CS 318 Principles of Operating Systems

Fall 2018

Lecture 14: I/O & Disks

Ryan Huang

JOHNS HOPKINS
WHITING SCHOOL of ENGINEERING

Slides adapted from David Mazières' lecture

Administrivia

- **Lab 3 is out, please start early**
  - workload increasing
  - absolute late penalty increasing
  - due 11/11 (Sunday) 11:59 pm
  - suggest checking design with the staff first
- **Lab 3 overview session Friday 5-6 pm G33/35**

Overview

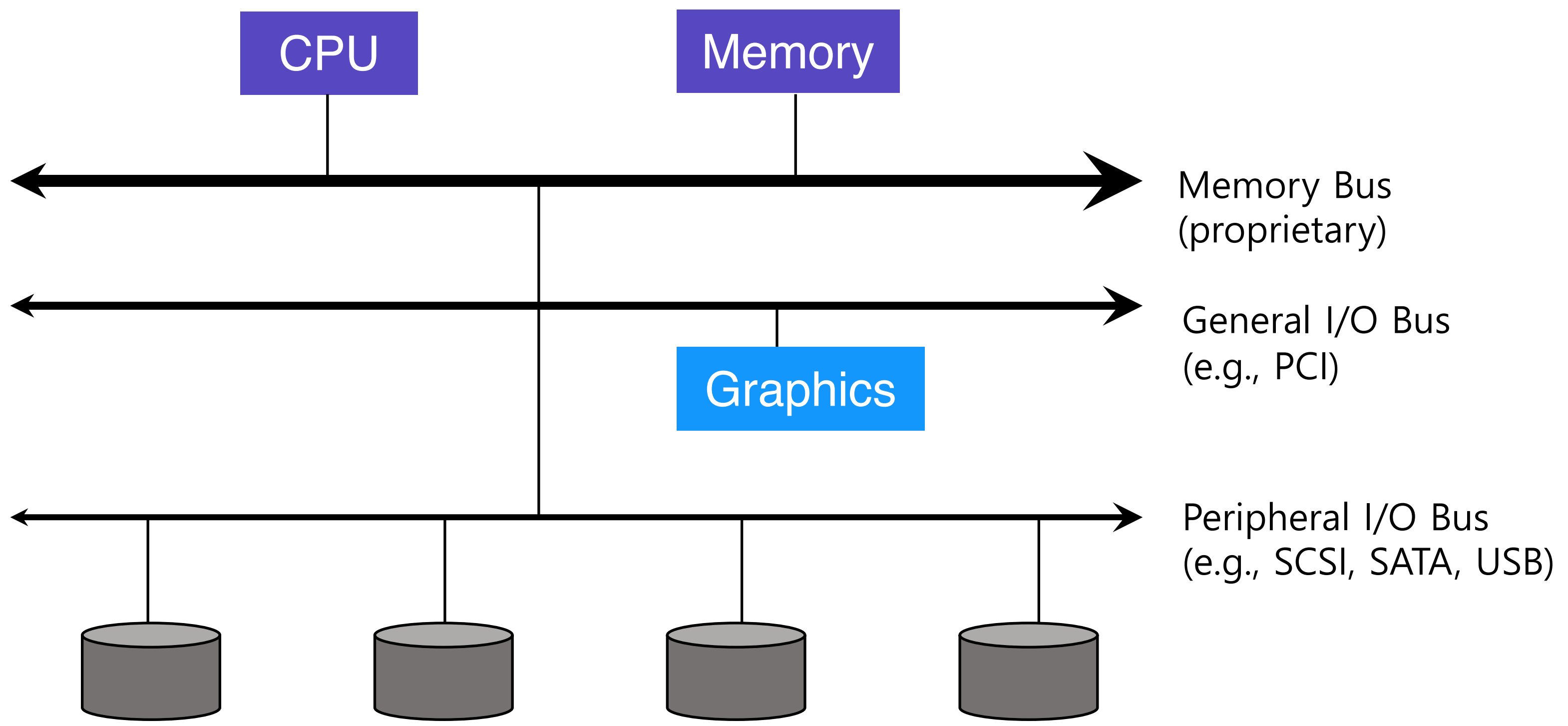
- **We've covered OS abstractions for CPU and memory so far**
- **I/O management is another major component of OS**
  - Important aspect of computer operation
  - I/O devices vary greatly
  - Various methods to control them
  - Performance management
  - New types of devices frequent
- **Ports, busses, device controllers connect to various devices**

I/O Devices

…

- **Issues to address:**
  - How should I/O be integrated into systems?
  - What are the general mechanisms?
  - How can we make the efficiently?

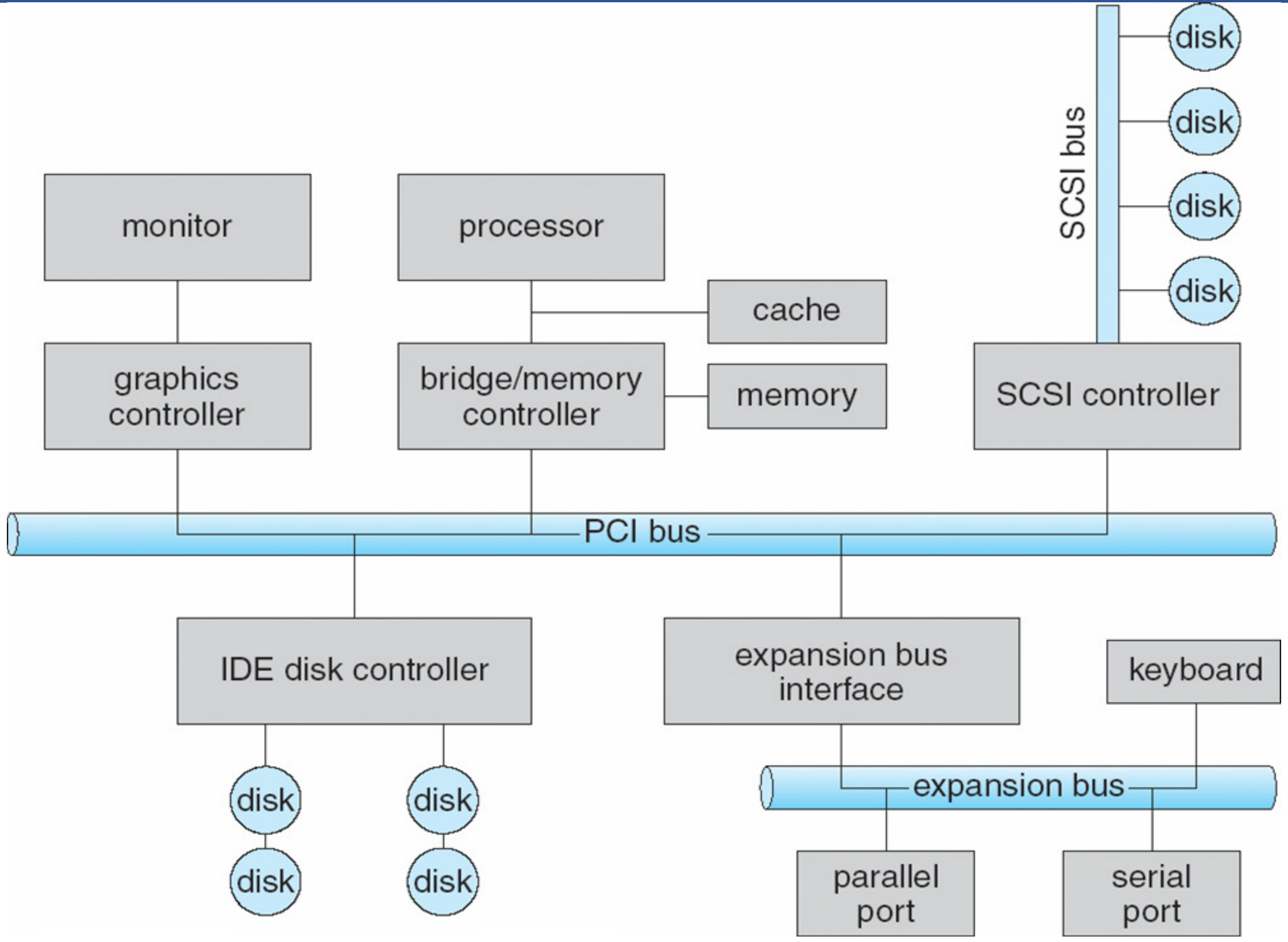
Structure of Input/Output (I/O) Device



I/O Device Interfaces

- **Port – connection point for device**
  - serial port
- **Bus – daisy chain or shared direct access**
  - PCI bus common in PCs and servers, PCI Express (PCIe)
  - expansion bus connects relatively slow devices
- **Controller (host adapter) – electronics that operate port, bus, device**
  - Sometimes integrated
  - Sometimes separate circuit board (host adapter)
  - Contains processor, microcode, private memory, bus controller, etc
  - Some talk to per-device controller with bus controller, microcode, memory, etc

