Operating Systems

Input/Output Devices

ENCE 360

Need for Input and Output

- An OS clearly needs input
  - How else can it know what services are required?
- An OS clearly provides output
  - How else are users/clients supposed to benefit from the services?

THE CRUX: HOW TO INTEGRATE I/O INTO OPERATING SYSTEMS?

How should I/O be integrated into OS?
What are the general mechanisms?
How can we make them efficient?

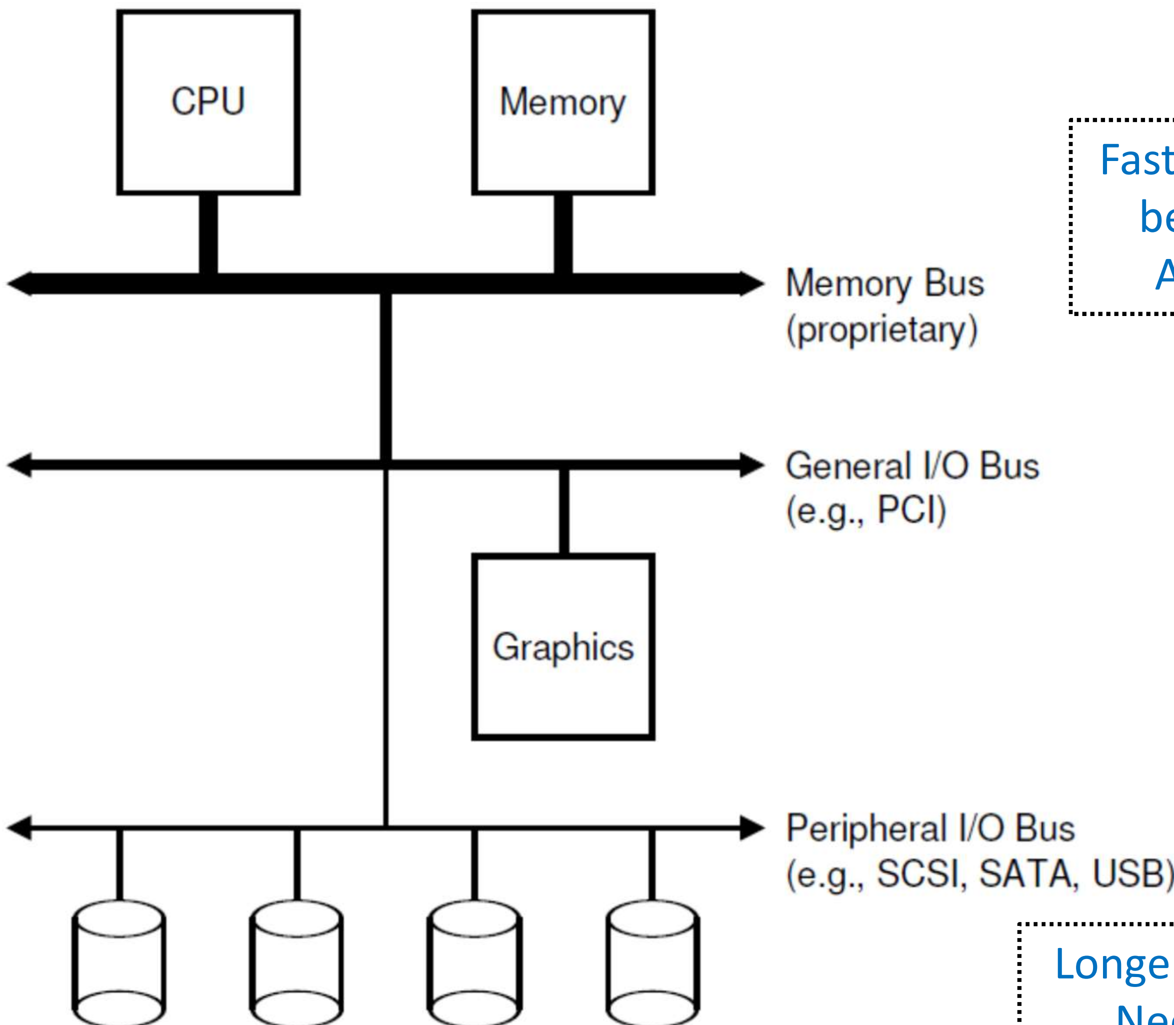
Outline

- Introduction (done)
- Device Controllers (next)
- Device Software
- Hard Disks

Chapter 5
MODERN OPERATING SYSTEMS (MOS)
By Andrew Tanenbaum

Chapter 36, 37
OPERATING SYSTEMS: THREE EASY PIECES
By Arpaci-Dusseau and Arpaci-Dusseau

Prototypical System Architecture



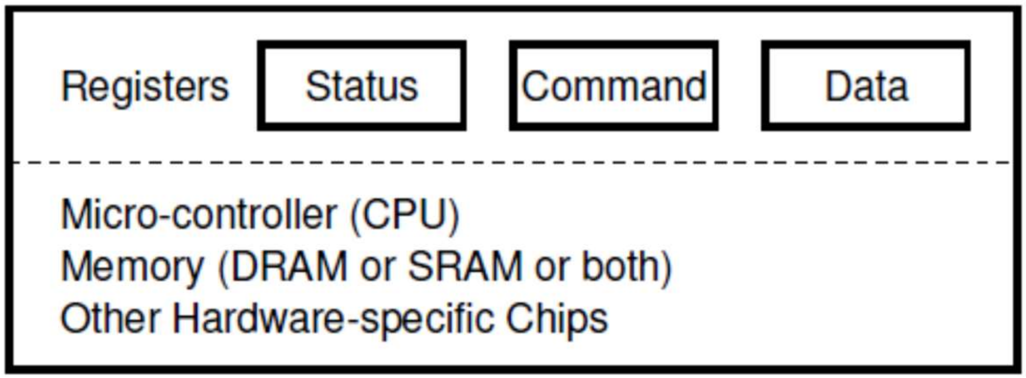
Fast, so must be short. Also $$

Devices that demand high perf generally closer to CPU

OS must deal with all devices!

Longer, so slower. Need many devices

Canonical Device



Internals can be simple (e.g., USB controller) to complex (e.g., RAID controller)

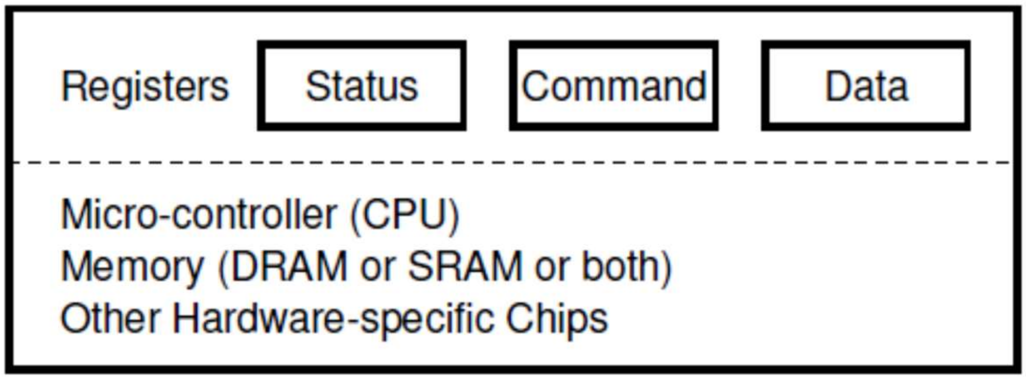
For OS, device is interface - like API of 3rd party system/library!

