Operating Systems

I/O, Hard Drives

Lecture Overview

- Canonical Device
- Direct Memory Access
- Hard Drives
- Solid State Drives

Last Week

Concurrency

- Semaphores

I/O Devices

What to do?

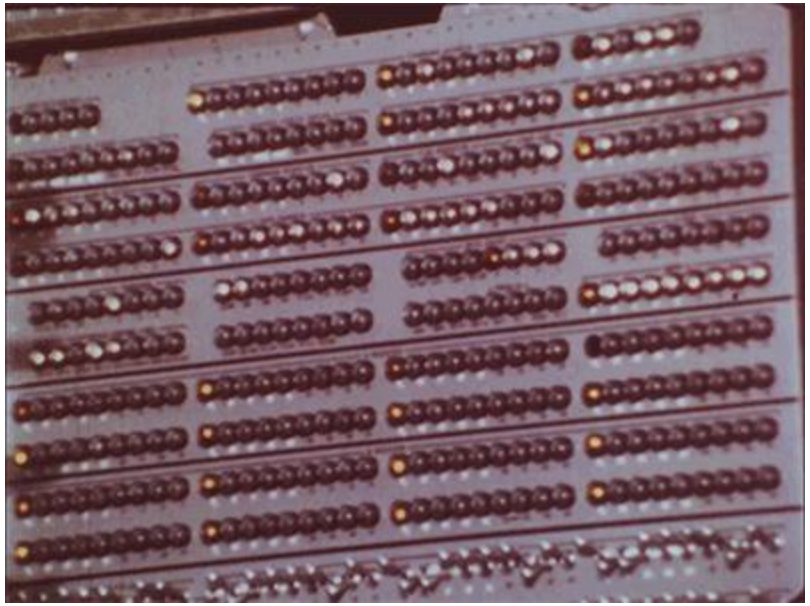
Motivation

What purpose is a computer that doesn’t interact with the outside world?

A bit of history

Original UI

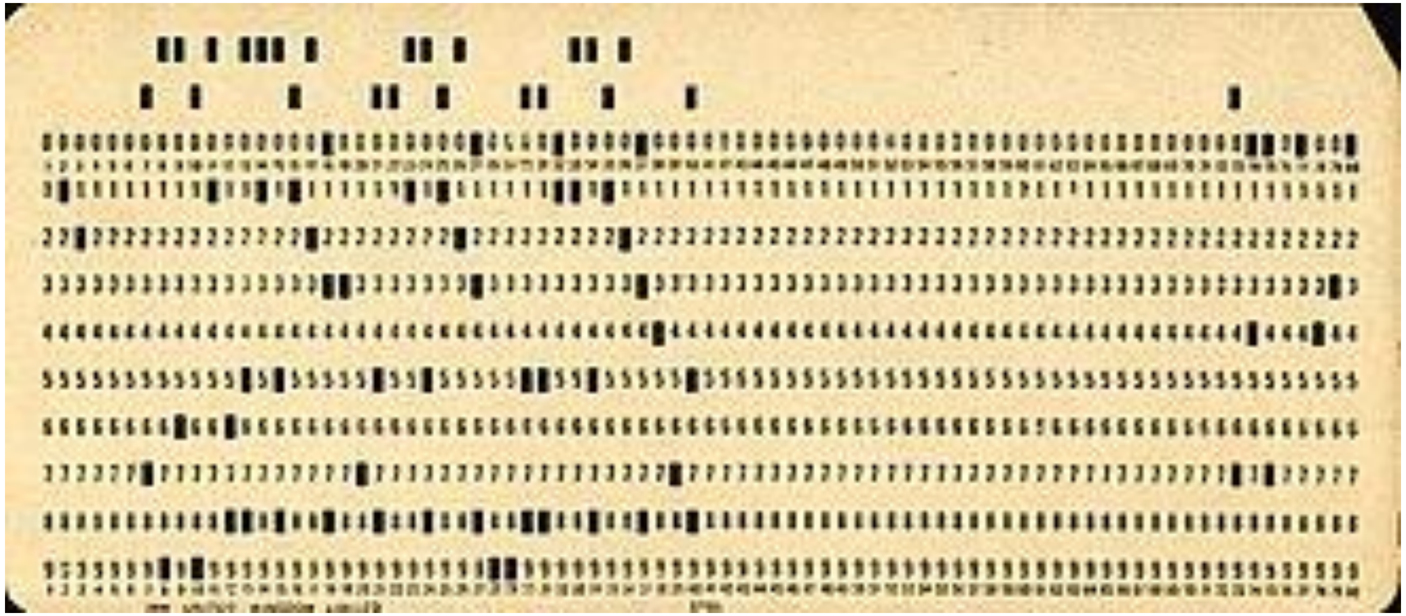
- Physical switches Input
- Lights Output



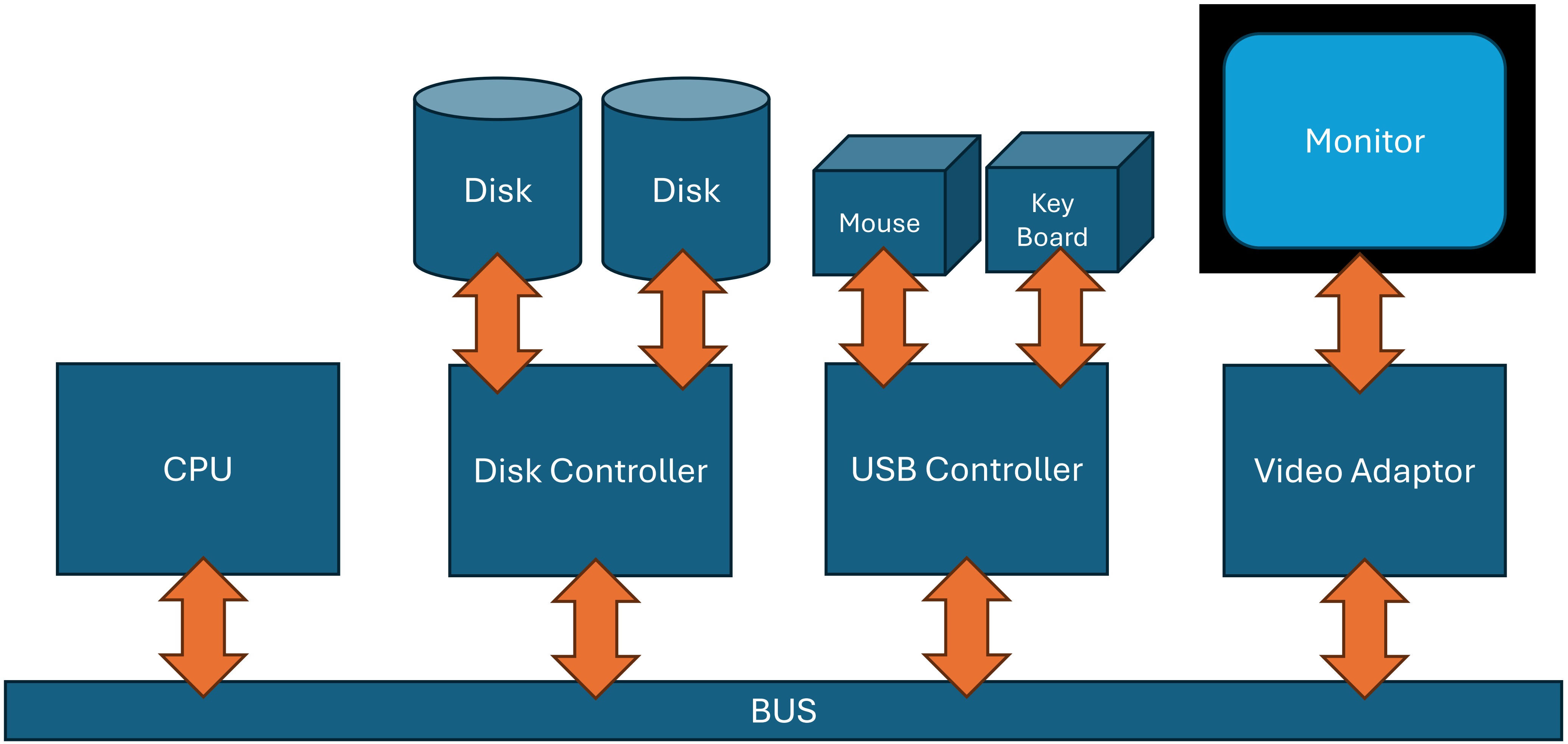
A bit of history

Punch Cards

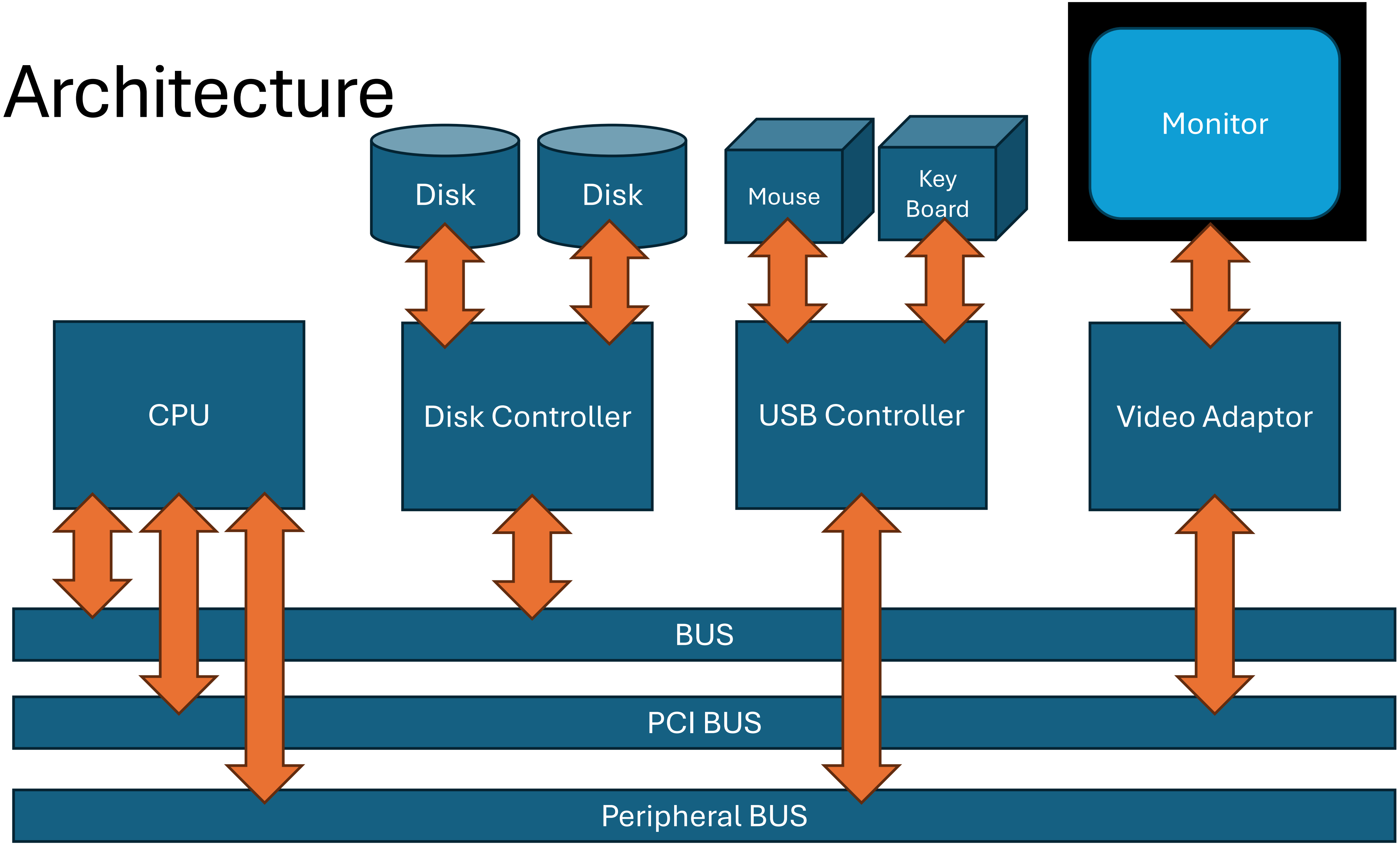
- Detect holes (physically)
- Print holes (physically)



Architecture



Architecture



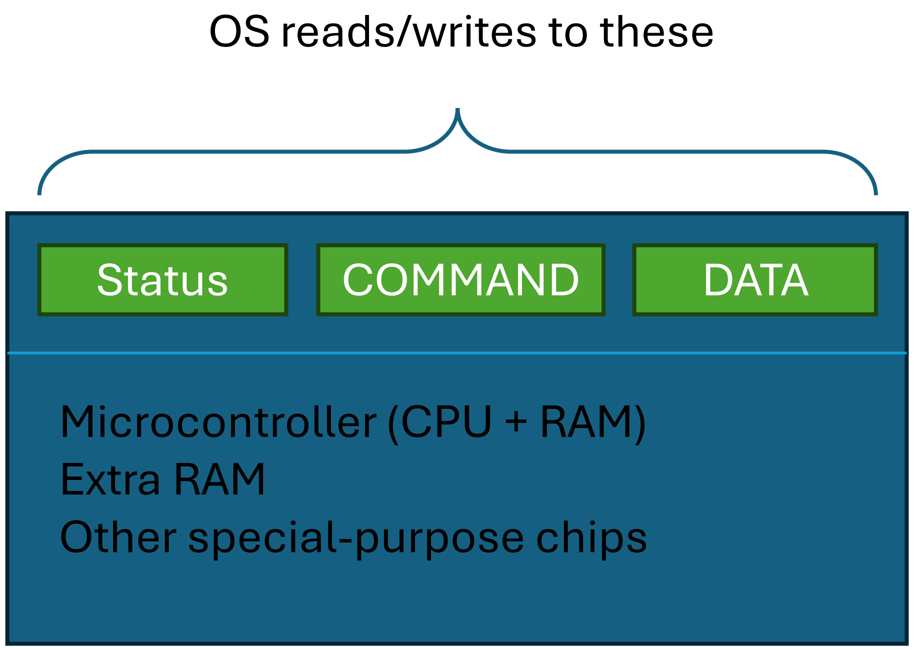
Device Controllers

- Local Buffer Storage
- Set of Special Purpose Registers

Canonical Device

Common Registers

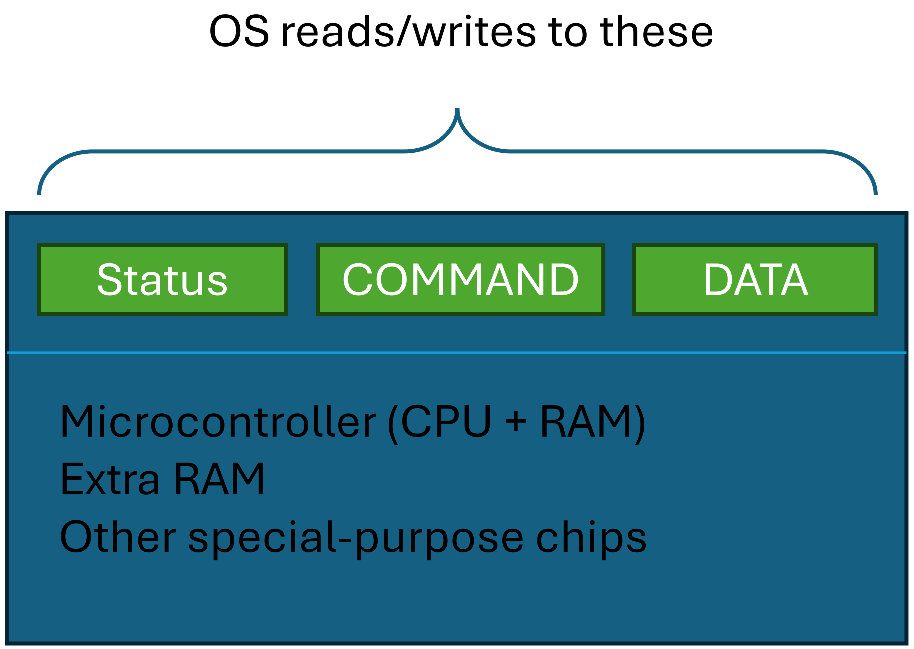
- Writing to a register may cause something to happen
- Reading from a register may cause something to happen
- Both can cause something to happen
- What you read isn't what your wrote



Example

```
while (STATUS == BUSY)
      ; // spin
Write data to DATA register
Write command to COMMAND register
while (STATUS == BUSY)
      ; // spin
```

This is a very simplified example



Example

```
while (STATUS == BUSY)
      ; // spin
Write data to DATA register
Write command to COMMAND register
while (STATUS == BUSY)
      ; // spin
```

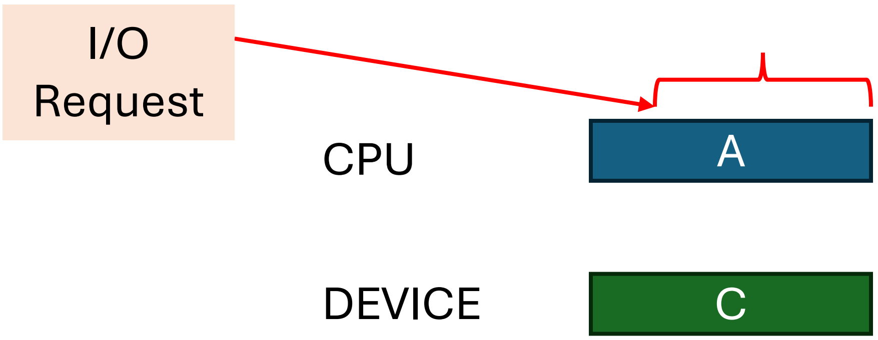
Burning CPU
(probably Kernel mode)

CPU A

DEVICE

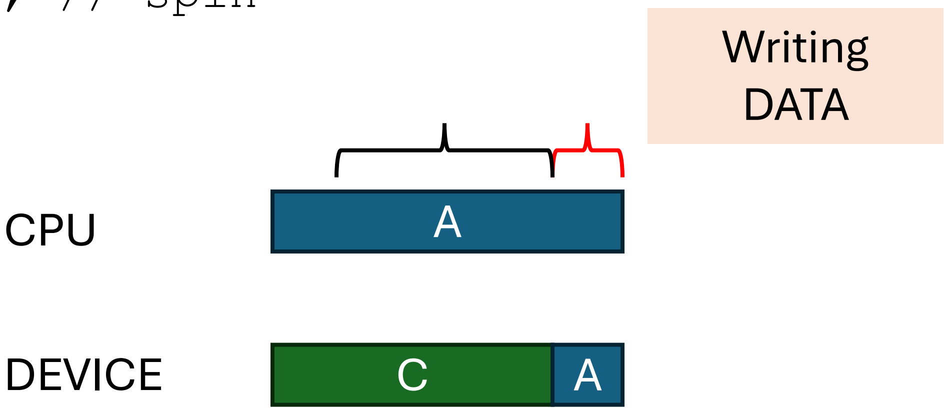
Example

```
while (STATUS == BUSY)
        ; // spin
Write data to DATA register
Write command to COMMAND register
while (STATUS == BUSY)
        ; // spin
```



Example

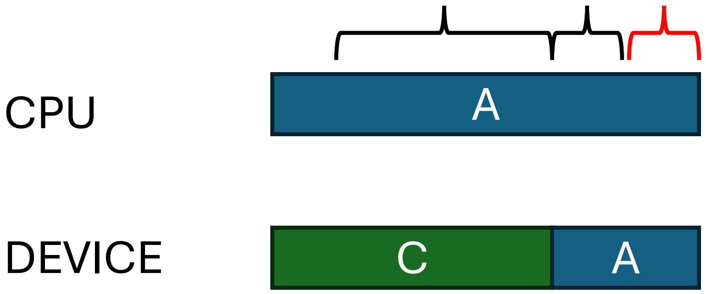
```
while (STATUS == BUSY)
      ; // spin
Write data to DATA register
Write command to COMMAND register
while (STATUS == BUSY)
      ; // spin
```



Example

```
while (STATUS == BUSY)
      ; // spin
Write data to DATA register
Write command to COMMAND register
while (STATUS == BUSY)
      ; // spin
```

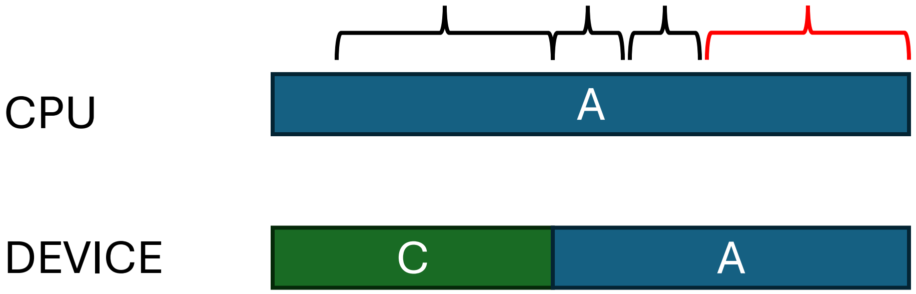
Writing COMMAND



Example

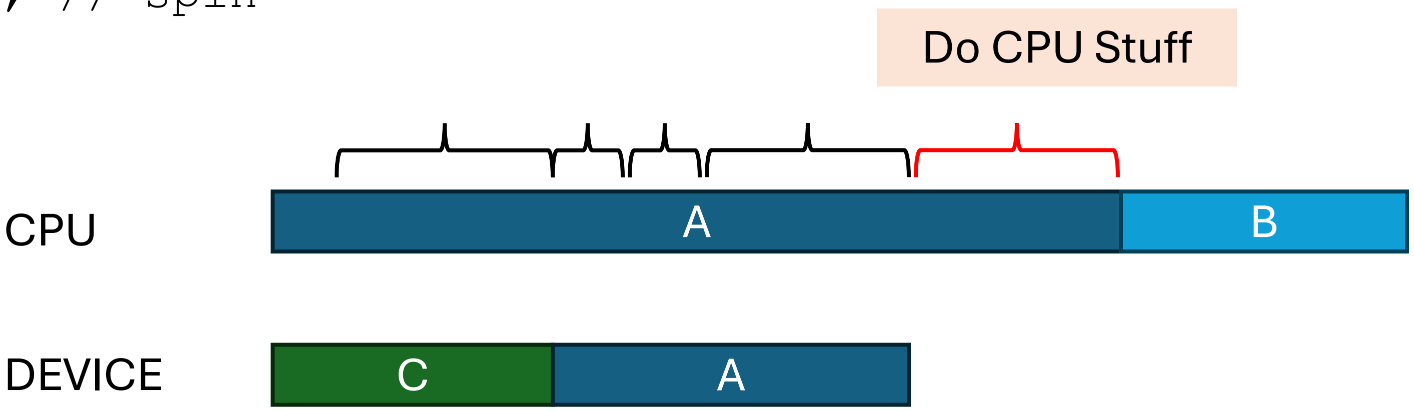
```
while (STATUS == BUSY)
      ; // spin
Write data to DATA register
Write command to COMMAND register
while (STATUS == BUSY)
      ; // spin
```

Do I/O Stuff



Example

```
while (STATUS == BUSY)
      ; // spin
Write data to DATA register
Write command to COMMAND register
while (STATUS == BUSY)
      ; // spin
```

