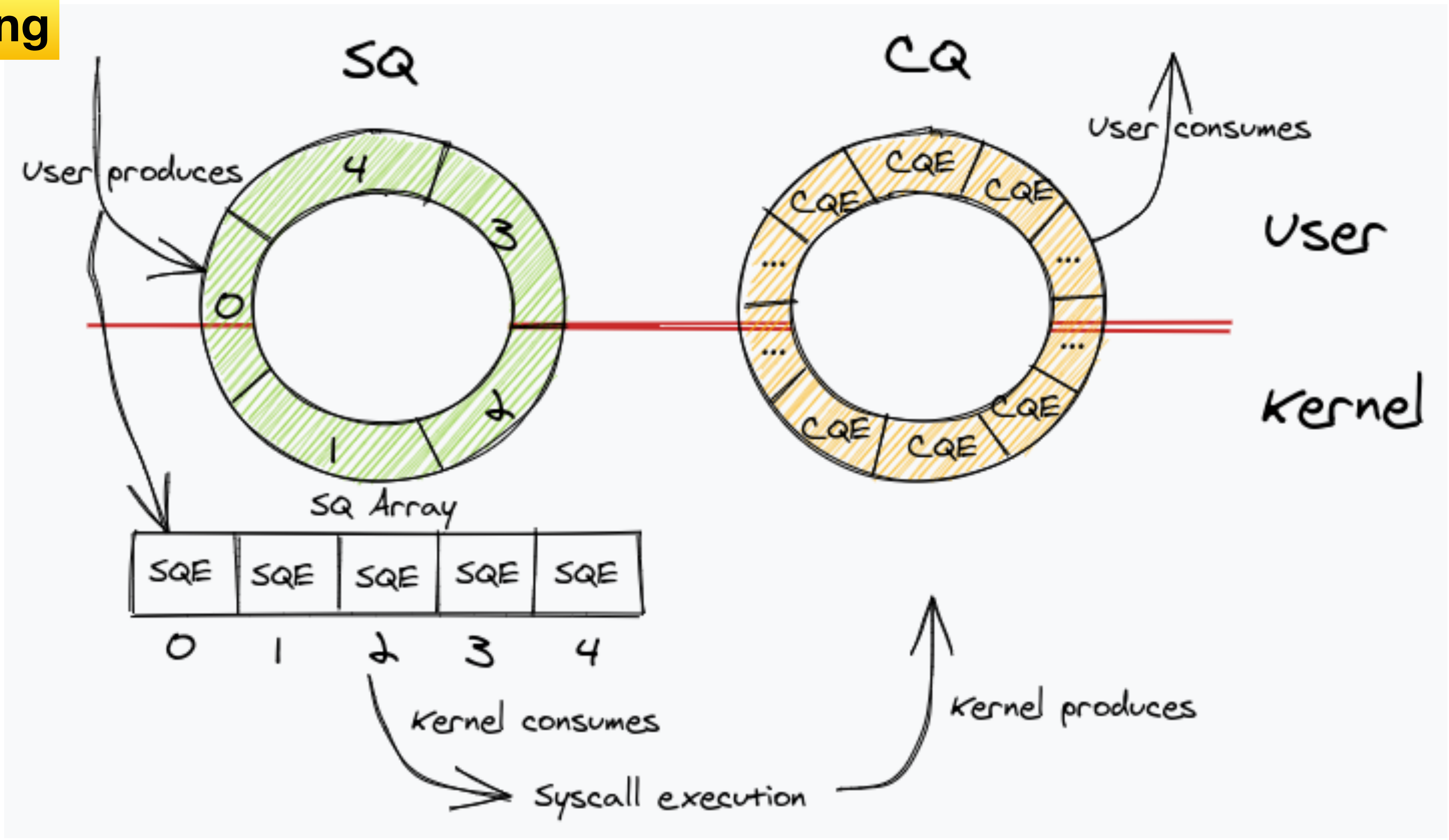
io_uring




```
io_uring_enter(5, 64, 0, 0, NULL, 8)     = 64
io_uring_enter(5, 64, 0, 0, NULL, 8)     = 64
io_uring_enter(5, 0, 1, IORING_ENTER_GETEVENTS, NULL, 8) = 0
io_uring_enter(5, 1, 0, 0, NULL, 8)      = 1
io_uring_enter(5, 1, 0, 0, NULL, 8)      = 1
io_uring_enter(5, 2, 0, 0, NULL, 8)      = 2
io_uring_enter(5, 2, 0, 0, NULL, 8)      = 2
io_uring_enter(5, 5, 0, 0, NULL, 8)      = 5
io_uring_enter(5, 5, 0, 0, NULL, 8)      = 5
io_uring_enter(5, 8, 0, 0, NULL, 8)      = 8
io_uring_enter(5, 8, 0, 0, NULL, 8)      = 8
```

image credit: https://developers.mattermost.com/blog/hands-on-iouring-go/

io_uring



io_uring has many features for even more throughput:

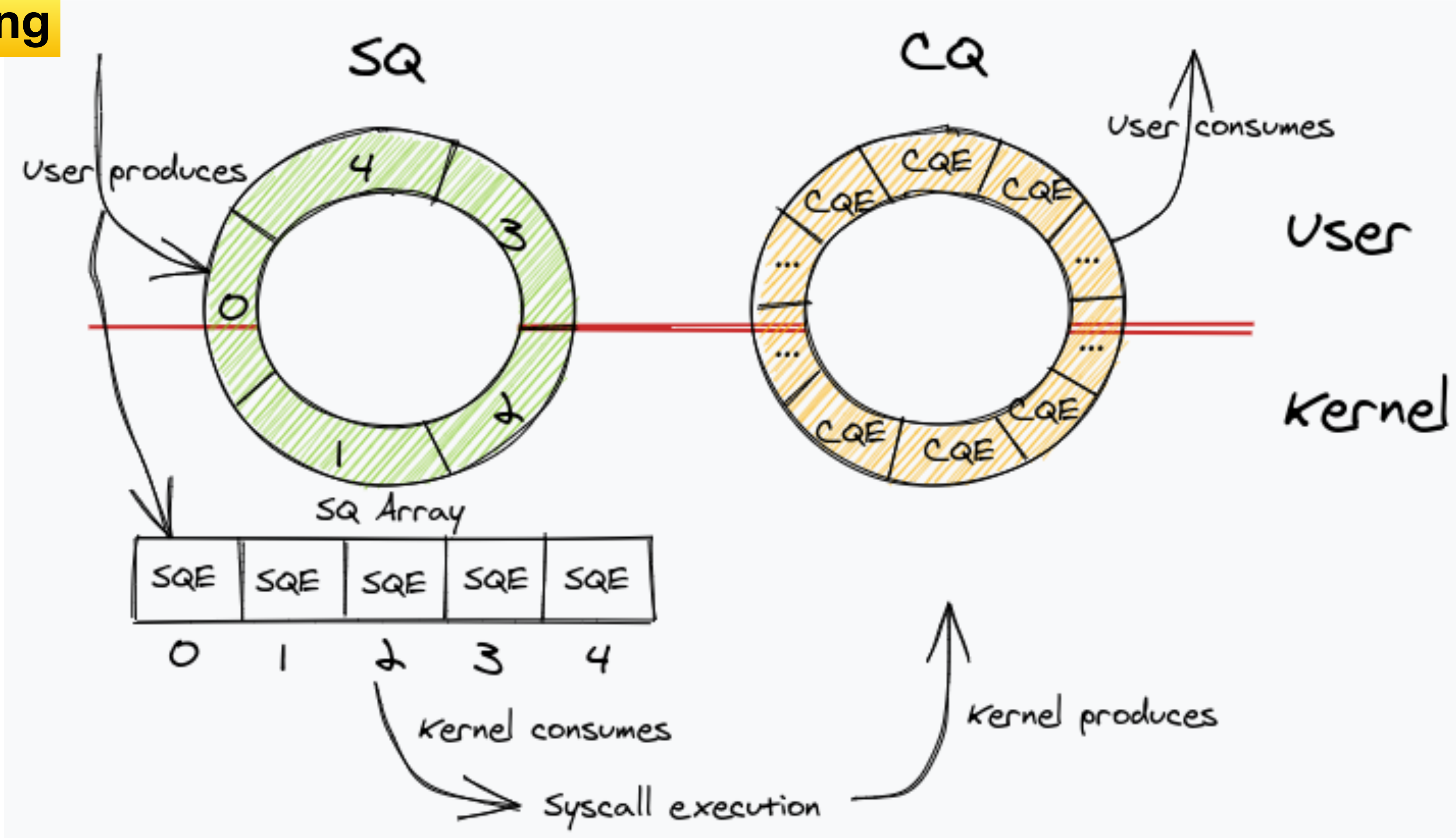
- "linked requests" and "barriers" for IO
- callback steering for responses across cores
- fixed buffers and descriptors to avoid page mapping

To take full advantage, the application must:

- submit IO requests in batches
- track a parallel data dependency graph across requests
- enforce linear buffer usage

image credit: https://developers.mattermost.com/blog/hands-on-iouring-go/

io_uring



io_uring has many similarities to other shared rings:

- how many userspace/kernel rings do we allocate?
- how big should each ring be?
- what does the runtime do when a ring is full?
- how many threads/cores should push to one ring?

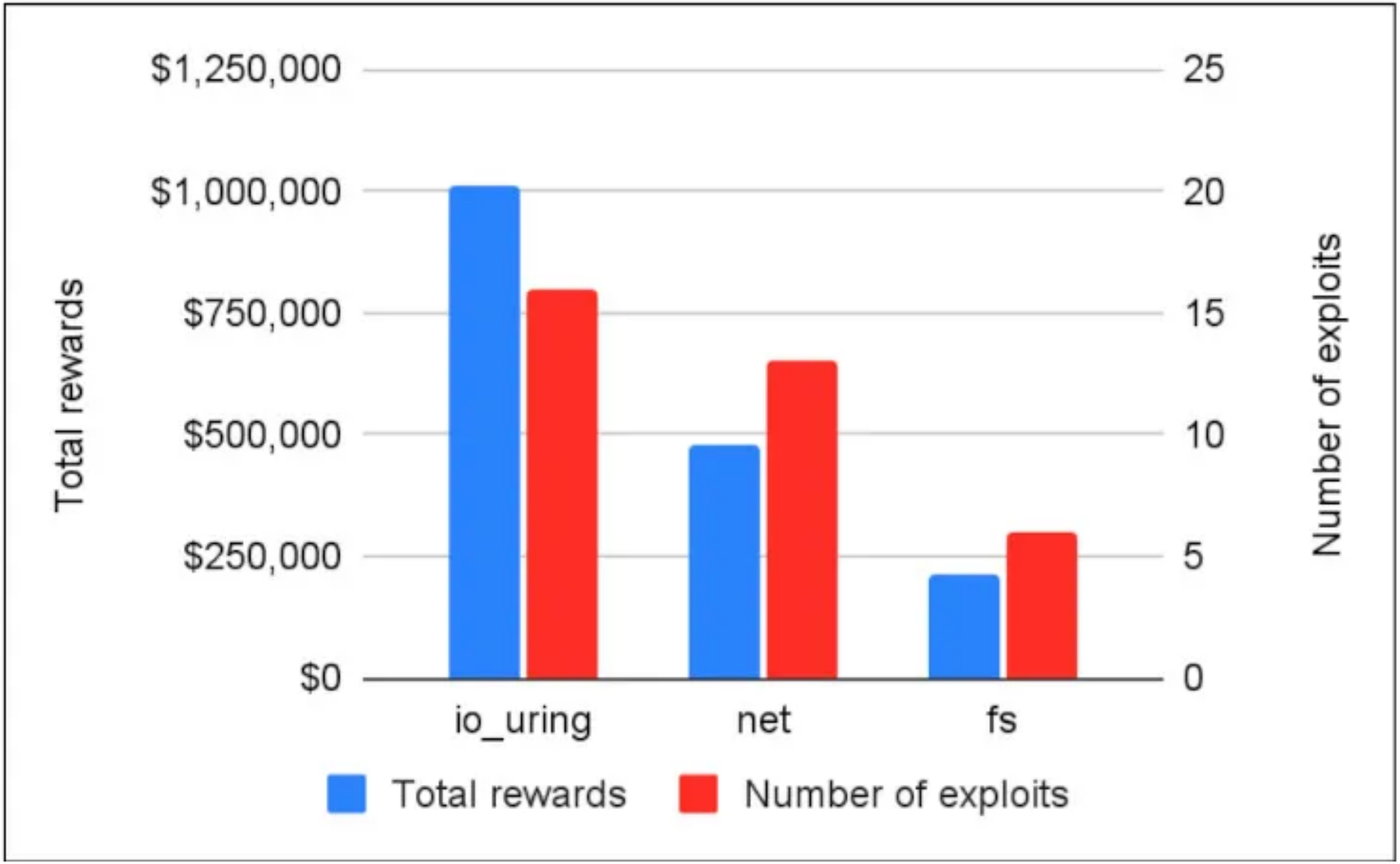
io_uring patterns show up and down the stack:

- Xen/KVM also have the same problems
- Docker for Desktop also plumbs macOS/Linux like this
- Security is an issue, as the Linux kernel isn't async-friendly

image credit: https://developers.mattermost.com/blog/hands-on-iouring-go/

io_uring

Google awards $1m worth of vulnerability reports in 2023 for....

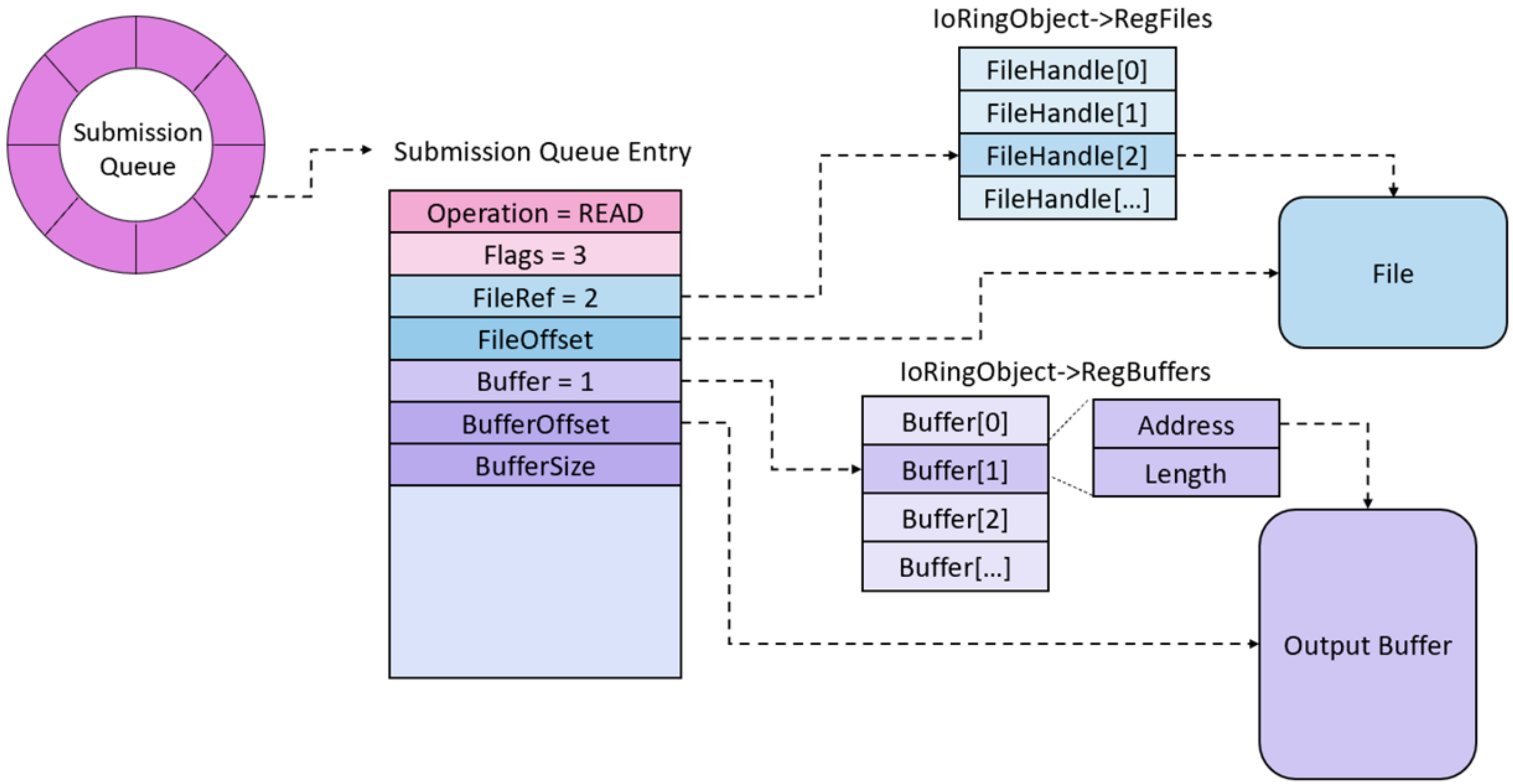


io_uring patterns show up up and down the stack:

- Xen/KVM also have the same problems
- Docker for Desktop also plumbs macOS/Linux like this
- Security is an issue, as the Linux kernel isn't async-friendly

image credit: https://developers.mattermost.com/blog/hands-on-iouring-go/

ioring

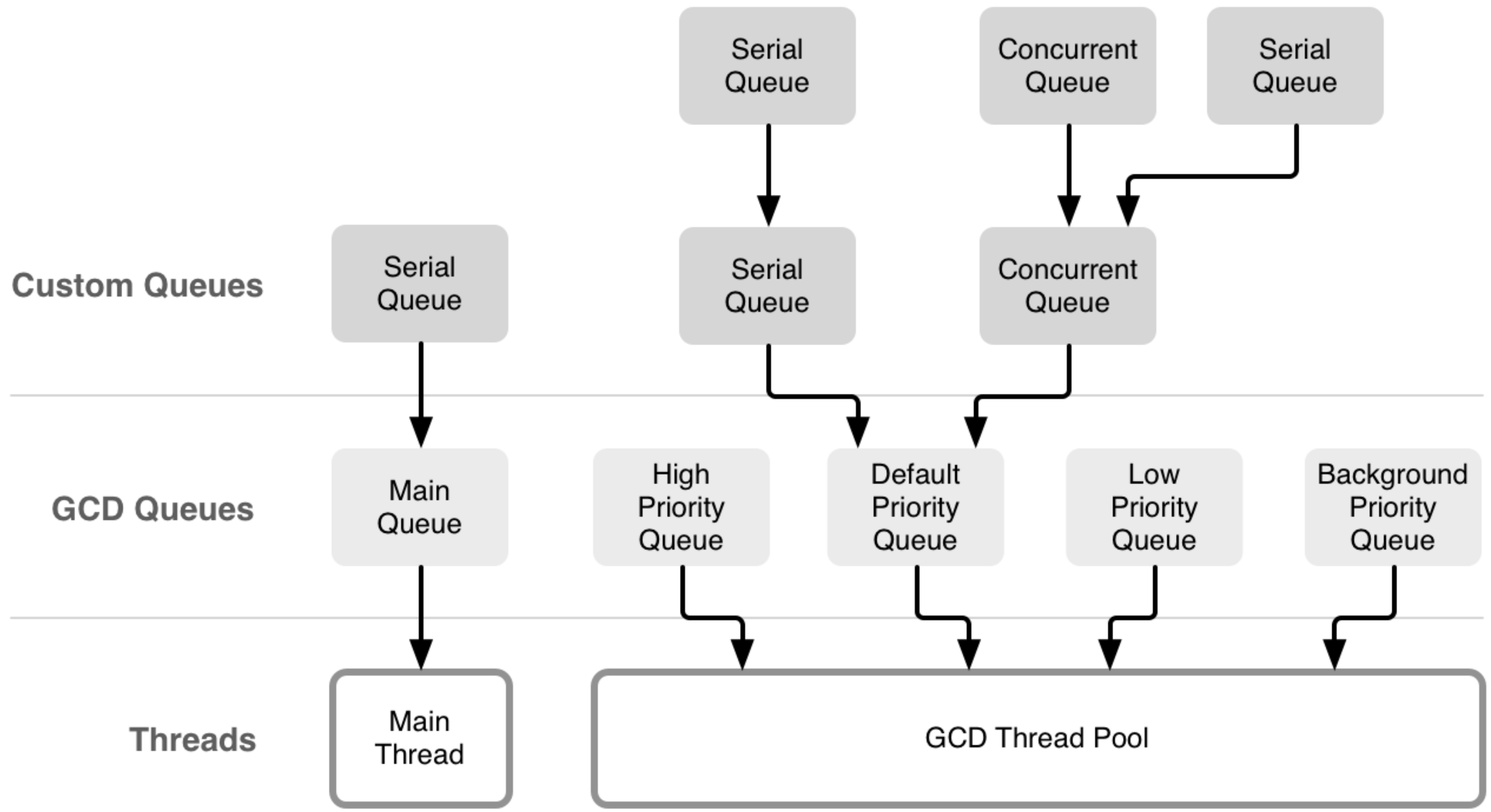


windows ioring is very similar to Linux:

- The NT kernel has long had full async ops (IOCP)
- The ioringapi.h has submission/completion queues
- Separate APIs for files (ioring) and networks (RIO)
- The overarching term is called "overlapping IO"

image credit: https://windows-internals.com/one-i-o-ring-to-rule-them-all-a-full-read-write-exploit-primitive-on-windows-11/