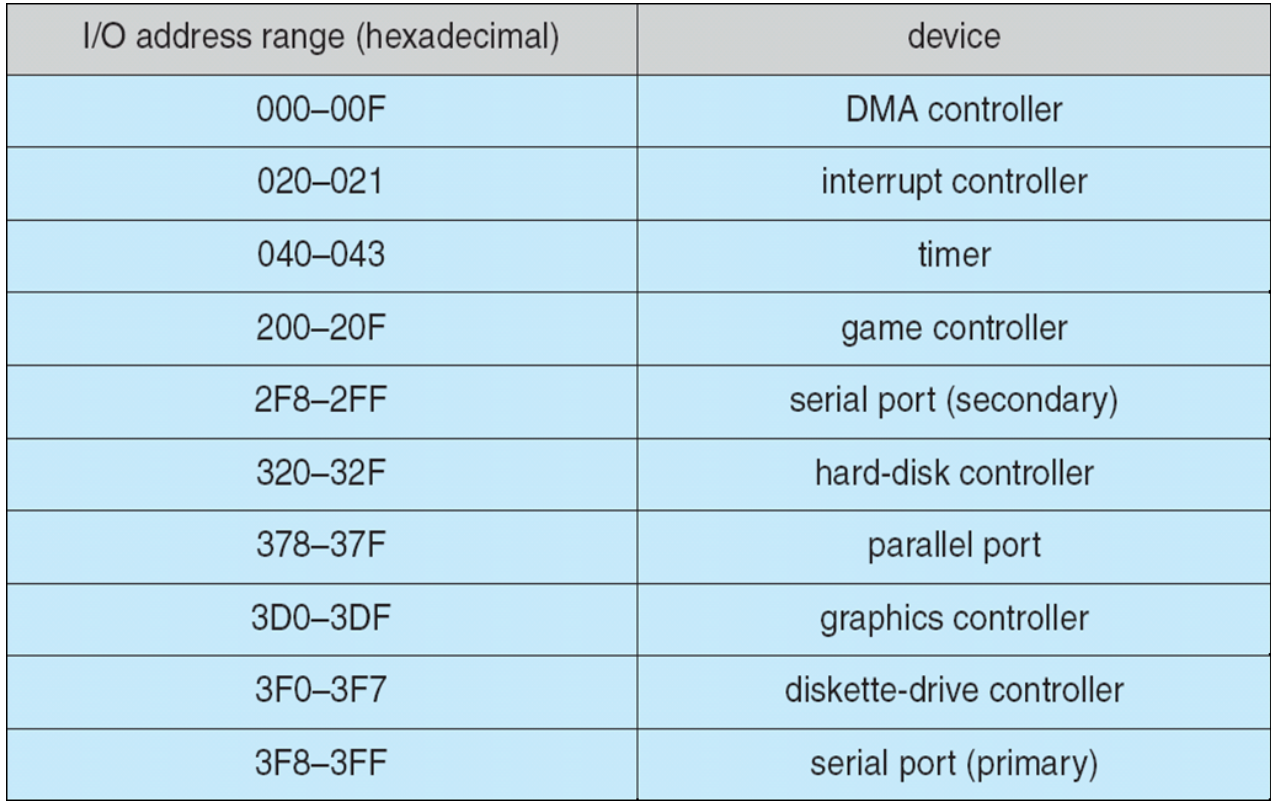
What Is I/O Bus? E.g., PCI



Device Interaction

- **How the OS communicates with the device?**
- **I/O instructions control devices**
  - `in` and `out` instructions on x86
  - Devices usually have registers
    - device driver places commands, addresses, and data there to read/write
- **Memory-mapped I/O**
  - Device registers available as if they were memory locations.
  - OS `load` (to read) or `store` (to write) to the device instead of main memory.

Device I/O Port Locations on PCs



I/O address range (hexadecimal)	device
000–00F	DMA controller
020–021	interrupt controller
040–043	timer
200–20F	game controller
2F8–2FF	serial port (secondary)
320–32F	hard-disk controller
378–37F	parallel port
3D0–3DF	graphics controller
3F0–3F7	diskette-drive controller
3F8–3FF	serial port (primary)

x86 I/O instructions

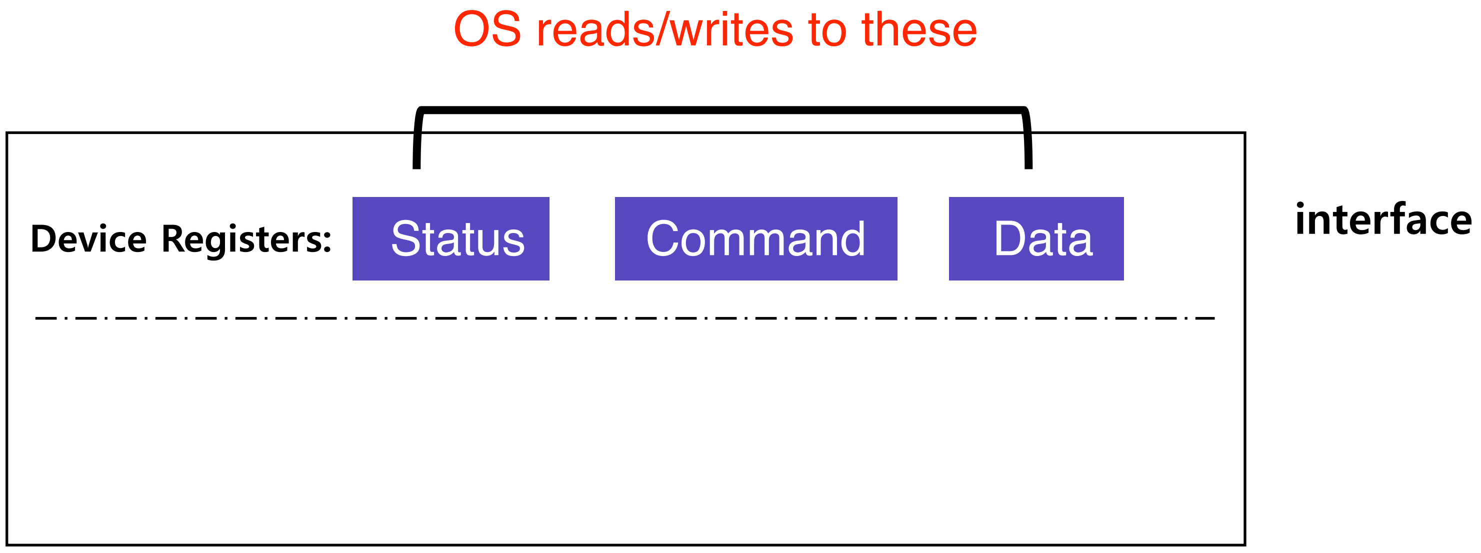
```
static inline uint8_t inb (uint16_t port)
{
  uint8_t data;
  asm volatile ("inb %w1, %b0" : "=a" (data) : "Nd" (port));
  return data;
}


static inline void outb (uint16_t port, uint8_t data)
{
  asm volatile ("outb %b0, %w1" : : "a" (data), "Nd" (port));
}


static inline void insw (uint16_t port, void *addr, size_t cnt)
{
  asm volatile ("rep insw" : "+D" (addr), "+c" (cnt)
                : "d" (port) : "memory");
}
```