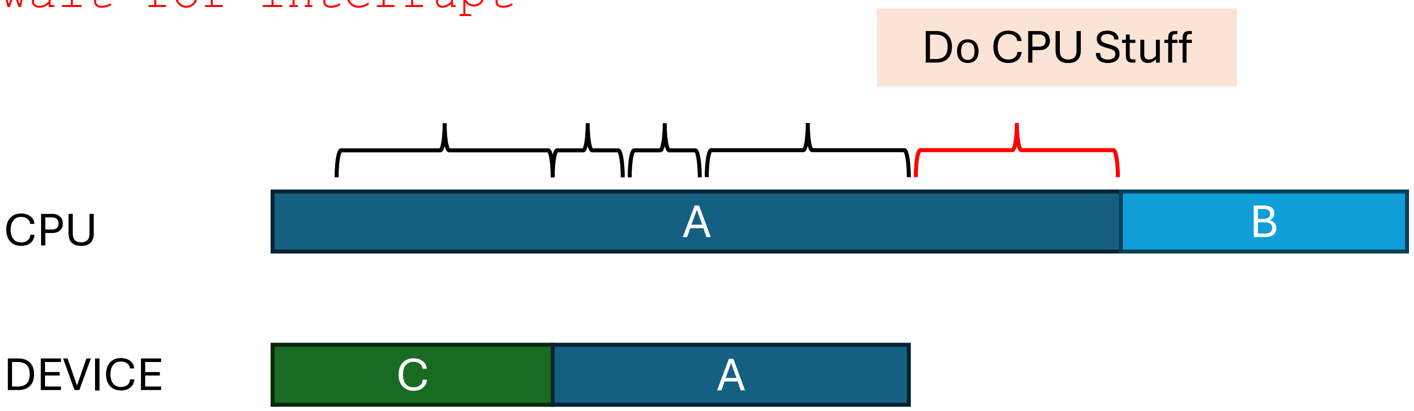


Example

```
while (STATUS == BUSY)
        wait for interrupt
Write data to DATA register
Write command to COMMAND register
while (STATUS == BUSY)
        wait for interrupt
```

Feels pretty inefficient…

Couldn’t we instead of **BUSY** waiting, perhaps we can perform an **interrupt**

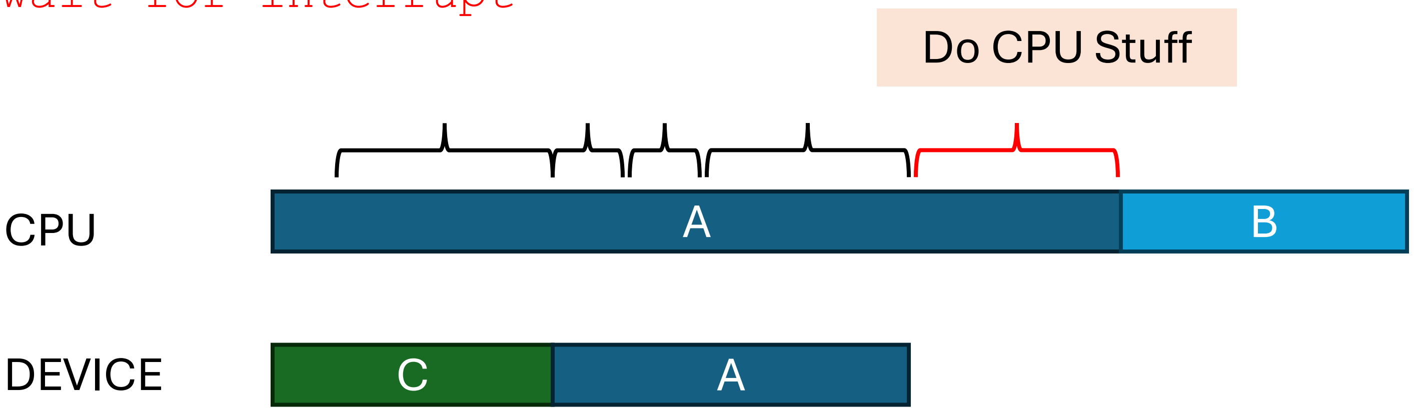


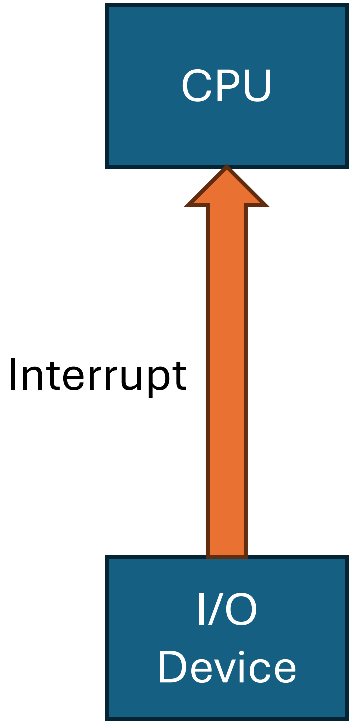
Example

Feels pretty inefficient…

Couldn’t we instead of **BUSY** waiting, perhaps we can perform an **interrupt**

```
while (STATUS == BUSY)
      wait for interrupt
Write data to DATA register
Write command to COMMAND register
while (STATUS == BUSY)
      wait for interrupt
```





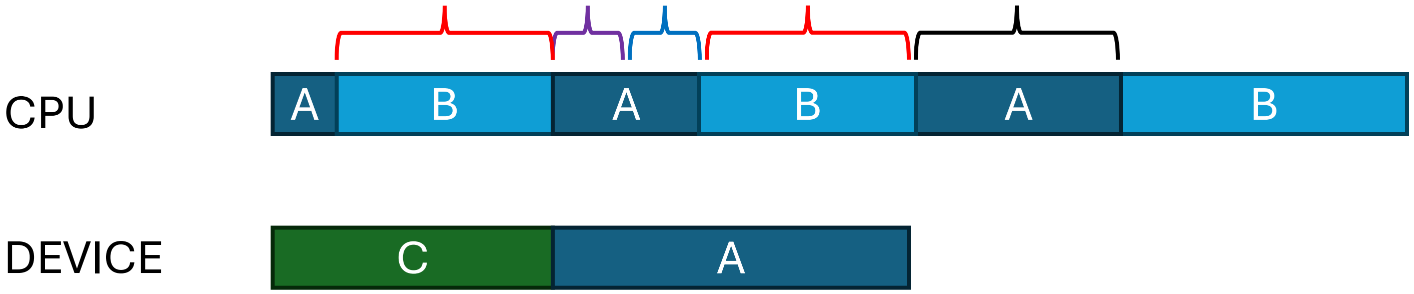
Example

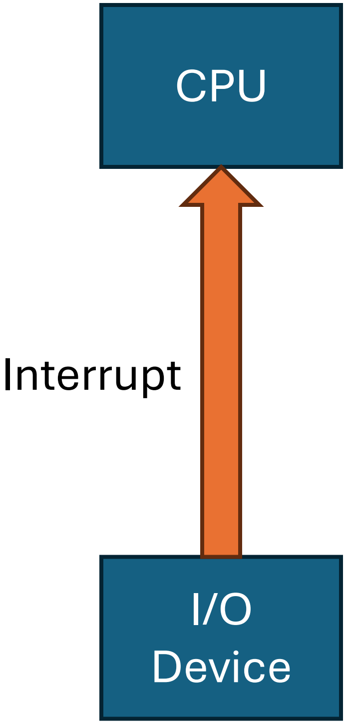
```
while (STATUS == BUSY)
      wait for interrupt
Write data to DATA register
Write command to COMMAND register
while (STATUS == BUSY)
      wait for interrupt
```

Feels pretty inefficient…

Couldn't we instead of **BUSY** waiting, perhaps we can perform an **interrupt**

Do CPU Stuff





Interrupts vs Polling

Polling

- Wastes CPU time
- Can work on fast devices

Interrupts

- Interrupt overheads
- Can result in live-lock (interrupt hell)
  - Better to ignore interrupts while some make progress
- Interrupt coalescing

Hybrid

- Spin, then use interrupts

Connecting I/O Devices

How is it done?

Protocol Variants

Port-mapped I/O

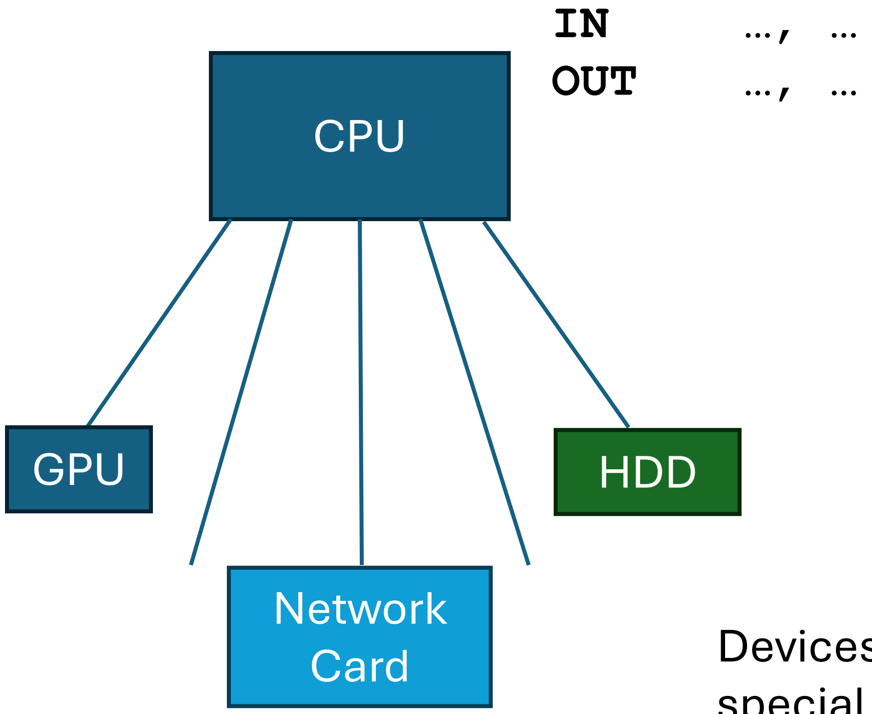
- Have an address space reserved for devices
- Use specialised CPU instruction (in, out) to read/write to this space
- Separates devices from normal memory
- Often one-byte at a time
- Examples: x86

Memory-mapped I/O

- Device registers are simply mapped into RAM
- OS reads these locations like any other location
- Simplified implementation
- Should not be **cached**

Protocol Variants

Port-mapped I/O



Devices have special **ports**

Memory-mapped I/O

Direct Memory Access

What would you say, you do around here?

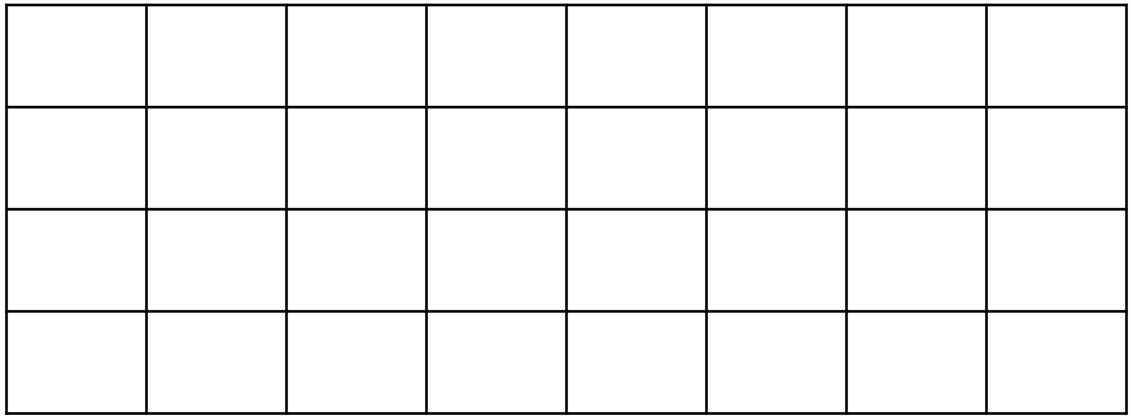
Problem

I/O Device

- I would like to copy some data into memory

CPU

- That is my job!



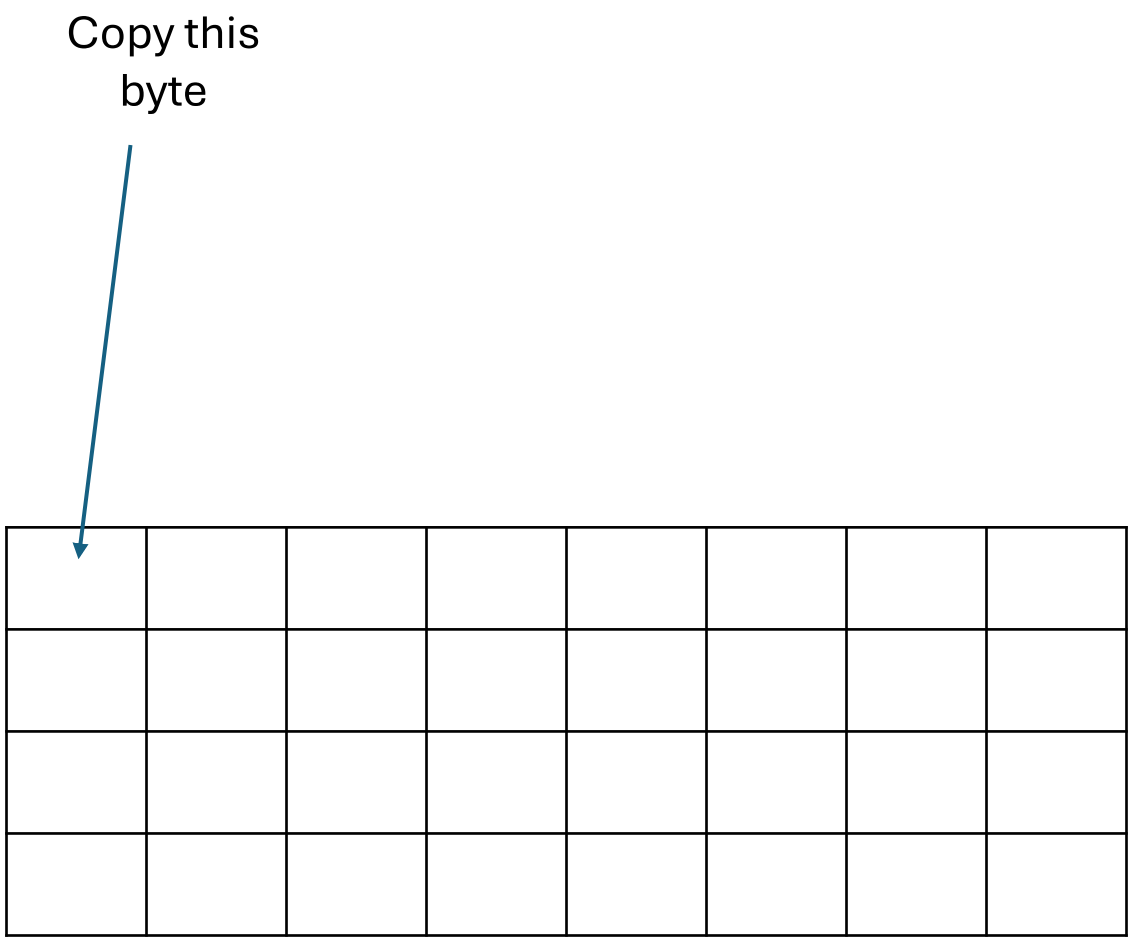
Problem

I/O Device

- I would like to copy some data into memory

CPU

- That is my job!



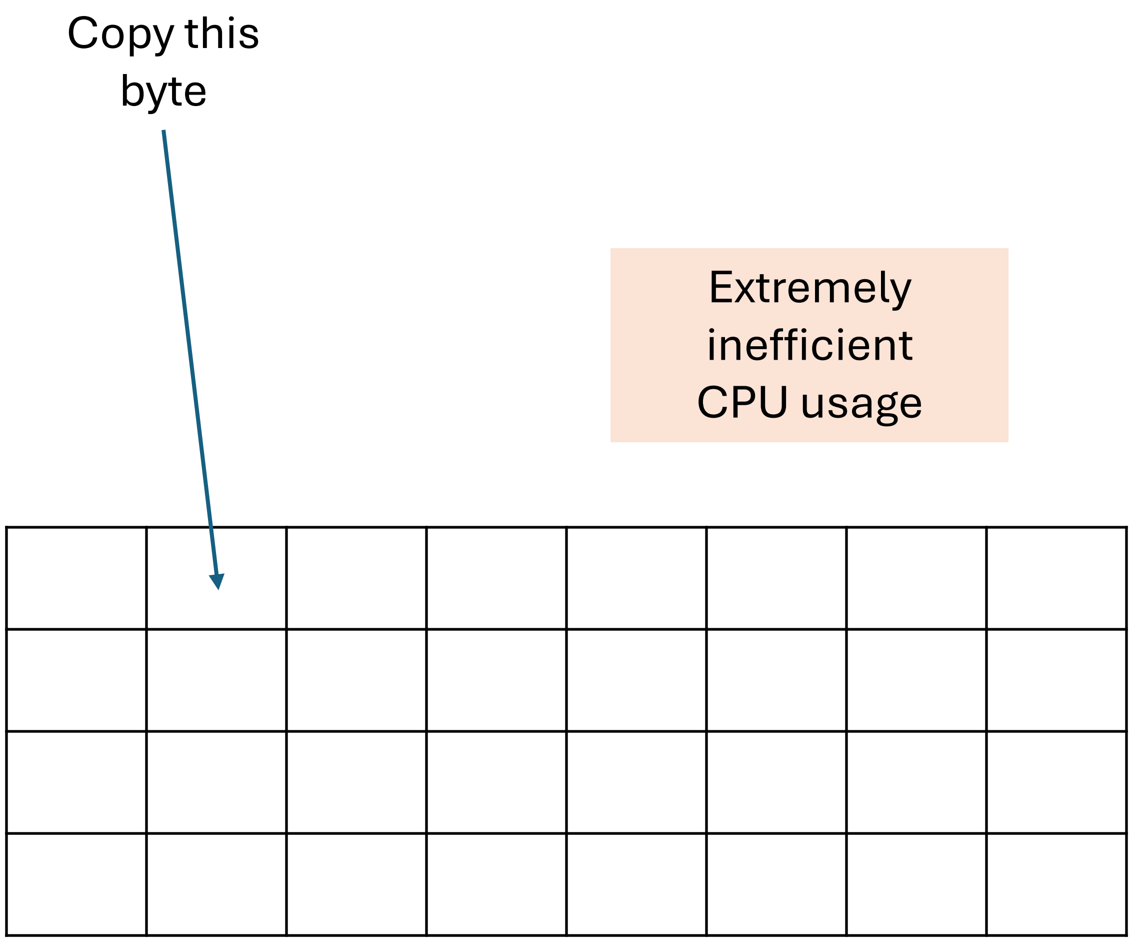
Problem

I/O Device

- I would like to copy some data into memory

CPU

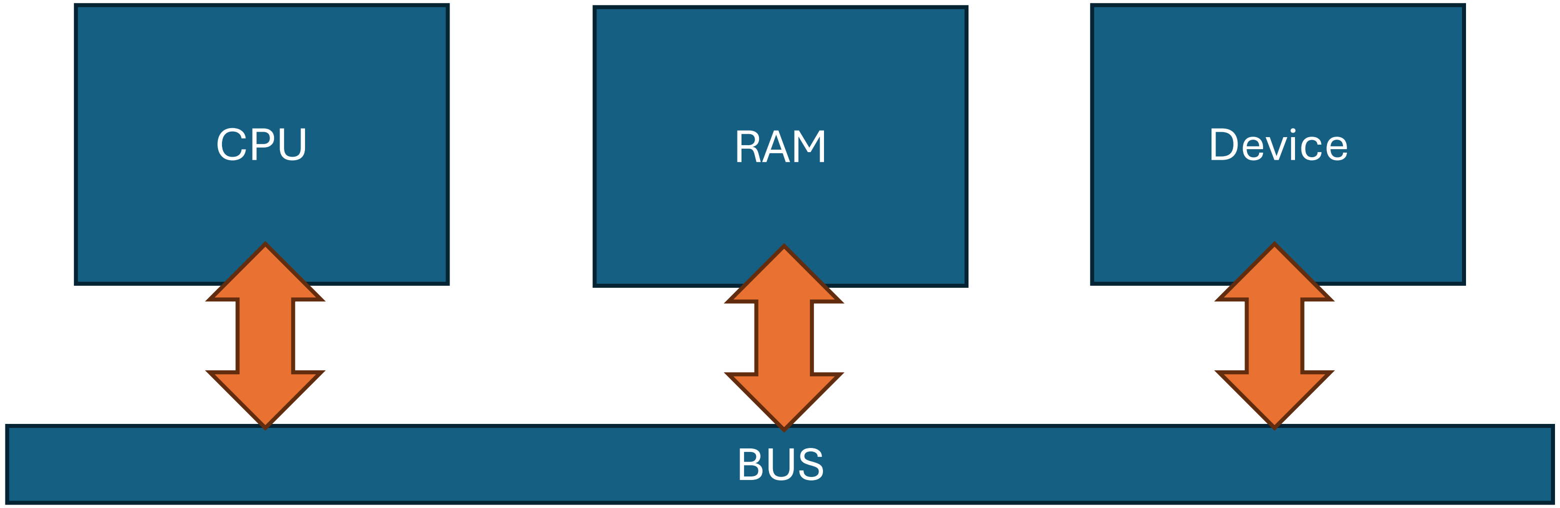
- That is my job!