Canonical Protocol

```
while (STATUS == BUSY)
  ; // wait until device is not busy
write data to DATA register
  ; // device may need to service request
write command to COMMAND register
  ; // starts device to execute command
while (STATUS == BUSY)
  ; // wait until device is done
```

See any problems?
Hint: remember,
devices can be slow!

Canonical Device



Internals can be simple (e.g., USB controller) to complex (e.g., RAID controller)

For OS, device is interface - like API of 3rd party system/library!

Canonical Protocol

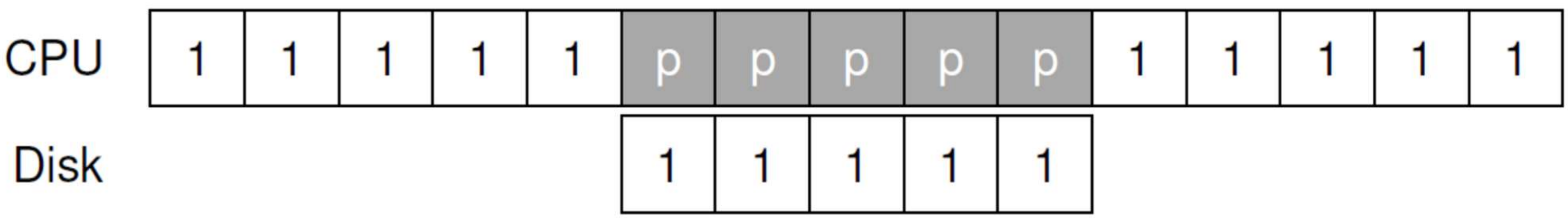
```
while (STATUS == BUSY)
   ; // wait until device is not busy
write data to DATA register
   ; // device may need to service request
write command to COMMAND register
   ; // starts device to execute command
while (STATUS == BUSY)
   ; // wait until device is done
```

THE CRUX:
HOW TO AVOID THE COST OF POLLING?

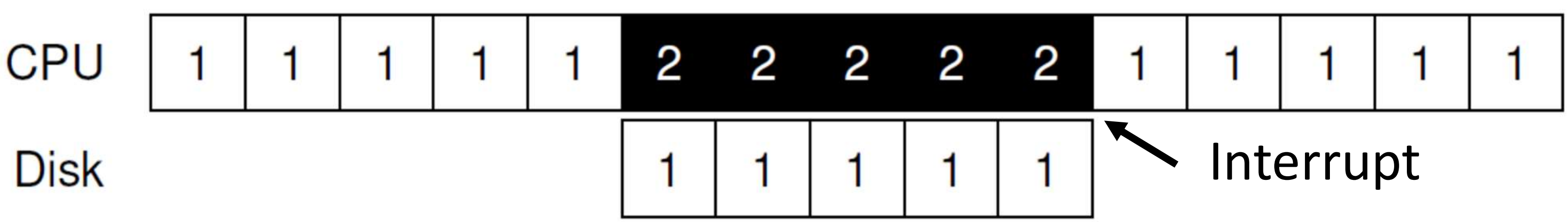
How can OS check device status without frequent polling?

Solution – the Interrupt (Again)

Polling

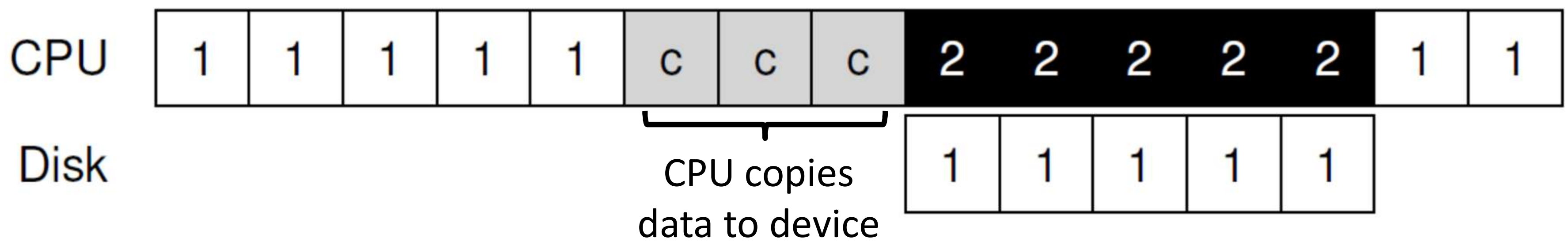


- Instead, CPU switches to new process
- Device raises interrupt when done
- Invokes interrupt handler



Copying Data? Ho, Hum

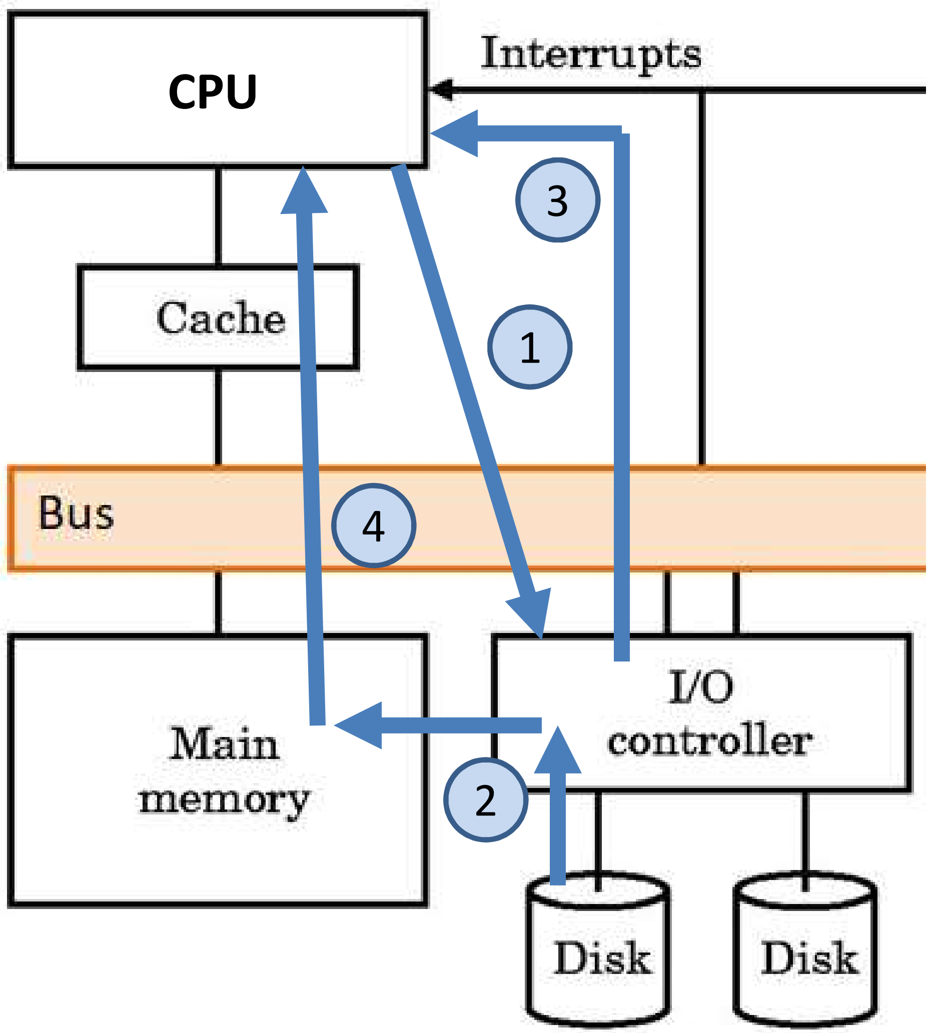
Process 1 wants to write data to disk



- CPU copying data (write and read) rather trivial
  - Could be better spent on other tasks!

