I/O and Filesystems

Based on slides by Matt Welsh, Harvard

Announcements

- Finals approaching, make sure you know when yours are
  - Ours: May 11, 1:30 – 4:30 pm

- Honors projects due soon
  - By April 30: contact us to schedule demo
  - Before final exam: complete demo
  - All members must contribute substantially and understand the entire project

But first, review

Threads in your web server

- Why are multiple threads useful in your web server – as opposed to serving all clients with a single thread in a single process? (Check all that apply)
  - Multiple threads can spread work across multiple cores / CPUs to decrease processing time.
  - Multiple threads have greater memory space to read files and write them to the network.
  - A single thread would have to switch back and forth between each connection, which is slow and annoying to program.
  - One thread can be reading/writing from the network while another is waiting to read a file off disk.

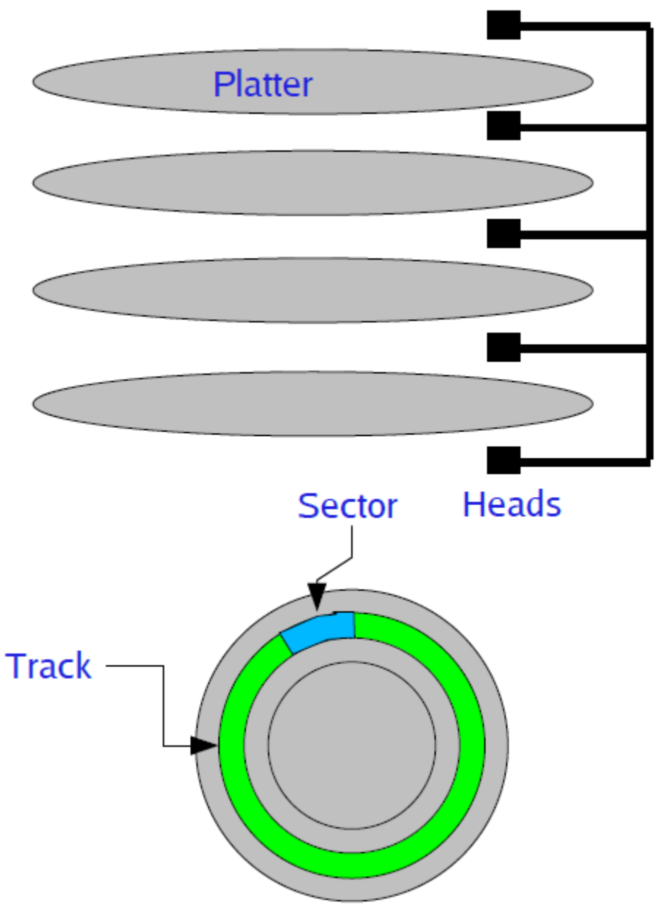
DNS caching

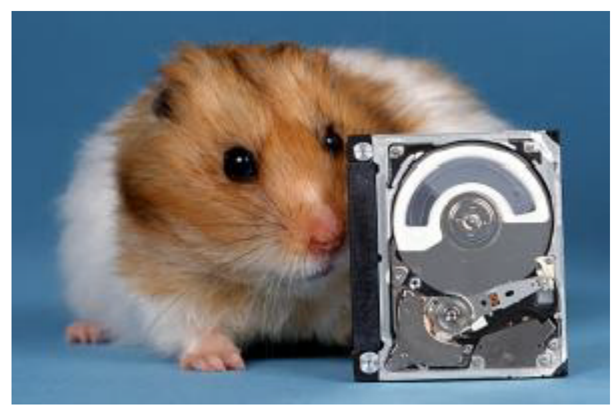
- Why does the DNS system use caching? (Check all that apply)
  - Returns more up-to-date results
  - Improves speed of response
  - Decreases workload on root and authoritative DNS servers
  - Improves security
  - Improves robustness (things still work even if some DNS servers fail)

Part 1: Disks

A Disk Primer

- Disks consist of one or more platters divided into tracks
  - Each platter may have one or two heads that perform read/write operations
  - Each track consists of multiple sectors
  - The set of sectors across all platters is a cylinder





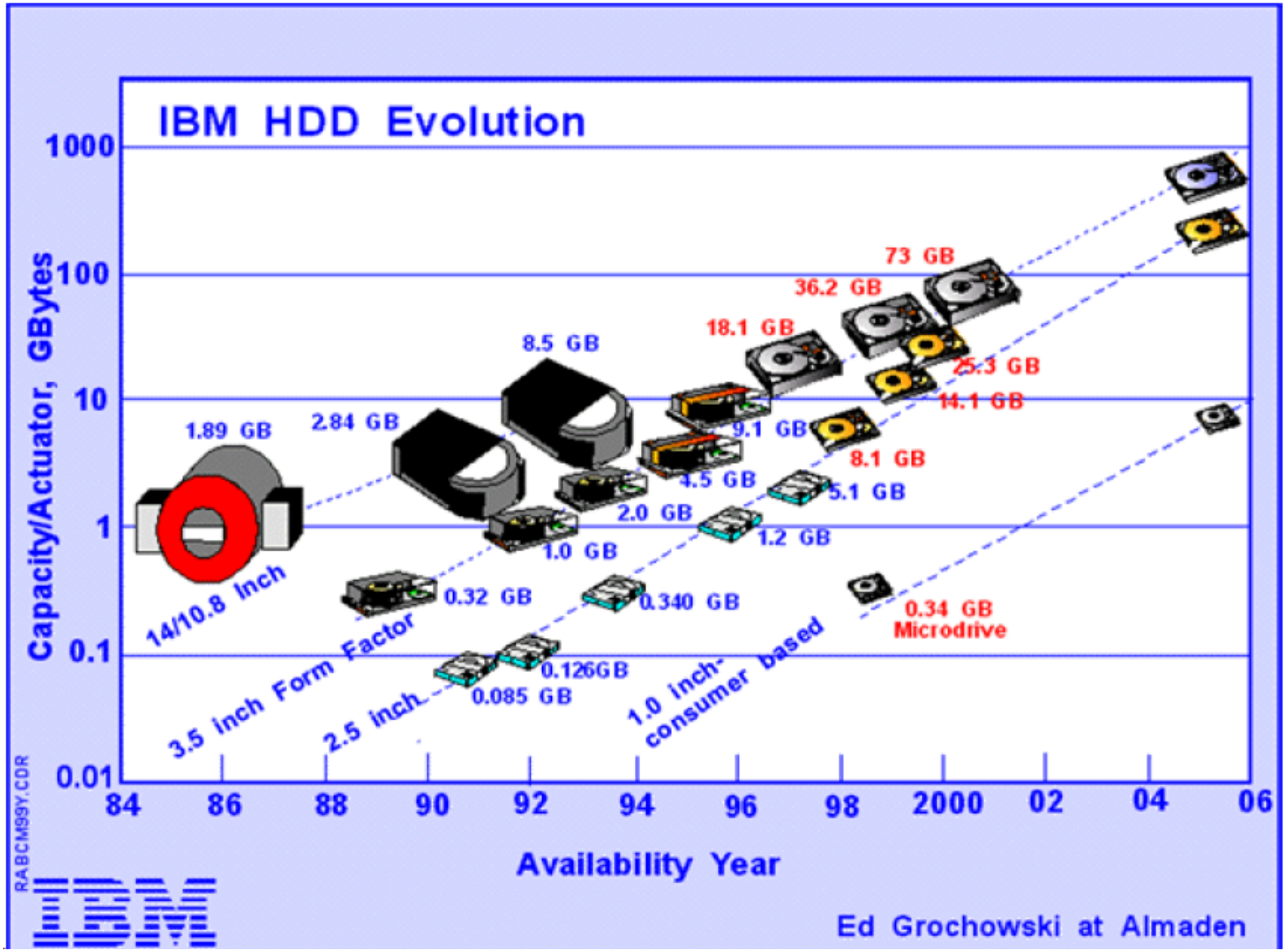
Hard Disk Evolution

- IBM 305 RAMAC (1956)
  - First commercially produced hard drive
  - 5 Mbyte capacity, 50 platters each 24” in diameter