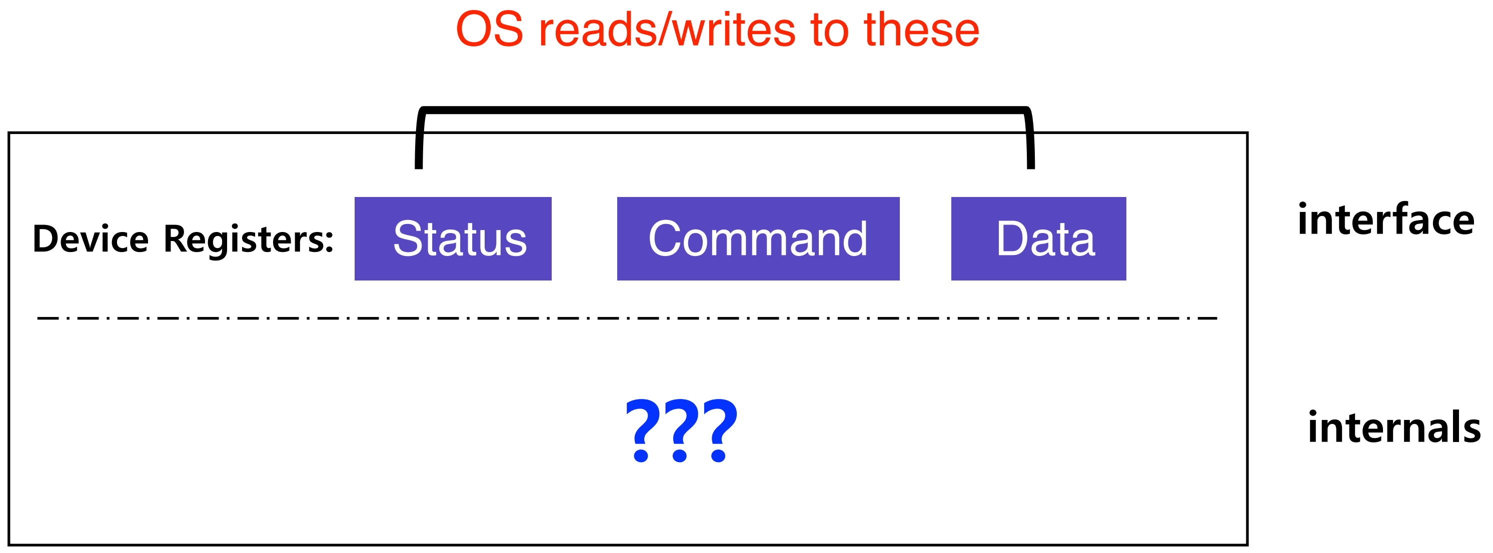
Pintos `threads/io.h`

Canonical I/O Device



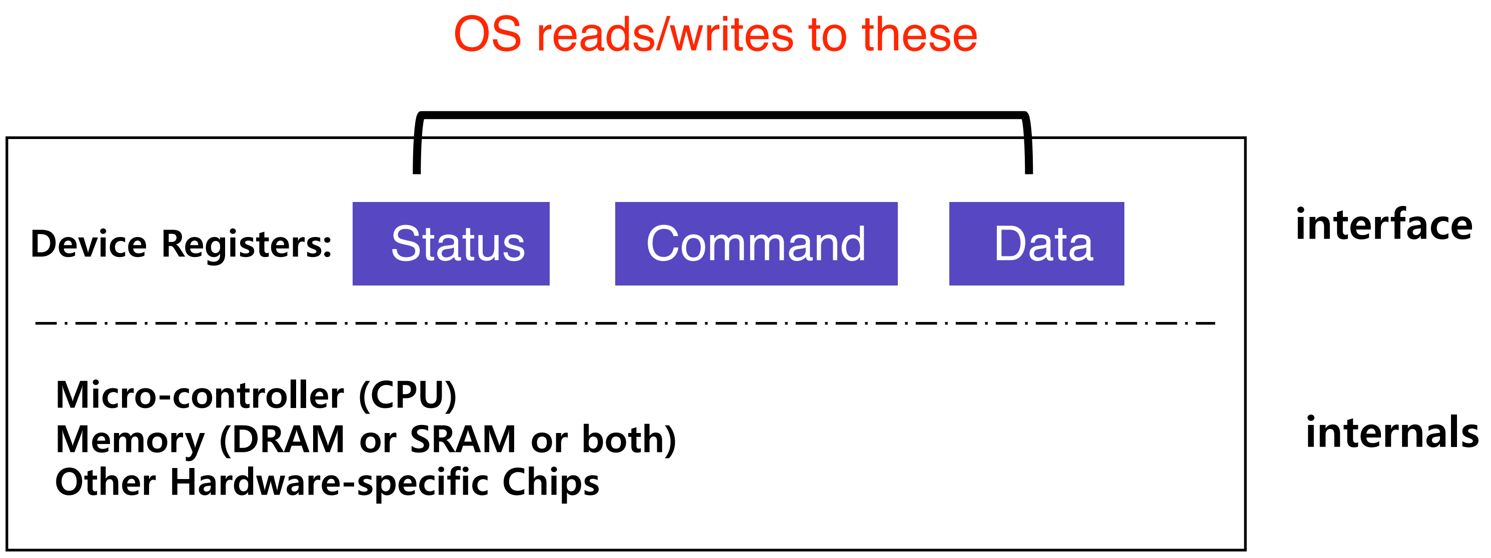
Canonical Device

Canonical I/O Device



Canonical Device

Canonical I/O Device



Canonical Device

Hardware Interface Of Canonical Device

- **status register**
  - See the current status of the device
- **command register**
  - Tell the device to perform a certain task
- **data register**
  - Pass data to the device, or get data from the device
- **By reading or writing the above three registers, the OS controls device behavior.**

Hardware Interface Of Canonical Device

- **Typical interaction example**

```
while (STATUS == BUSY)
    ; //wait until device is not busy
write data to data register
write command to command register
    Doing so starts the device and executes the command
while (STATUS == BUSY)
    ; //wait until device is done with your request
```

IDE Disk Driver

```
void IDE_ReadSector(int disk, int off,
      void *buf)
{
  // Select Drive
  outb(0x1F6, disk == 0 ? 0xE0 : 0xF0);
  IDEWait();
  // Read length (1 sector = 512 B)
  outb(0x1F2, 1);
  outb(0x1F3, off); // LBA low
  outb(0x1F4, off >> 8); // LBA mid
  outb(0x1F5, off >> 16); // LBA high
  outb(0x1F7, 0x20); // Read command
  insw(0x1F0, buf, 256); // Read 256 words
}
```

```
void IDEWait()
{
  // Discard status 4 times
  inb(0x1F7); inb(0x1F7);
  inb(0x1F7); inb(0x1F7);
  // Wait for status BUSY flag to clear
  while ((inb(0x1F7) & 0x80) != 0);
}
```

Memory-mapped IO

- `in/out` **instructions slow and clunky**
  - Instruction format restricts what registers you can use
  - Only allows 2^{16} different port numbers
  - Per-port access control turns out not to be useful (any port access allows you to disable all interrupts)
- **Devices can achieve same effect with physical addresses, e.g.:**

```
volatile int32_t *device_control
   = (int32_t *) (0xc0100 + PHYS_BASE);
*device_control = 0x80;
int32_t status = *device_control;
```

  - OS must map physical to virtual addresses, ensure non-cachable

Polling

- **OS waits until the device is ready by repeatedly reading the status register**
  - Positive aspect is simple and working.
  - **However, it wastes CPU time just waiting for the device**.
    - Switching to another ready process is better utilizing the CPU.

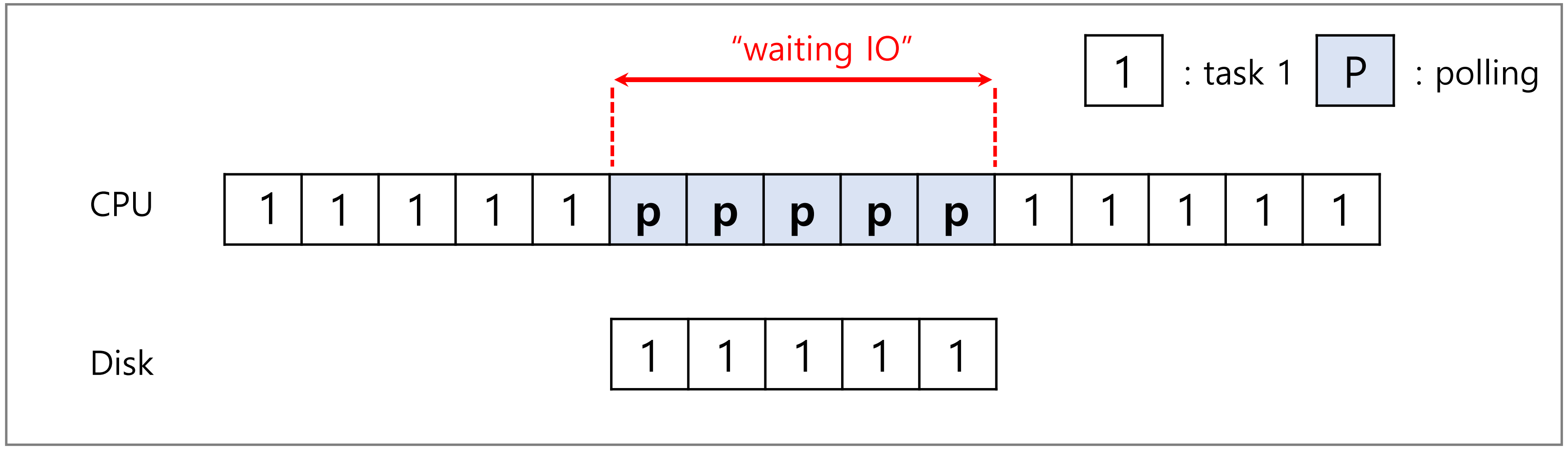


Diagram of CPU utilization by polling

Interrupts

- **Put the I/O request process to sleep and context switch to another**
- **When the device is finished, wake the process waiting for the I/O by interrupt**
  - Positive aspect is allow to **CPU and the disk are properly utilized.**