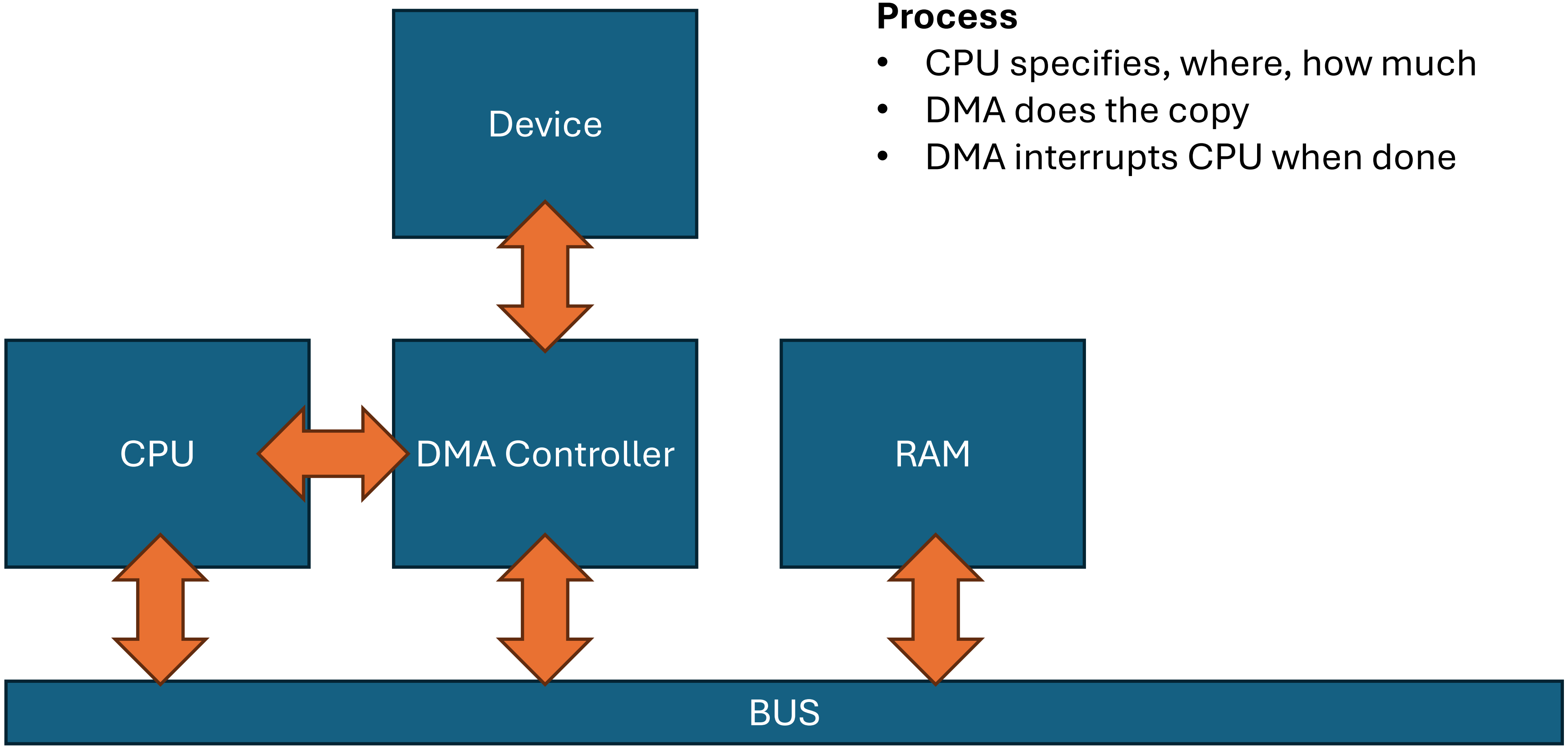
In Pictures: Programmed I/O

Process

- CPU copies a byte
- CPU copies a byte
- …



In Pictures: Direct Memory Access



Process

- CPU specifies, where, how much
- DMA does the copy
- DMA interrupts CPU when done

Copying can go both way

Updating our model

PIO

```
while (STATUS == BUSY)
        wait for interrupt
Write data to DATA register
Write command to COMMAND register
while (STATUS == BUSY)
        wait for interrupt
```

DMA

```
while (STATUS == BUSY)
        wait for interrupt
Write data to DATA register
Write command to COMMAND register
while (STATUS == BUSY)
        wait for interrupt
```

Device Drivers

Writing lots of code

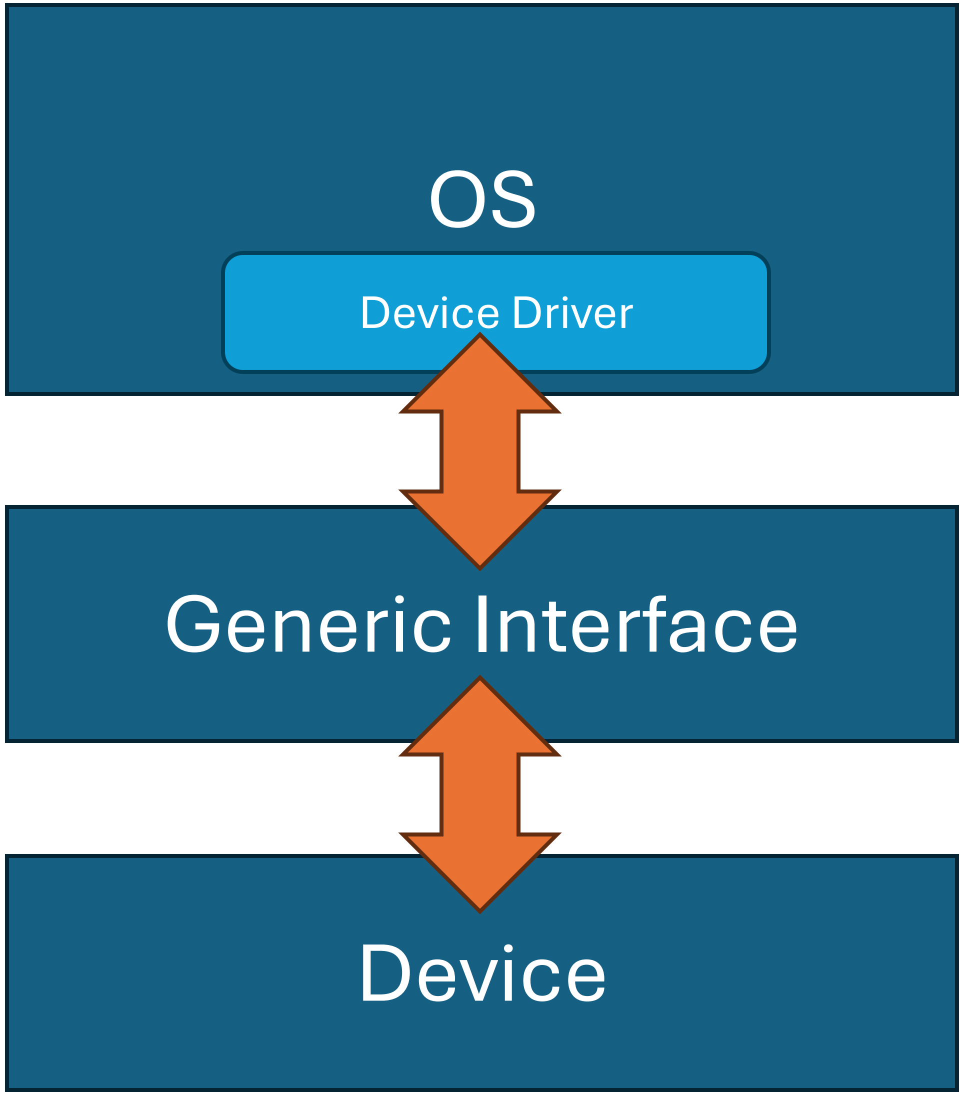
Device Driver

OS

- Kernel stuff

Device

- Works a particular way



Device Driver

Device Drivers

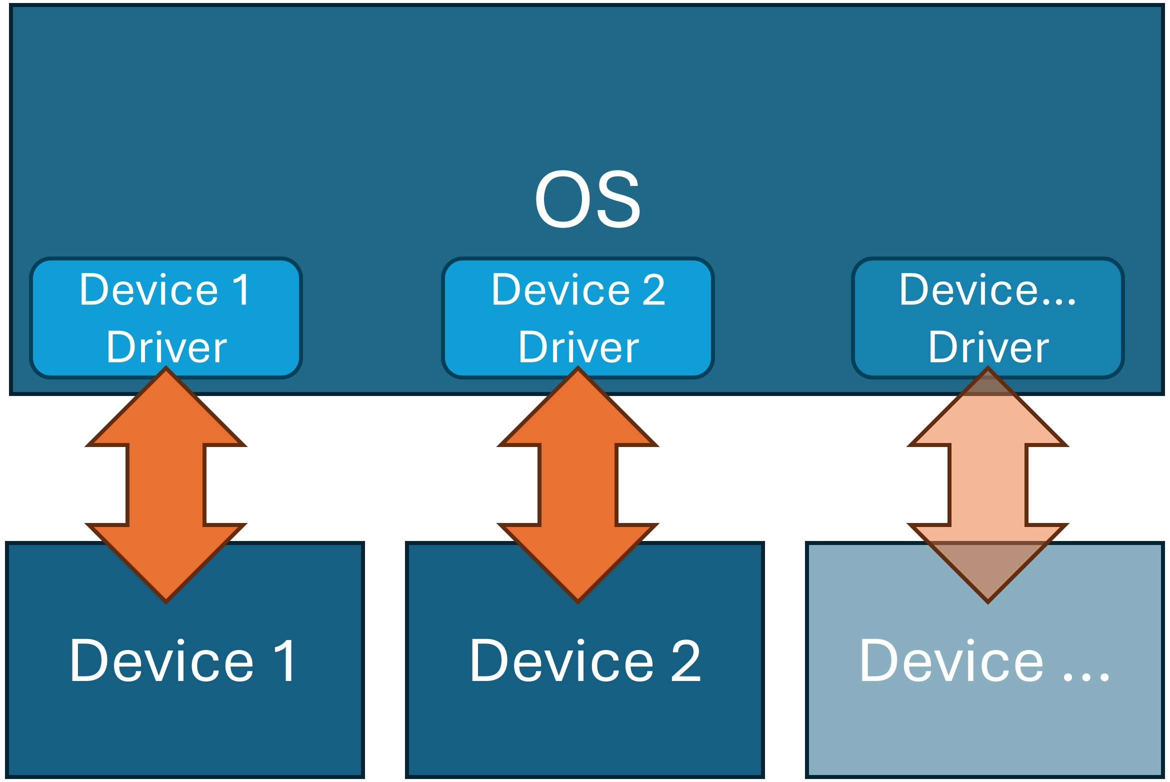
- 70% of linux code is **drivers**

Mac

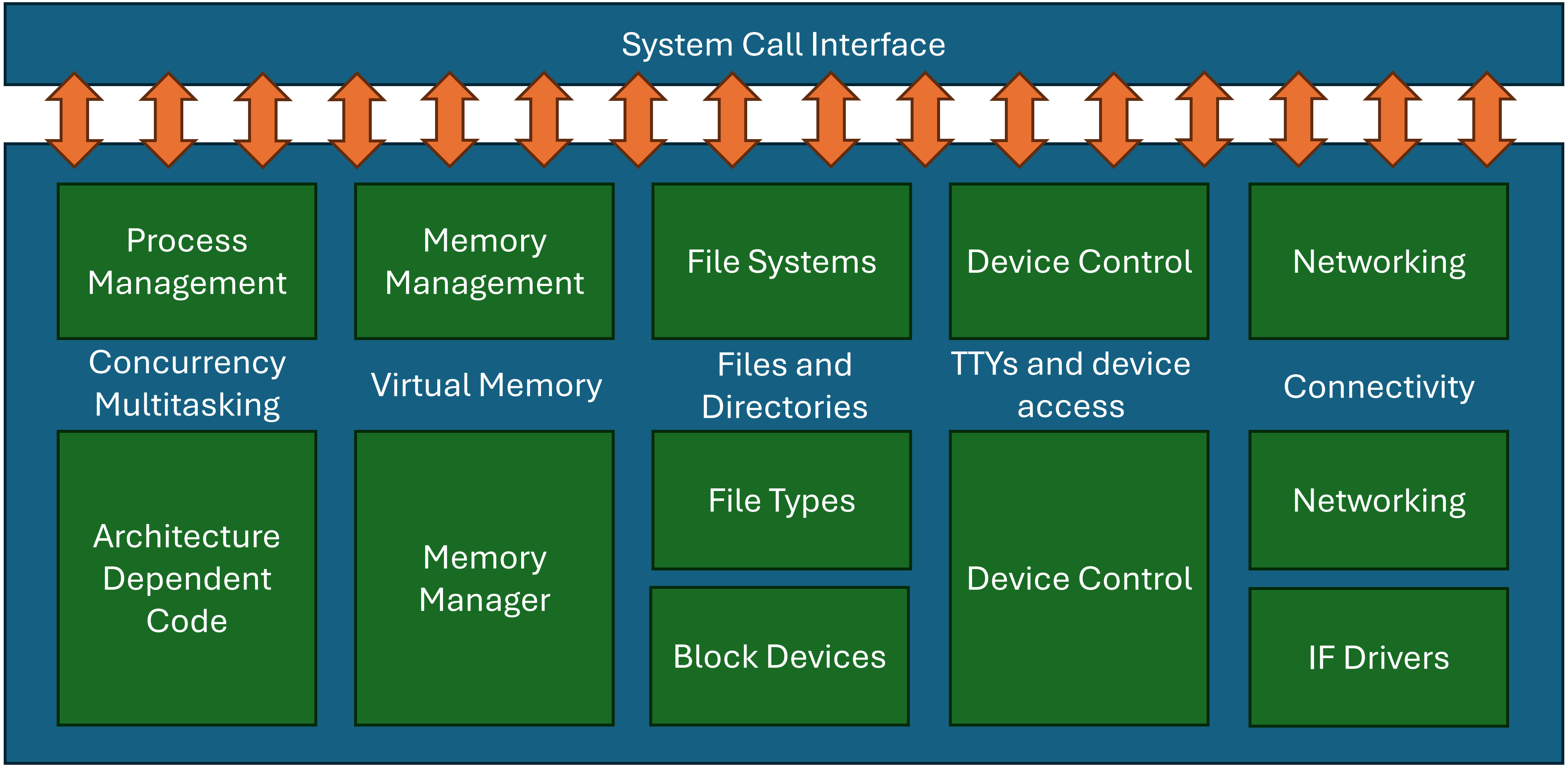
- Integrated HW (i.e., no issues)
- 3rd party driver instructions

Windows

- Historically separate driver CD's



Kernel Device Structure



Device Drivers

What is it?

- Device-specific code in the kernal that interacts directly with the device hardware
  - (Translation of the generic interface)
- Allows a generic interface

Device Drivers

Top Half

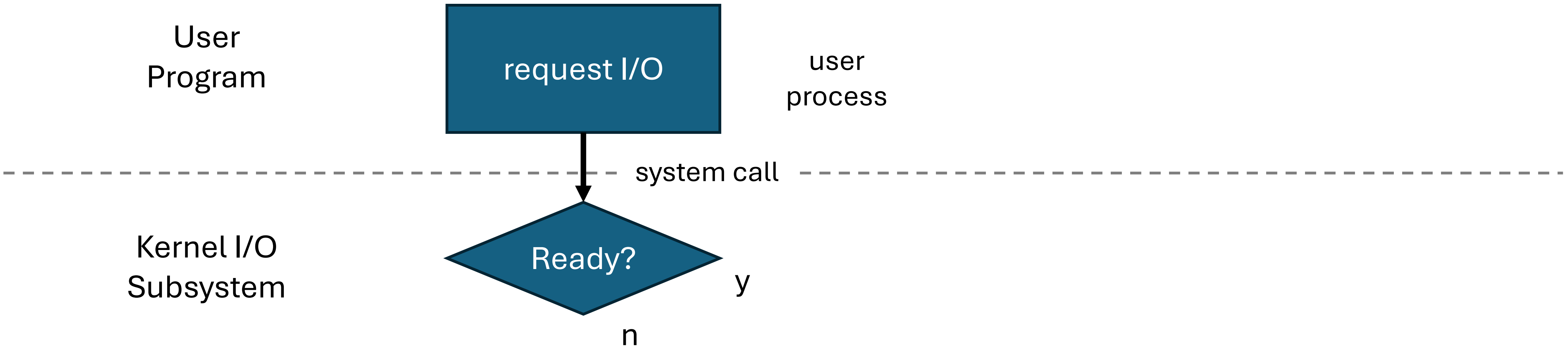
- kernel’s interface to the **device driver** (standardised interface)
  - Linux “everything is a file”
- read() / write()
- open() / close()
- ioctl() system call (Linux)
  - Special device-specific configuration supported by the

Bottom Half

- Interrupt service routines
- DMA operations

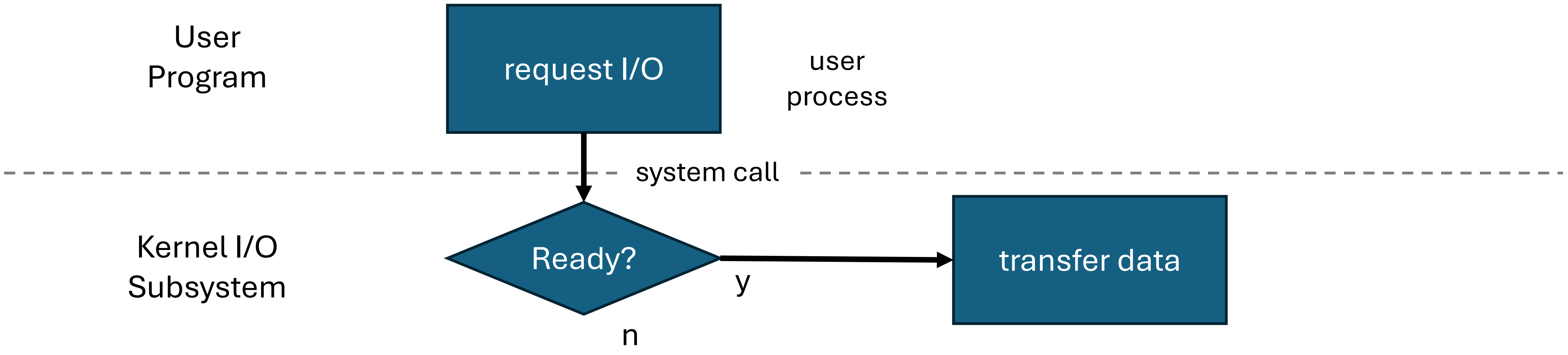
Lifecycle of an I/O request

We request some I/O use in our program



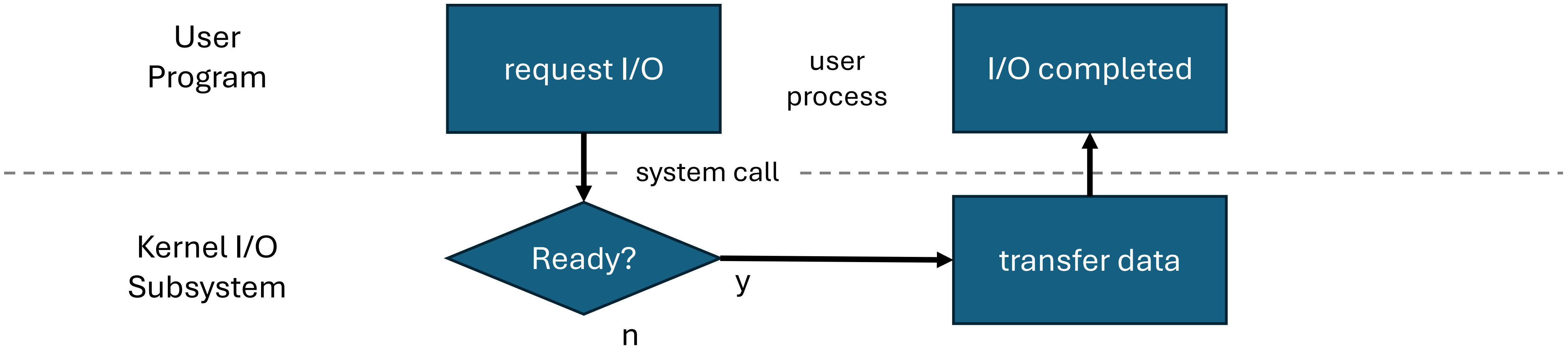
Lifecycle of an I/O request

We transfer the data, and return completion (or **error**)