Cost of recording your life

- How much does the disk storage to record the audio of your entire life cost?
  - The whole thing ... even when you're asleep and even the part you haven't lived yet
  - Assume pretty good quality audio

Hard Drive Evolution



Disk access time

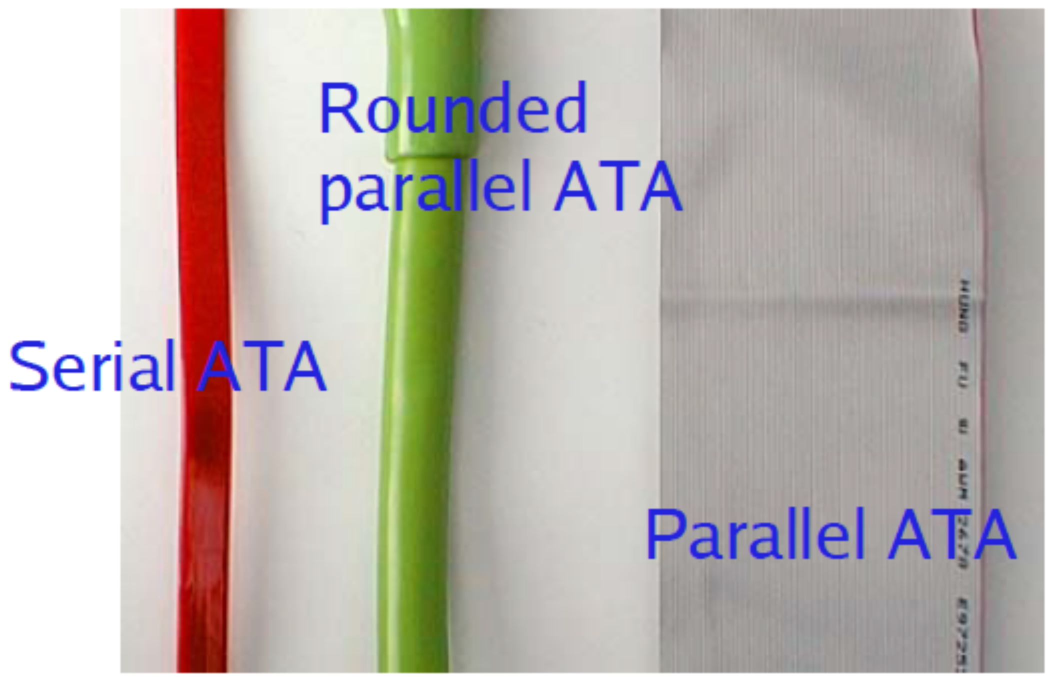
- Command overhead:
  - Time to issue I/O, get the HDD to start responding, select appropriate head
- Seek time:
  - Time to move disk arm to the appropriate track
  - Depends on how fast you can physically move the disk arm
    - These times are not improving rapidly!
- Settle time:
  - Time for head position to stabilize on the selected track
- Rotational latency:
  - Time for the appropriate sector to move under the disk arm
  - Depends on the rotation speed of the disk (e.g., 7200 RPM)
- Transfer time
  - Time to transfer a sector to/from the disk controller
  - Depends on density of bits on disk and RPM of disk rotation

Disks are messy and slow

- Low-level interface for reading and writing sectors
  - Generally allow OS to read/write an entire sector at a time
  - No notion of “files” or “directories” – just raw sectors
  - So, what do you do if you need to write a single byte to a file?
  - Disk may have numerous bad blocks – OS may need to mask this from filesystem
- Access times are still very slow
  - Disk seek times are around 10 ms
    - Although raw throughput has increased dramatically
  - Compare to several nanosec to access main memory
  - Requires careful scheduling of I/O requests

ATA Interfaces

- Serial ATA (SATA): Today's standard for connecting hard drives to the motherboard
  - Using a serial (not parallel) interface
    - Earlier versions used a parallel interface (PATA)
  - Speeds starting at 1.5 Gbit/sec (SATA 1.0)
    - SATA 2.0 (3.0 Gbit/sec), SATA 3.0 (6.0 Gbit/sec)
- Can drive longer cables at much higher clock speeds than parallel cable



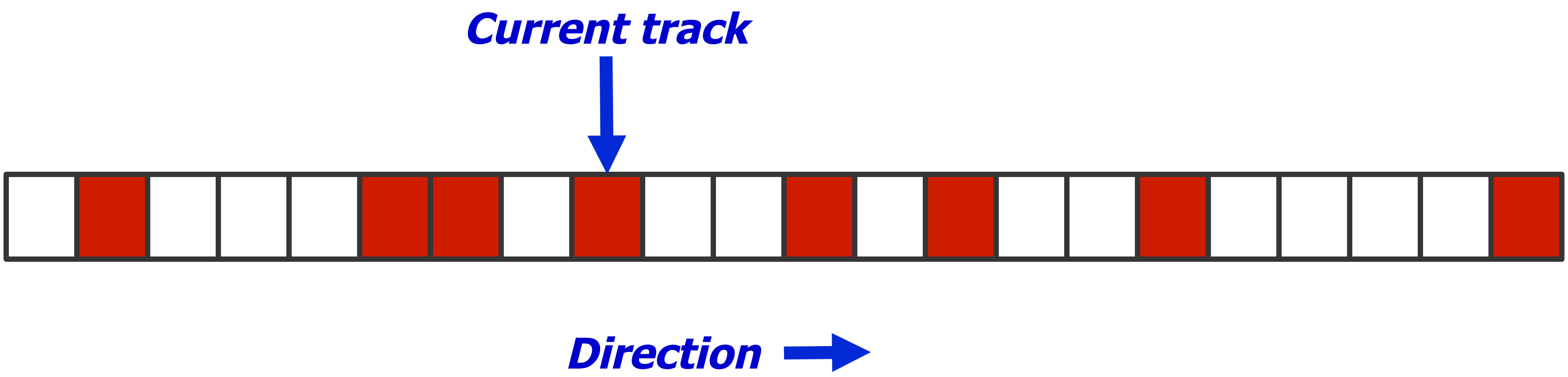
Disk I/O Scheduling

- Given multiple outstanding I/O requests, what order to issue them?
- Why does it matter?
- Major goals of disk scheduling:
- 1) Minimize latency for small transfers
  - Primarily: Avoid long seeks by ordering accesses according to disk head locality
- 2) Maximize throughput for large transfers
  - Large databases and scientific workloads often involve enormous files and datasets
- Note that disk block layout also has a large impact on performance
  - Where we place file blocks, directories, file system metadata, etc.
  - This will be covered in future lectures