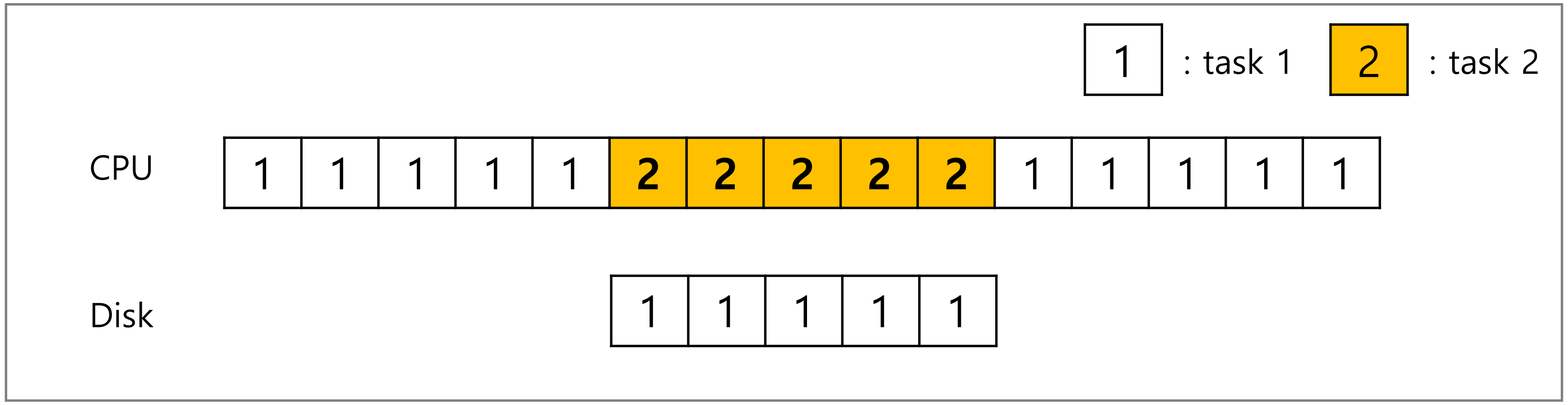


Diagram of CPU utilization by interrupt

Polling vs Interrupts

- ***However,*** **“interrupts is not always the best solution”**
  - If, device performs very quickly, interrupt will “slow down” the system.
- **E.g., high network packet arrival rate**
  - Packets can arrive faster than OS can process them
  - Interrupts are very expensive (context switch)
  - Interrupt handlers have high priority
  - In worst case, can spend 100% of time in interrupt handler and never make any progress – *receive livelock*
  - Best: Adaptive switching between interrupts and polling

If a device is fast → poll is best.
If it is slow → interrupts is better.

One More Problem: Data Copying

- **CPU wastes a lot of time in copying *a large chunk of data* from memory to the device.**

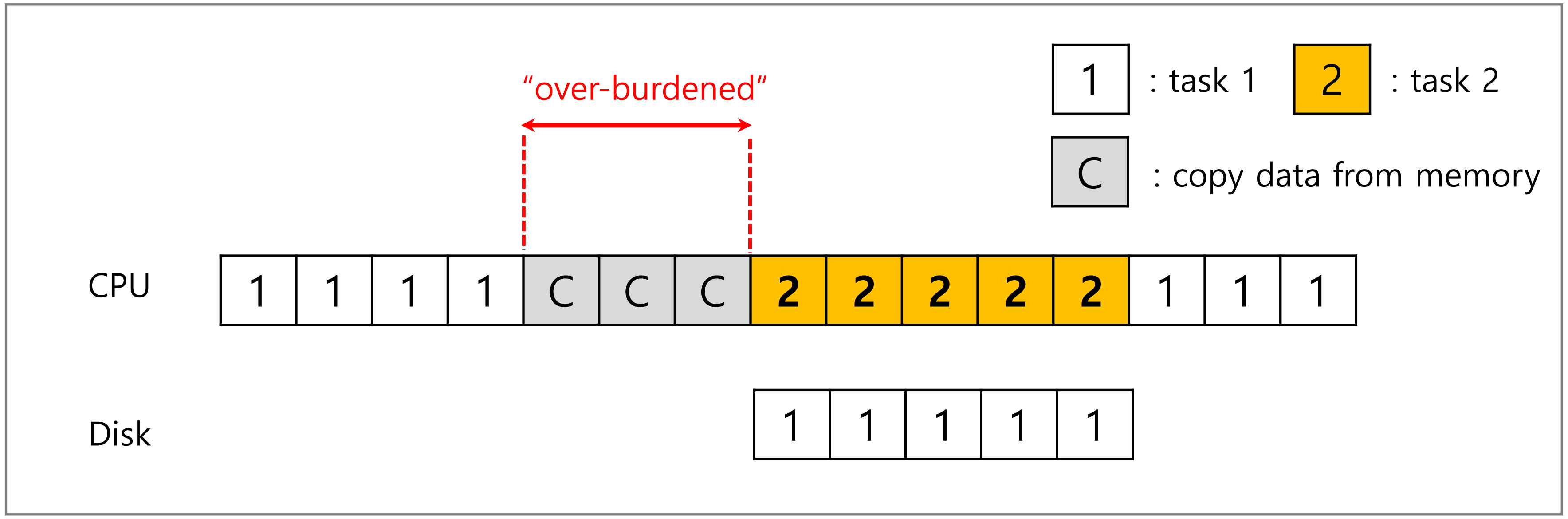
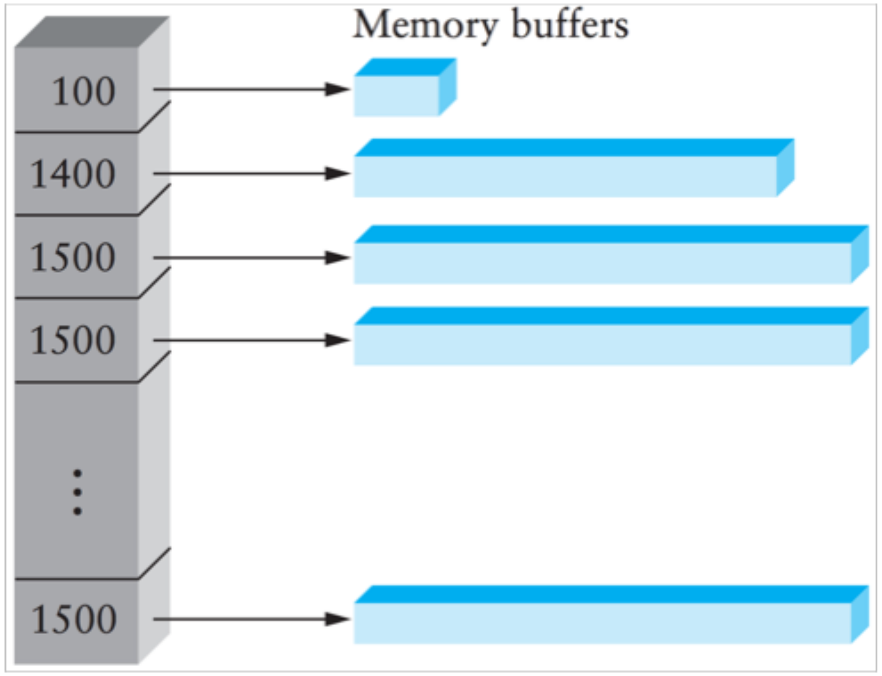


Diagram of CPU utilization

DMA (Direct Memory Access)

Buffer descriptor list



- **Idea: only use CPU to transfer control requests, not data**
- **Include list of buffer locations in main memory**
  - Device reads list and accesses buffers through DMA
  - Descriptions sometimes allow for scatter/gather I/O

DMA (Direct Memory Access) Cont.