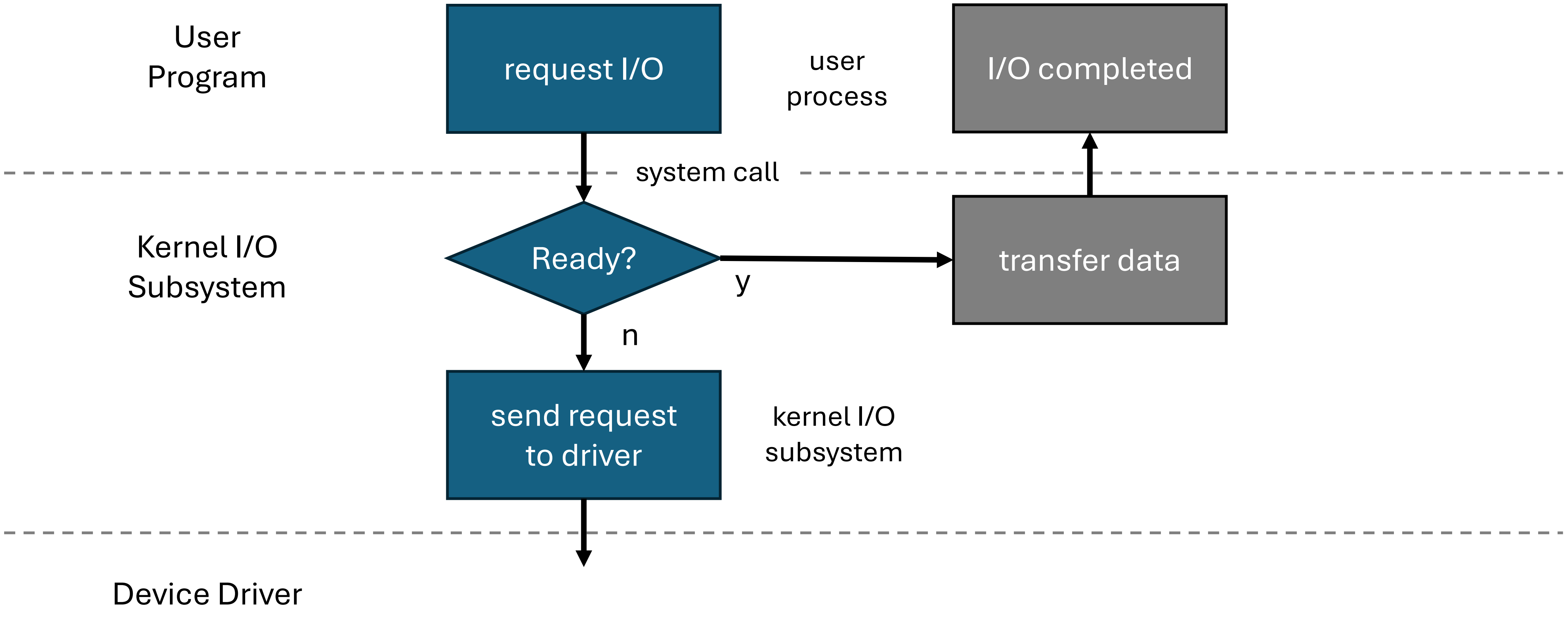
Lifecycle of an I/O request

We're done, either we have received **input** or pushed **output**



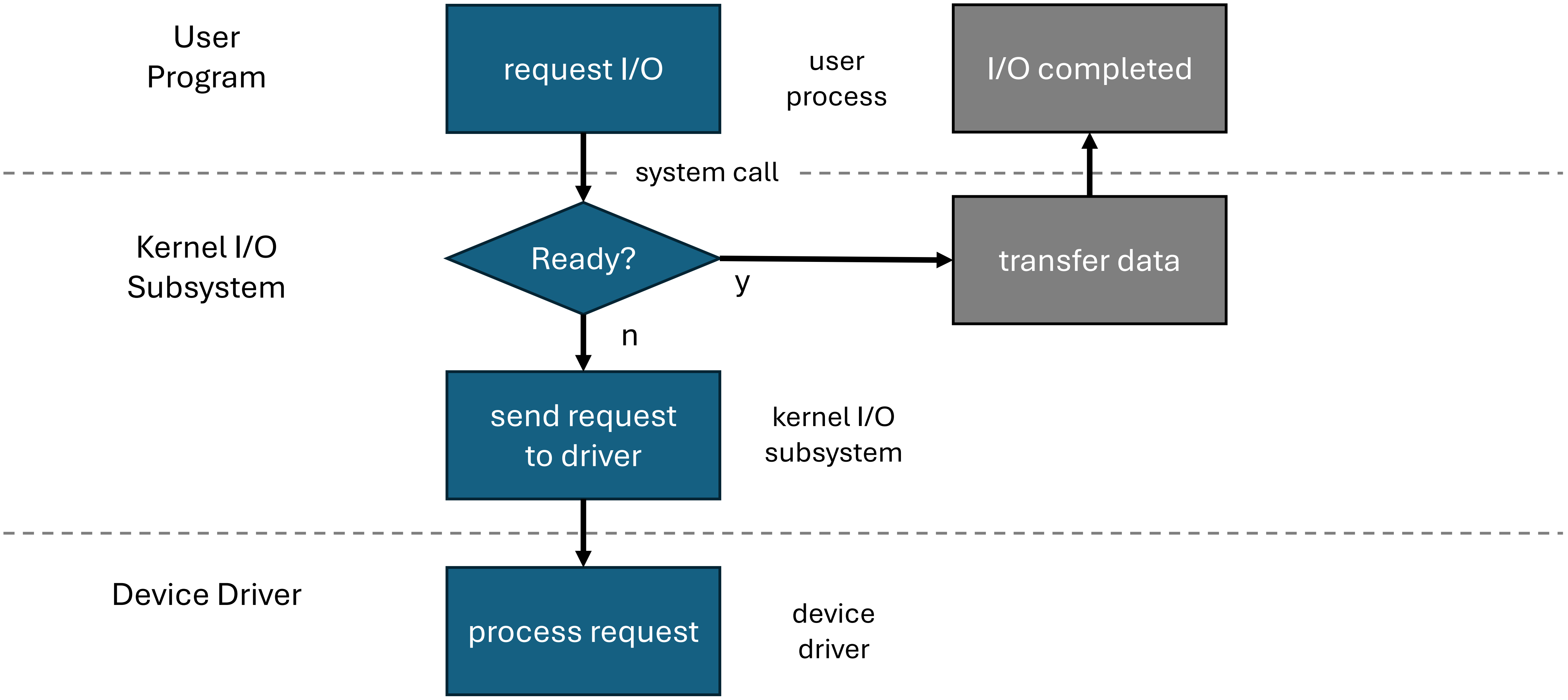
Lifecycle of an I/O request

Now we need to talk to the device driver
Block the process



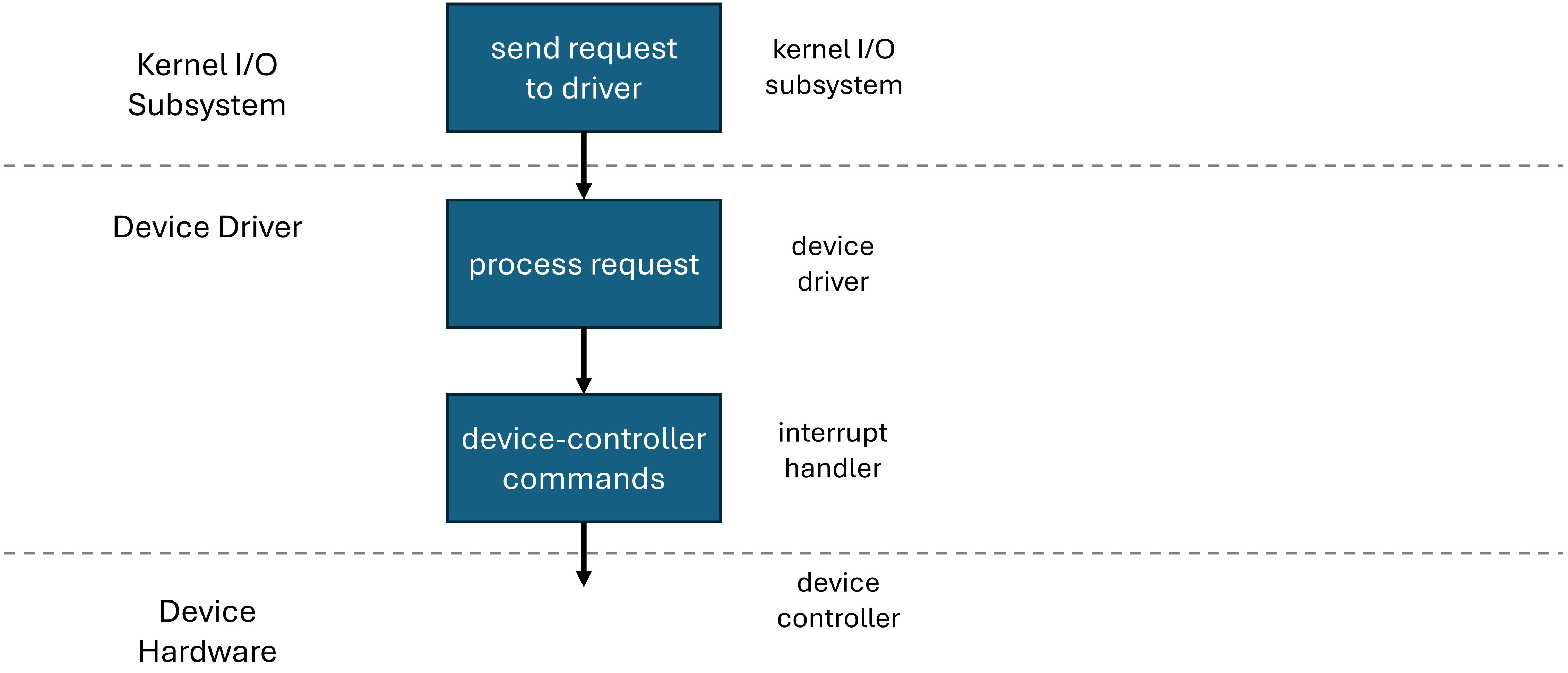
Lifecycle of an I/O request

process the request
issue commands
configure controller to block



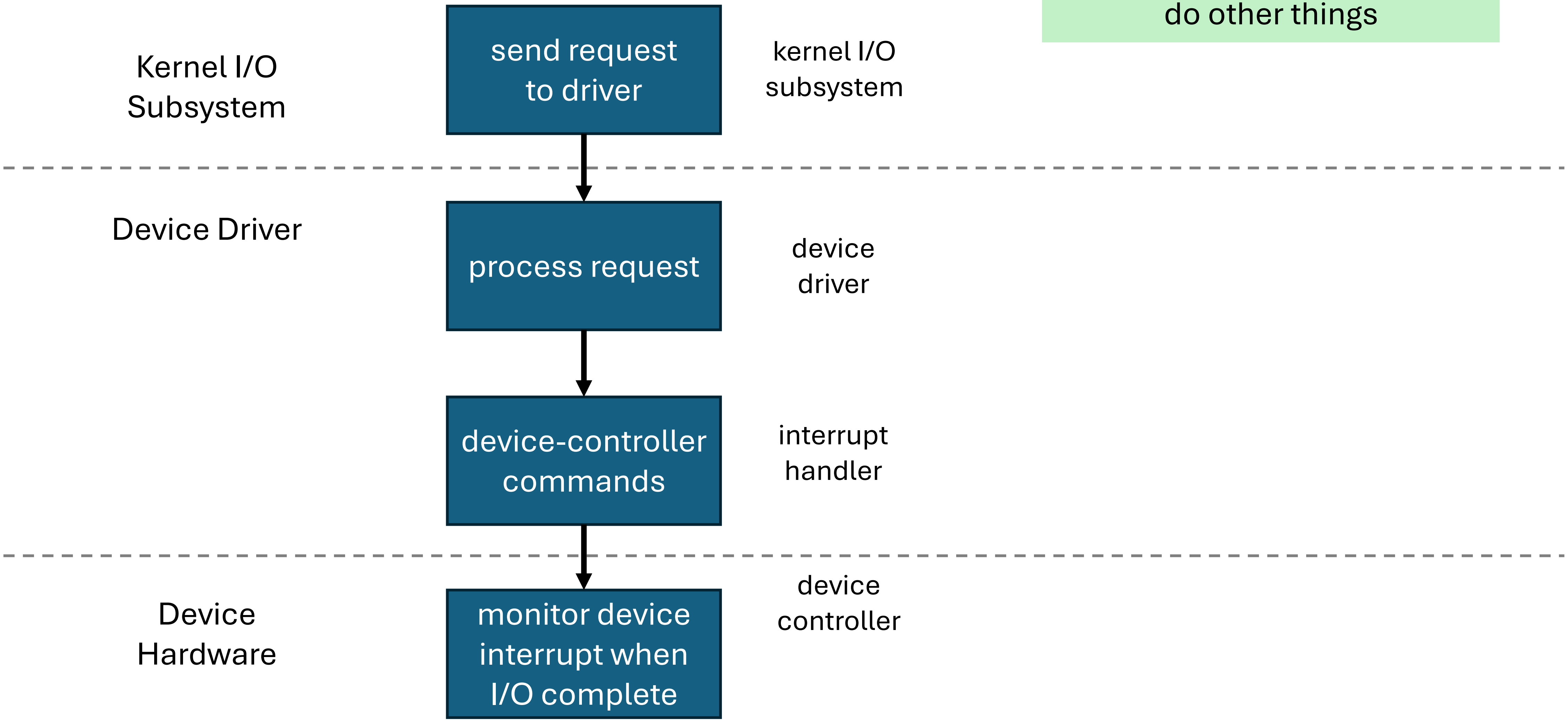
Lifecycle of an I/O request

time to send the commands to the physical hardware

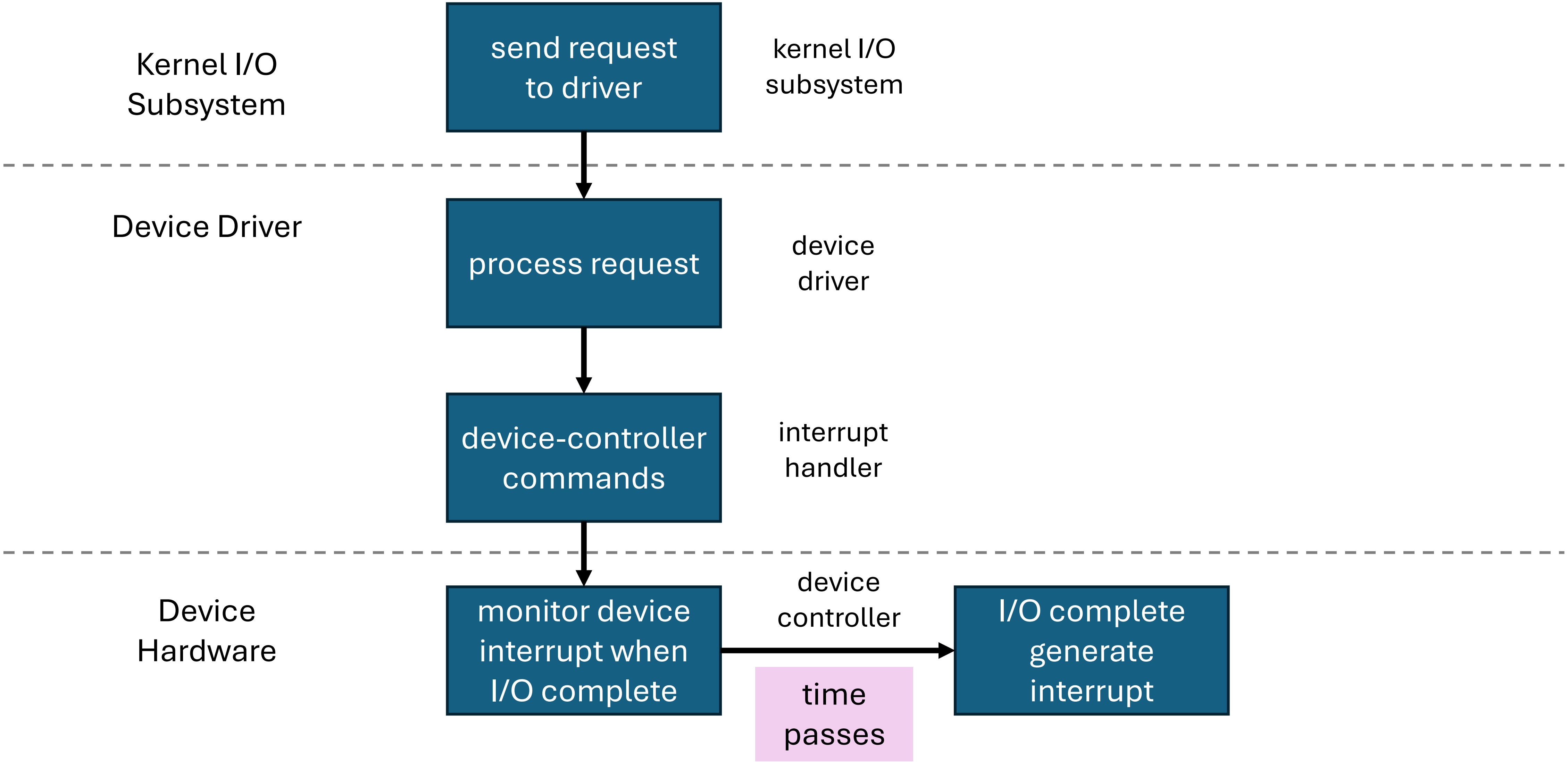


Lifecycle of an I/O request

device does the real work
deals with pending interrupts
(or polls)
CPU is now ‘free’ to
do other things