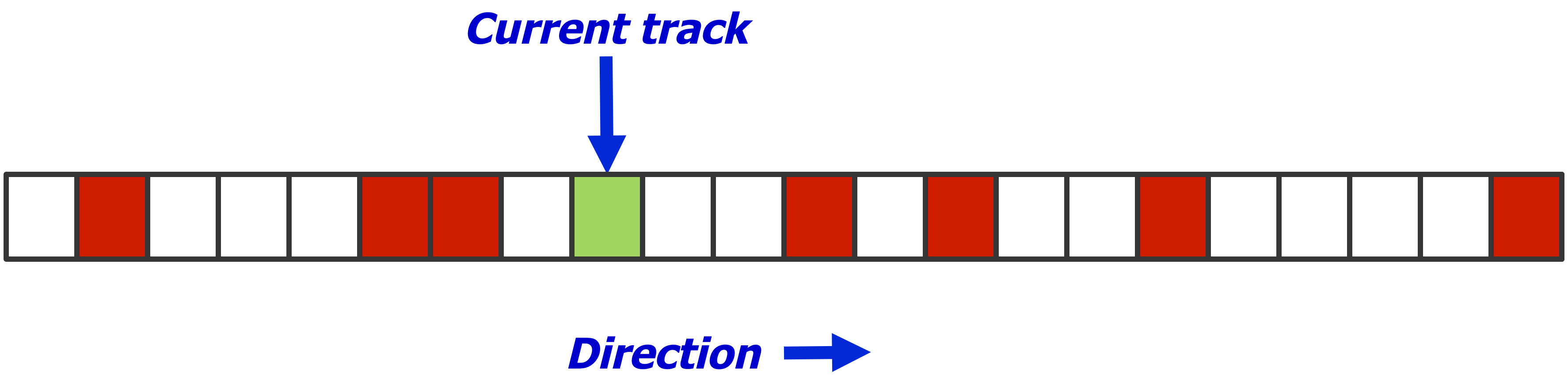
Disk I/O Scheduling

- Given multiple outstanding I/O requests, what order to issue them?
- FIFO: Just schedule each I/O in the order it arrives
  - What's wrong with this? Potentially lots of seek time!
- SSTF: Shortest seek time first
  - Issue I/O with the nearest cylinder to the current one
  - Favors middle tracks: Head rarely moves to edges of disk
- SCAN (or Elevator) Algorithm:
  - Head has a current direction and current cylinder
  - Sort I/Os according to the track # in the current direction of the head
  - If no more I/Os in the current direction, reverse direction
- CSCAN Algorithm:
  - Always move in one direction, "wrap around" to beginning of disk when moving off the end
  - Idea: Reduce variance in seek times, avoid discriminating against the highest and lowest tracks