THE CRUX:
HOW TO LOWER DEVICE OVERHEADS?
How can OS offload work so CPU can be more efficient?

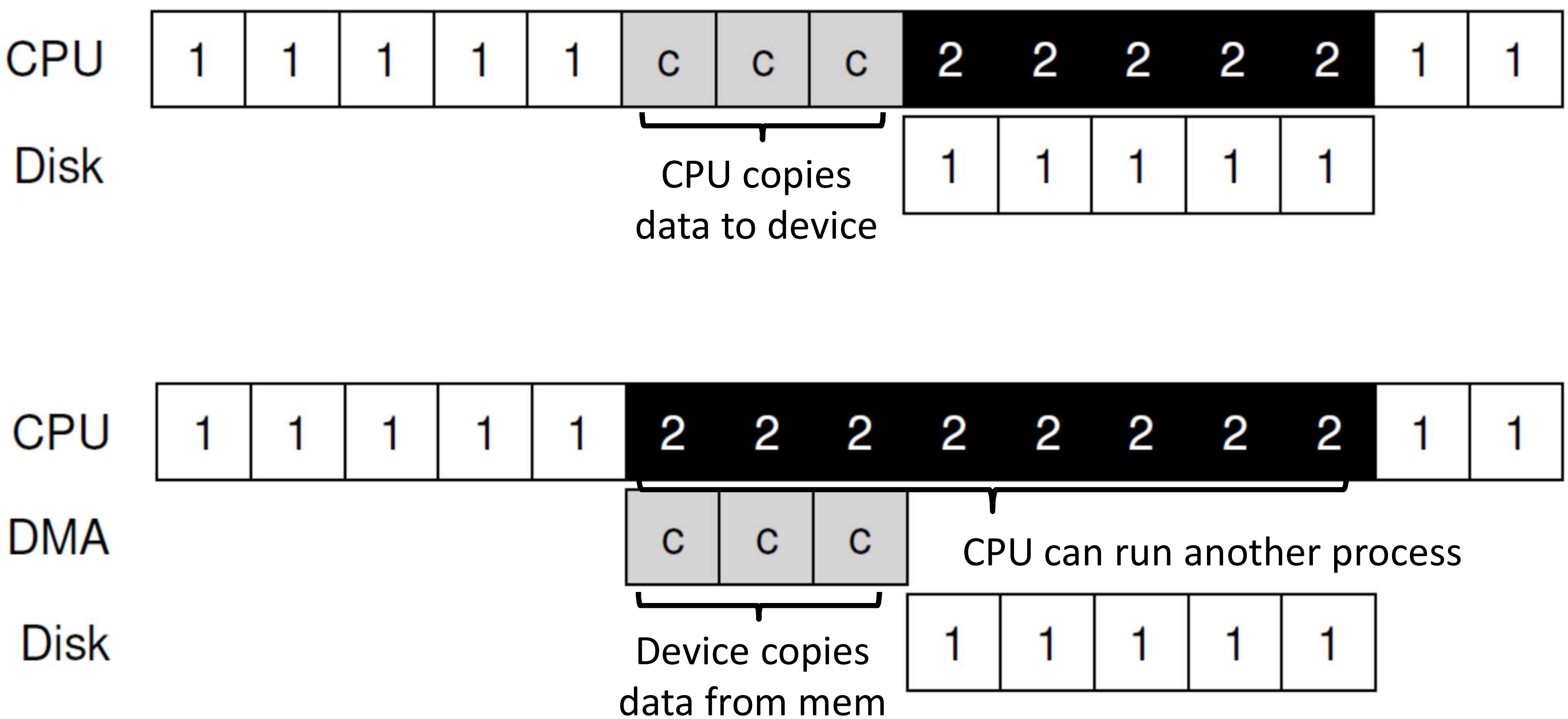
Solution – Direct Memory Access (DMA)



1. CPU provides DMA address
2. Device performs direct transfer to memory
3. Device interrupts processor
4. Processor accesses device data from memory

The Benefits of DMA

Process 1 wants to write data to disk



Outline

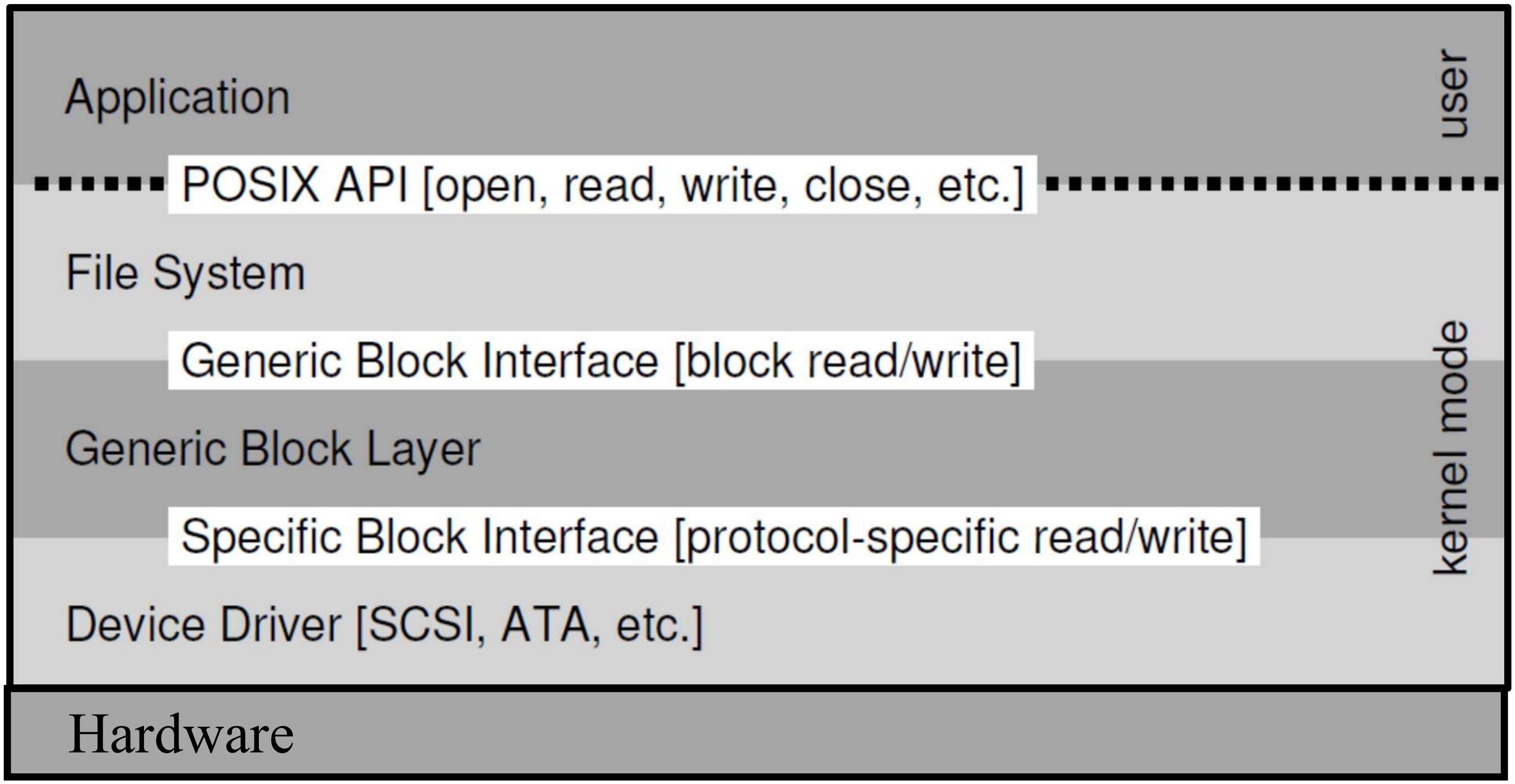
- Introduction (done)
- Device Controllers (done)
- Device Software (next)
- Hard Disks