Grand Central Dispatch

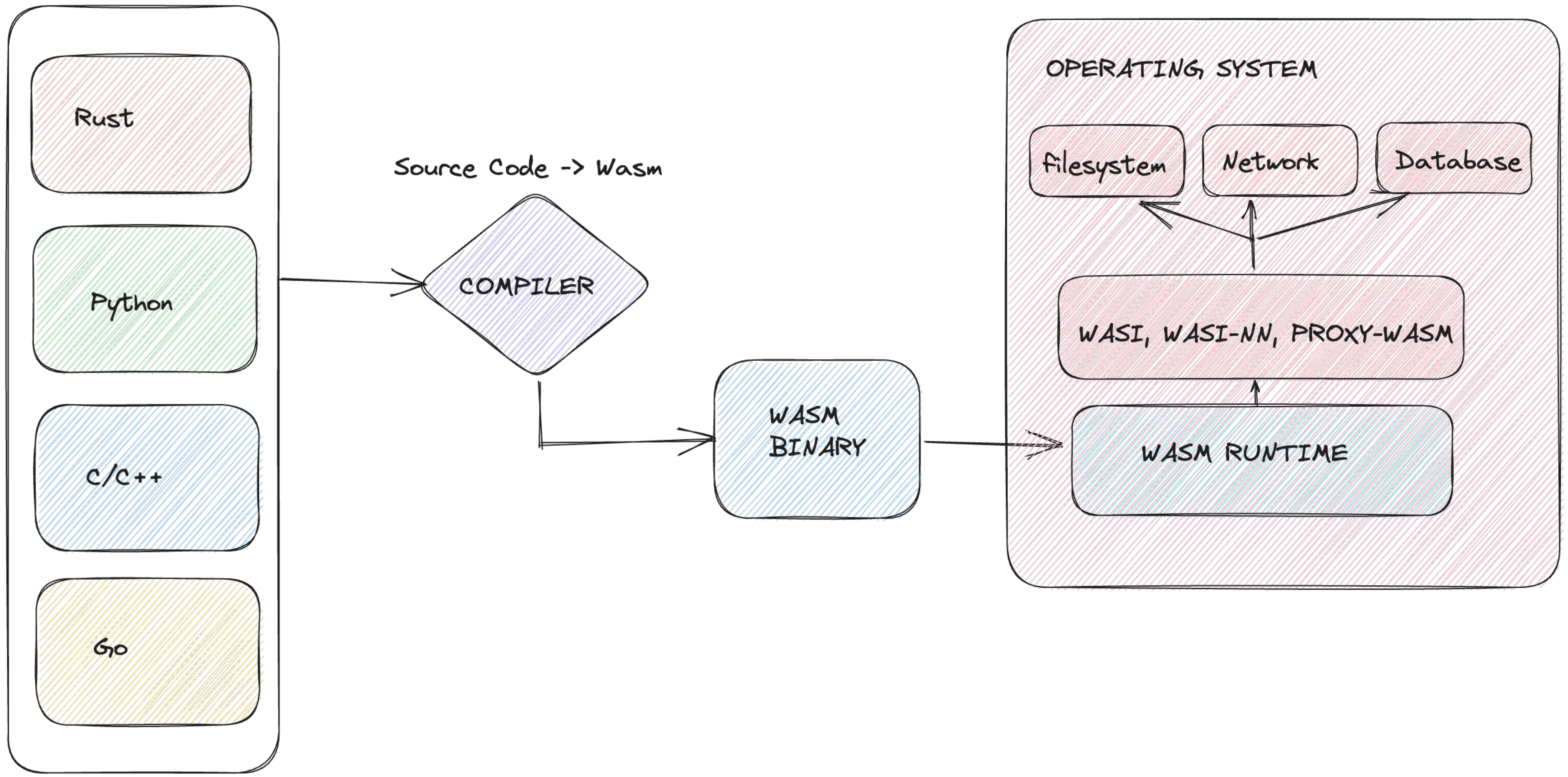


but macOS diverges dramatically!

- libdispatch is based on M:N threading
- the runtime defines a series of queues and priorities
- the kernel creates and destroys threads depending on backpressure
- the shared memory rings are hidden here (dynamically resized)

image credit: https://windows-internals.com/one-i-o-ring-to-rule-them-all-a-full-read-write-exploit-primitive-on-windows-11/

WebAssembly / WASI



and we have emerging WebAssembly ("WASI") system interfaces:

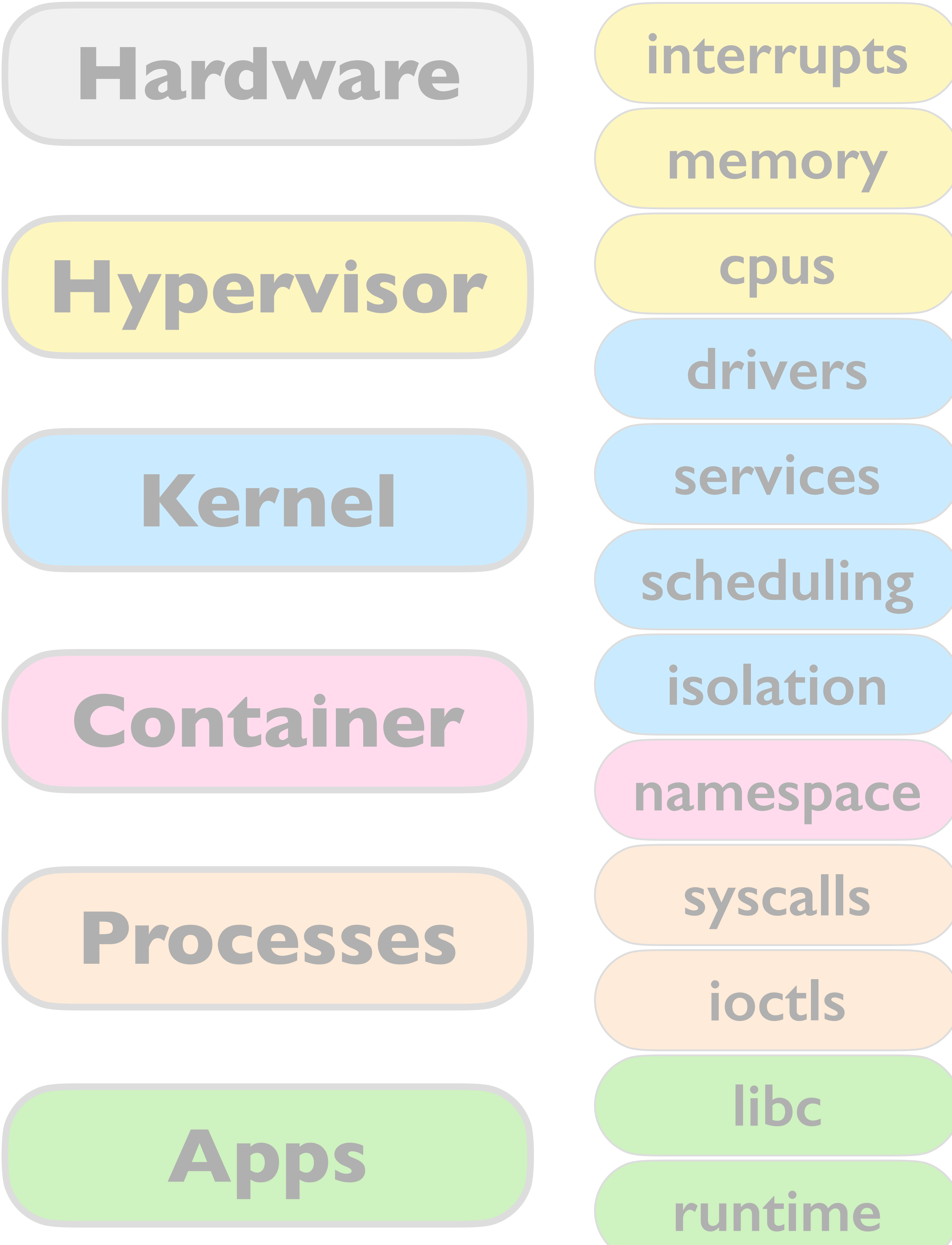
- these have no kernel/userspace, but are sandboxed
- **WASI programs interact with the outside world via *capabilities***
- capabilities are defined at link-time or at run-time
- the application bindings are high-level IO streams
- the WASM runtime can implement these using io_uring/etc.

image credit: https://collabnix.com/docker-and-wasm-containers-better-together/

POSIX is now firmly a historical myth

Instead, a modern stack needs to directly support these sorts of interfaces:

Linux (io_uring, eBPF, seccomp)
macOS (Grand Central Dispatch)
Windows (IOCP, ioring)
FreeBSD (aio, jails)
OpenBSD (kqueue, pledge)
wasm (wasi, browser)
JavaScript (webworkers)
Xen/KVM (paravirtual devices)
Bare metal (direct hardware)



Deploying software on modern infrastructure needs shared memory parallelism everywhere

What are some of these post-POSIX interfaces and do they share anything in common?

We're having a go at supporting this in OCaml 5 with our Eio library using the new effect handlers feature

```ocaml
module Main
 (Console: Mirage_types_lwt.CONSOLE)
 (Time: Mirage_types_lwt.TIME) = struct

  let start c _ =
    let rec loop = function
      | 0 -> Lwt.return_unit
      | n ->
        Console.log c "hello" >>= fun () ->
        Time.sleep_ns (Duration.of_sec 1) >>= fun () ->
        Console.log c "world" >>= fun () ->
        loop (pred n)
    in
    loop 4

end
```

ML functors for portability across hardware

Monadic cooperative concurrency

```
module Main
 (Console: Mirage_types_lwt.CONSOLE)
 (Time: Mirage_types_lwt.TIME) = struct

  let start c _ =
    let rec loop = function
      | 0 -> Lwt.return_unit
      | n ->
        Console.log c "hello" >>= fun () ->
        Time.sleep_ns (Duration.of_sec 1) >>= fun () ->
        Console.log c "world" >>= fun () ->
        loop (pred n)
    in
    loop 4

end
```

"Capabilities" to access drivers passed in as args

```
let start () =
  Eio_main.run @@ fun env ->
  for i = 0 to 5 do
     traceln "hello";
     Time.sleep (Stdenv.clock env) 1.0;
     traceln "world"
  done
```

Normal OCaml code. No functors!

Capabilities passed in as subtype-friendly value

```
let start () =
  Eio_main.run @@ fun env ->
  for i = 0 to 5 do
     traceln "hello";
     Time.sleep (Stdenv.clock env) 1.0;
     traceln "world"
  done
```

Can use imperative affordances in the language

Direct-style blocking! No monadic binds!

```ocaml
module Main
 (Console: Mirage_types_lwt.CONSOLE)
 (Time: Mirage_types_lwt.TIME) = struct

  let start c _ =
    let rec loop = function
      | 0 -> Lwt.return_unit
      | n ->
        Console.log c "hello" >>= fun () ->
        Time.sleep_ns (Duration.of_sec 1) >>= fun () ->
        Console.log c "world" >>= fun () ->
        loop (pred n)
    in
    loop 4
end
```

OCaml 5.0 (multicore+effects)

```ocaml
let start () =
  Eio_main.run @@ fun env ->
  for i = 0 to 5 do
     traceln "hello";
     Time.sleep (Stdenv.clock env) 1.0;
     traceln "world"
  done
```

```
open Printf

let _ = run (fun _ ->
  fork (fun _ ->
    printf "[t1] Sending 0\n";
    let v = xchg 0 in
    printf "[t1] received %d\n" v);
  fork (fun _ ->
    printf "[t2] Sending 1\n";
    let v = xchg 1 in
    printf "[t2] received %d\n" v))
[t1] Sending 0
[t2] Sending 1
[t2] received 0
[t1] received 1
```