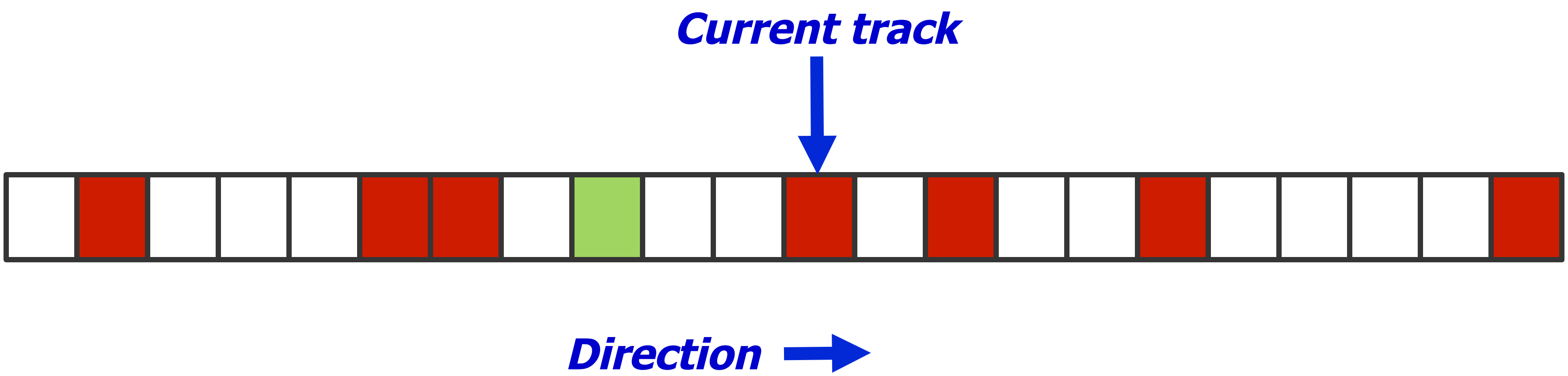
SCAN example



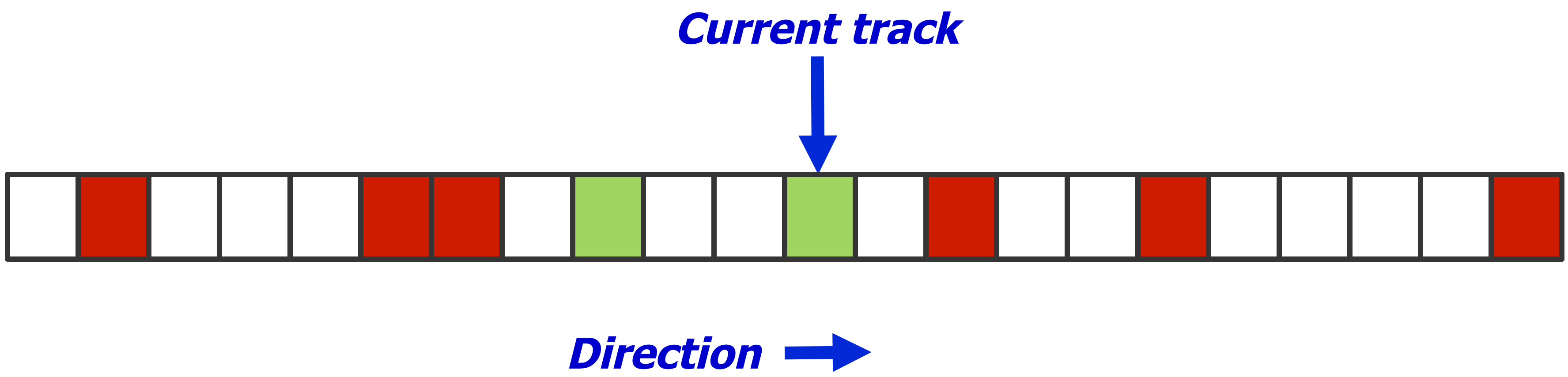
SCAN example



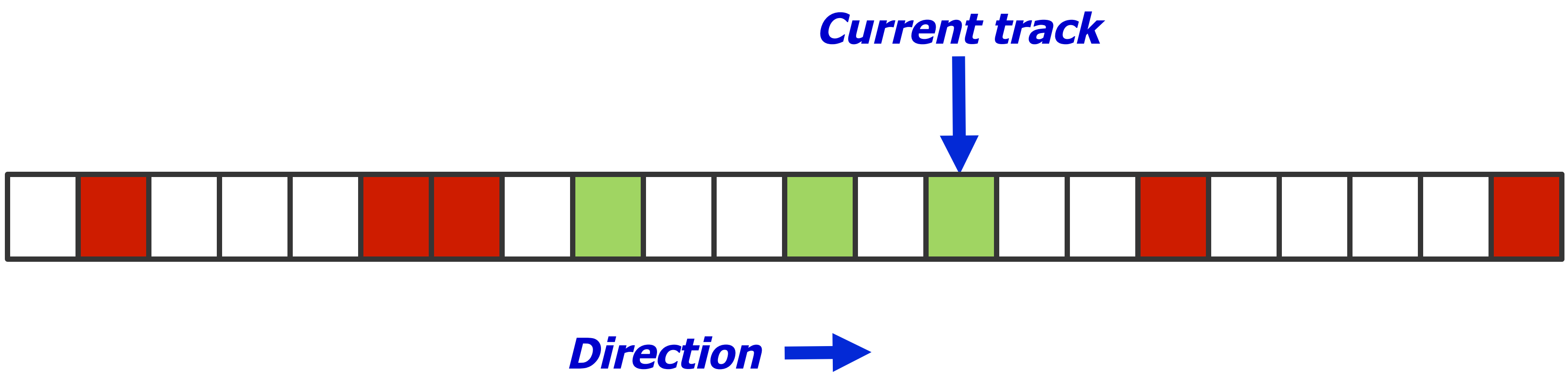
SCAN example



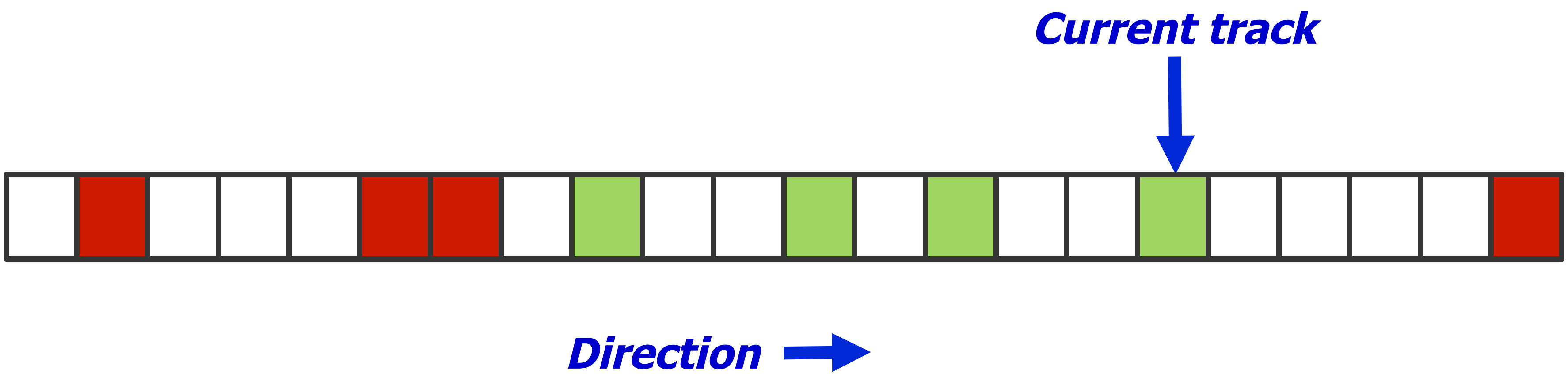
SCAN example



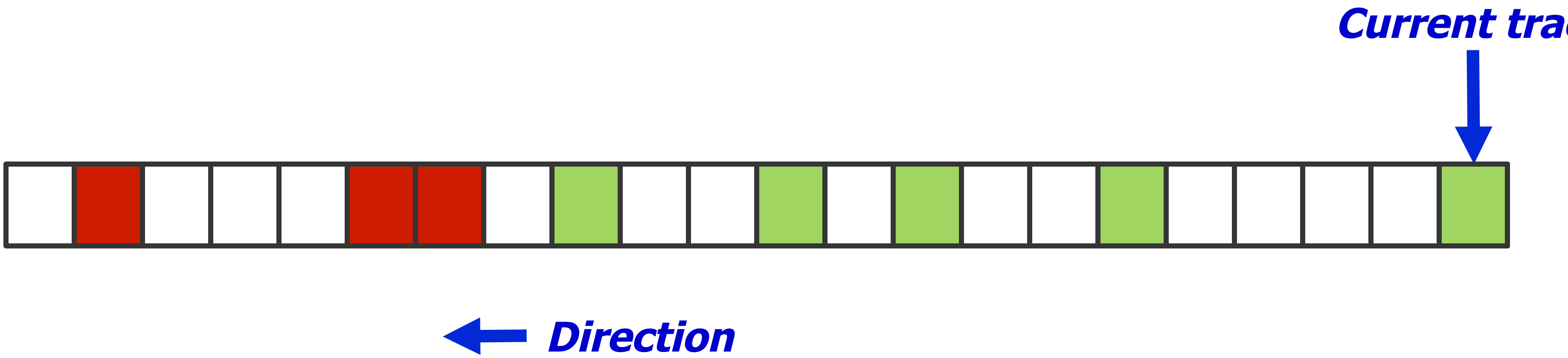
SCAN example



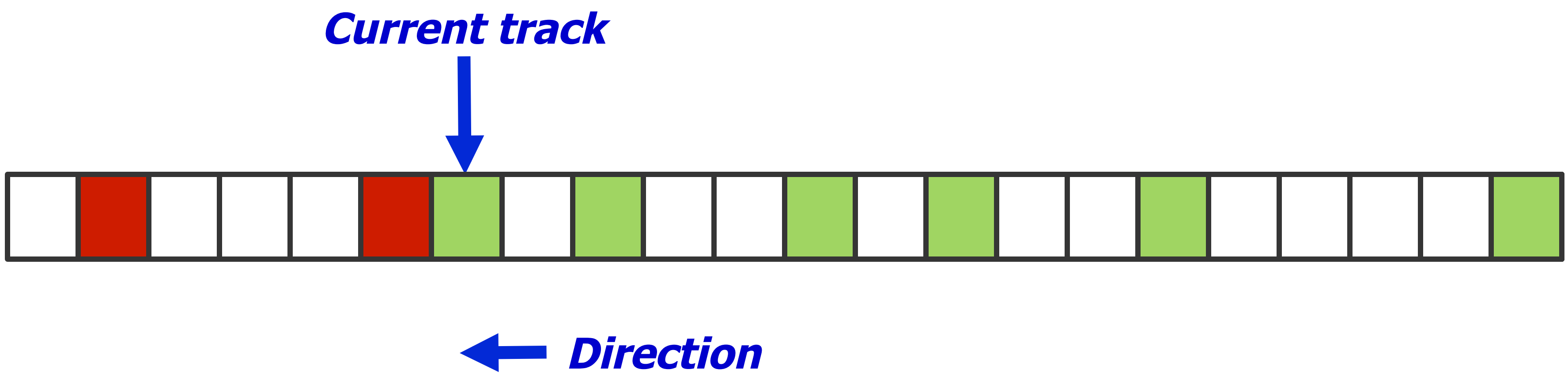
SCAN example



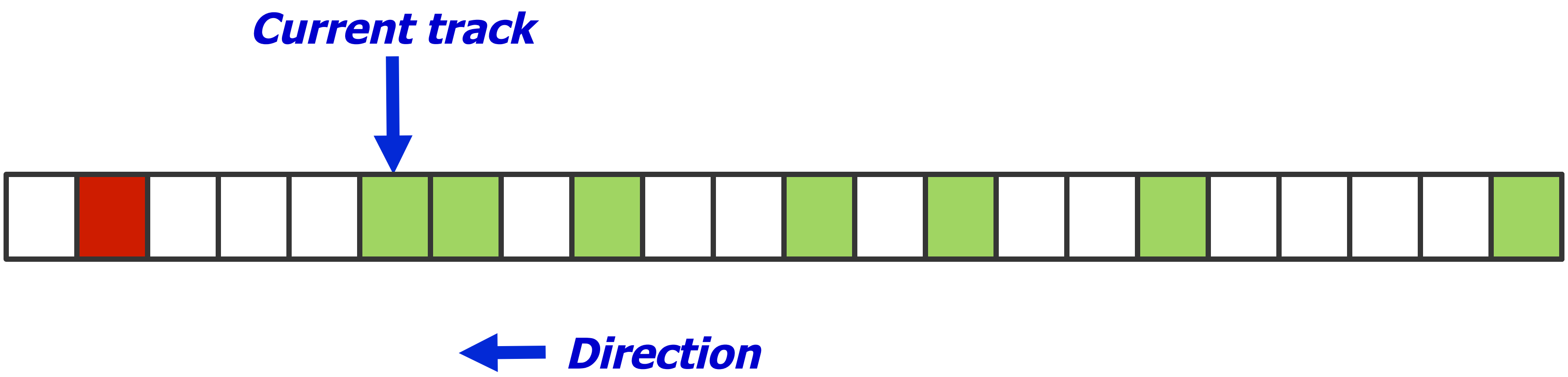
SCAN example



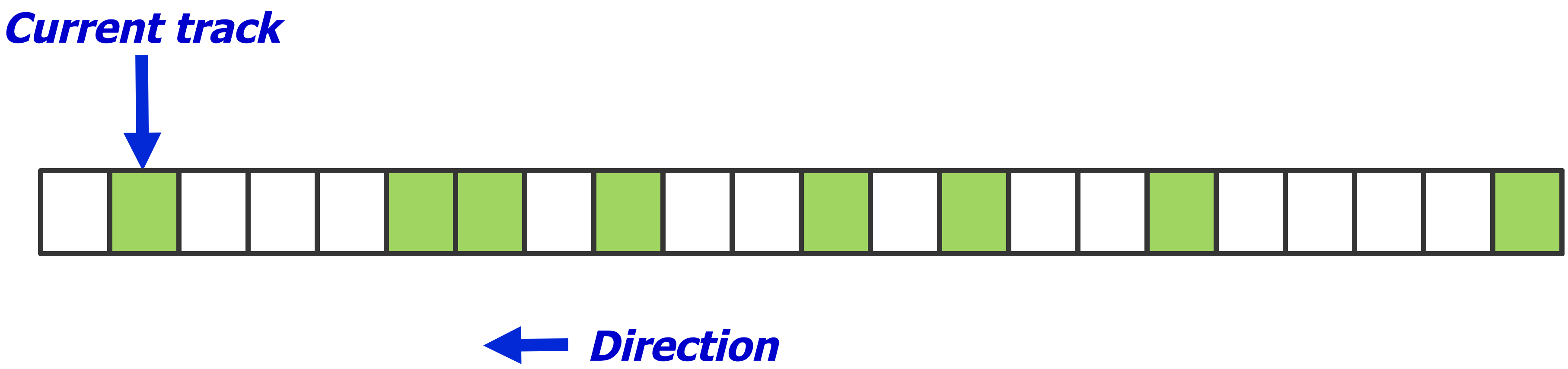
SCAN example



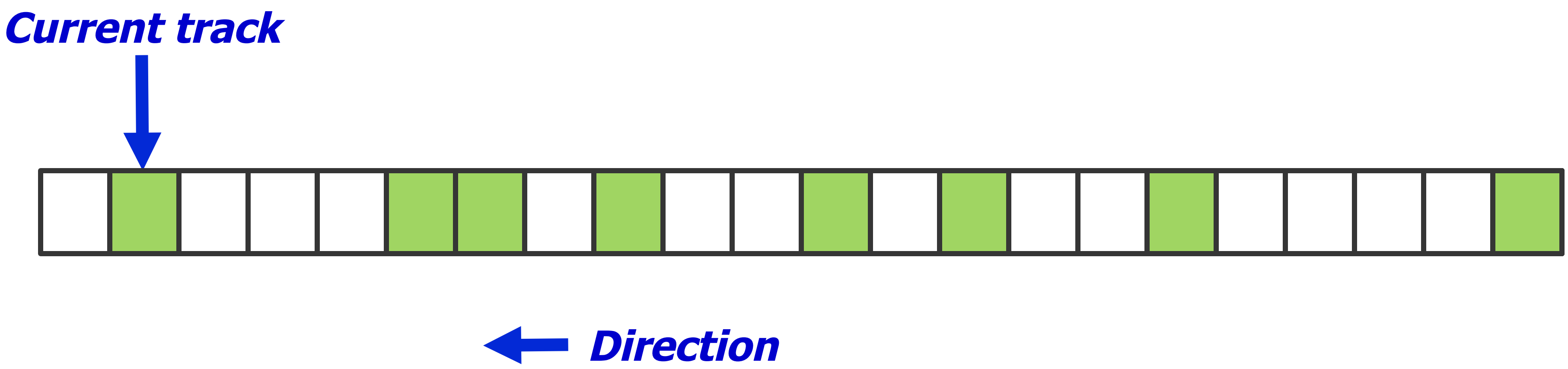
SCAN example



SCAN example



SCAN example



- What is the overhead of the SCAN algorithm?
  - Count the total amount of seek time to service all I/O requests
    - I.e., count total number of track changes
  - In this case, 12 tracks in --> direction
  - 15 tracks for long seek back
  - 5 tracks in <-- direction
    - Total: 12+15+5 = 32 tracks

What about flash?

- Non-volatile, solid state storage
  - No moving parts!
  - Fast access times (about 0.1 msec)
  - Can read and write individual bytes at a time
- Limitations
  - Block erasure: However, must erase a whole "block" before writing to it