Integration

- Devices interfaces are very specific
  - Even for functionally similar devices!
    - e.g., SCSI disk vs. IDE disk vs. USB thumb drive …
  - Not to mention functionally different devices!
    - e.g., keyboard vs. disk vs. network card …
- Want system to be (mostly) oblivious to differences

> THE CRUX:
> HOW TO BUILD DEVICE-NEUTRAL OS?
> How to hide details of device interactions from OS interactions?

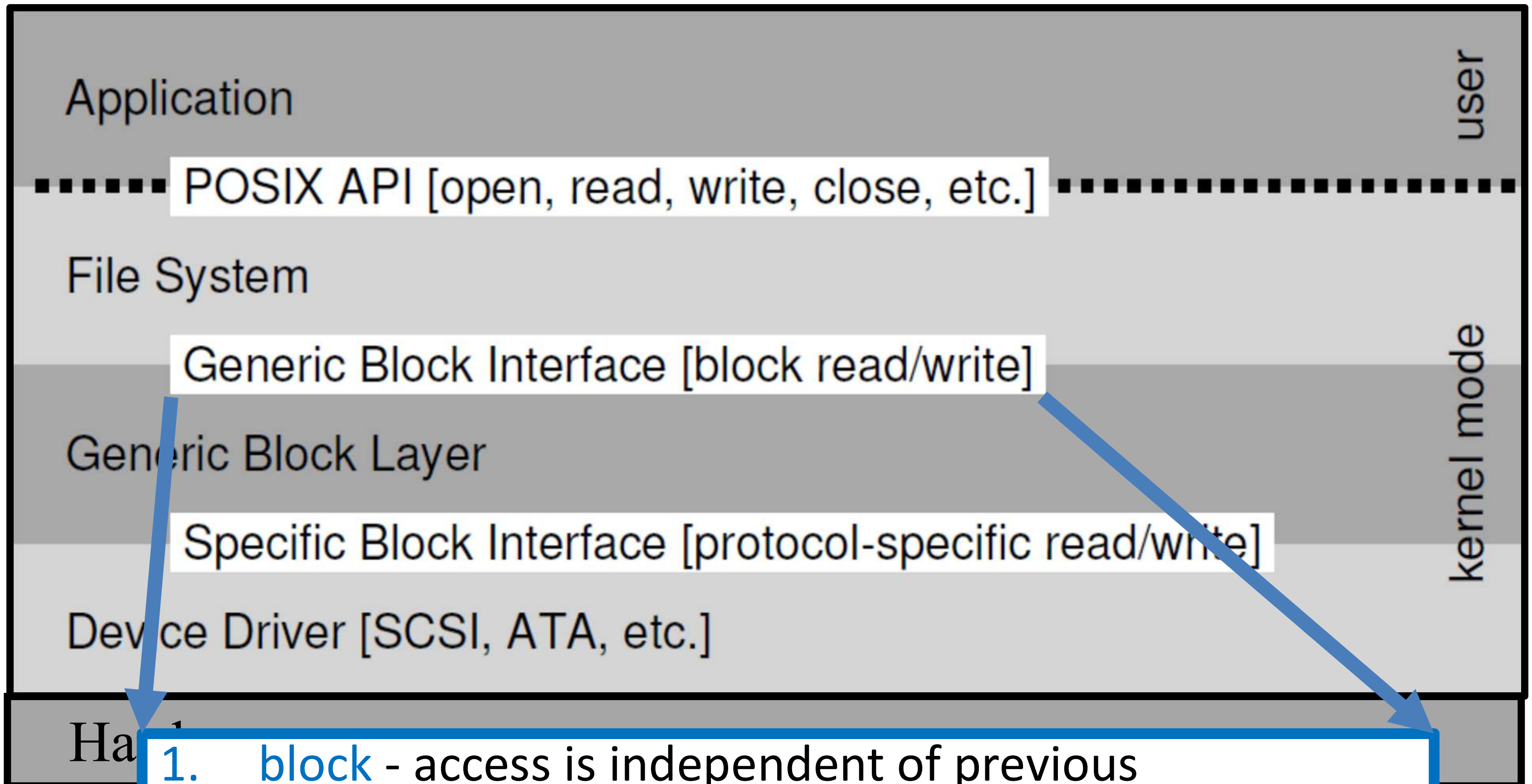
Solution – Abstraction



- Application oblivious to file system details
- File system oblivious layer specific details
- Device layer oblivious device specific details
- Device driver knows specifics of device hardware