- **When completed, DMA raises an interrupt, I/O begins on Disk.**

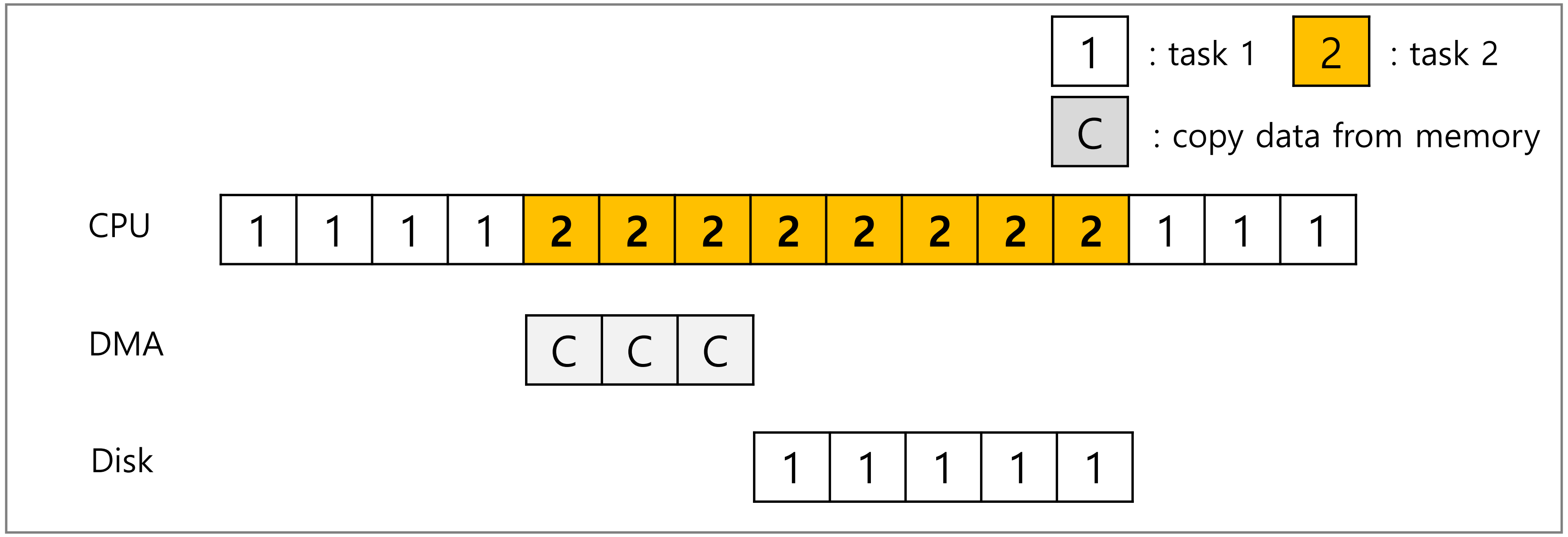
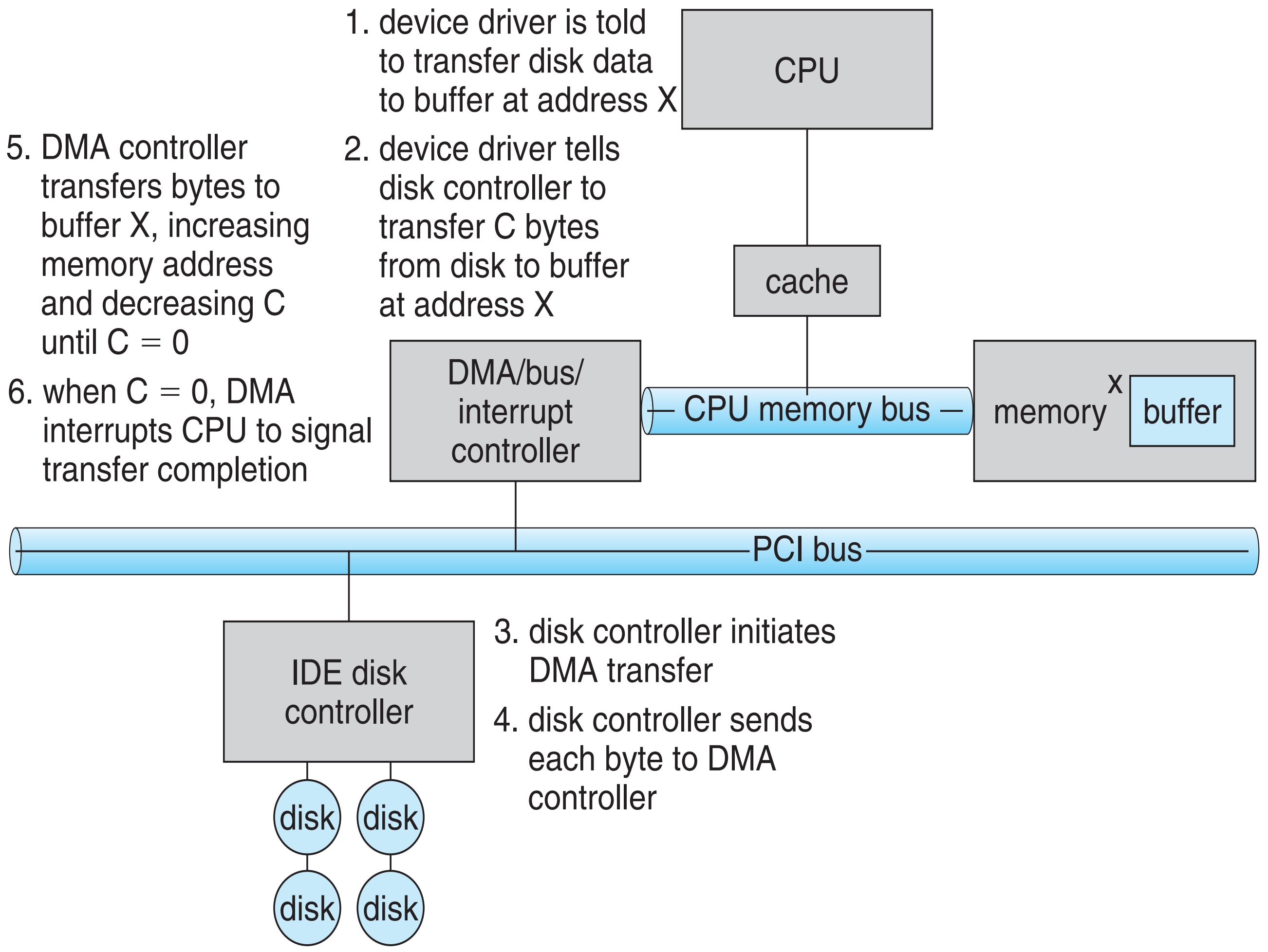


Diagram of CPU utilization by DMA

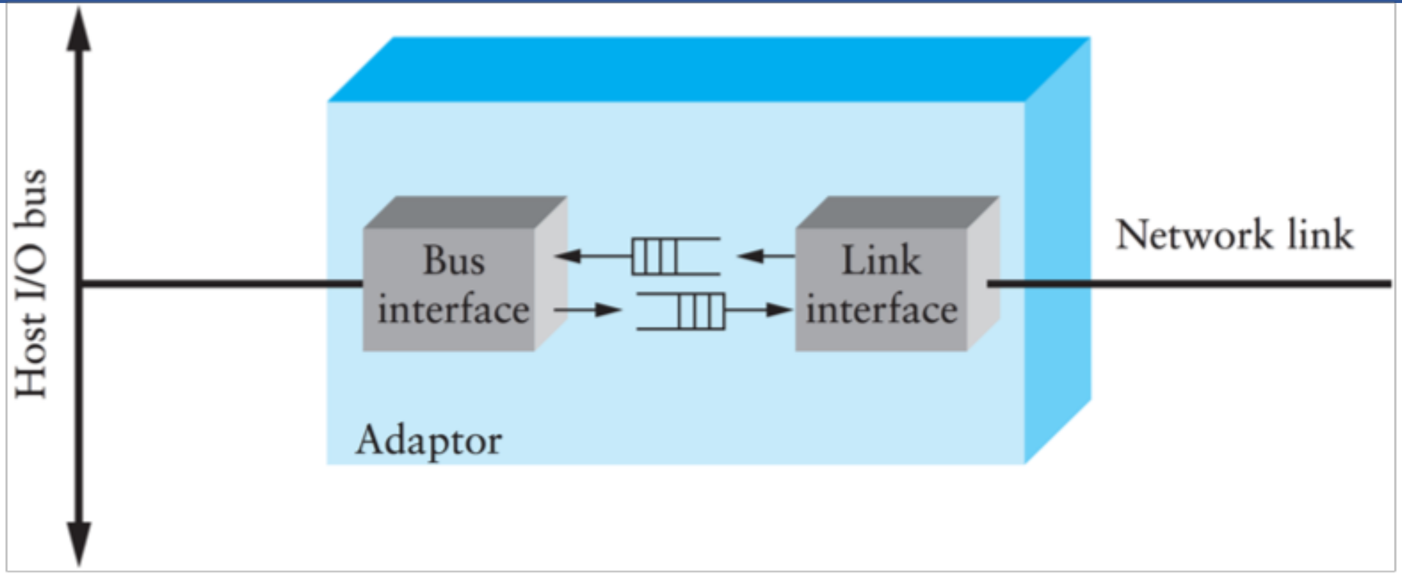
Direct Memory Access

- **Avoid programmed I/O for large data movement**
- **Requires DMA controller**
- **Bypasses CPU to transfer data directly between I/O device and memory**
- **OS writes DMA command block into memory**
  - Source and destination addresses
  - Read or write mode
  - Count of bytes
  - Writes location of command block to DMA controller

Example: IDE disk read w. DMA

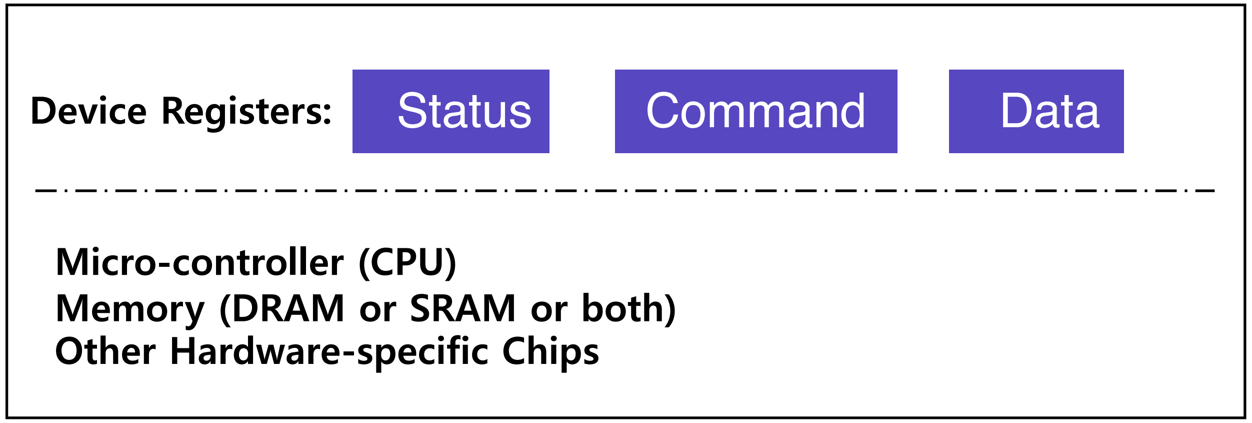

1. device driver is told to transfer disk data to buffer at address X
2. device driver tells disk controller to transfer C bytes from disk to buffer at address X
3. disk controller initiates DMA transfer
4. disk controller sends each byte to DMA controller
5. DMA controller transfers bytes to buffer X, increasing memory address and decreasing C until C = 0
6. when C = 0, DMA interrupts CPU to signal transfer completion
CPU
cache
CPU memory bus
memory
x
buffer
DMA/bus/interrupt controller
PCI bus
IDE disk controller
disk
disk
disk
disk