  - Read disturb: Reads can cause cells near the read cell to change
    - Solution: Periodically re-write blocks
  - Limited number of erase/write cycles
    - Most flash on the market today can withstand up to 1 million erase/write cycles
    - Flash Translation Layer (FTL): writes to a different cell each time to wear-level device, cache to avoid excessive writes
- How does this affect how we design filesystems???

Part 2: I/O

Input and Output

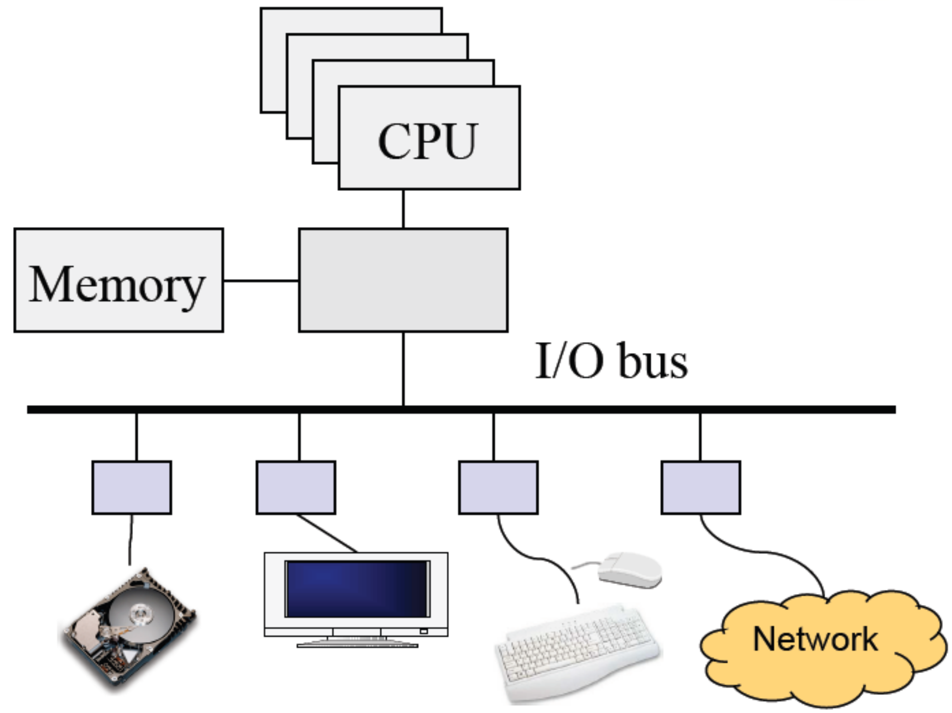
- A computer's job is to process data
  - Computation (CPU, cache, and memory)
  - Move data into and out of a system (between I/O devices and memory)
- Challenges with I/O devices
  - Different categories: storage, networking, displays, etc.
  - Large number of device drivers to support
  - Device drivers run in kernel mode and can crash systems
- Goals of the OS
  - Provide a generic, consistent, convenient and reliable way to
  - access I/O devices
  - As device-independent as possible
  - Don't hurt the performance capability of the I/O system too much

How does the CPU talk to devices?