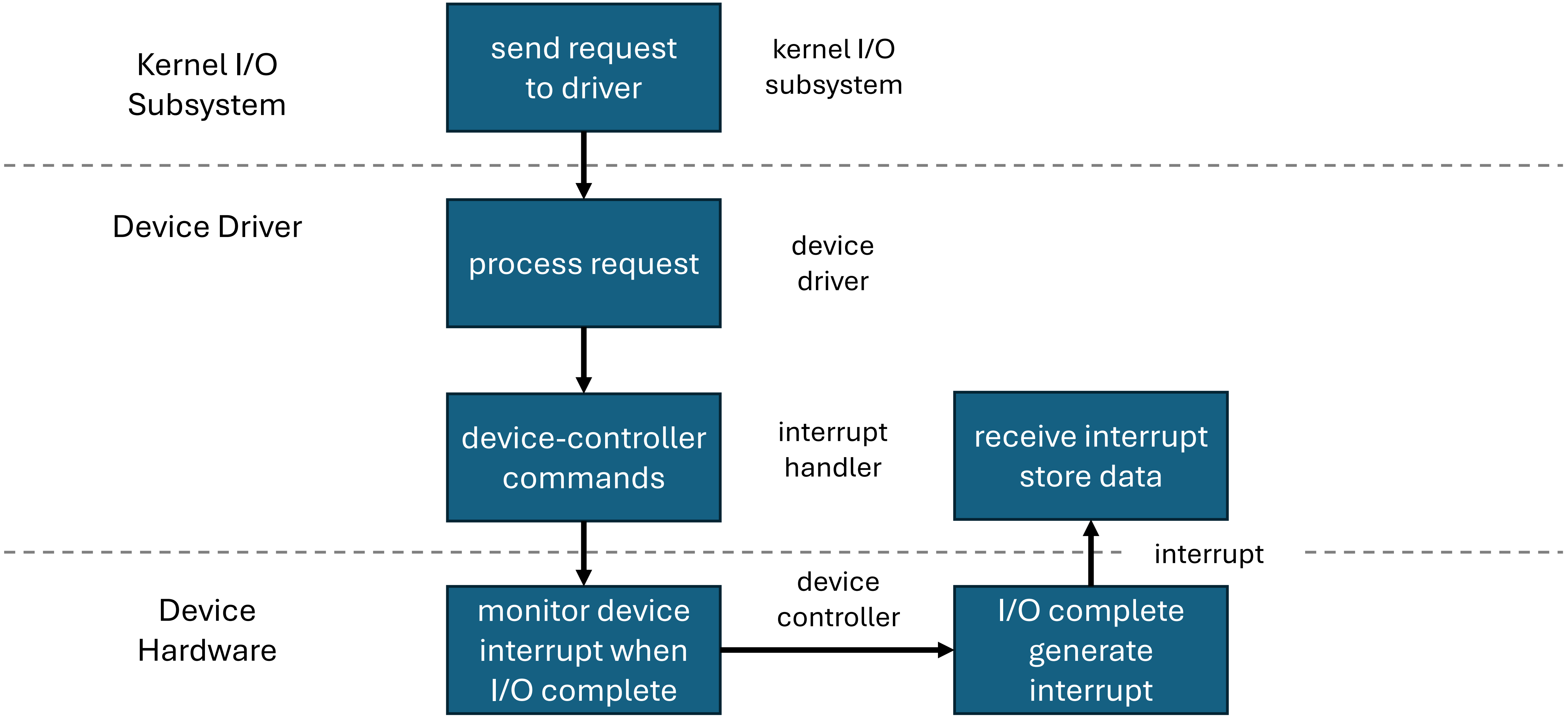


Lifecycle of an I/O request



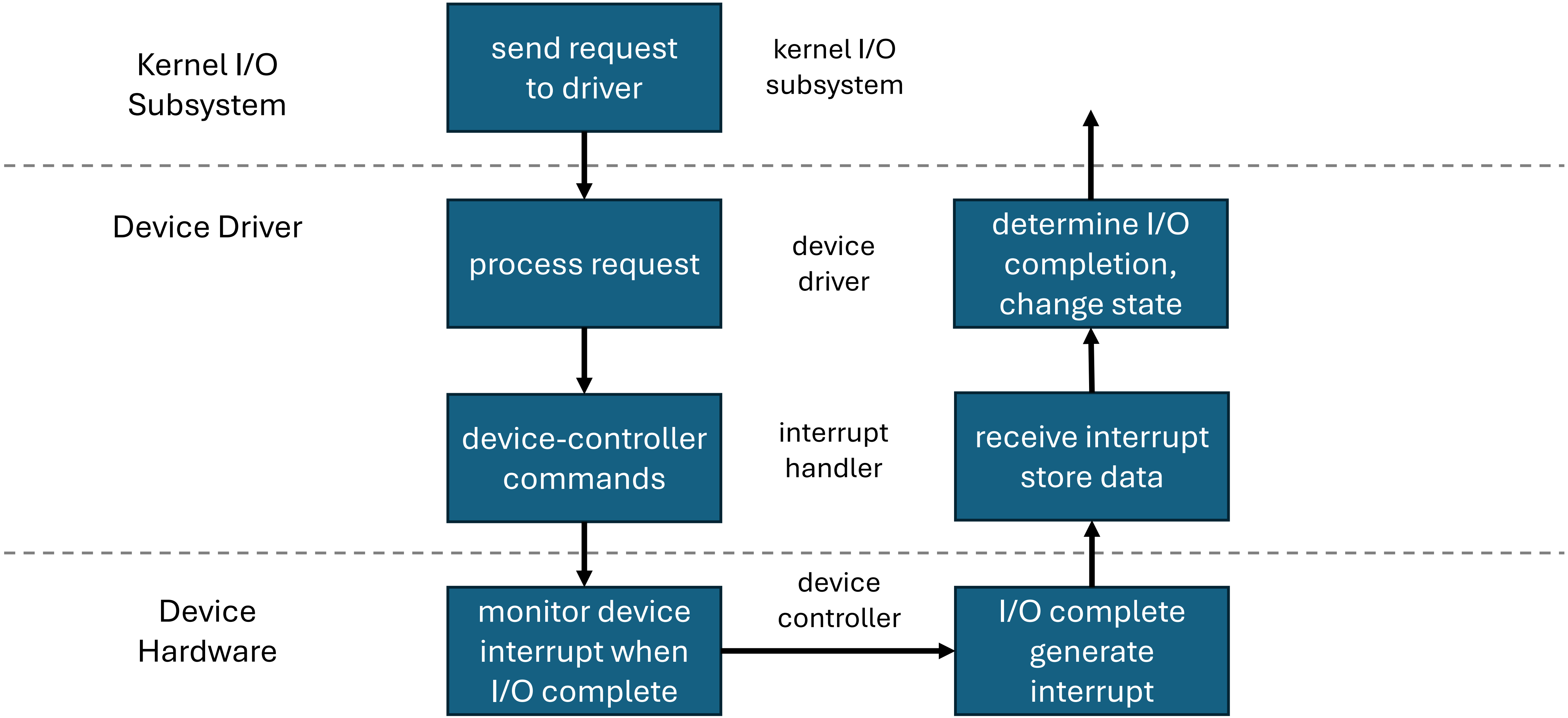
Lifecycle of an I/O request

register that the interrupt
arrived
queue?
unblock device

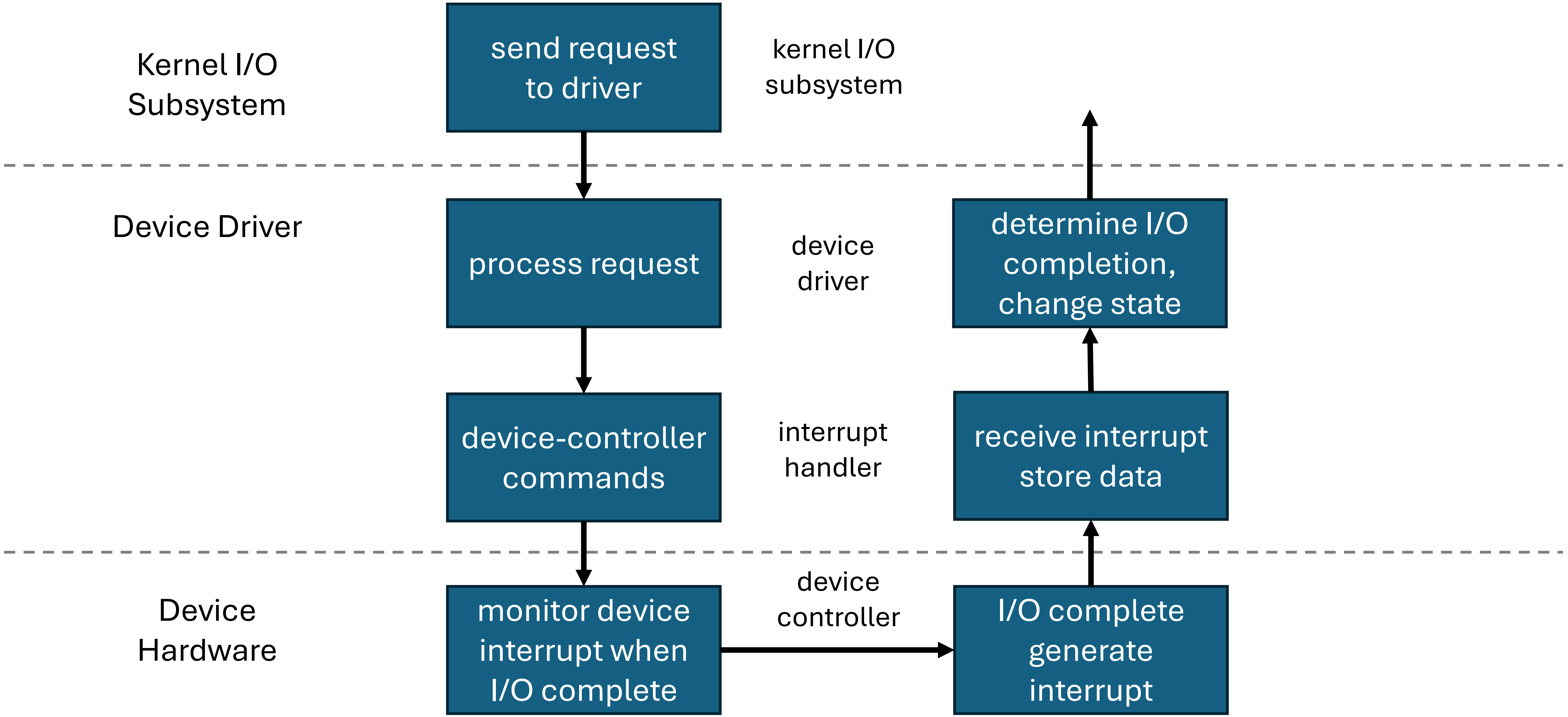


Lifecycle of an I/O request

do work (copying etc)
inform kernel that the actual
work has been done

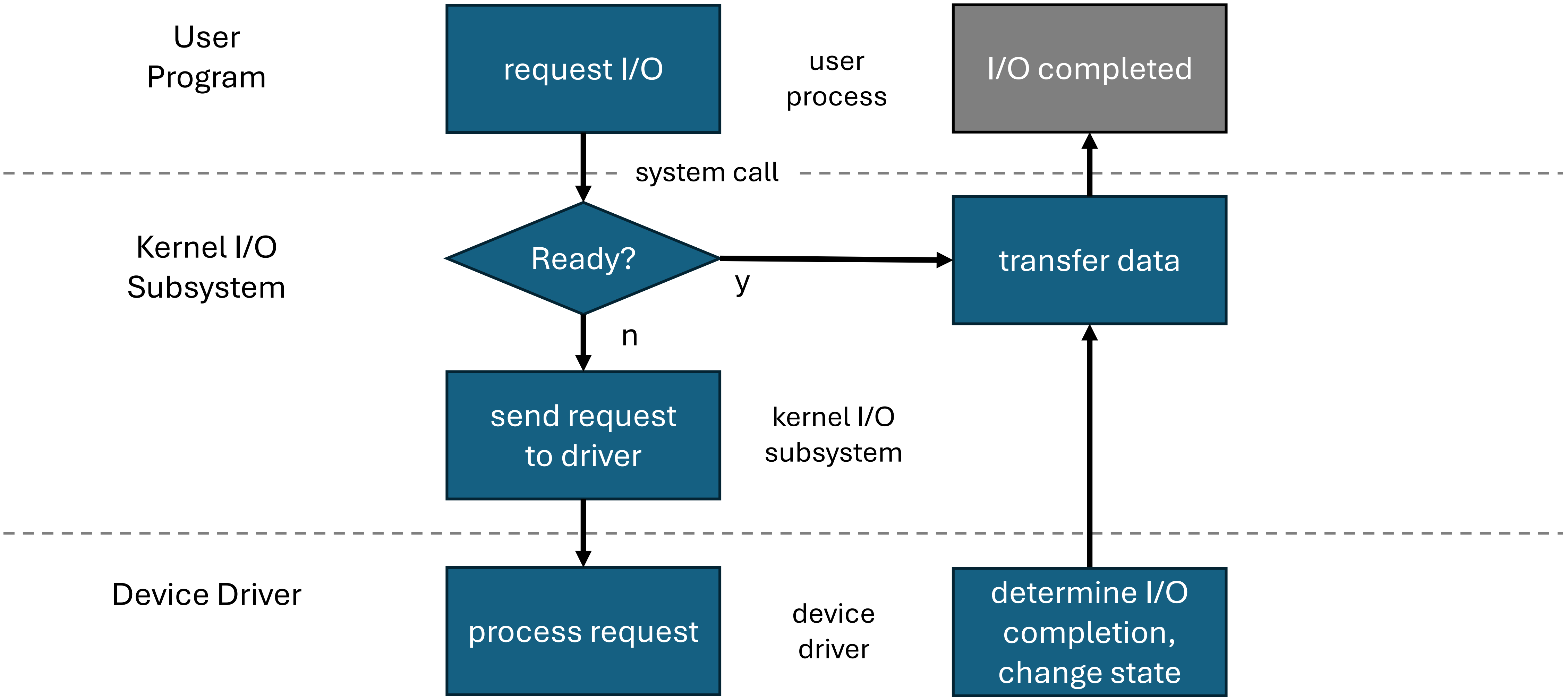


Lifecycle of an I/O request



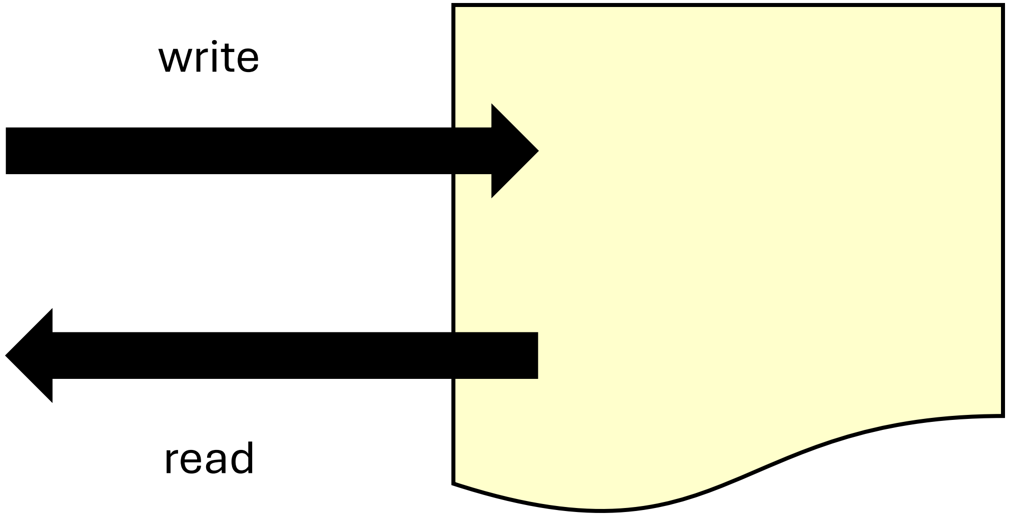
back to transfer data!

Lifecycle of an I/O request



Everything is a file

```
FILE fd = fopen("/dev/something", "rw");
for (int i = 0; i < 10; i++) {
  fprintf(fd, "Count %d\n", i);
}
close(fd);
```



Everything is a file...

Block Devices: Drives

- Access blocks of data
- Commands:
  - open/read/write/**seek**
- Raw I/O or file-system access
- Memory mapped file access possible

Everything is a file...

```
emper@Dooly:/dev$ ls -l
total 0
crw-r--r-- 1 root root     10, 235 Sep  1 16:55 autofs
drwxr-xr-x 2 root root         580 Sep  1 16:55 block
drwxr-xr-x 2 root root         100 Sep  1 16:55 bsg
crw-rw---- 1 root disk     10, 234 Sep  1 16:55 btrfs-control
drwxr-xr-x 3 root root          60 Sep  1 16:55 bus
drwxr-xr-x 2 root root        2780 Sep  1 16:55 char
crw------- 1 root root      5,   1 Sep  1 16:55 console
lrwxrwxrwx 1 root root          11 Sep  1 16:55 core -> /proc/kcore
crw------- 1 root root     10, 125 Sep  1 16:55 cpu_dma_latency
crw------- 1 root root     10, 203 Sep  1 16:55 cuse
drwxr-xr-x 6 root root         120 Sep  1 16:55 disk
drwxr-xr-x 3 root root         100 Sep  1 16:55 dri
crw-rw-rw- 1 root root     10, 127 Sep  1 16:55 dxg
lrwxrwxrwx 1 root root          13 Sep  1 16:55 fd -> /proc/self/fd
crw-rw-rw- 1 root root      1,   7 Sep  1 16:55 full
crw-rw-rw- 1 root root     10, 229 Sep  1 16:55 fuse
drwxr-xr-x 2 root root           0 Sep  1 16:55 hugepages
crw------- 1 root root    229,   0 Sep  1 16:55 hvc0
crw--w---- 1 root tty     229,   1 Sep  1 16:55 hvc1
crw------- 1 root root    229,   2 Sep  1 16:55 hvc2
crw------- 1 root root    229,   3 Sep  1 16:55 hvc3
crw------- 1 root root    229,   4 Sep  1 16:55 hvc4
crw------- 1 root root    229,   5 Sep  1 16:55 hvc5
crw------- 1 root root    229,   6 Sep  1 16:55 hvc6
crw------- 1 root root    229,   7 Sep  1 16:55 hvc7
```

d:	directory
l:	link
c:	character device file
b:	block device file
p:	named pipe
s:	socket
tty:	terminal

Everything is a file

Character Devices: keyboards, mice, serial ports

- Single characters at a time
- Commands:
  - get(), put()
- Libraries layered on top allow line editing

Everything is a file

Network Devices: Ethernet, Wireless, Bluetooth

- Different enough to have their own interface
- Use sockets()   (see CAN)
- Functions
  - select()
- Usage: pipes, FIFOs, streams, queues, mailboxes

Timing Paradigms

Blocking Interface: “Wait”

- When request data, ‘sleep’ process until data ready
- When write data, ‘sleep’ until device is ready for data

Non-blocking Interface: “Don’t wait”

- Returns quickly from read or write request with count of bytes transferred
- Read may return nothing, write may return nothing

Asynchronous Interface: “Tell me later”

- When request data, take pointer to user’s buffer, return immediately; later kernel fills buffer and notifies user
- When send data, take pointer to user’s buffer, return immediately; later kernel takes data and notifies user

I/O Device Types

Summary

- Different I/O
  - Speeds (0.1 bytes/s => Gbytes/s)
  - Access patterns?
  - Access timing? (blocking, non-blocking, asynchronous)
  - Notification mechanisms (interrupts vs polling)
  - Types (block, character, network)

Hard Disks

Huzzahs

Tape Drives

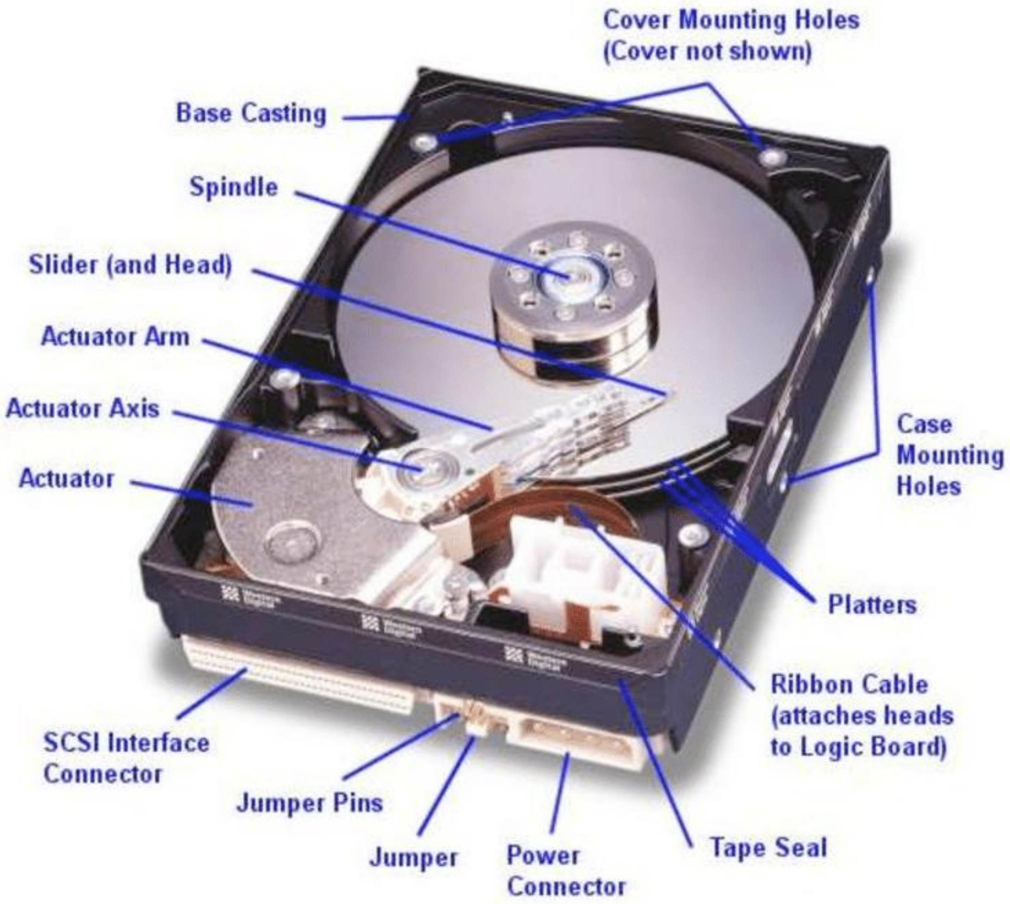
Used for storage

- Need to wind tape to find correct section of ‘information’ to load into memory.
- Incredibly slow!
- Incredibly dense!

Hard Disk Drive

Used for storage

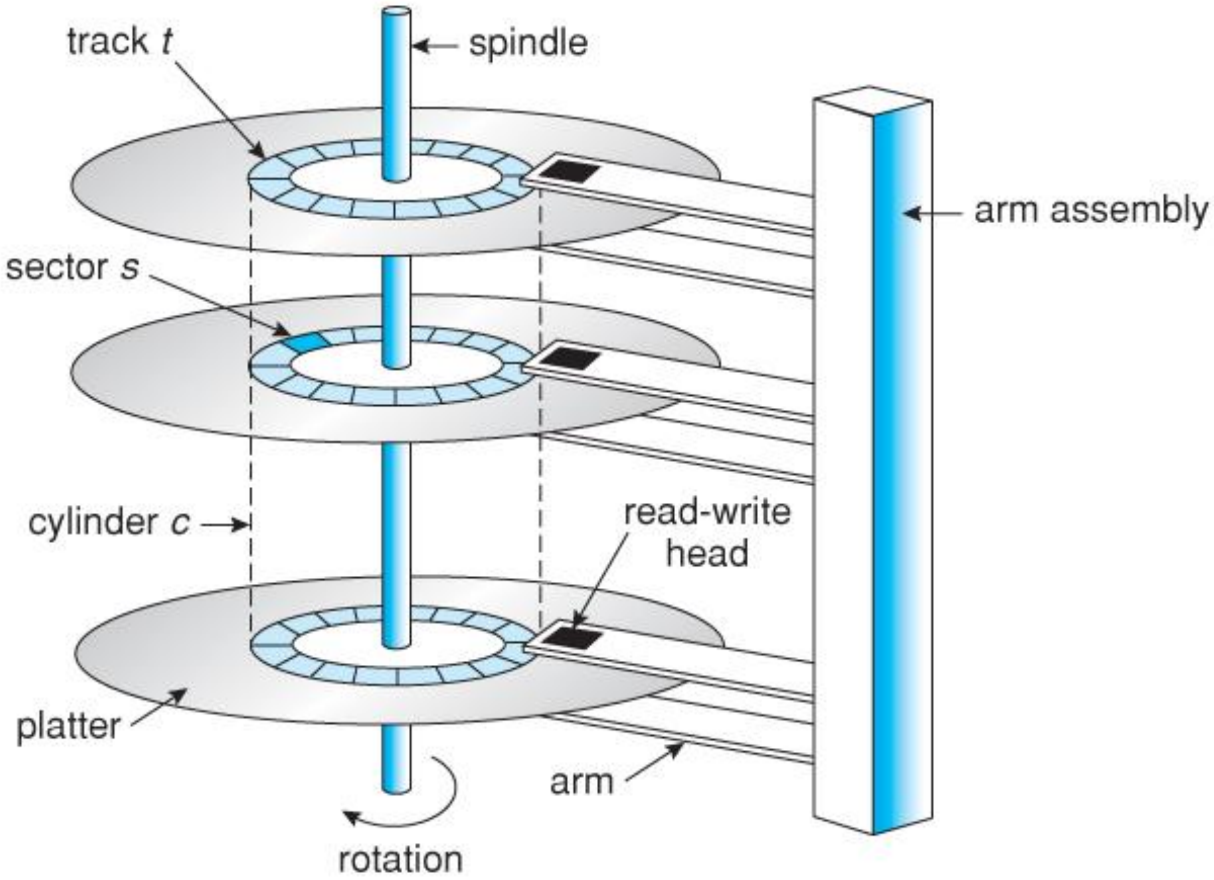
- Wheel allows quick access to different locations on the disk
- Pretty quick?
- Pretty dense?



Hard Disk Drive

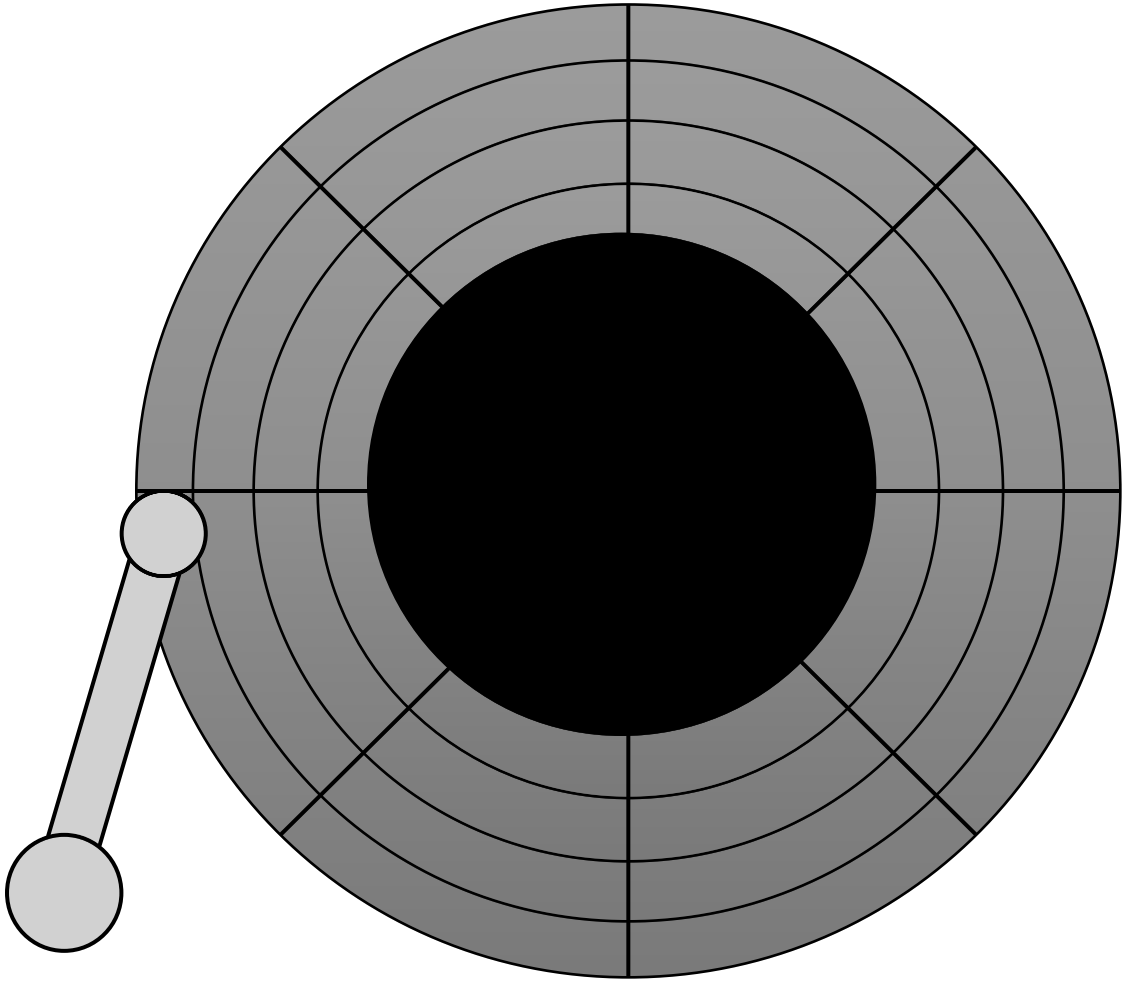
Contain:

- Tracks
- Sectors (512B or 4096B)
- Cylinders Many heads...
- Platters
- Spindle

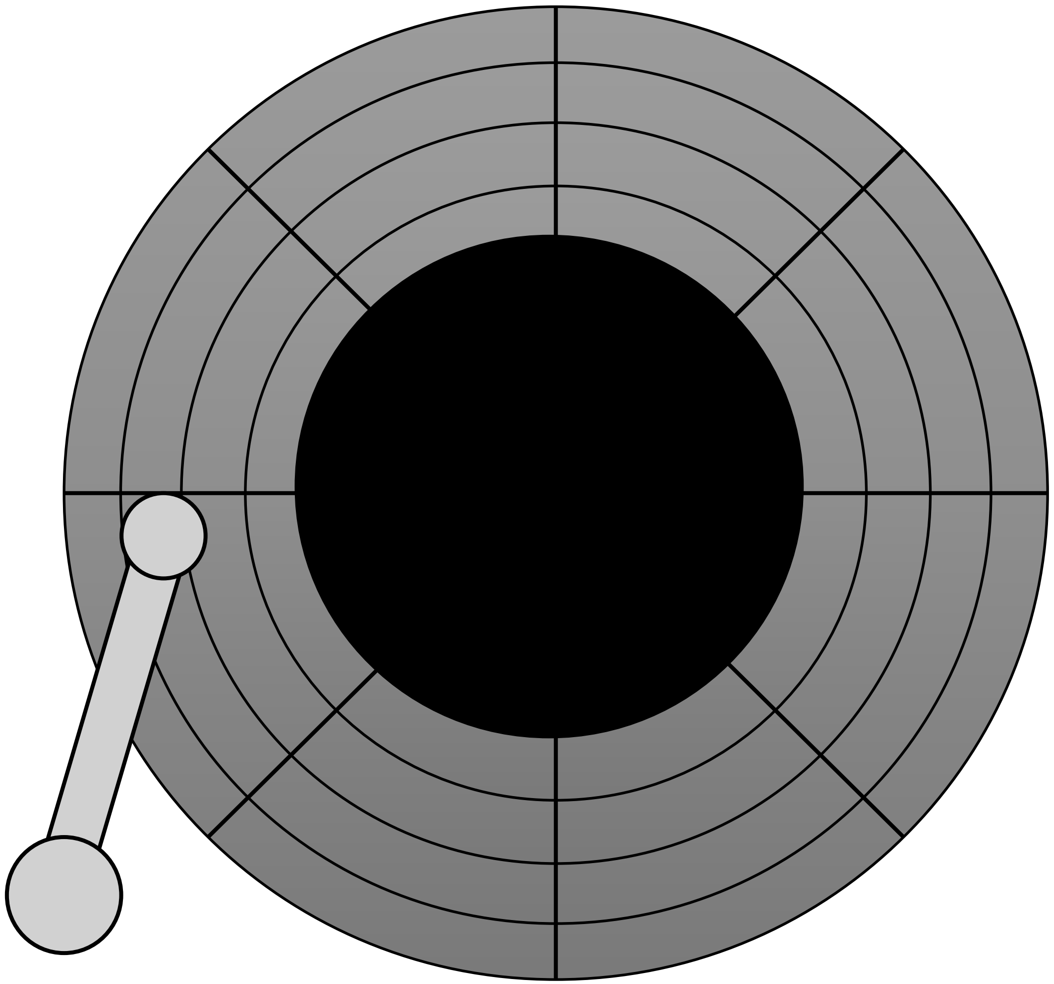


Reading/Writing

Spindle… spins



Reading/Writing



Some questions:

- How does the disk head know where it is?
  - Platters misaligned
  - Tracks not perfectly concentric
- High precision manufacturing
  - Some contain helium??

Reading/Write

Magnetic disk

- Pass a field over a region to change its state (lasts a long time).