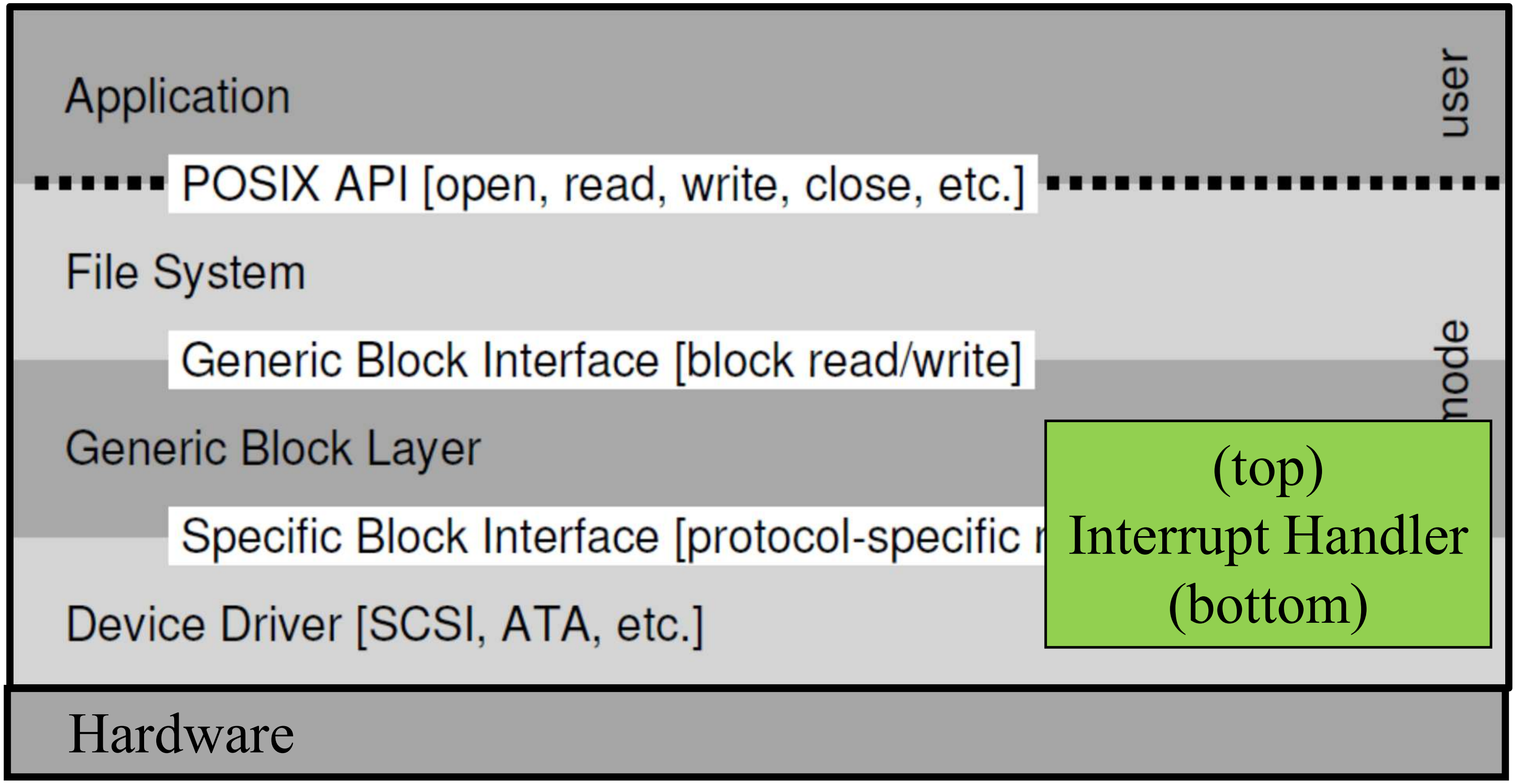
70% of Linux is device driver code!

Generic Device Types



1. block - access is independent of previous
   - e.g., hard disk
2. stream - access is serial
   - e.g., keyboard, network
3. other (e.g., timer/clock (just generate interrupts))

Interrupt Handler (1 of 2)

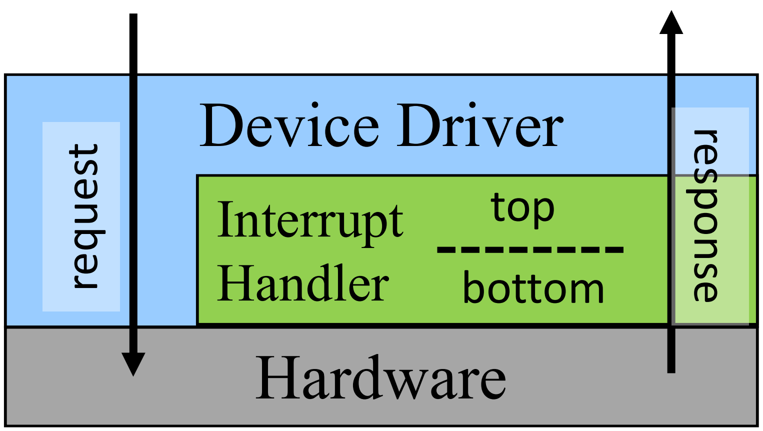


- Interrupts handled by device in two parts
  - Short at first/top (generic)
  - Longer next/bottom (device specific)

(Next slide)

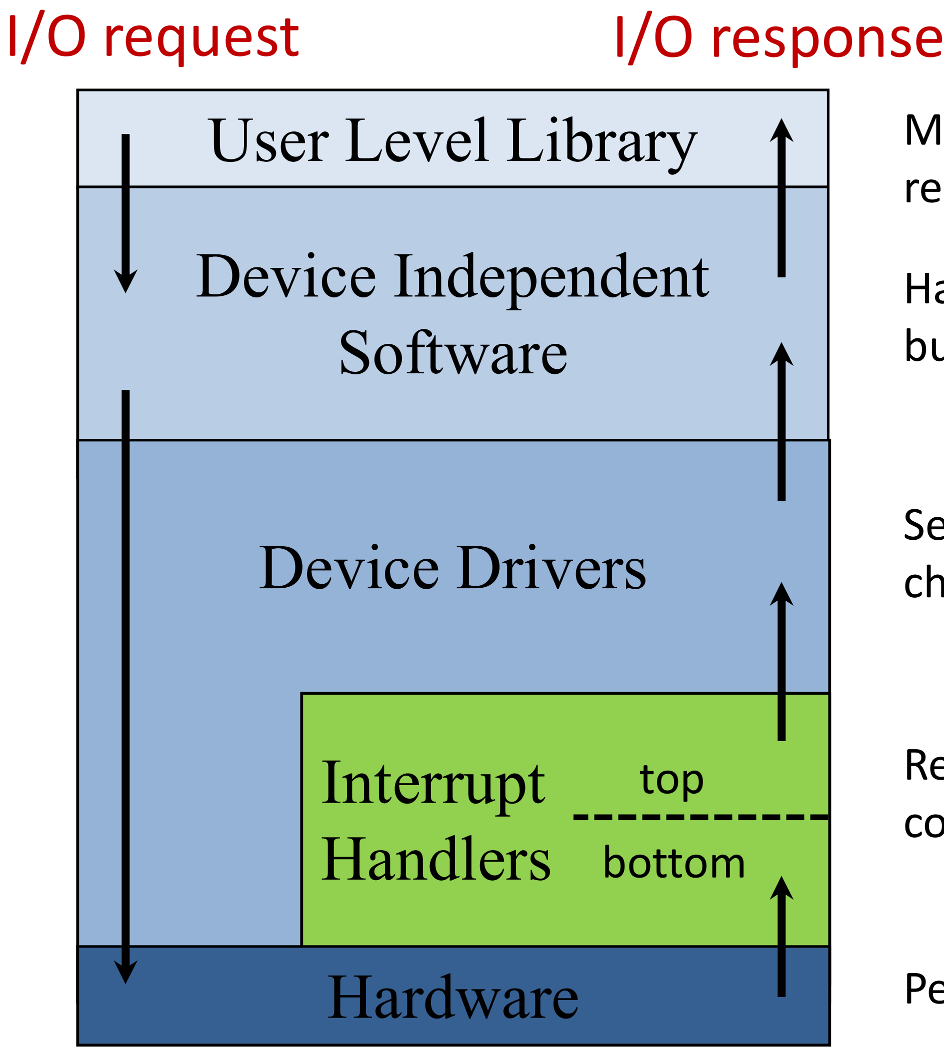
Interrupt Handler (2 of 2)

- When handling interrupt, other interrupts disabled
  - Incoming ones may be lost
  - So, make as small as possible
- Solution → Split into two pieces



- First part minimal amount of work
  - Defer rest until later
  - Effectively, queue up rest
  - Re-enable interrupts
  - Linux: "top-half" handler
- Second part does most of work
  - Run device-specific code
  - Windows: "deferred procedure call"
  - Linux: "bottom-half" handler