- Device controller: Circuit that enables devices to talk to the peripheral bus
- Host adapter: Circuit that enables the computer to talk to the peripheral bus
- Bus: Wires that transfer data between components inside computer
- Device controller allows OS to specify simpler instructions to access data
- Example: a disk controller
  - Translates "access sector 23" to "move head reader 1.672725272 cm from edge of platter"
  - Disk controller "advertises" disk parameters to OS, hides internal disk geometry
  - Most modern hard drives have disk controller embedded as a chip on the physical device

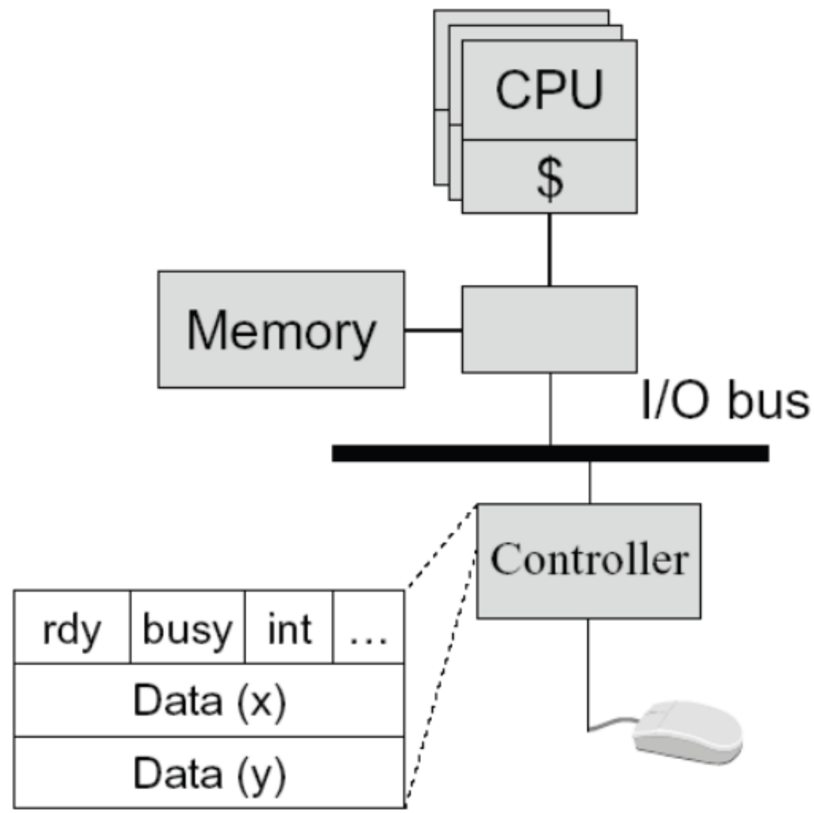
Review: Computer Architecture

- Compute hardware
  - CPU and caches
  - Chipset
  - Memory
- I/O Hardware
  - I/O bus or interconnect
  - I/O controller or adaptor
  - I/O device
- Two types of I/O
  - Programmed I/O (PIO)
    - CPU does the work of moving data
  - Direct Memory Access (DMA)
    - CPU offloads the work of moving data to DMA controller



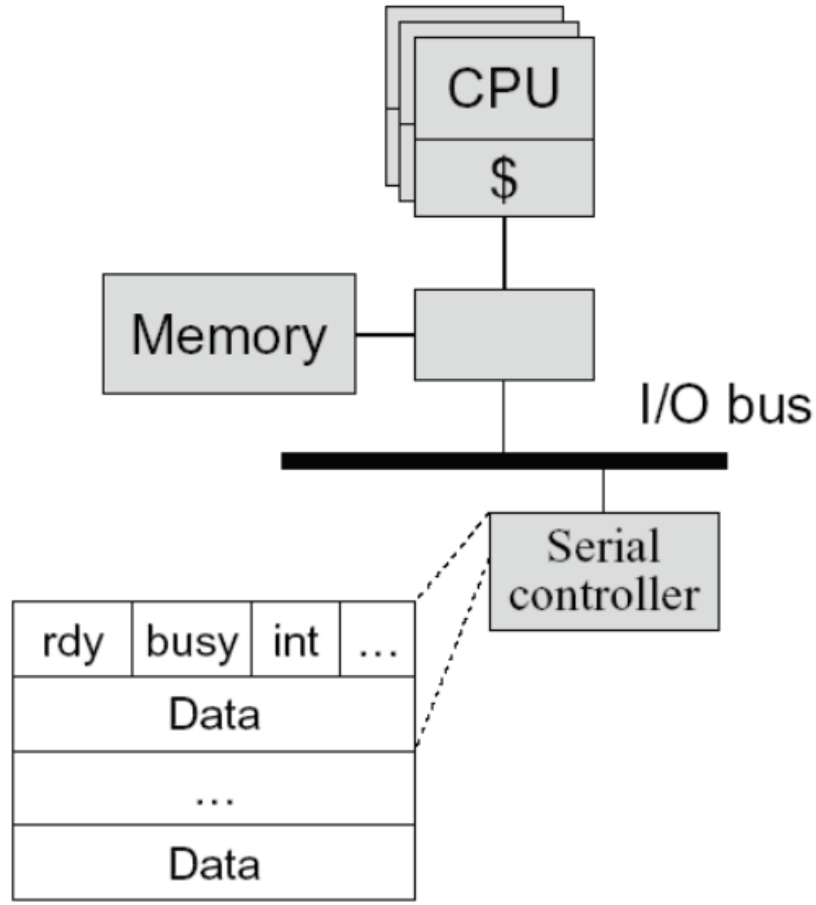
Programmed Input Device

- Device controller
  - Status register
    - ready: tells if the host is done
    - busy: tells if the controller is done
    - int: interrupt
    - …
  - Data registers
- A simple mouse design
  - When moved, put (X, Y) in mouse's device controller's data registers
  - Interrupt CPU
- Input on an interrupt
  - CPU saves state of currently-executing program
  - Reads values in X, Y registers
  - Sets ready bit
  - Wakes up a process/thread or execute a piece of code to handle interrupt