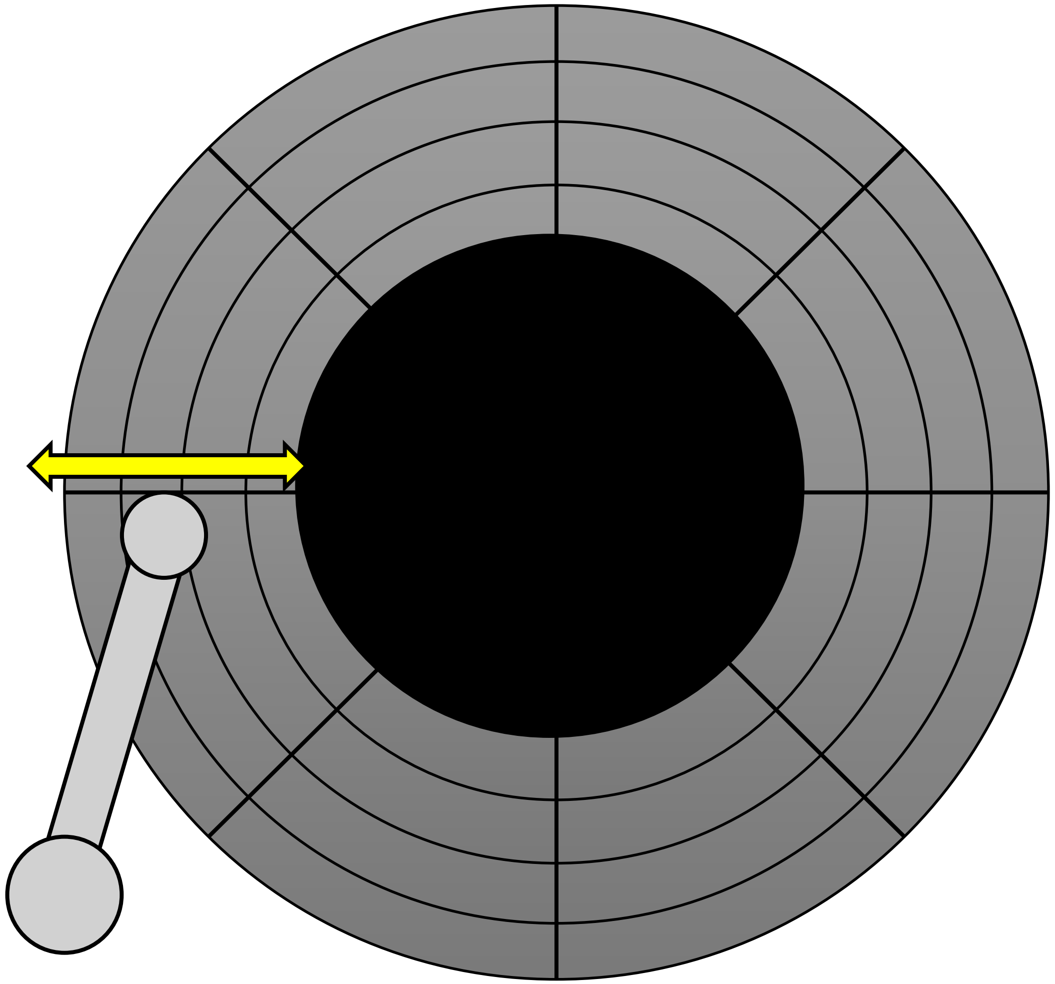
Drive Servo Systems

- Need to work out where you are (lets you choose which sector to read/write to
  - Notches (old)
  - Timing information within a disk

Seek, Rotate, Transfer

Seek Cost:

- Depends on the cylinder distance
  - Not purely linear
  - Must accelerate/coast/slow/settle
- Entire seek often takes several milliseconds (4-10ms)
- Average seek distance ~1/3 of max seek distance



Seek, Rotate, Transfer

Rotate:

- Depends on the rotations per minute (RPM)
  - 3600
  - 5400
  - 7200
  - 15000

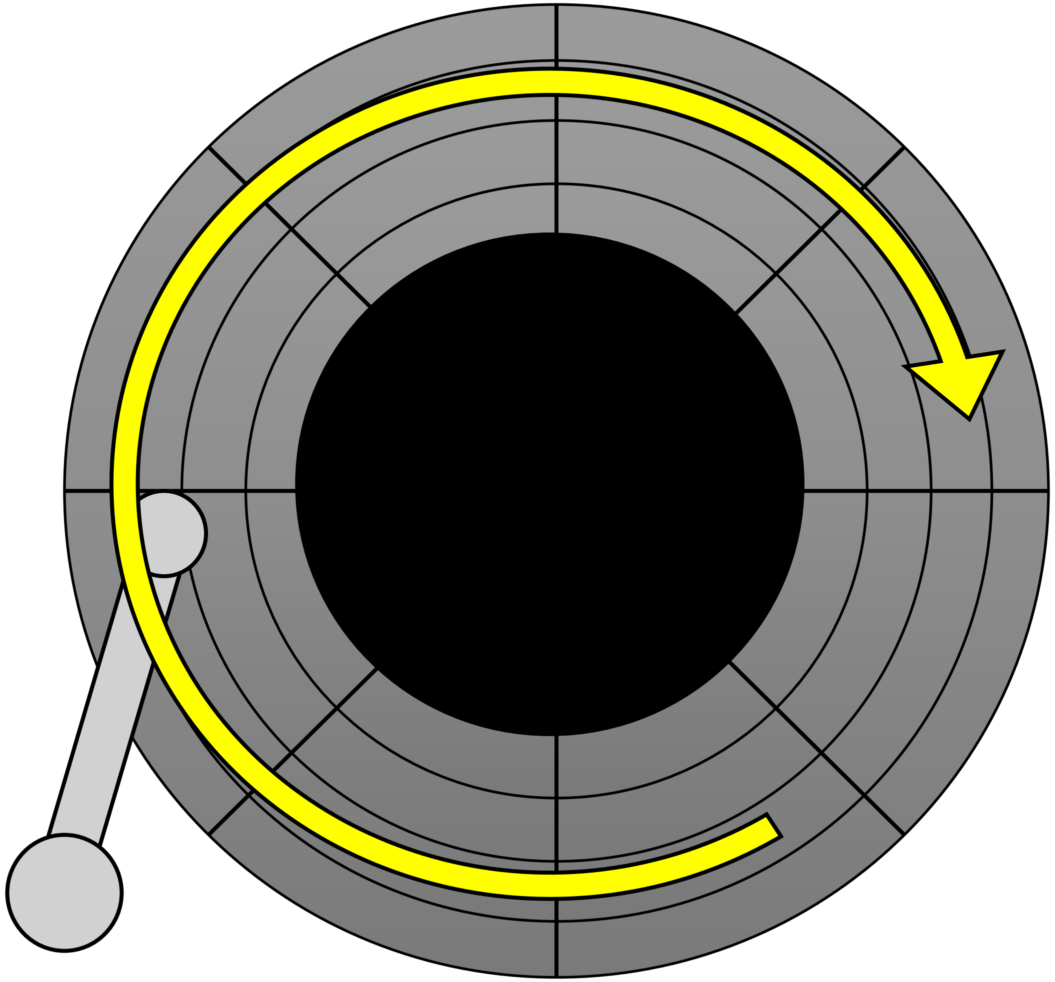
7200 RPM

$\Rightarrow$1 minute / 7200

$\Rightarrow$ 8.3 ms / rotation

Average rotation

= 8.3/2 = 4.15ms

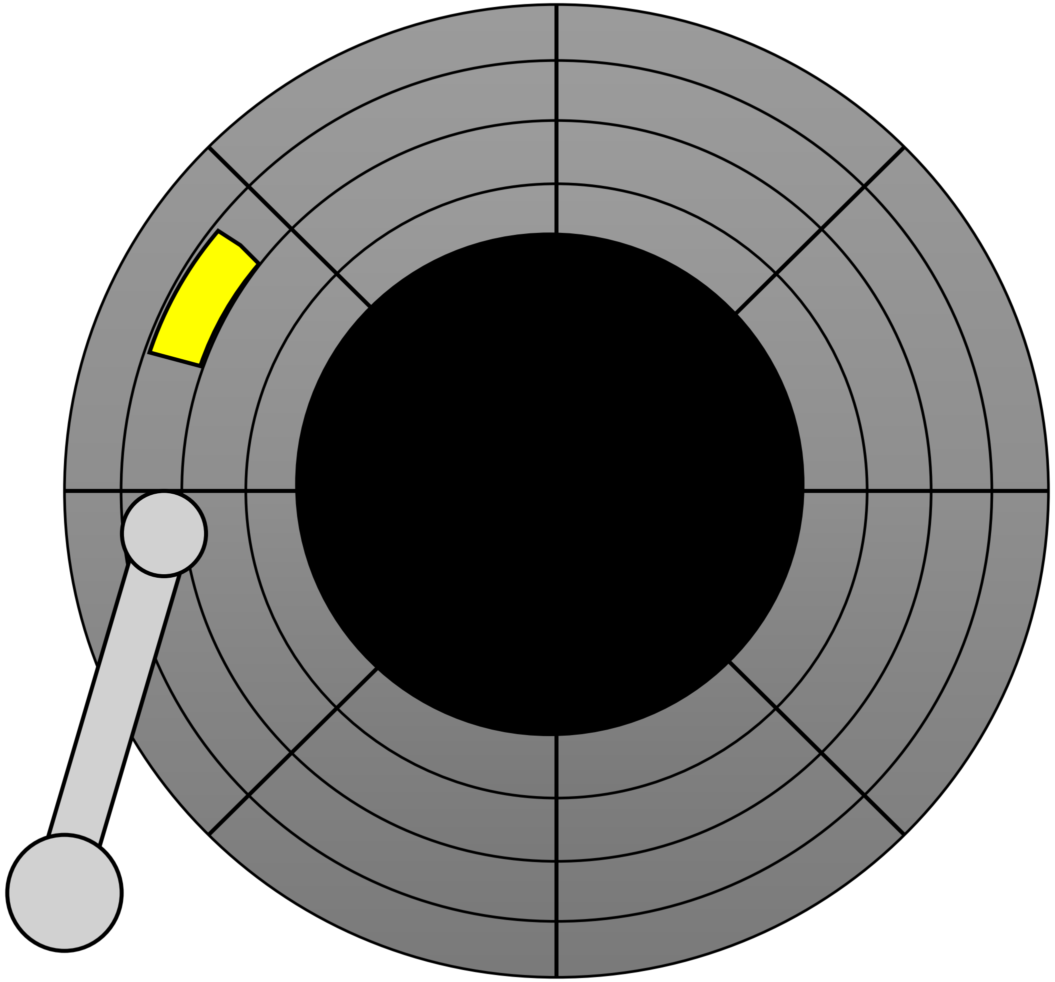


Seek, Rotate, Transfer

Transfer

- Pretty fast
  - RPM (fast = more)
  - Data density (more = more)
  - Request size (max = more)
- Typically, GB/s

4096 B * 1GB/s = 4 ms transfer



Disk Comparison

	Cheetah	Barracuda
Capacity	300 GB	1 TB
RPM	15000	7200
Av Seek (ms)	4	9
Max Transfer (MB/s)	125	105
Platters	4	4
Cache MB	16	32

What is the throughput?

Cheetah: 125 MB/s
Barracuda: 105 MB/s

Disk Comparison: Calculations

	Cheetah	Barracuda
RPM	15000	7200
Av Seek (ms)	4	9
Max Transfer (MB/s)	125	105

How long would an average **random** 16 KB read take with the Cheetah?

Seek: 4ms

Disk Comparison: Calculations

	Cheetah	Barracuda
RPM	15000	7200
Av Seek (ms)	4	9
Max Transfer (MB/s)	125	105

How long would an average **random** 16 KB read take with the Cheetah?

Seek: 4ms

$$\frac{1}{2} \times \frac{1}{15000} \times \frac{60}{1} \times \frac{1000}{1}$$

Unit Conversions

2ms

Disk Comparison: Calculations

	Cheetah	Barracuda
RPM	15000	7200
Av Seek (ms)	4	9
Max Transfer (MB/s)	125	105

How long would an average **random** 16 KB read take with the Cheetah?

$$\frac{16}{125} \times \frac{1}{1024} \times \frac{1000}{1}$$

0.125ms

Unit Conversions

Seek: 4ms
Rotation: 2ms

Disk Comparison: Calculations

	Cheetah	Barracuda
RPM	15000	7200
Av Seek (ms)	4	9
Max Transfer (MB/s)	125	105

How long would an average **random** 16 KB read take with the Cheetah?

Seek	**Rotate**	**Transfer**
4	2	0.125

= 6.125

Seek:	4ms
Rotation:	2ms
Transfer:	0.125ms

Buffering

Internal Memory

- 2-16 MB cache
- Read contents of entire track into memory during rotational delay
  - Cache leverages spatial locality
  - “Read ahead”

Write caching with volatile memory

- Immediate reporting
- Problem: You claim to have written when you haven’t (power failure = bad)

Tagged Command Queuing

- Have multiple outstanding requests (SATA… 16?)
- Disk can re-order requests

More Disk ‘Smarts’

Depends upon what the external abstraction is:

- Disks may re-order operations
- Makes reasoning about their performance very difficult
- Operating systems may not have any useful optimisations available
  - SCSI devices do not expose internals – just a linear space of blocks

Solid State Disks

Flash!

Solid State Disks (SSD)

No moving parts

- Flash memory
- Written in pages (not bytes)
- Not byte-level addressable
  - Read whole page
  - Modify
  - Write whole page
  - Can only write to erased blocks

SSD – Brief history

Battery-backed DRAM (1995)

- Battery needed to avoid data loss

NAND Multi-level Cell (2 or 3 bit cell) **flash memory** (2009)

1991

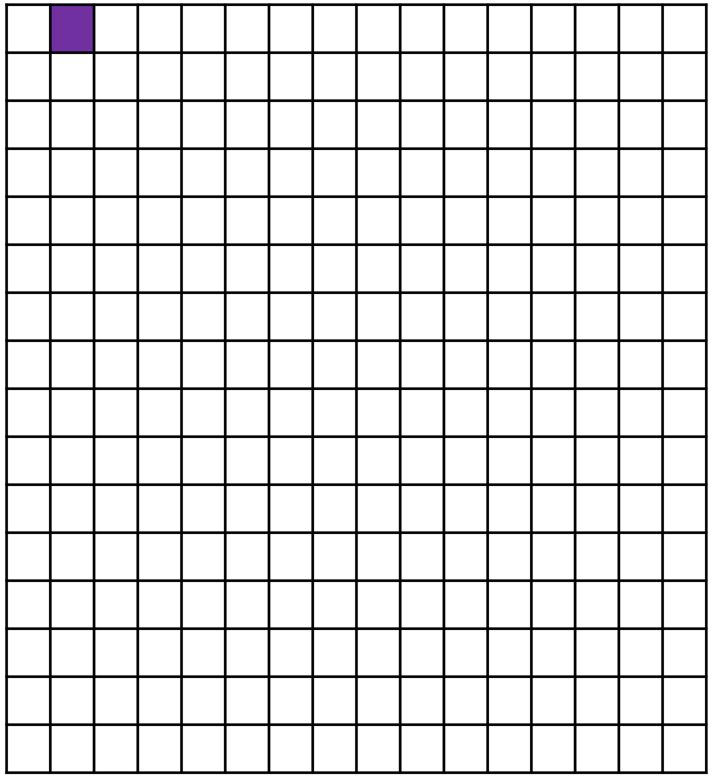
The logic: NAND Flash Memory

Memory consists of cells

- Originally 1-bit but can now be multi-bit

Memory is stored in **pages**

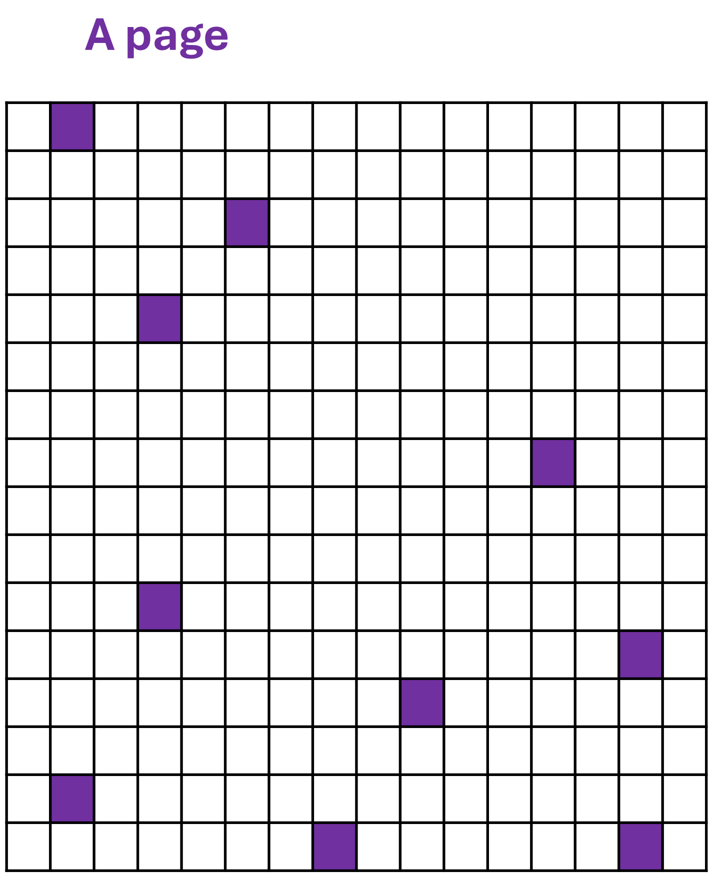
Pages are grouped into **blocks**



The logic: NAND Flash Memory

Pages are initially (empty – this usually actually means all 1's)

When you need a page you find one and write new memory into it.

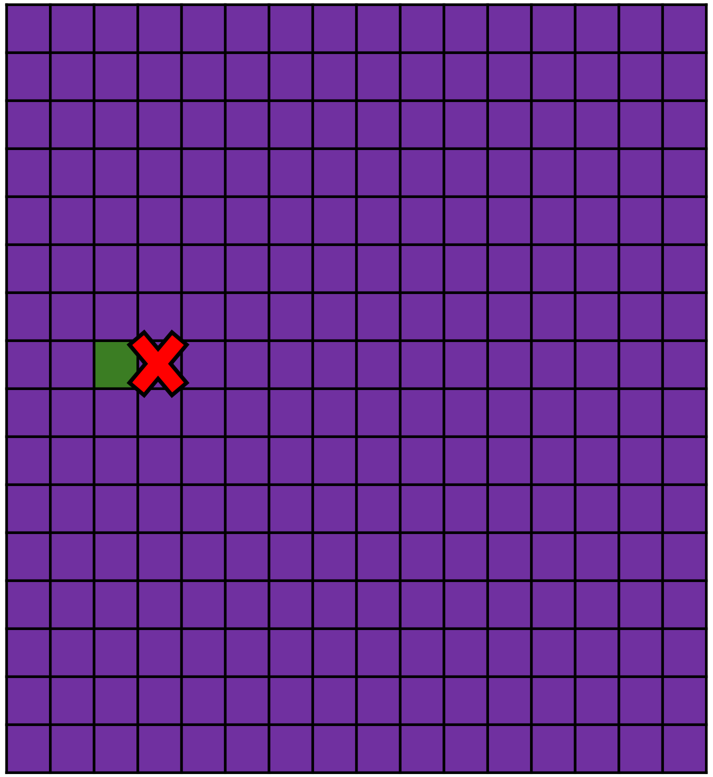


A block

The logic: NAND Flash Memory

If all the **pages** are full… you can't overwrite them

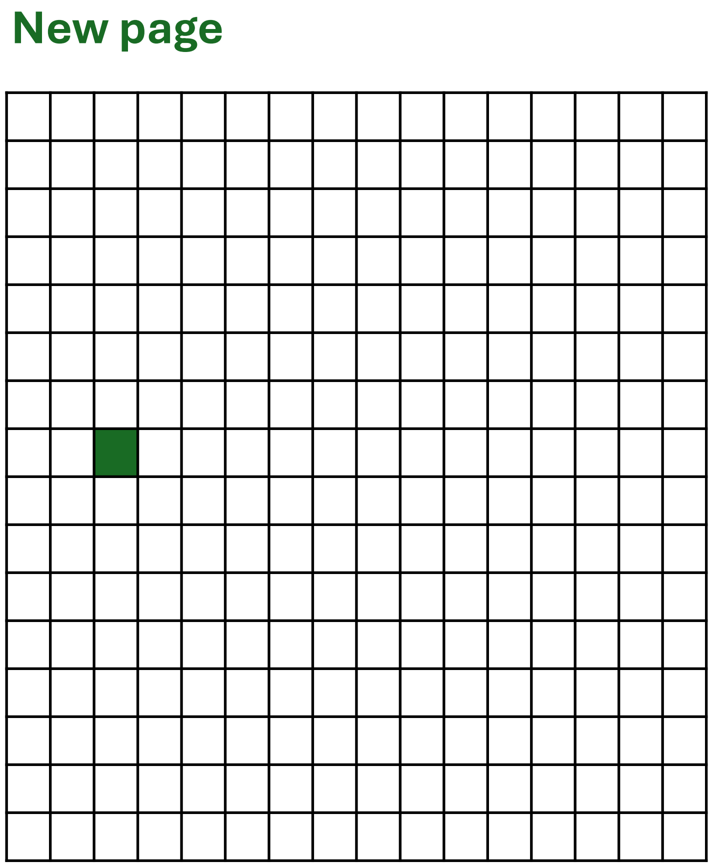
Many pages



A block

The logic: NAND Flash Memory

Instead, you would need to 'empty' an entire **block**, then write the new **page**.

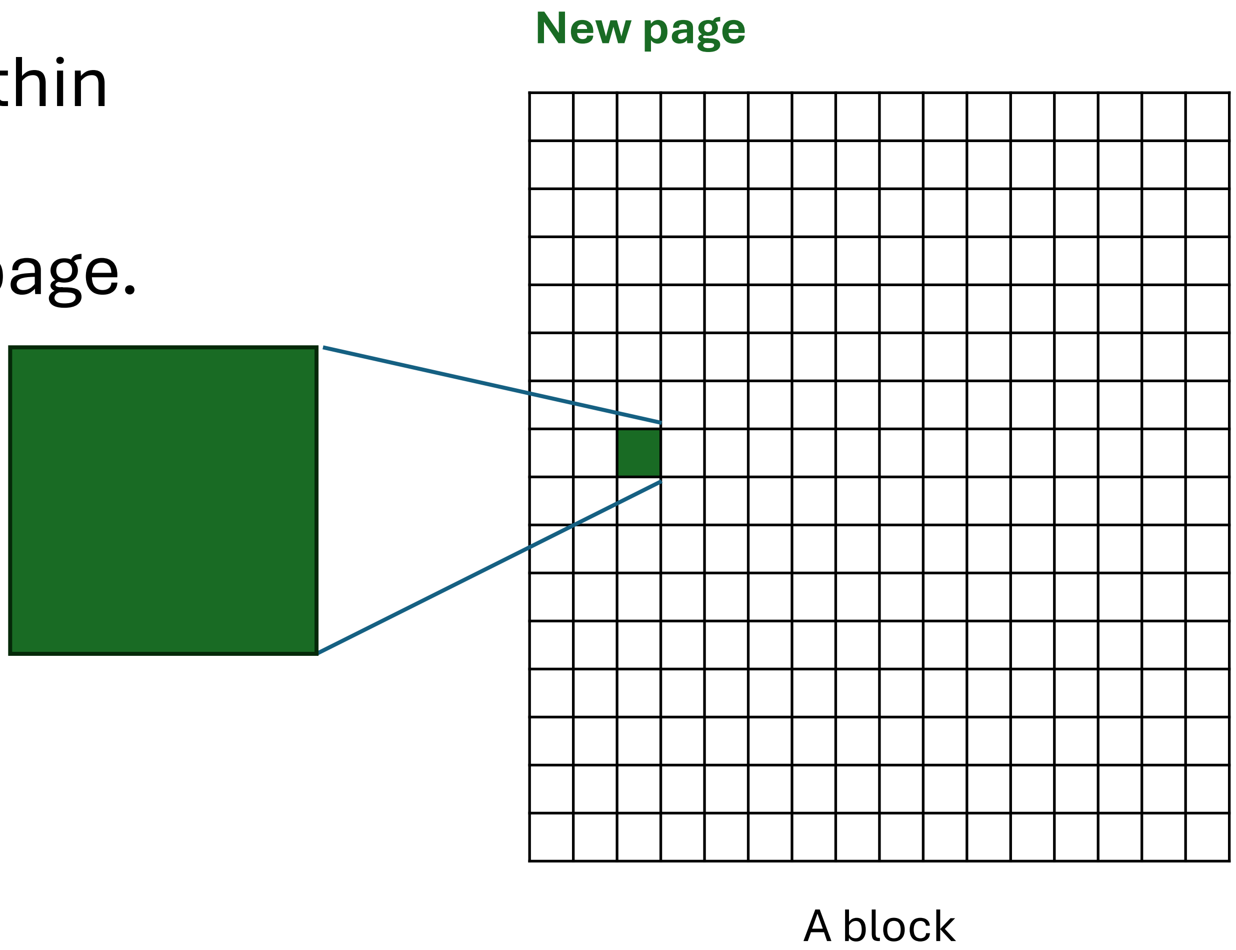


A block

The logic: NAND Flash Memory

You can’t grab bytes from within a page.