We can now implement "fork" in OCaml

No monadic concurrency tricks required

And xchg for application-specified message passing

```
type _ Effect.t +=   Fork : (unit -> unit) -> unit t
                   | Yield : unit t

let fork f = perform (Fork f)
let yield () = perform Yield
let xchg v = perform (Xchg v)
```

Effect is an extensible type with a GADT for each effect

These helper functions actually raise the effect

```
type _ Effect.t +=   Fork : (unit -> unit) -> unit t
                   | Yield : unit t

let fork f = perform (Fork f)
let yield () = perform Yield
let xchg v = perform (Xchg v)
```

```
let run (main : unit -> unit) : unit =
  let xchger = ref None in
  let enqueue k v = Queue.push (continue k v) run_q in
  let dequeue () =
    if Queue.is_empty run_q then ()
    else Queue.pop run_q () in
  let rec spawn f =
    match f () with
    | () ->                    dequeue ()
    | exception e ->           dequeue ()
    | effect Yield, k ->       enqueue k (); dequeue ()
    | effect (Fork f), k ->    enqueue k (); spawn f
    | effect (Xchg n), k ->
        match !exchanger with
        |Some (n',k') -> xchger := None; enqueue k' n; continue k n'
        |None -> xchger := Some (n, k); dequeue ()
  in
  spawn main
```

Effects supply a one-shot continuation

Effects are handled alongside exceptions

Eio uses modern IO APIs with the new OCaml 5 effects

Things that worked well in 20 years of OCaml happiness	**Things to improve after 20 years of OCaml pain**
No forced preemption (either concurrency or parallelism)	Error handling mixed up monadic+exceptions+result
Arranging libraries as functors led to good systems hygiene	Deep functor stacks are just impenetrable to figure out
Strict execution model made systems bindings easy	Monadic concurrency led to high heap memory usage
Customising runloop for application is straightforward	Exception backtraces not preserved with Lwt/Async
Sequential OCaml performance is very very good.	Lifetimes/cancels difficult to track systematically

Eio looks like a DSL interpreted by the IO backend

io_uring has many fancy features for even more speed:

- “linked requests” and “barriers” for IO
- callback steering for responses across cores
- fixed buffers and descriptors to avoid page mapping

To take full advantage, the application must:

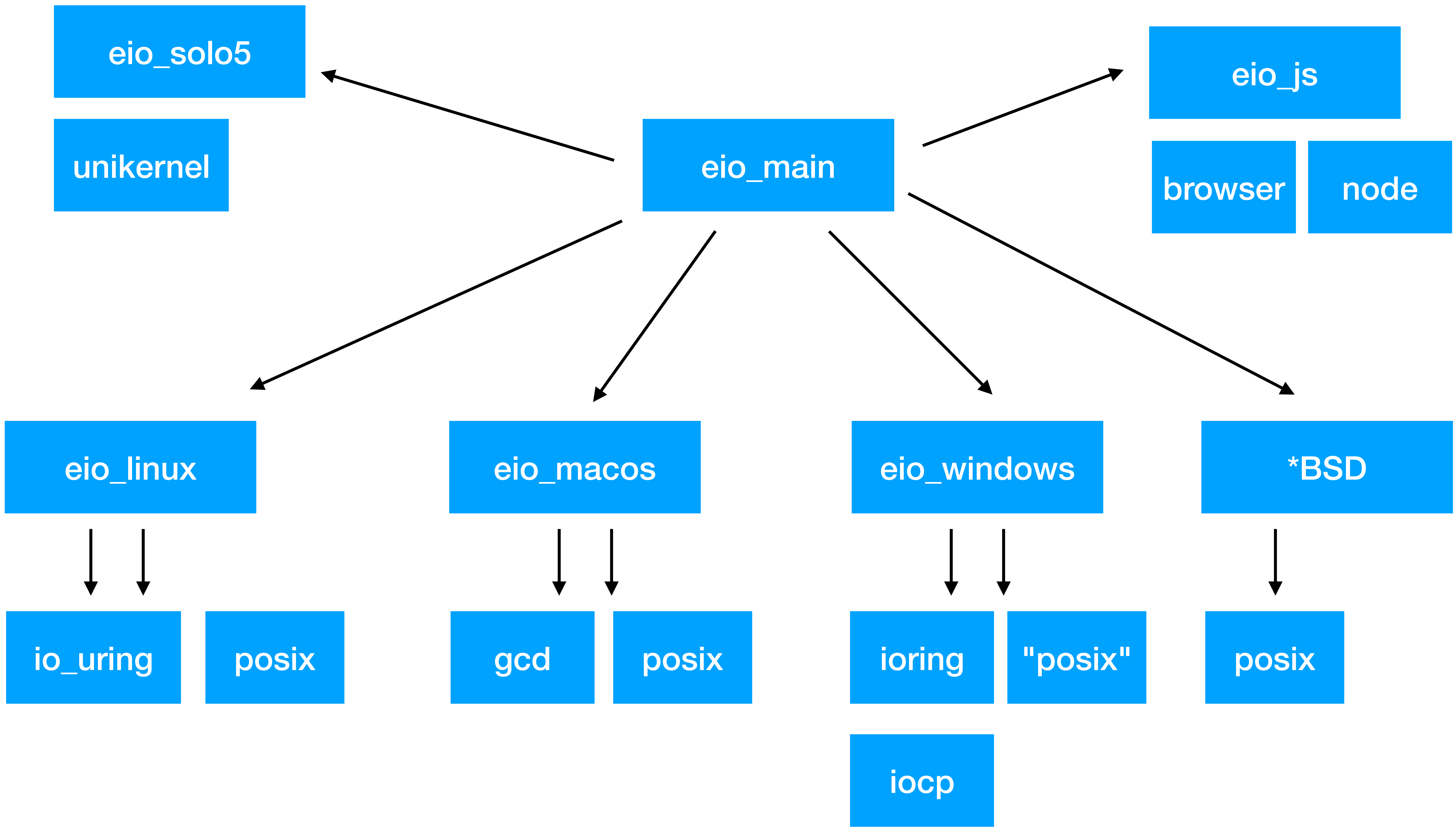
- submit IO requests in batches
- track a parallel data dependency graph across requests
- enforce linear buffer usage

```
# let () =
    let buf = Bytes.create 4096 in
    let rec copy () =
      match input stdin buf 0 4096 with
      | 0 -> ()
      | got ->
        output stdout buf 0 got;
        copy ()
    in
    copy ()
```

```
# let () =
    Eio_main.run @@ fun env ->
    Eio.(
      Flow.copy
        (Stdenv.stdin env)
        (Stdenv.stdout env)
    )
```