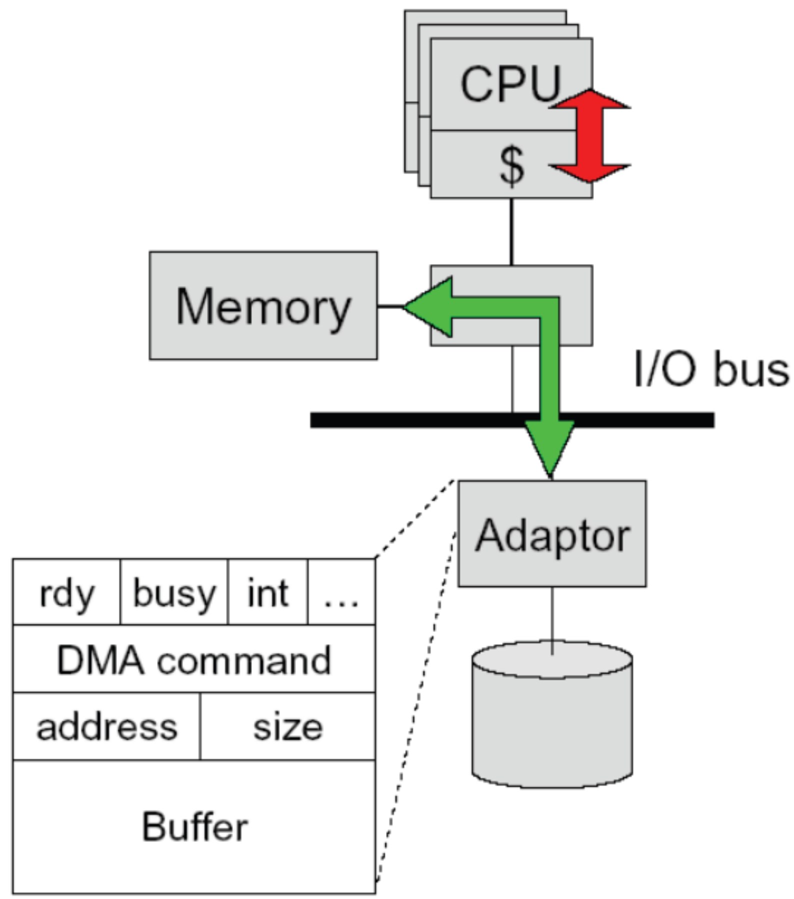
Programmed Output Device

- Device
  - Status registers (ready, busy, …)
  - Data registers
- Example
  - A serial output device
- Perform an output
  - CPU: Poll the busy bit
  - Writes the data to data register(s)
  - Set ready bit
  - Controller sets busy bit and transfers data
  - Controller clears the busy bit



Direct Memory Access (DMA)

- DMA controller or adaptor
  - Status register (ready, busy, interrupt, …)
  - DMA command register
  - DMA register (address, size)
  - DMA buffer
- Host CPU initiates DMA
  - Device driver call (kernel mode)
  - Wait until DMA device is free
  - Initiate a DMA transaction
  - (command, memory address, size)
  - Block
- Controller performs DMA
  - DMA data to device (size--; address++)
  - Interrupt on completion (size == 0)
- Interrupt handler (on completion)
  - Wakeup the blocked process



Memory-mapped I/O

- Use the same address bus to address both memory and I/O devices
  - The memory and registers of I/O devices are mapped to address values
  - Allows same CPU instructions to be used with regular memory and devices
- I/O devices, memory controller, monitor address bus
  - Each responds to addresses they own
- Orthogonal to DMA
  - May be used with, or without, DMA

Polling- vs. Interrupt-driven I/O

- Polling
  - CPU issues I/O command
  - CPU directly writes instructions into device's registers
  - CPU busy waits for completion
- Interrupt-driven I/O
  - CPU issues I/O command
  - CPU directly writes instructions into device's registers
  - CPU continues operation until interrupt
- Direct Memory Access (DMA)
  - Typically done with Interrupt-driven I/O
  - CPU asks DMA controller to perform device-to-memory transfer
  - DMA issues I/O command and transfers new item into memory
  - CPU module is interrupted after completion
- Which is better, polling or interrupt-driven I/O?

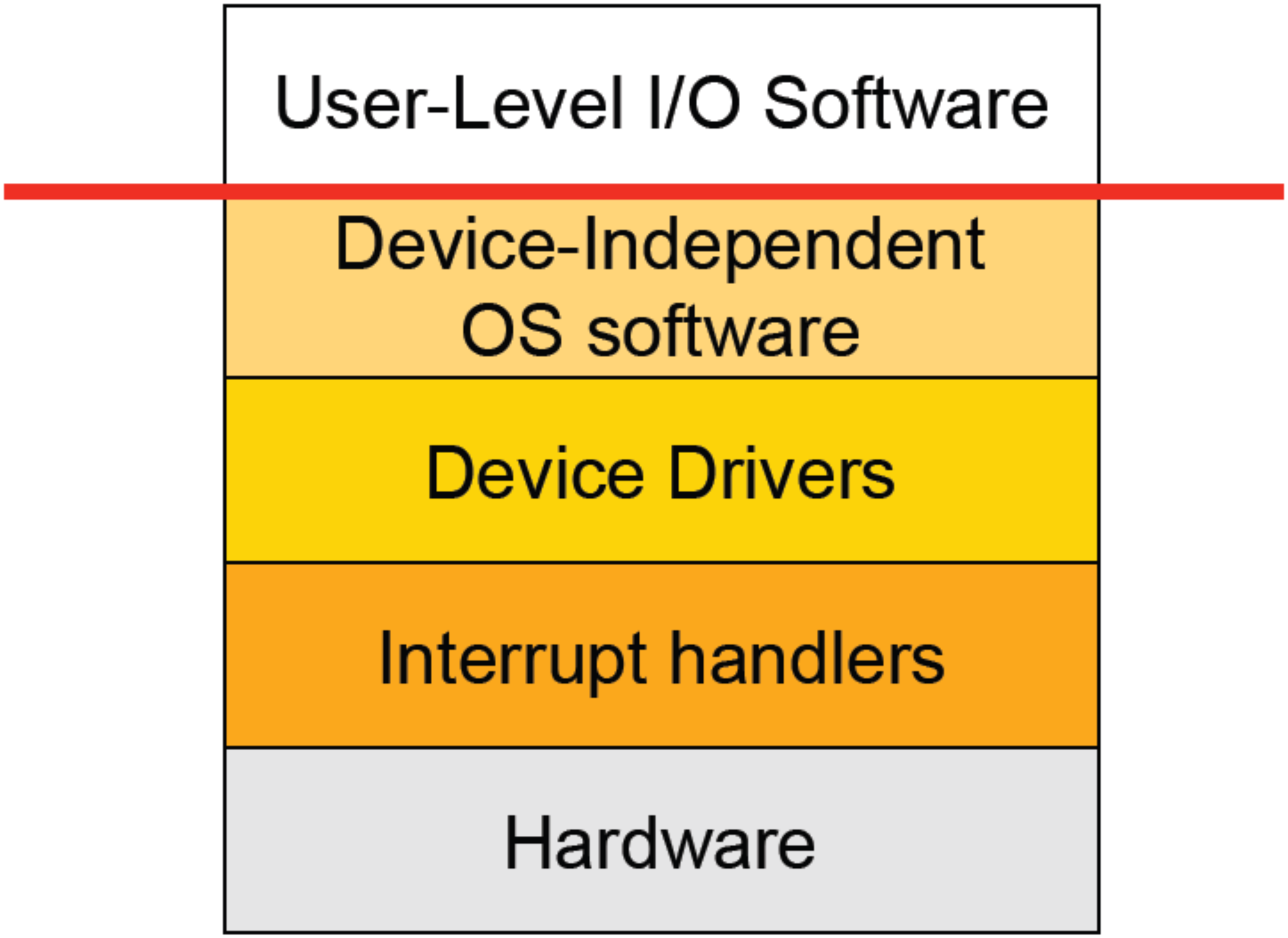
Polling- vs. Interrupt-driven I/O

- Polling
  - Expensive for large transfers
  - Better for small, dedicated systems with infrequent I/O
- Interrupt-driven
  - Overcomes CPU busy waiting
  - I/O module interrupts when ready: event driven