I/O System Summary



Make I/O call, format request and response

Handle naming, protection, blocking, buffering, allocation

Setup device registers for request, check status upon response

Respond to interrupt when device completes I/O

Perform I/O operation

Outline

- Introduction (done)
- Device Controllers (done)
- Device Software (done)
- Hard Disks (next)

Hard Drive Overview

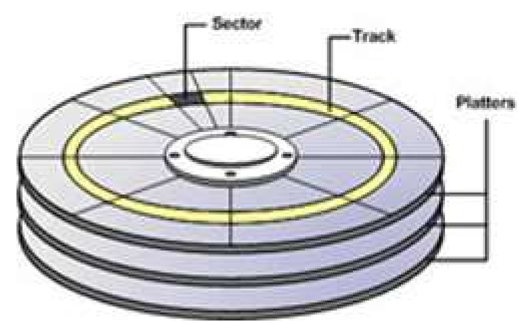
File

The quick brown
fox jumped over
the lazy dogs

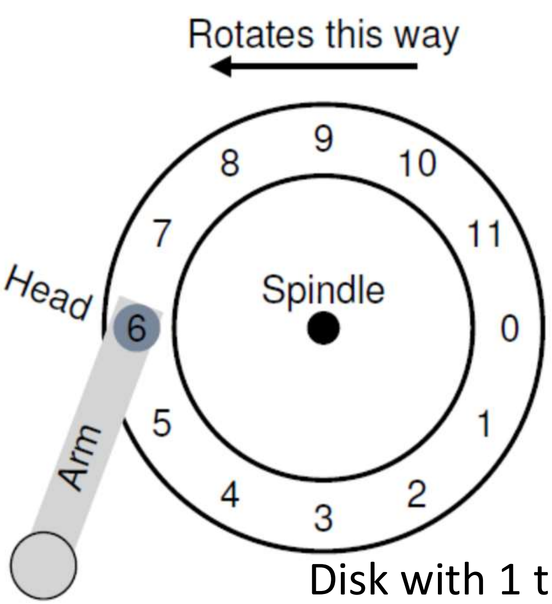
???

Hard Disk

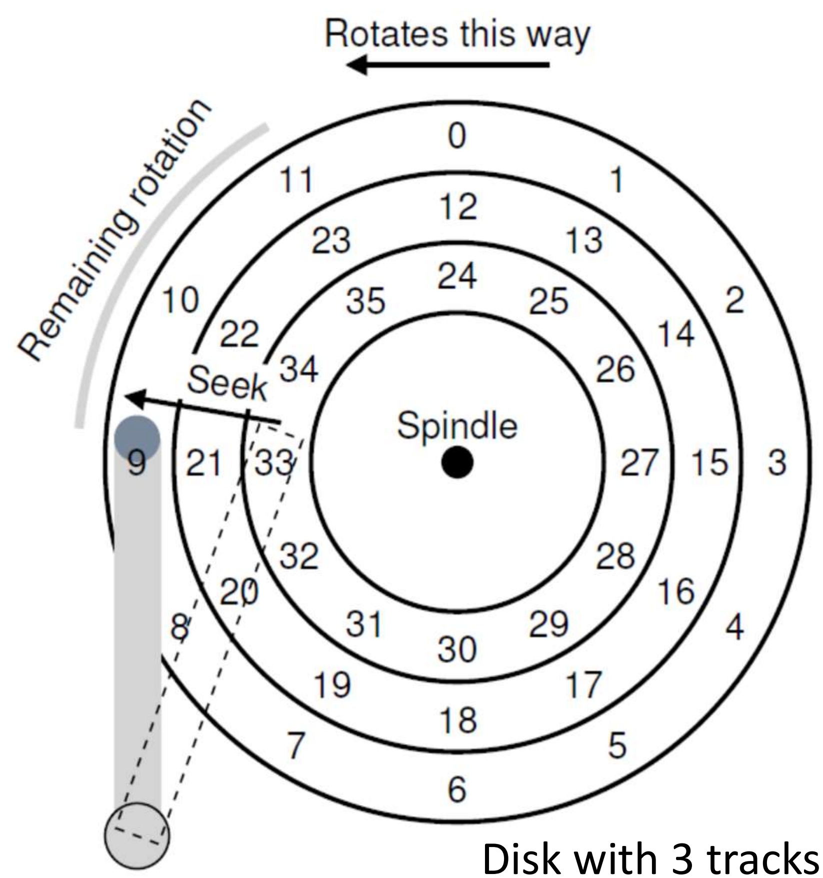
- Hard disk has series of platters
- How do bytes get arranged on disk?



Reading/Writing Disk Blocks



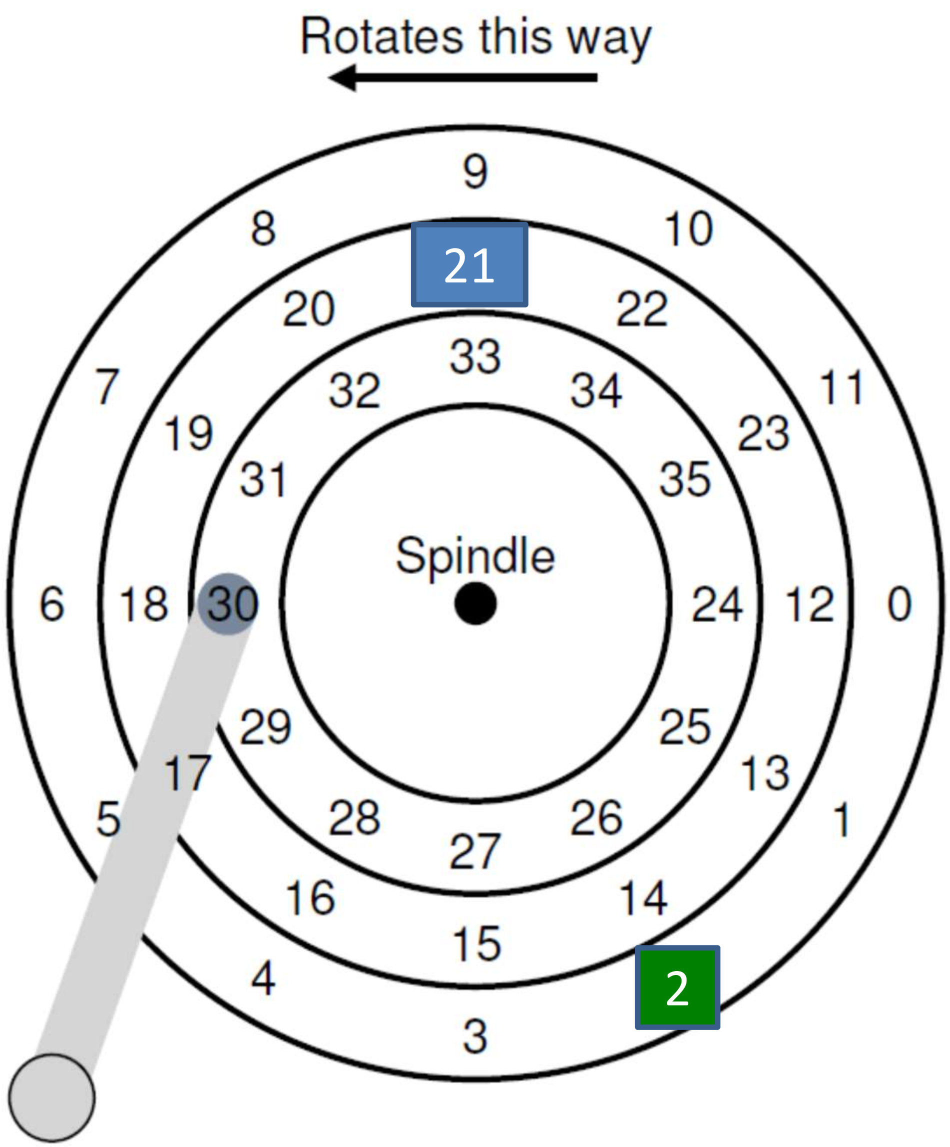
Disk with 1 track



Disk with 3 tracks

Time to read/write block:
Seek time – move arm to position
Rotation time – spin disk to right block
Transfer time – data on/off disk