The Eio library spawns fibres to keep the uring filled asynchronously

eio has a “low-level” interface to each supported system.
Hard to program portably but everything is exposed to the programmer.



eio has a “low-level” interface to each supported system.
Hard to program portably but everything is exposed to the programmer.

Linux is increasingly adding many new extensions
To take advantage of these, we do type-safe bindings to io_uring with tight integration with the OCaml GC

```
# let with_id_full t fn datum ~extra_data =
   match Heap.alloc t.data datum ~extra_data with
  | exception (Invalid_argument _ as ex) -> check t; raise ex
  | entry ->
 let ptr = Heap.ptr entry in
 let has_space = fn ptr in
 if has_space then Some entry
 else (ignore (Heap.free t.data ptr : a); None)
```

```
# let read t ~file_offset fd buf user_data =
  with_id_full t (fun id ->
    Uring.submit_read t.uring fd id buf file_offset
  ) user_data ~extra_data:buf
```

```
# let fn_on_ring fn t =
  match fn t.uring with
  | Uring.Cqe_none -> None
  | Uring.Cqe_some { user_data_id; res } ->
    let data = Heap.free t.data user_data_id in
    Some { result = res; data }
```

eio has a “low-level” interface to each supported system.
Hard to program portably but everything is exposed to the programmer.

eio has a “high-level” portable interface for applications.
Does not expose ambient resources, but instead an (OCaml) object/class-based interface that exposes capabilities **and** a functional direct style of IO

```
type Stdenv.t = <
    stdin  : Flow.source;
    stdout : Flow.sink;
    stderr : Flow.sink;
    net : Net.t;
    domain_mgr : Domain_manager.t;
    clock : Time.clock;
    mono_clock : Time.Mono.t;
    fs : Fs.dir Path.t;
    cwd : Fs.dir Path.t;
    secure_random : Flow.source;
    debug : Debug.t;
  >

val stdin  : <stdin  : #Flow.source as 'a; ..> -> 'a
val stdout : <stdout : #Flow.sink   as 'a; ..> -> 'a
val stderr : <stderr : #Flow.sink   as 'a; ..> -> 'a
val secure_random : <secure_random : #Flow.source as 'a; ..> -> 'a
```

```
(** [initialise ~ctx ~endpoint ~email priv] constructs a [t] by
    looking up the directory and account of [priv] at [endpoint]. If no
    account is registered yet, a new account is created with contact
    information of [email]. The terms of service are agreed on. *)
val initialise : < net : #Eio.Net.t; .. > -> endpoint:Uri.t -> ?email:string ->
    X509.Private_key.t -> (t, [> `Msg of string ]) result

(** [sign_certificate ~ctx solver t sleep csr] orders a certificate for
      the names in the signing request [csr], and solves the requested
      challenges. *)
val sign_certificate : < net : #Eio.Net.t; .. > ->
    solver -> t -> (int -> unit) ->
    X509.Signing_request.t ->
    (X509.Certificate.t list, [> `Msg of string ]) result
```

```
let cert_dir env = Stdenv.cwd env
let token_dir env = Eio.Path.(Stdenv.fs env / “tokens”)

let priv_pem = Eio.Path.(load (cert_dir / priv_pem)) in
let csr_pem = Eio.Path.(load (cert_dir / csr_pem)) in
let* account_key = X509.Private_key.decode_pem (Cstruct.of_string priv_pem) in
let* request = X509.Signing_request.decode_pem (Cstruct.of_string csr_pem) in
let solver =
   match solver, acme_dir, ip, key with
   | _, Some path, None, None ->
     traceln "using http solver, writing to %s" path);
     let solve_challenge _ ~prefix:_ ~token ~content =
       let path = Eio.Path.(token_dir / path / token) in
       Eio.Path.save ~create:(`Or_truncate 0o600) path content;
       Ok ()
```

The Wild West of post-POSIX IO

Deploying software on modern hardware uses shared memory parallelism all over

Modern runtimes need to not block:

- shared memory channels run throughout the stack
- make rings safer and easier to share.
- POSIX is anathema to concurrent, parallel IO
- ...but POSIX is still around making trouble

What are some of these post-POSIX interfaces and do they share anything in common?

Design for post-POSIX first in your runtime/language:

- io_uring forces you to design for concurrency first
- macOS is an awkward blocker here with GCD
- but Windows has the best-of-breed support
- the evolution of WASI[X]/WALI are critical here

We're having a go at supporting this in OCaml 5 with our Eio library using the new effect handlers feature

eio does (https://github.com/ocaml-multicore/eio):

- zero-copy memory that can be mapped elsewhere
- programmer-controlled context switching
- lower pauses due to reduced GC activity + batching
- very high throughput IO by default
- reasonable migration path from Lwt/Async
- working backtraces and capabilities by default