How Interrupts are implemented

- CPU hardware has an interrupt report line that the CPU tests after executing every instruction
  - If a(ny) device raises an interrupt by setting interrupt report line
    - CPU catches the interrupt and saves the state of current running process into PCB
    - CPU dispatches/starts the interrupt handler
    - Interrupt handler determines cause, services the device and clears the interrupt report line
- Other uses of interrupts: exceptions
  - Division by zero, wrong address
  - System calls (software interrupts/signals, trap)
  - Virtual memory paging

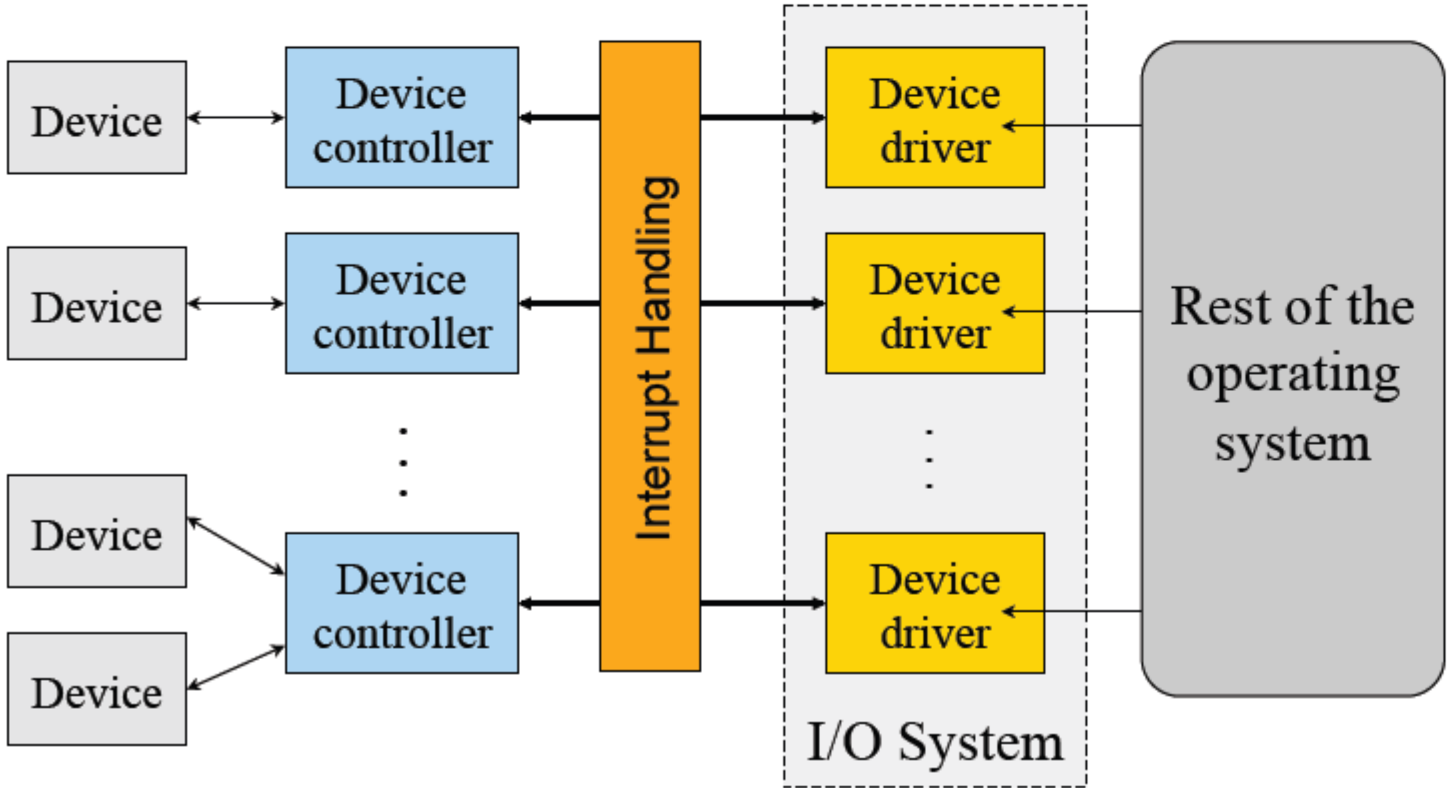
I/O Software Stack



Interrupt Handling

- Save context (registers that hw hasn't saved, PSW etc)
- Mask interrupts if needed
- Set up a context for interrupt service
- Set up a stack for interrupt service
- Acknowledge interrupt controller, perhaps enable it
- Save entire context to PCB
- **Run the interrupt service**
- Unmask interrupts if needed
- Possibly change the priority of the process
- Run the scheduler
- Then OS will set up context for next process, load registers and PSW, start running process …

Device Drivers



- Manage the complexity and differences among specific types of devices (disk vs. mouse, different types of disks …)
- Each handles one type of device or small class of them (eg SCSI)

Typical Device Driver Design

- Operating system and driver communication
  - Commands and data between OS and device drivers
- Driver and hardware communication
  - Commands and data between driver and hardware
- Driver responsibilities
  - Initialize devices
  - Interpreting commands from OS
  - Schedule multiple outstanding requests
  - Manage data transfers
  - Accept and process interrupts
  - Maintain the integrity of driver and kernel data structures

Device Driver Behavior

- Check input parameters for validity, and translate them to device specific language
- Check if device is free (wait or block if not)
- Issue commands to control device
  - Write them into device controller's registers
  - Check after each if device is ready for next (wait or block if not)
- Block or wait for controller to finish work
- Check for errors, and pass data to device-independent software
- Return status information
- Process next queued request, or block waiting for next
- Challenges:
  - Must be reentrant (can be called by an interrupt while running)
  - Handle hot-pluggable devices and device removal while running
  - Complex and many of them; bugs in them can crash system

Types of I/O Devices

- Block devices
  - Organize data in fixed-size blocks
  - Transfers are in units of blocks
  - Blocks have addresses and data are therefore addressable
  - E.g. hard disks, USB disks, CD-ROMs
- Character devices
  - Delivers or accepts a stream of characters, no block structure
  - Not addressable, no seeks
  - Can read from stream or write to stream
  - Printers, network interfaces, terminals
- Like everything, not a perfect classification
  - E.g. tape drives have blocks but not randomly accessed
  - Clocks are I/O devices that just generate interrupts

Char/Block Device Interfaces

- Character device interface
  - read(deviceNumber, bufferAddr, size)
    - Reads "size" bytes from a byte stream device to "bufferAddr"
  - write(deviceNumber, bufferAddr, size)
    - Write "size" bytes from "bufferAddr" to a byte stream device
- Block device interface
  - read(deviceNumber, deviceAddr, bufferAddr)
    - Transfer a block of data from "deviceAddr" to "bufferAddr"
  - write(deviceNumber, deviceAddr, bufferAddr)
    - Transfer a block of data from "bufferAddr" to "deviceAddr"
  - seek(deviceNumber, deviceAddress)
    - Move the head to the correct position
    - Usually not necessary

Sync vs Asynchronous I/O

- Synchronous I/O
  - read() or write() will block a user process until its completion
  - OS overlaps synchronous I/O with another process
- Asynchronous I/O
  - read() or write() will not block a user process
  - user process can do other things before I/O completion
  - I/O completion will notify the user process

Example: Blocked Read

- A process issues a read call which executes a system call
- System call code checks for correctness
- If it needs to perform I/O, it will issues a device driver call
- Device driver allocates a buffer for read and schedules I/O
- Controller performs DMA data transfer
- Block the current process and schedule a ready process
- Device generates an interrupt on completion
- Interrupt handler stores any data and notifies completion
- Move data from kernel buffer to user buffer
- Wakeup blocked process (make it ready)
- User process continues when it is scheduled to run