You need to take the whole page.

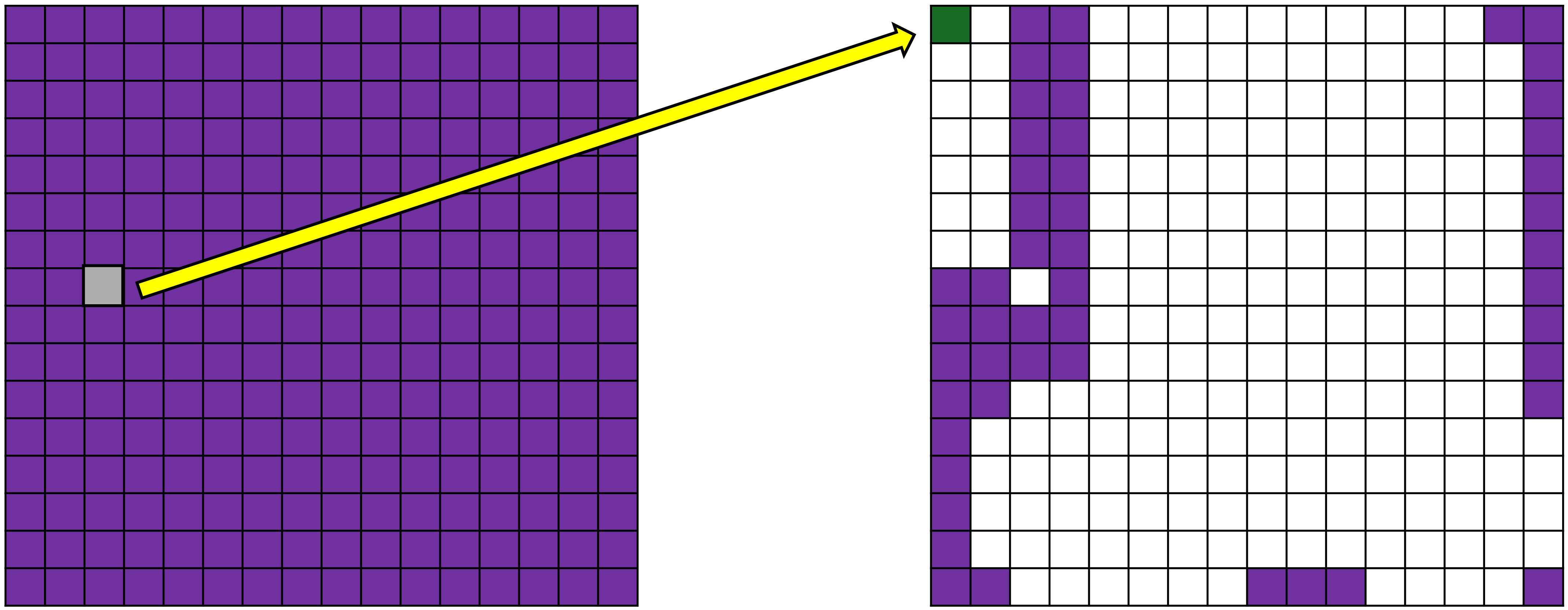


How can this possibly be good?

Where are the problems?

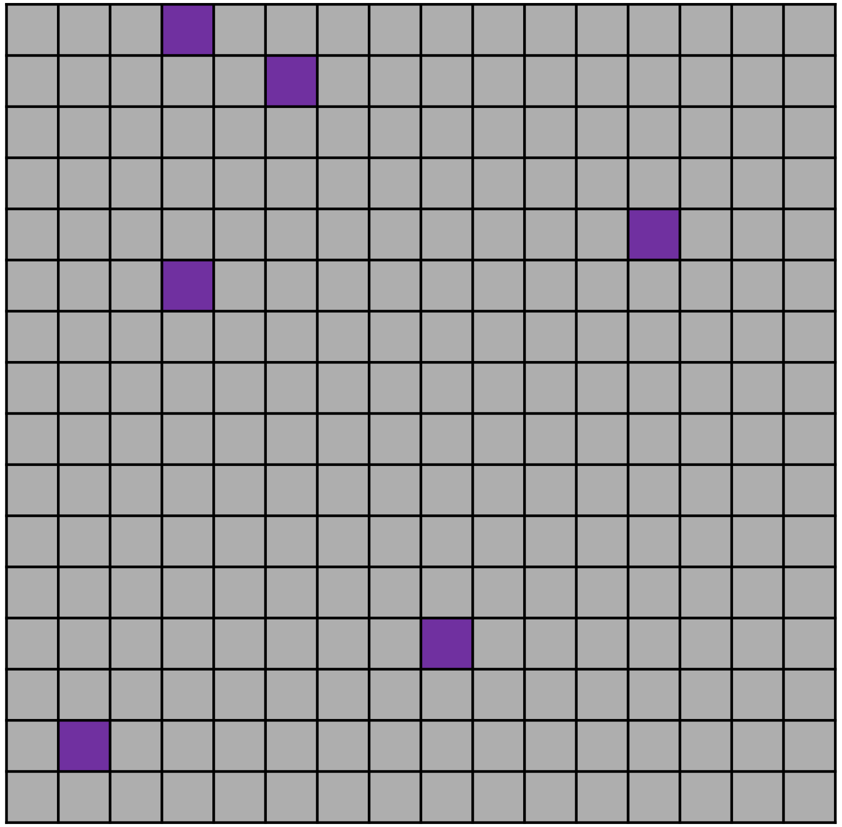
What can we do to fix these?

Copy and Paste



Old page is read, copied to a new page with modifications... then marked **invalid**

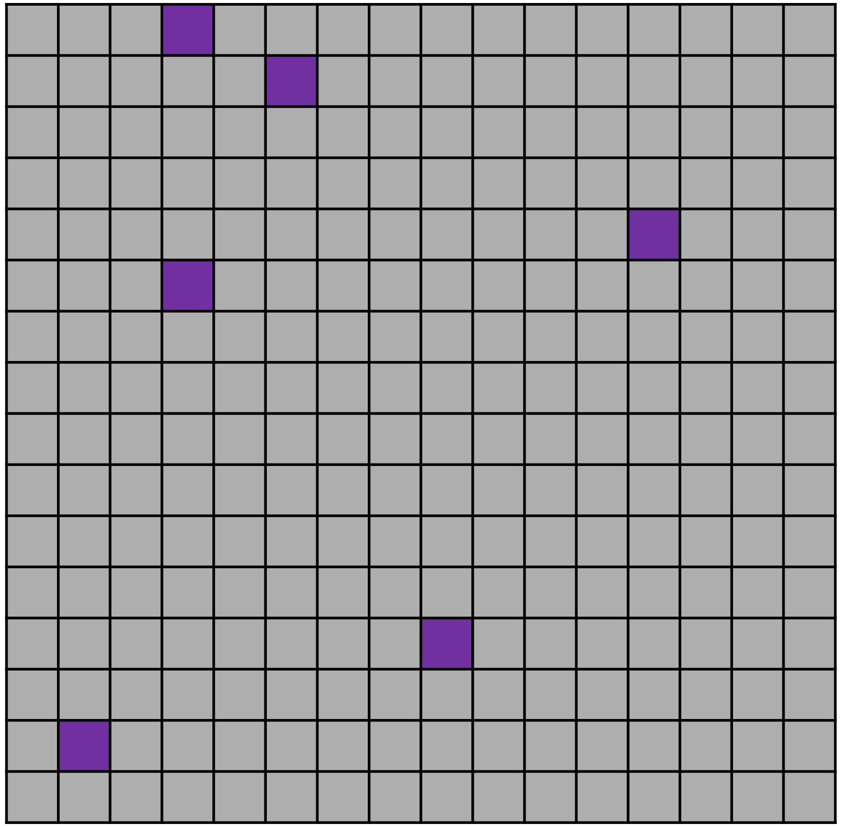
Garbage Collection



Garbage Collection

Once a block is mostly invalid, it can be scrubbed.

Deletion

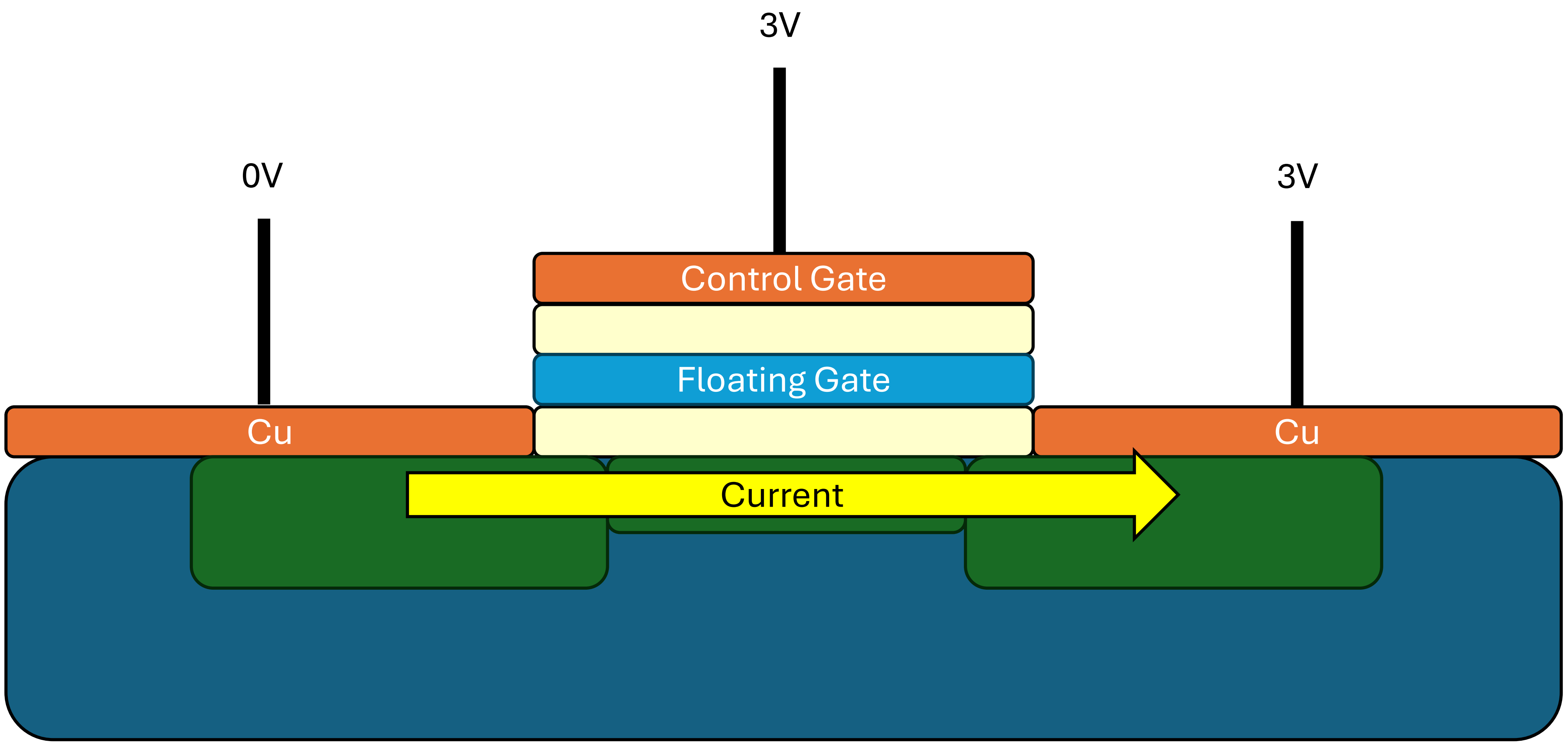


TRIM Command

- Delete just means… mark a page as **invalid**

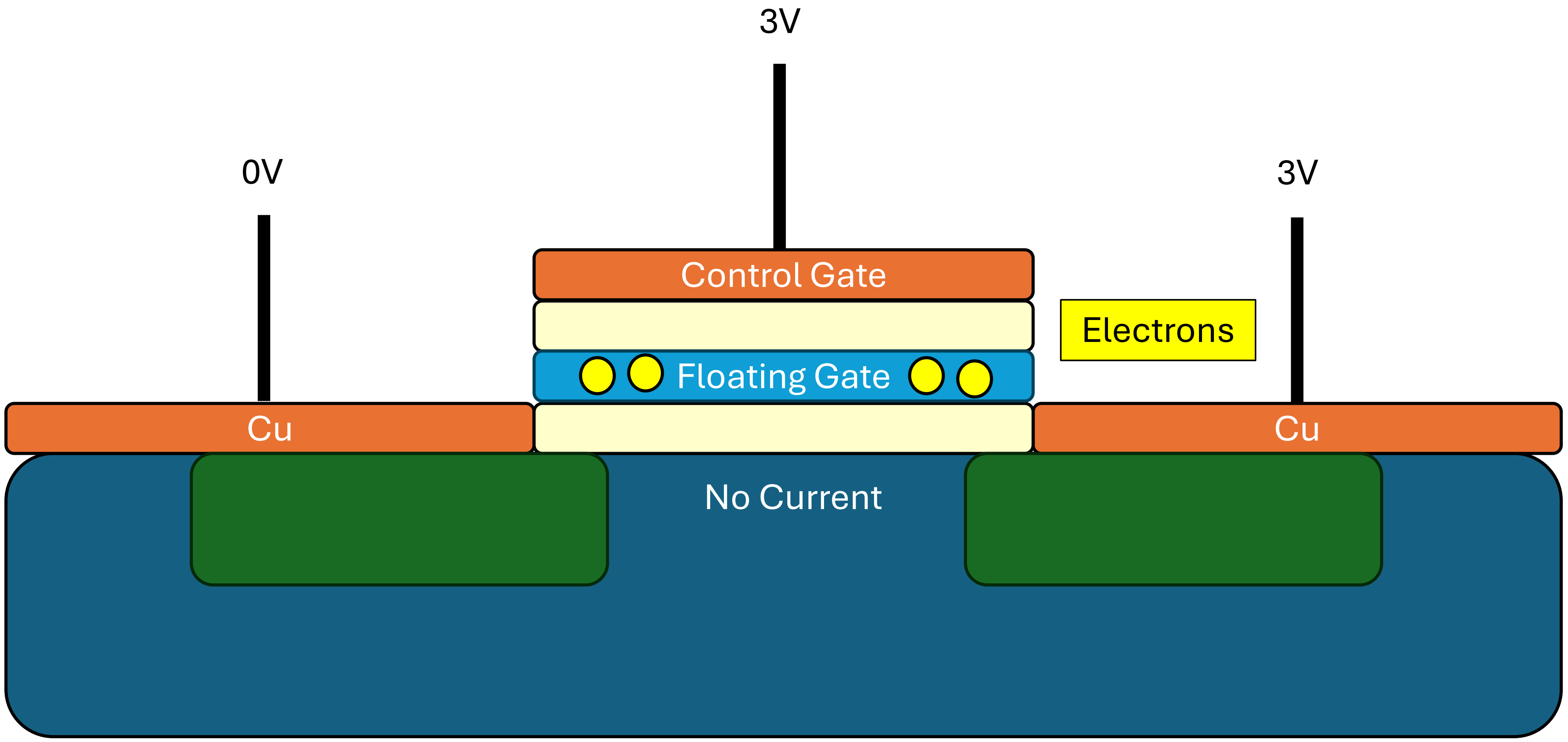
Flash Memory (1)

Current implies 1



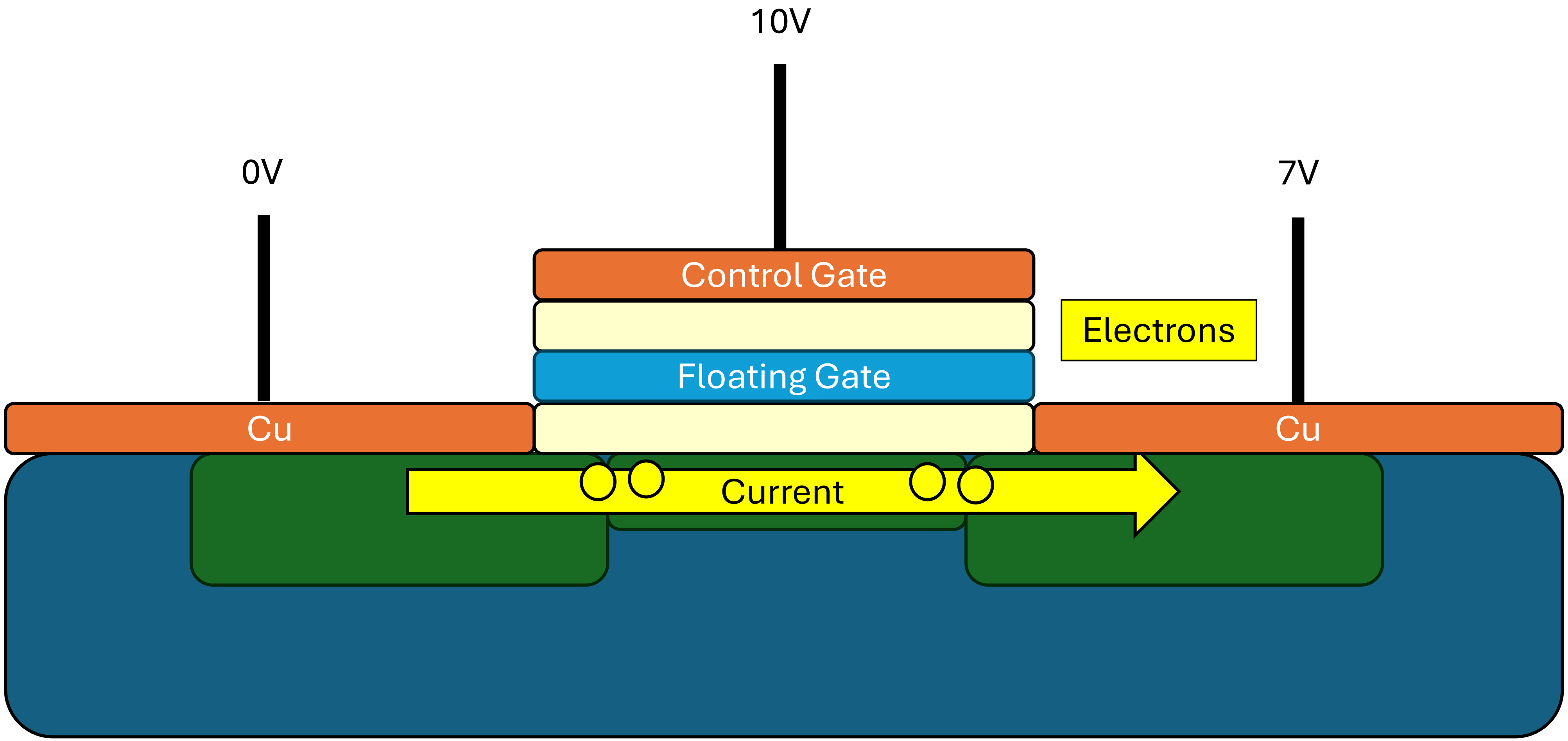
Flash Memory (0)

No Current implies 0



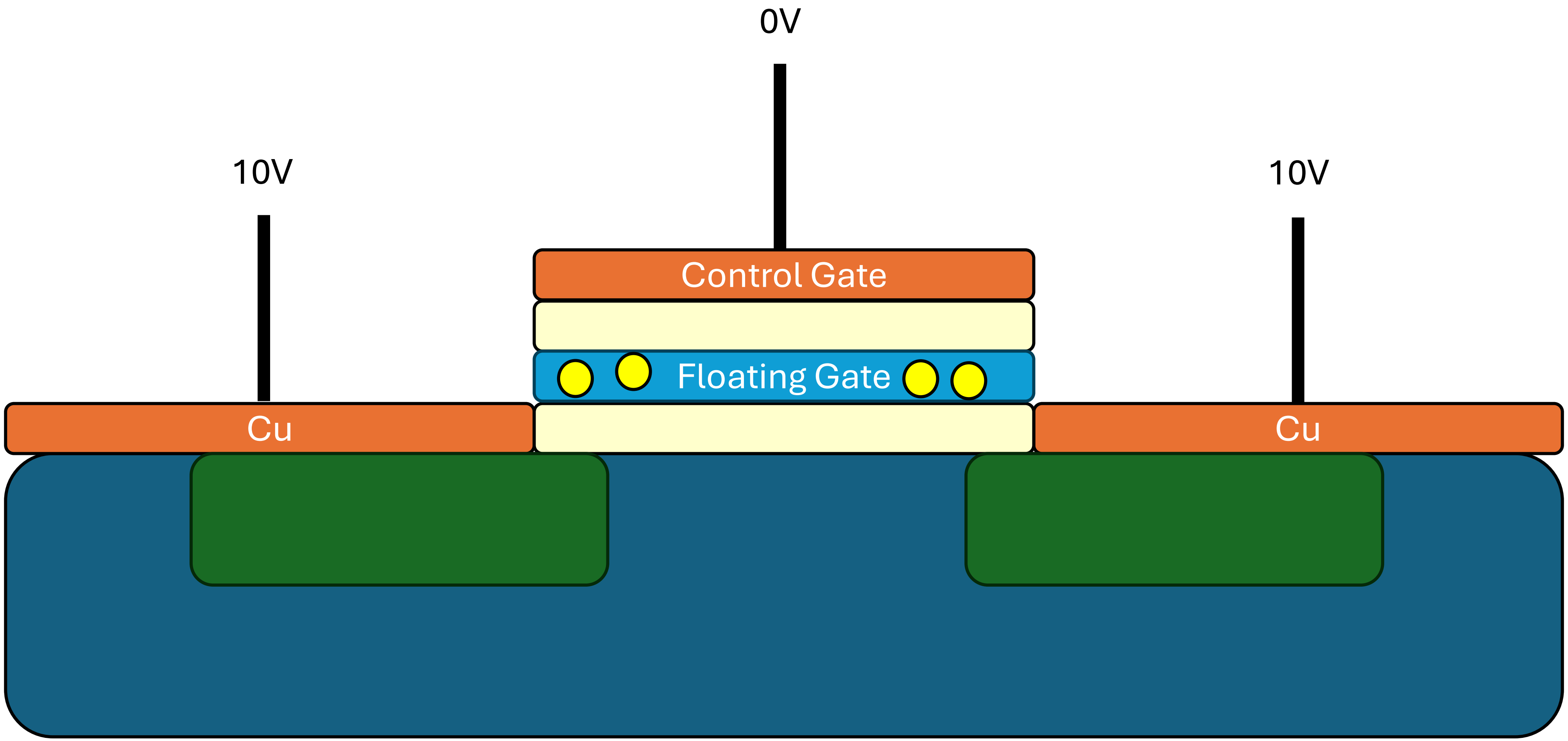
Flash Memory (Writing zero)

No Current implies 0



Flash Memory (1)

Current implies 1



Flash Memory: Problems

Erasing does damage to the cell itself.