Organizing Disk Block Requests



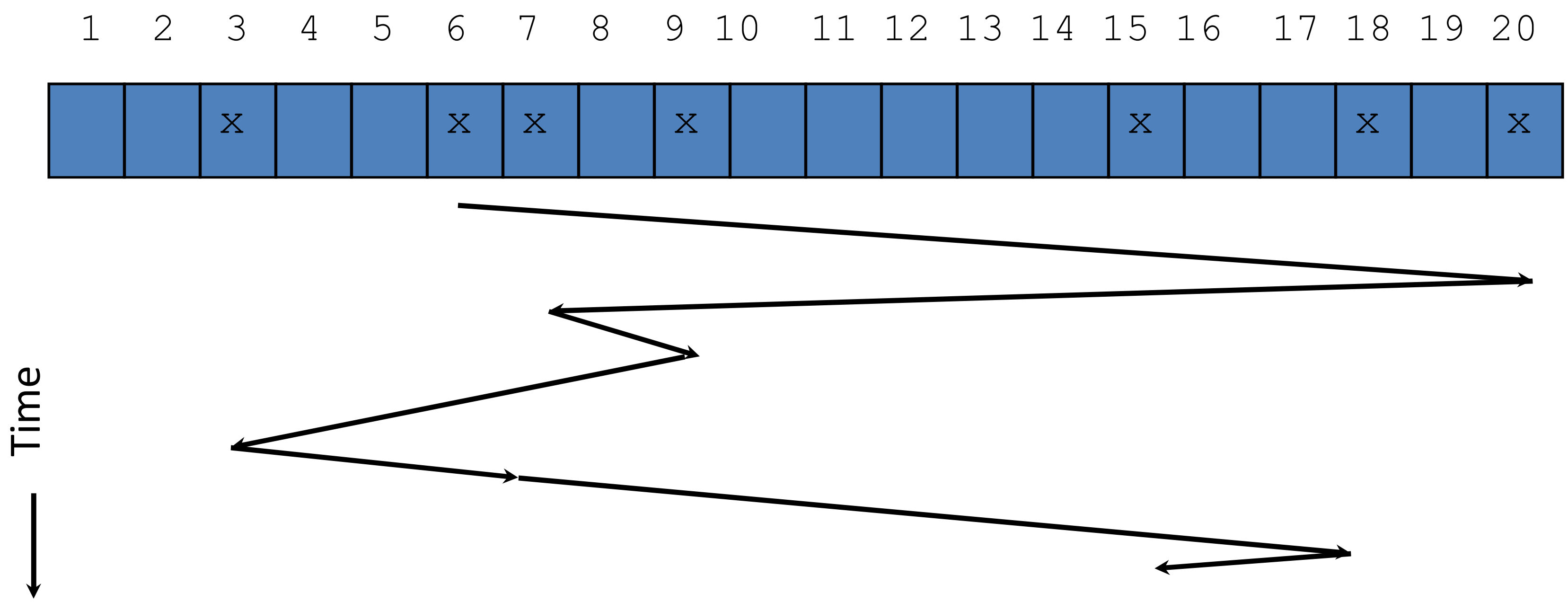
- Rotation fast
- Arm movement relatively slower

→ Seek time dominates

So, if 2 and 21, then which next?

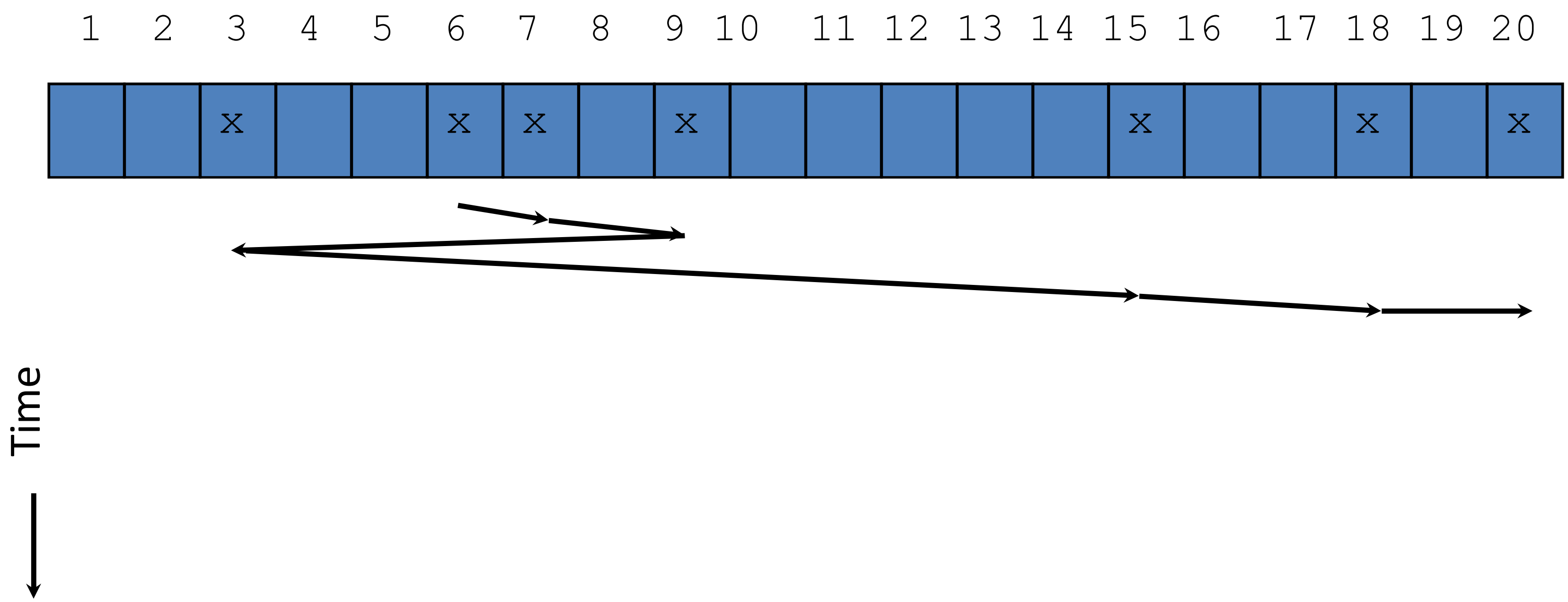
Because matters so much, OS often organizes requests for efficiency
→ But how?