Example: Network Interface Card



- **Link interface talks to wire/fiber/antenna**
  - Typically does framing, link-layer CRC
- **FIFOs on card provide small amount of buffering**
- **Bus interface logic uses DMA to move packets to and from buffers in main memory**

Protocol Variants



- **Status checks: *polling* vs. *interrupts***
- **Data: *programmed I/O* (PIO) vs. *direct memory access* (DMA)**
- **Control: *special instructions* vs. *memory-mapped I/O***

Variety Is a Challenge

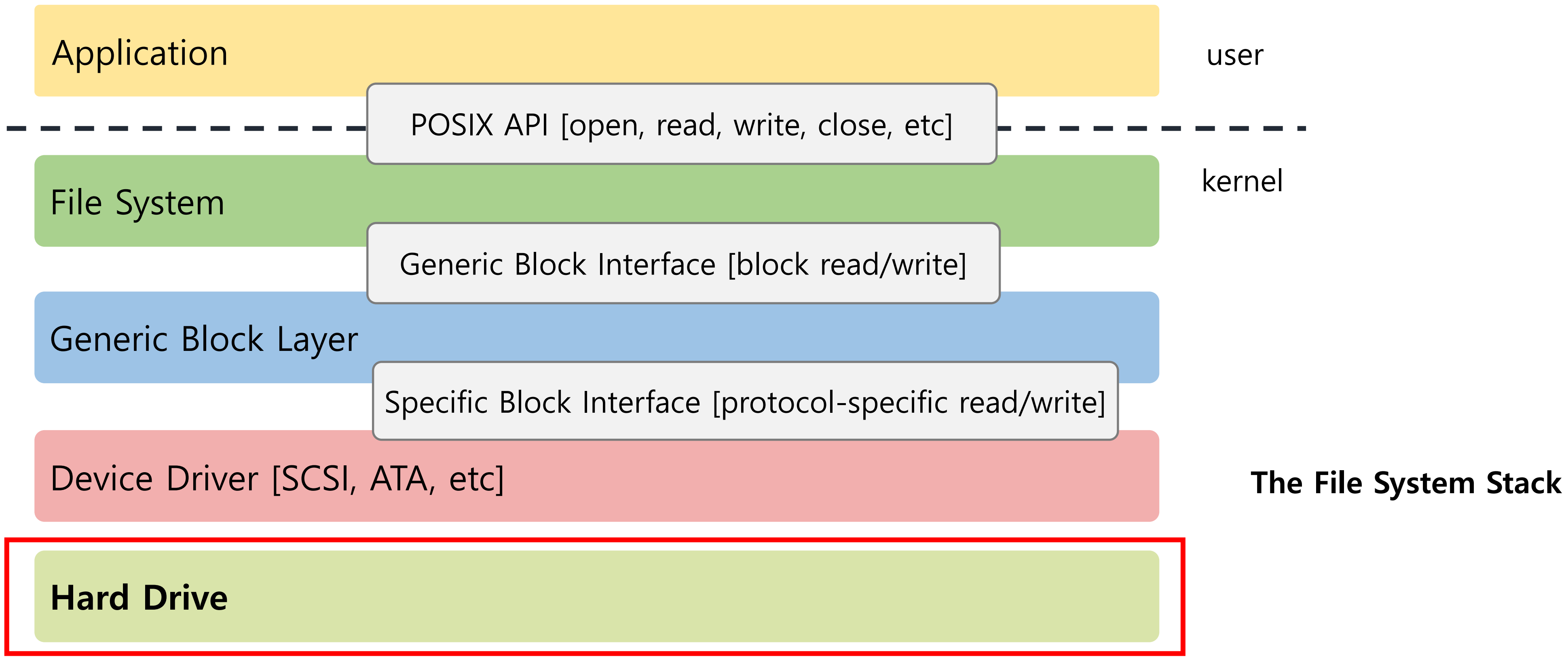
- **Problem:**
  - many, many devices
  - each has its own protocol
- **How can we avoid writing a slightly different OS for each H/W combination?**

Variety Is a Challenge

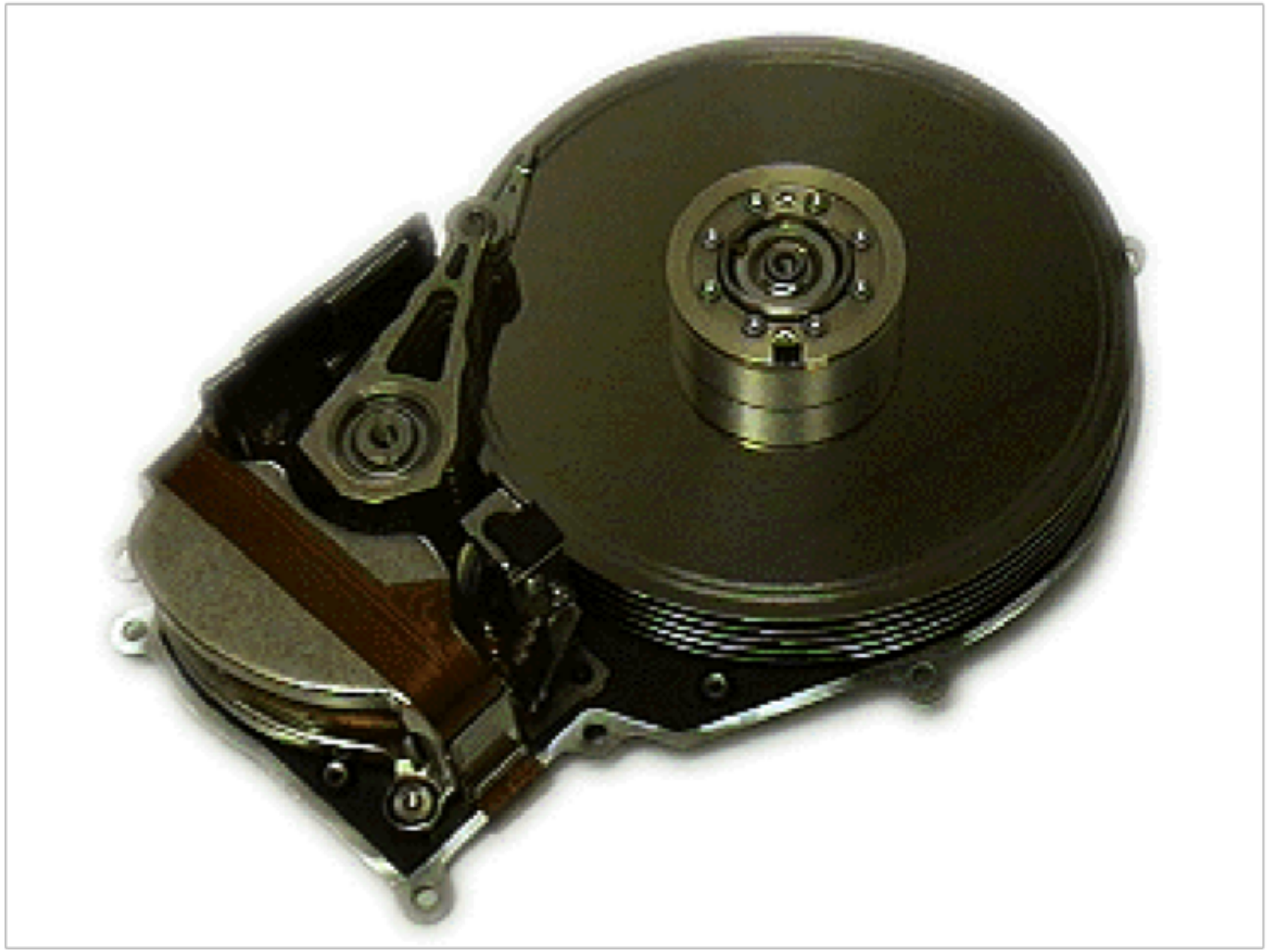
- **Problem:**
  - many, many devices
  - each has its own protocol
- **How can we avoid writing a slightly different OS for each H/W combination?**
- **Solution: Abstraction!**
  - Build a common interface
  - Write device driver for each device
  - Drivers are 70% of Linux source code