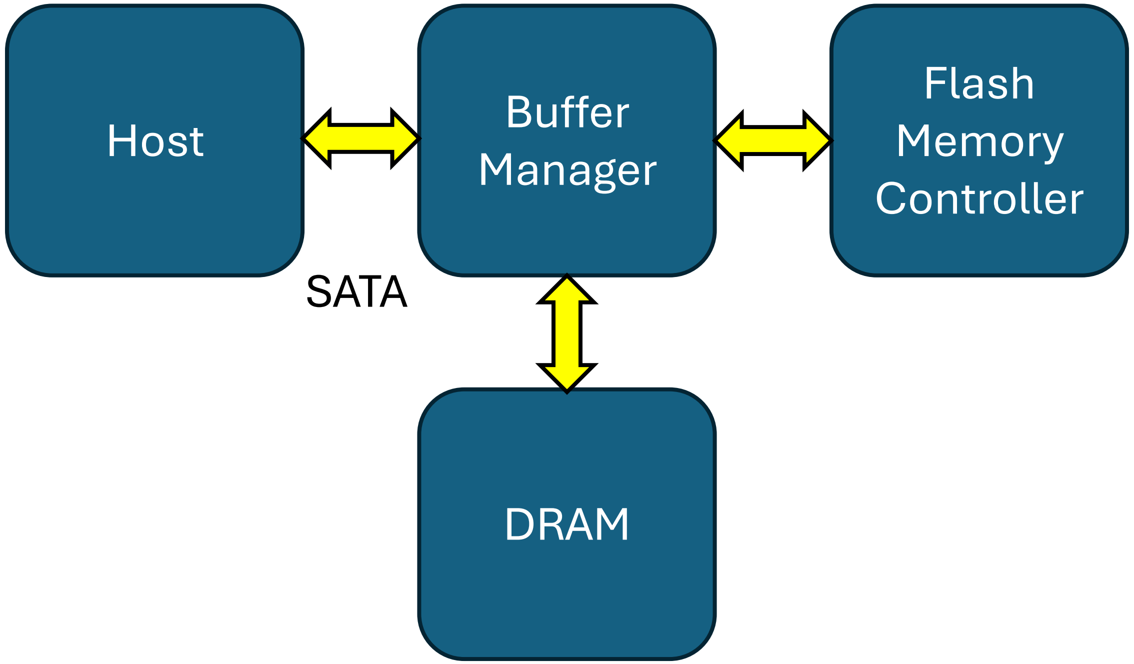
$\Rightarrow$Each cell has a limited lifecycle (~10K)

$\Rightarrow$ minimise its use

SSD Architecture

Write 4KB Page ~200 μs – 1.7 ms

- Only empty pages
- Erase ~ 1.5ms



New Bottlenecks

Hard Drives: Slow

⇒Use SATA bus (600 MB/s)

SSD Drives: Fast

⇒Use PCIe bus (7-14 GB/s)

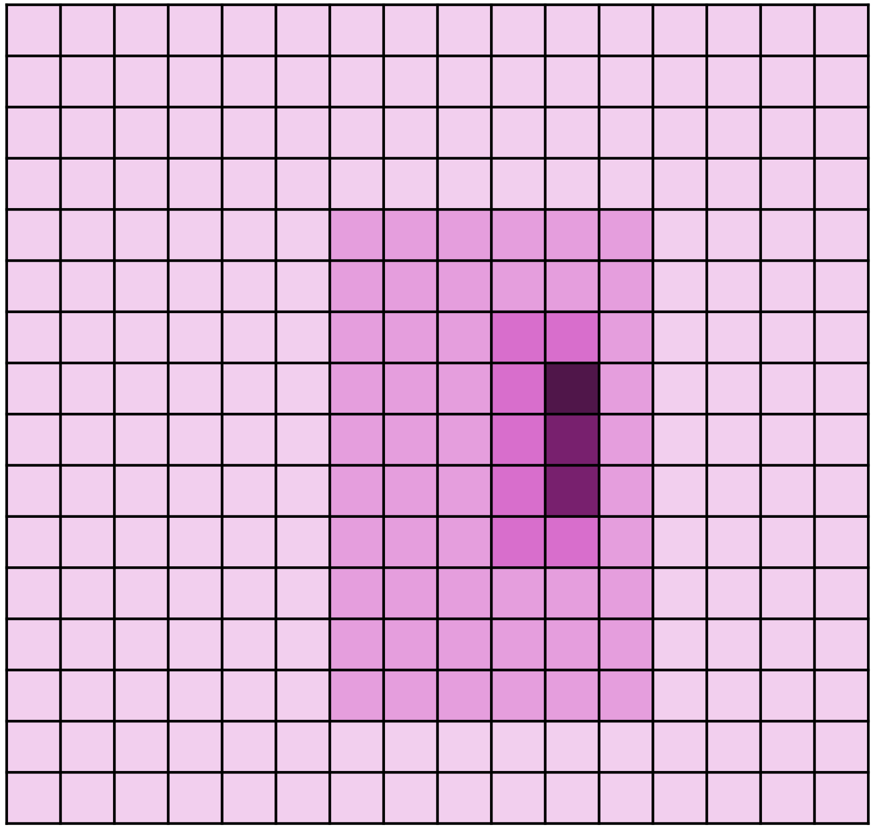
M2 Slots added

Software Changes needed too:

- AHCI (Advance Host Controller Inteface)
  - Single command queue, 32 entries
- NVMe
  - Reduced latency
  - Massive parallelism

Wear Leveling

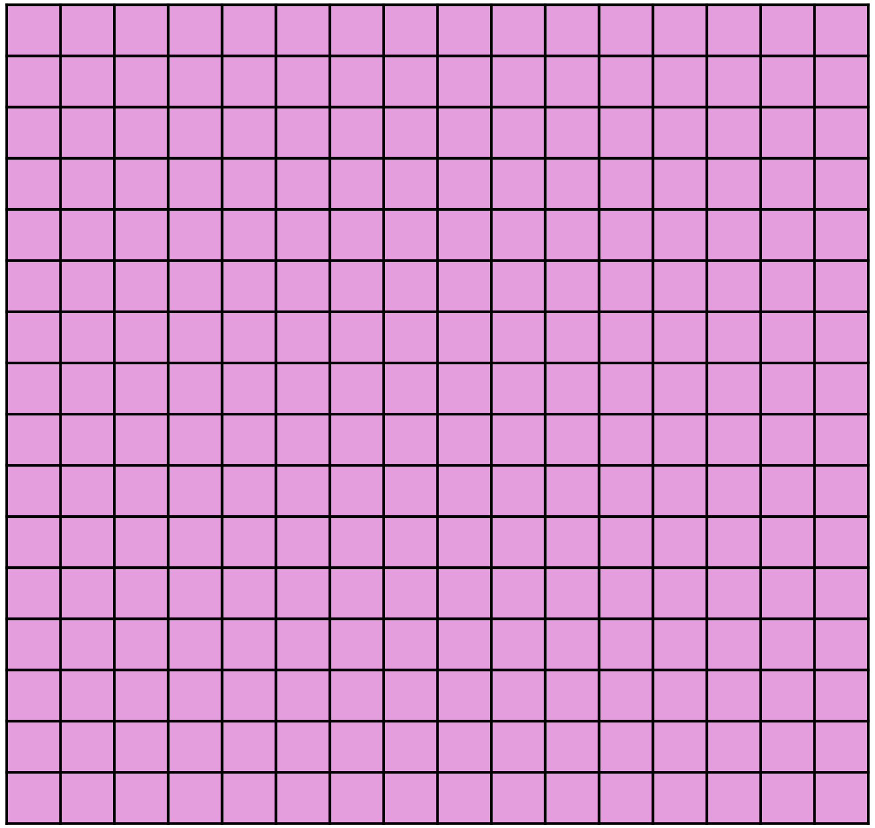
If memory blocks can wear out:



This is bad

Wear Leveling

If memory blocks can wear out:



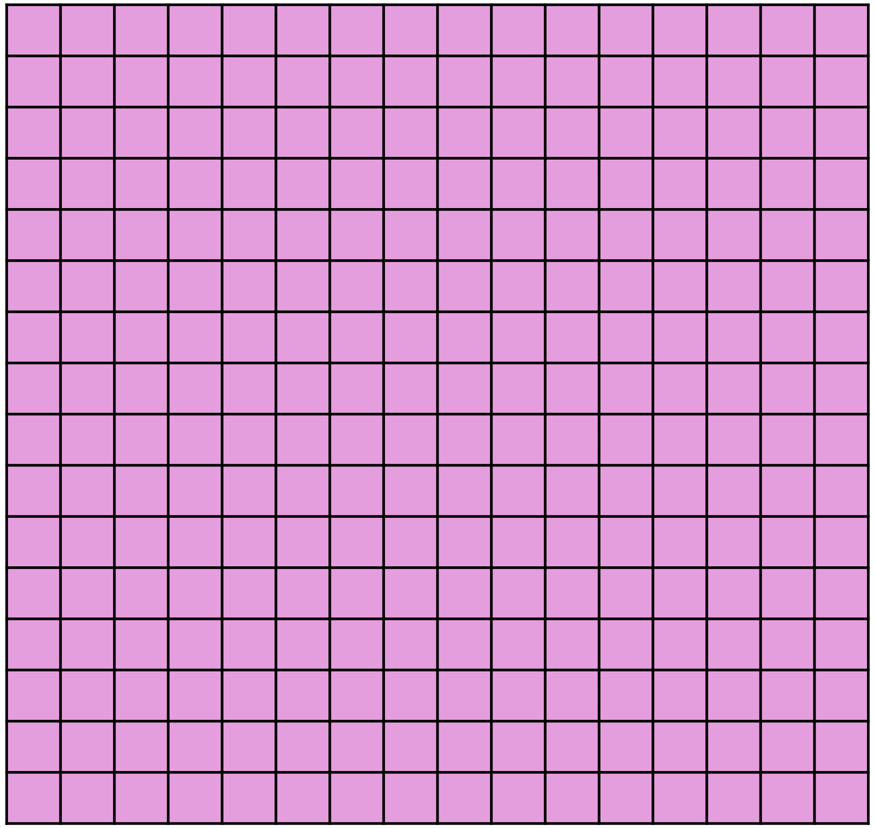
This is better

Wear Leveling

If memory blocks can wear out:

But…

- 10K writes
- Static data is also bad… (can get stuck)
- Some data shuffling occurs



This is better

Wear Leveling

Layer indirection

- Maintain a Flash Translation Layer (FTL in SSD)
- Map virtual block numbers (SO) to physical page numbers (flash memory controller uses these)

The memory can move without the OS knowing

Copy on Write

- Do not overwrite a page when OS updates its data
- Instead write a new version to a free page.
- Update FTL mapping

SSD Summary

Pros (vs Hard Drives)

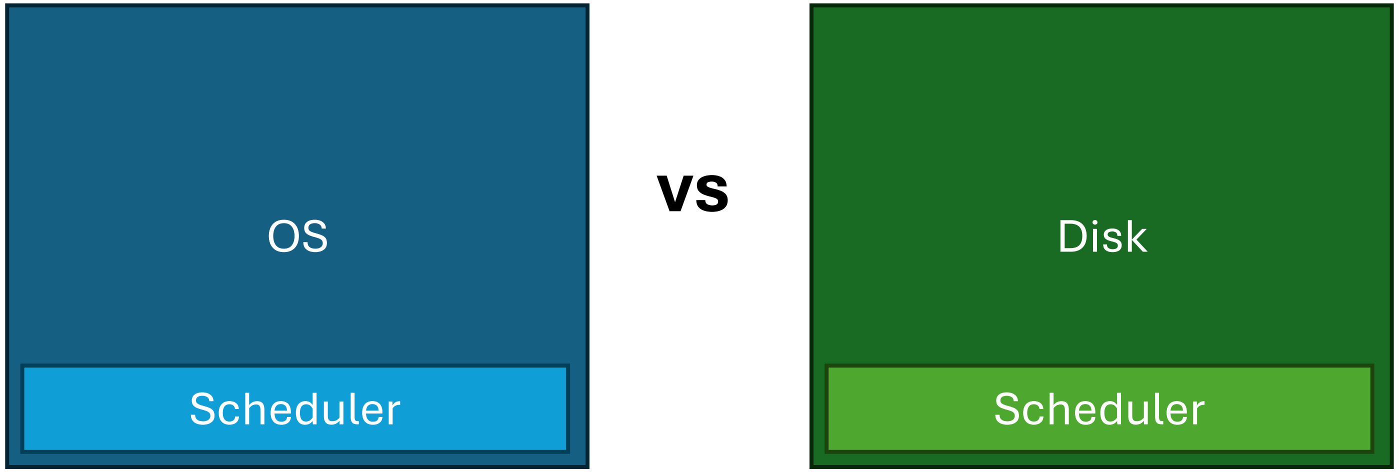
- Faster (lower latentcy, not seeking/rotating)
- No moving parts (less fragile)

Cons

- Assymetric performance (read/write/erase)
- Complicated
  - Wear Leveling
  - Garbage collections
- Lifetime (pretty minor)

Schedulers

Where to put it?



First Come, First Serve (FCFC)

You work this out...

Shortest Positioning Time First (SPTF)

Greedy Algorithm

- Choose the shortest one first

Implementation

OS

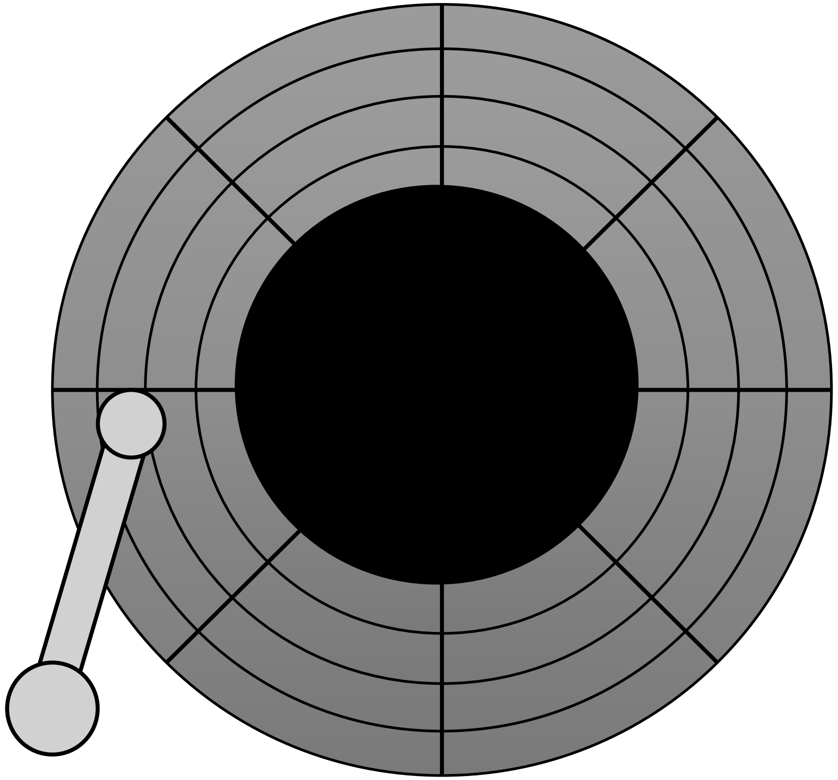
- Shortest Seek Time First
  - Starves distant requests

SCAN

Elevator Algorithm

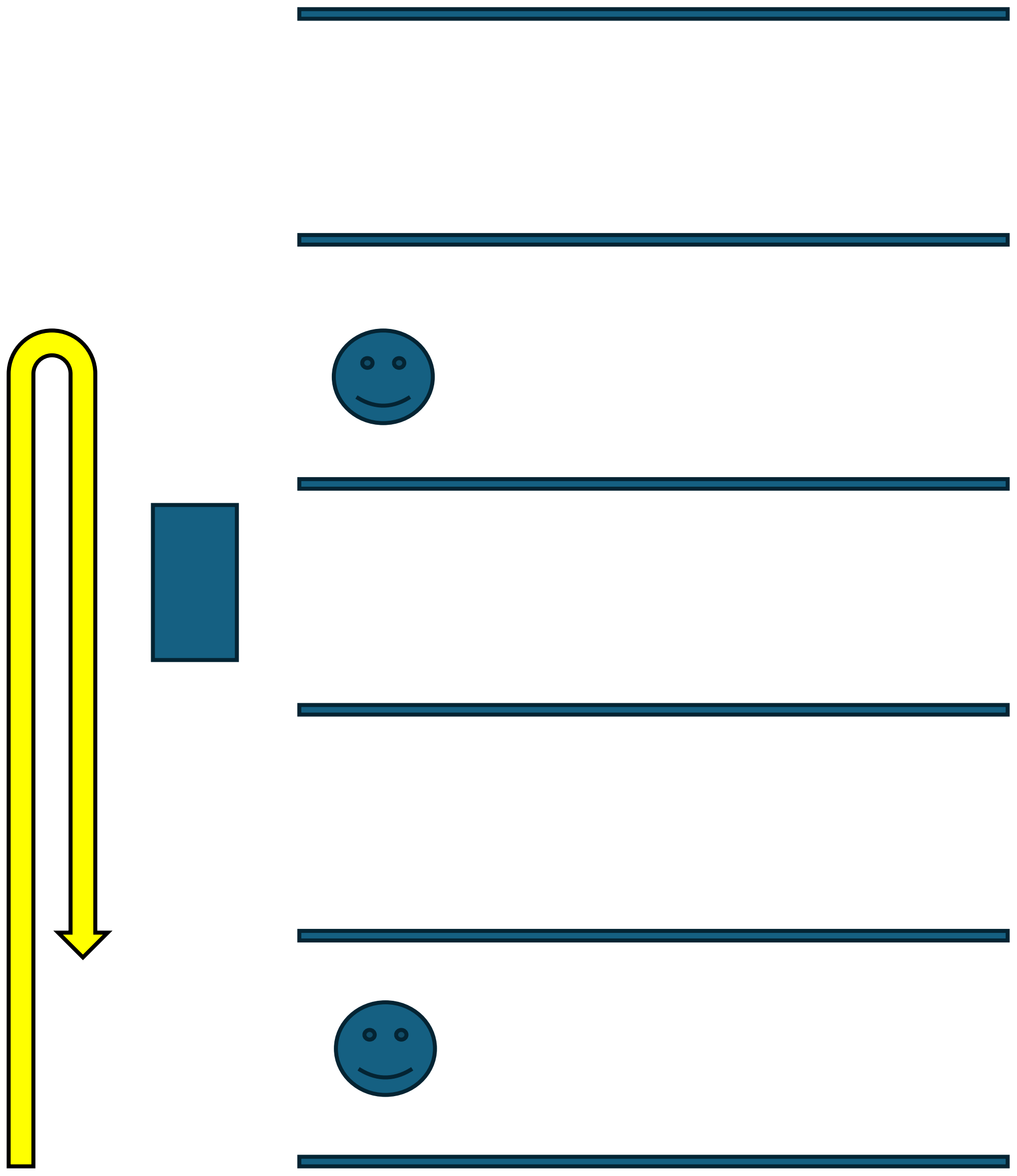
- Go up and down sweeping and servicing requests as you pass them
  - Sorts by cylinder (ignores rotation delays)

Circular Scan



LOOK

Like SCAN, but reverse when you reach the last request.



I/O Device Summary

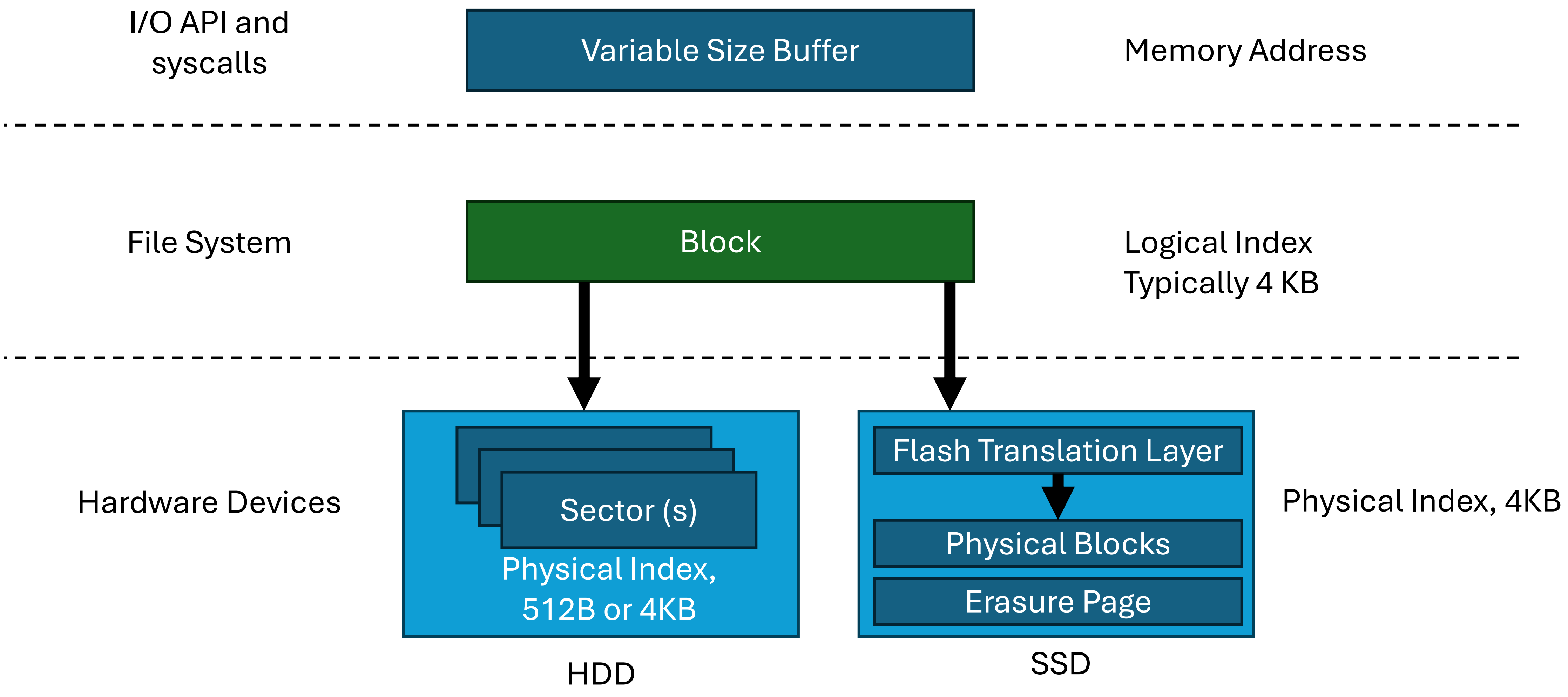
Summary

- Overlap I/O with CPU where possible
  - Interrupts and DMA are your friend
- Storage devices pretend to be ‘one big block’ (even when they are not)
- Never to random access I/O unless you really need to (i.e., linked lists are bad)
- Scheduling can be effective for otherwise slow devices

Filesystems

How?

From Storage to File Systems



An interesting issue

Windows:

C:

Linux

****

Summary

- Canonical Device
- Direct Memory Access
- Hard Drives
- Solid State Drives

Questions?