First-Come First-Served (FCFS)



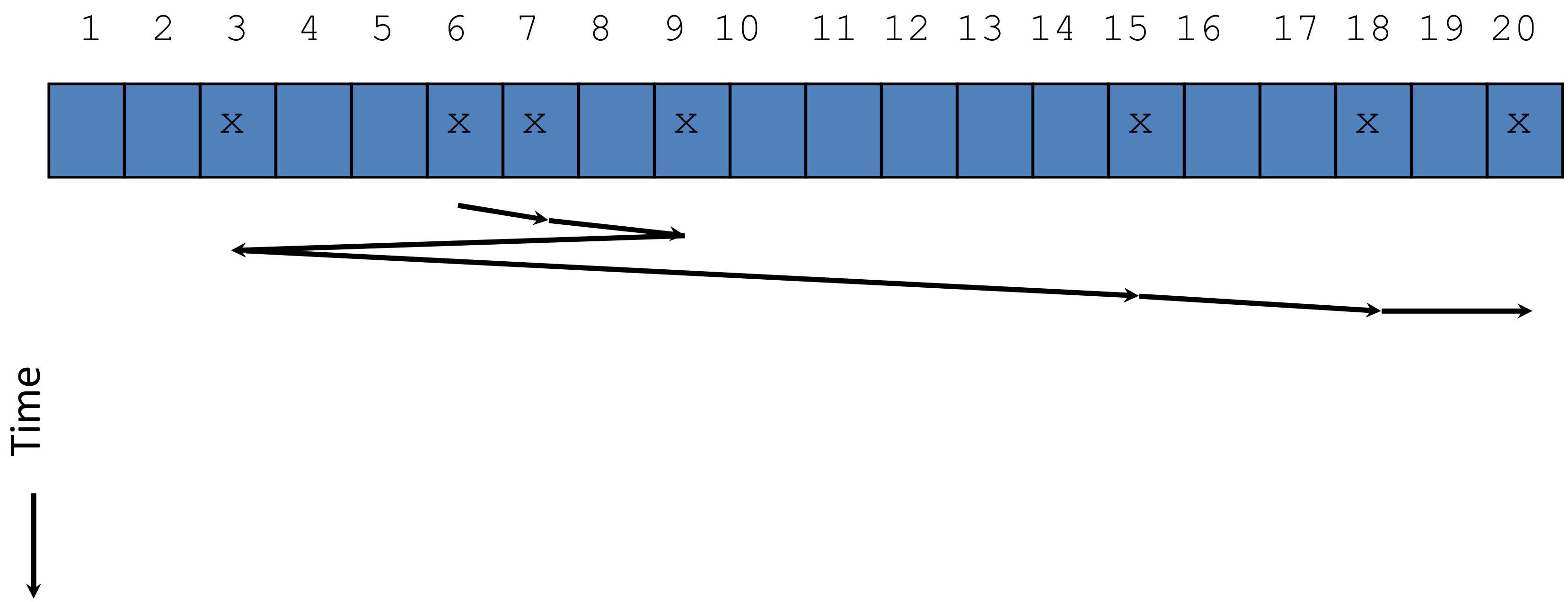
- Service requests in order that they arrive
  - Total time: 14+13+2+6+3+12+3=53
- Little done to optimize
- How can we make more efficient?

Shortest Seek First (SSF)



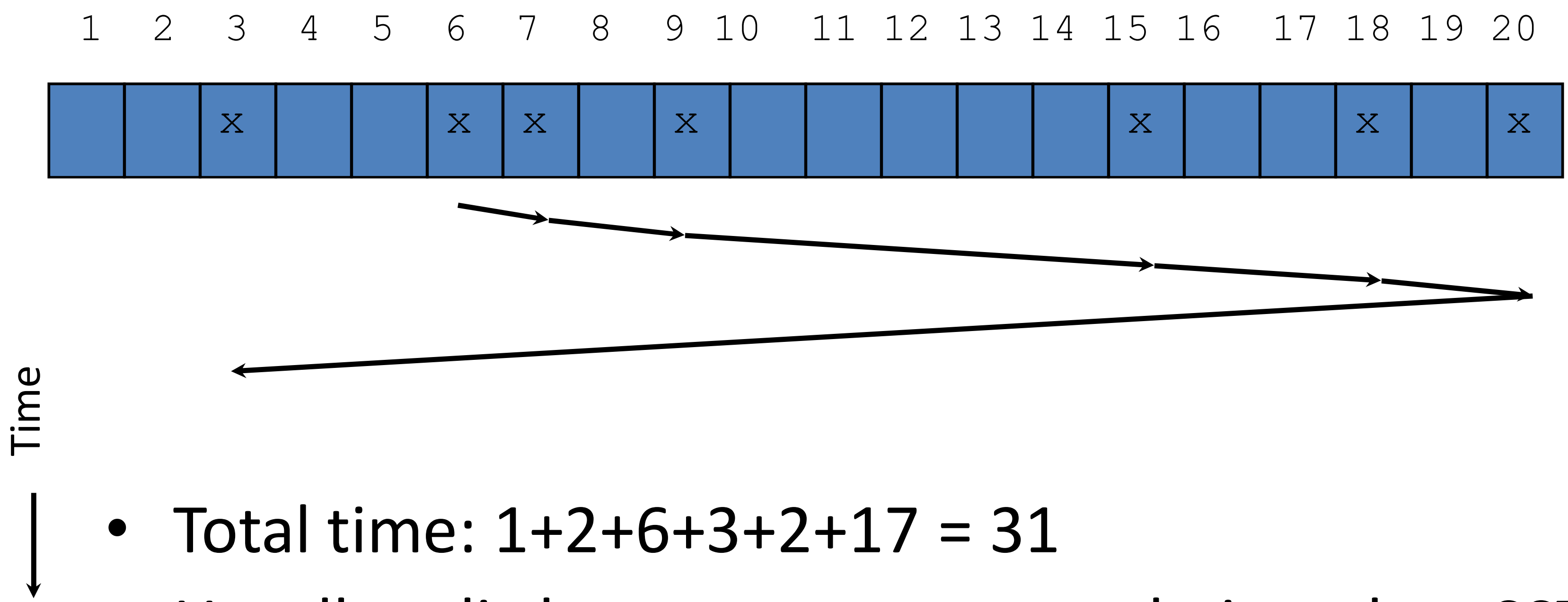
- Service request closest to read arm
  - Total time: 1+2+6+9+3+2 = 23
- What might happen that is bad?
  - Hint: something similar happened with scheduling

Shortest Seek First (SSF)



- Service request closest to read arm
  - Total time: 1+2+6+9+3+2 = 23
- What might happen that is bad?
  - Continual request near arm → starvation!

Elevator (SCAN)



- Total time: 1+2+6+3+2+17 = 31
- Usually, a little worse average seek time than SSTF
  - But more fair, avoids starvation
- Alternate C-SCAN has less variance
- Note, seek getting faster, rotational not
  - Someday, change algorithms

State of the Art – a Mixed Bag

- Disks evolving (e.g., rotation + seek converging), so OS may not always know best
- Instead, issue cluster of requests that are likely to be best
  - Send to disk and let disk handle
- Linux – no one-size fits all (sys admins tune)
  - Complete Fair Queueing (CFQ) – queue per processes, so fair but can optimize within process
    - Default for many systems
  - Deadline – optimize queries (better perf), but hard limit on latency to avoid starvation
  - Noop – no-sorting of requests at all (good for SSD. Why?)

Outline

- Introduction (done)
- Device Controllers (done)
- Device Software (done)
- Hard Disks (done)