File System Abstraction

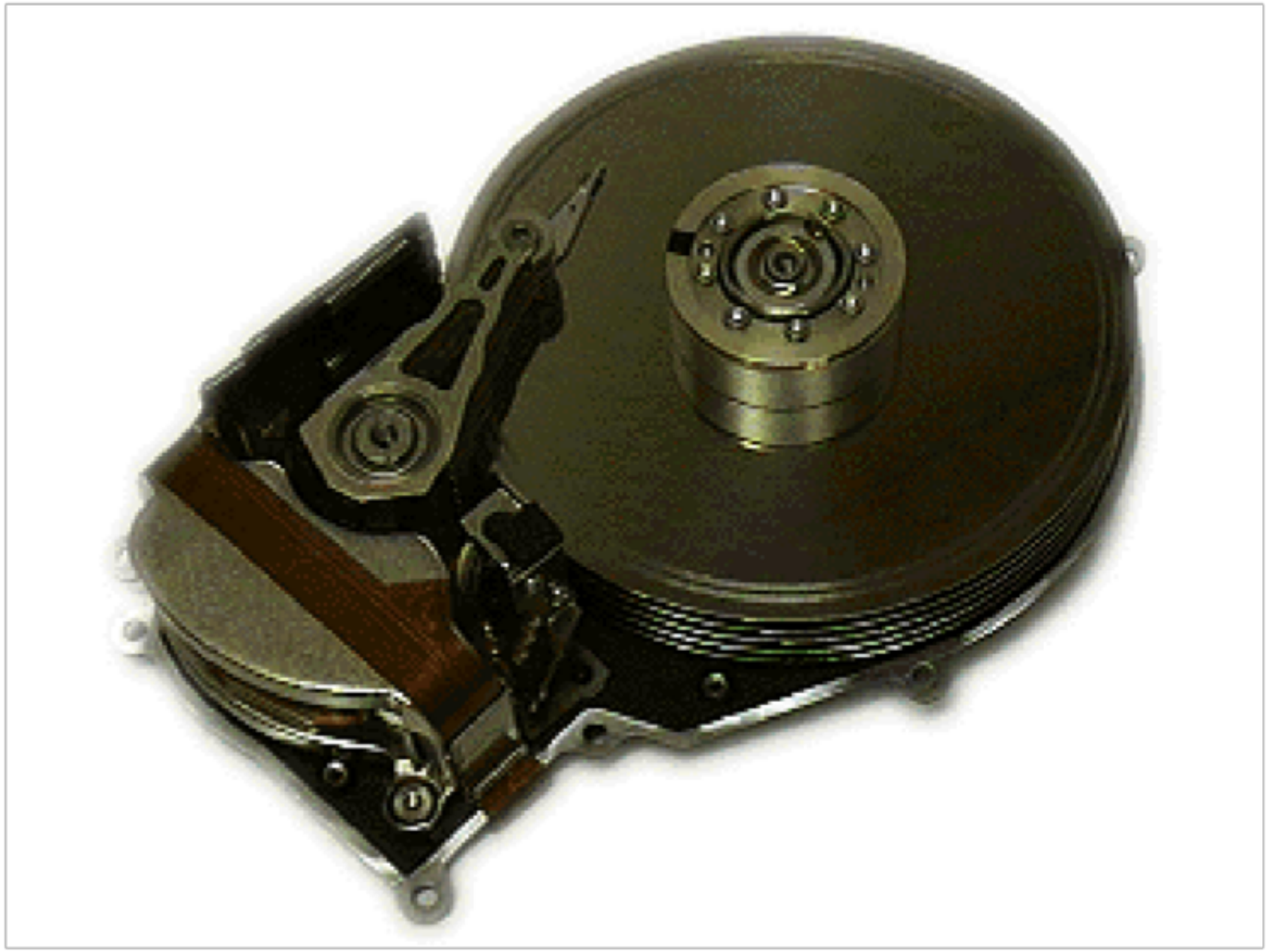
- **File system** specifics **of which disk class it is using.**
  - Ex) It issues **block read** and **write** request to the generic block layer.


Application
user
POSIX API [open, read, write, close, etc]
kernel
File System
Generic Block Interface [block read/write]
Generic Block Layer
Specific Block Interface [protocol-specific read/write]
Device Driver [SCSI, ATA, etc]
The File System Stack
Hard Drive

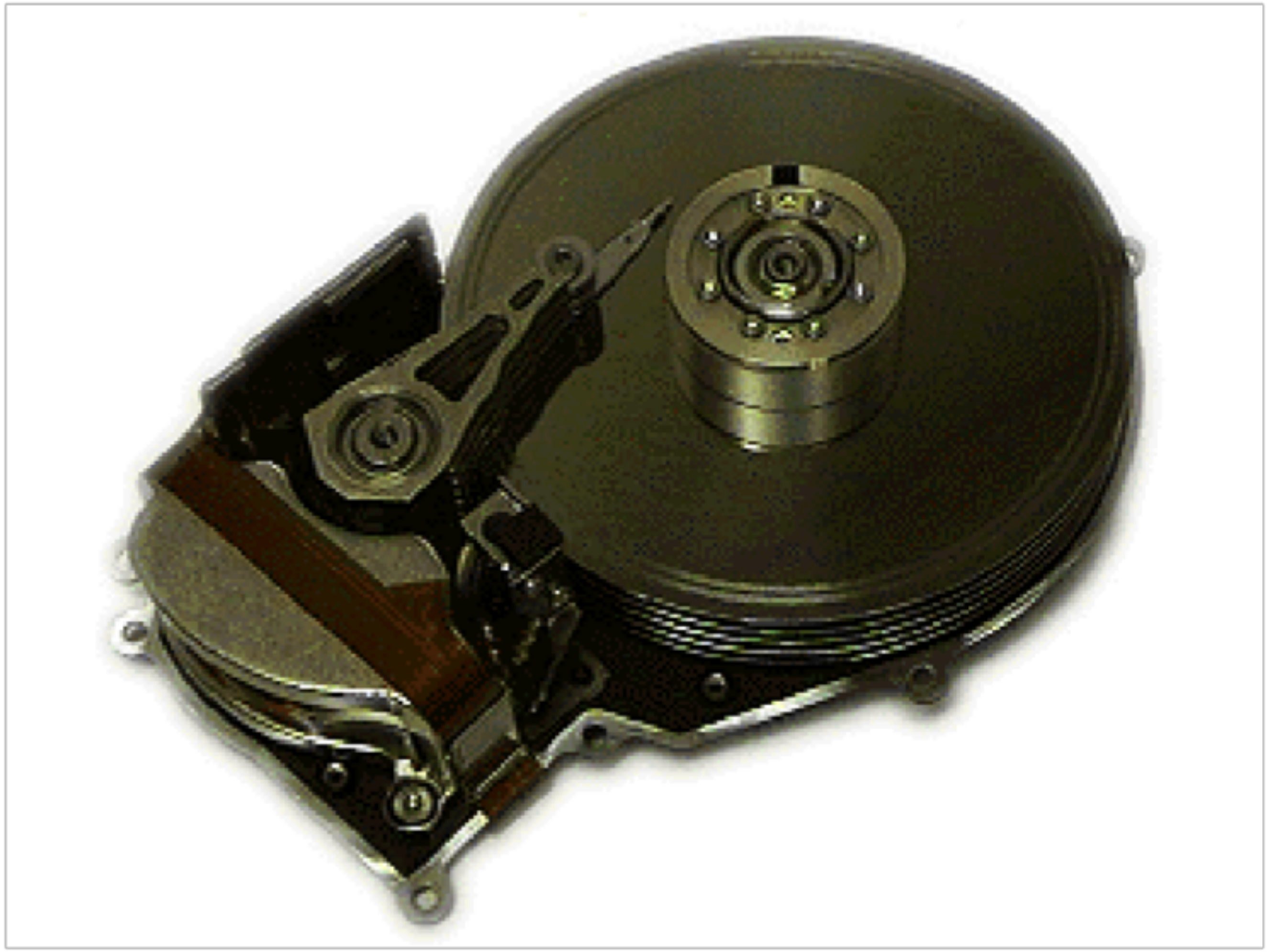
Hard Disks



Hard Disks



Hard Disks



Basic Interface

- **Disk interface presents linear array of sectors**
  - Historically 512 Bytes
  - Written atomically (even if there is a power failure)
  - 4 KiB in “advanced format” disks
    - Torn write: If an untimely power loss occurs, only a portion of a larger write may complete
- **Disk maps logical sector #s to physical sectors**
- **OS doesn’t know logical to physical sector mapping**

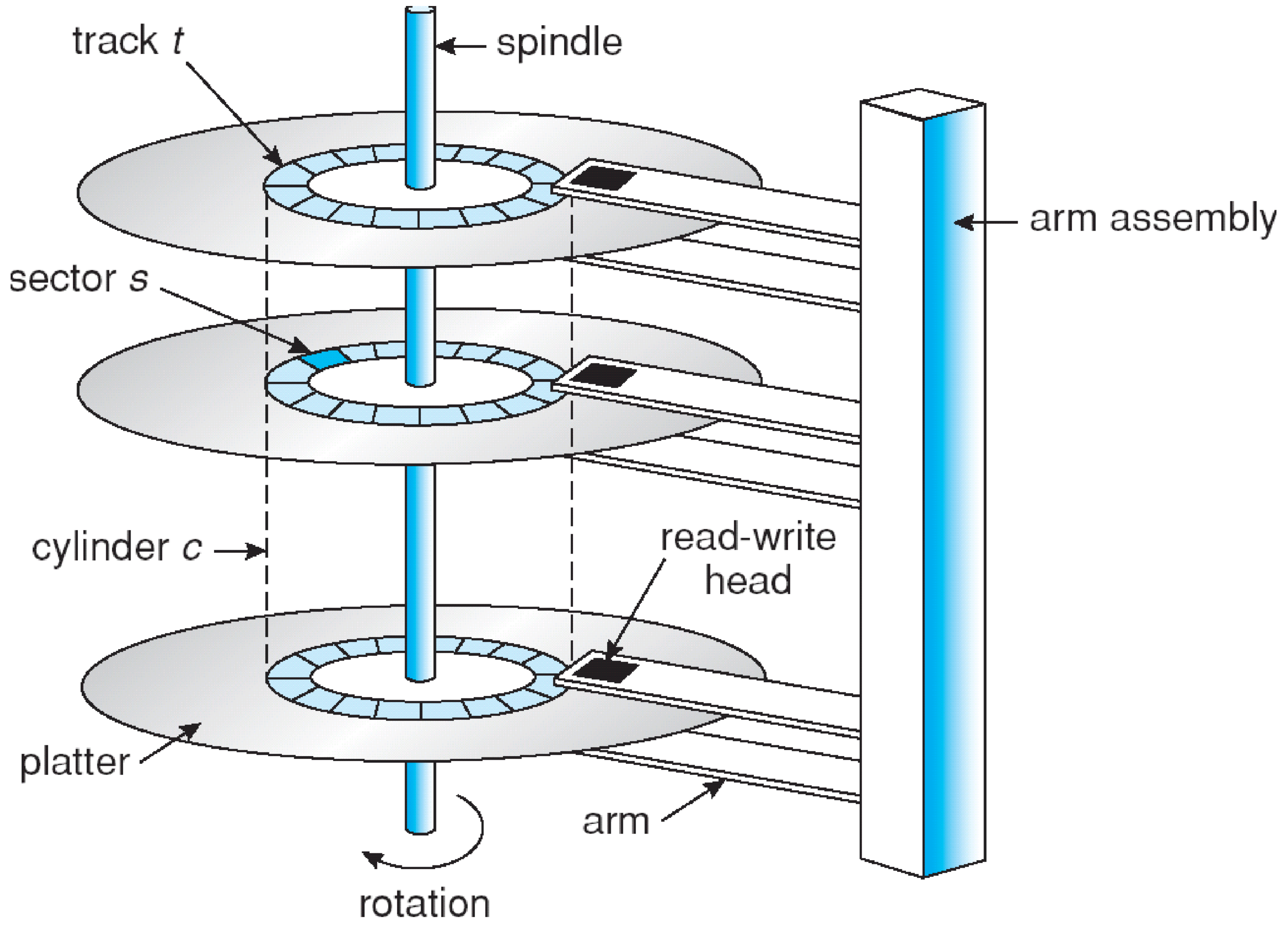
Basic Geometry

- **Platter (Aluminum coated with a thin magnetic layer)**
  - A circular hard surface
  - Data is stored persistently by inducing magnetic changes to it.
  - Each platter has 2 sides, each of which is called a **surface**.

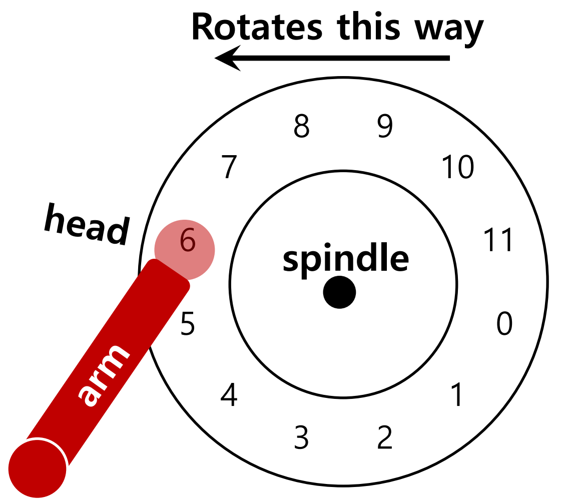
Basic Geometry (Cont.)

- **Spindle**
  - Spindle is connected to a motor that spins the platters around.
  - The rate of rotations is measured in **RPM** (Rotations Per Minute).
    - Typical modern values : 7,200 RPM to 15,000 RPM.
- **Track**
  - Concentric circles of sectors
  - Data is encoded on each surface in a track.
  - A single surface contains many thousands and thousands of tracks.
- **Cylinder**
  - A stack of tracks of fixed radius
  - Heads record and sense data along cylinders
  - Generally only one head active at a time

Cylinders, Tracks, & Sectors



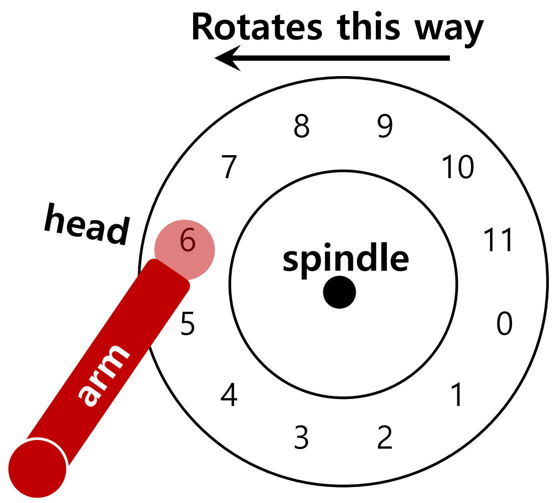
A Simple Disk Drive



A Single Track Plus A Head

- **Disk head (One head per surface of the drive)**
  - The process of *reading* and *writing* is accomplished by the **disk head**.
  - Attached to a single disk arm, which moves across the surface.

Single-track Latency: The Rotational Delay

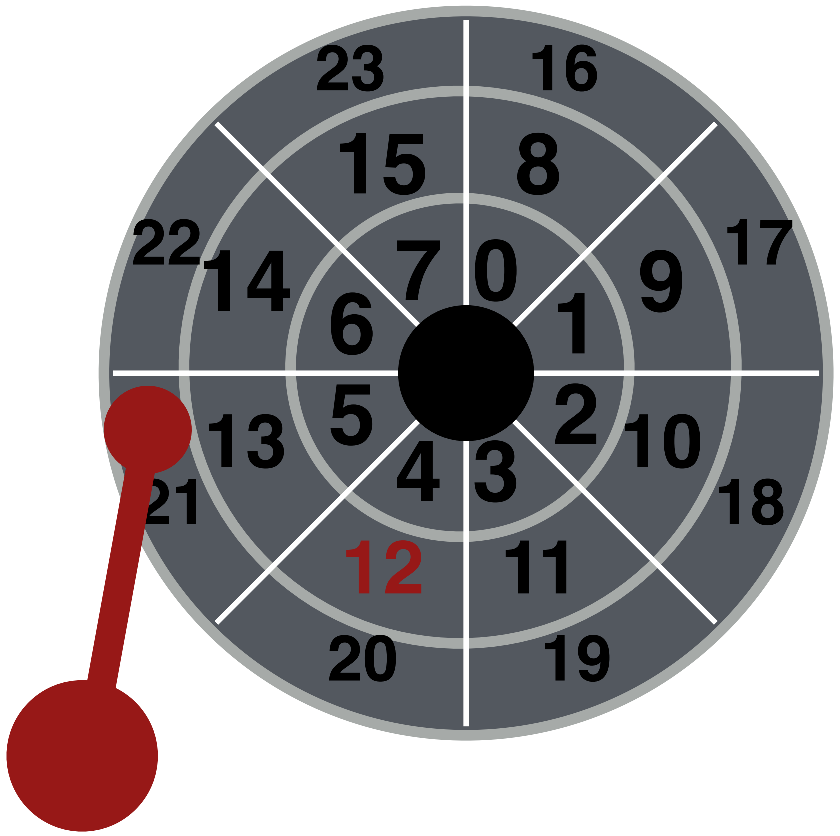


A Single Track Plus A Head

- **Rotational delay: Time for the desired sector to rotate**
  - Ex) Full rotational delay is `R` and we start at `sector 6`
    - Read sector 0: Rotational delay = $\frac{R}{2}$
    - Read sector 5: Rotational delay = `R-1` (worst case.)

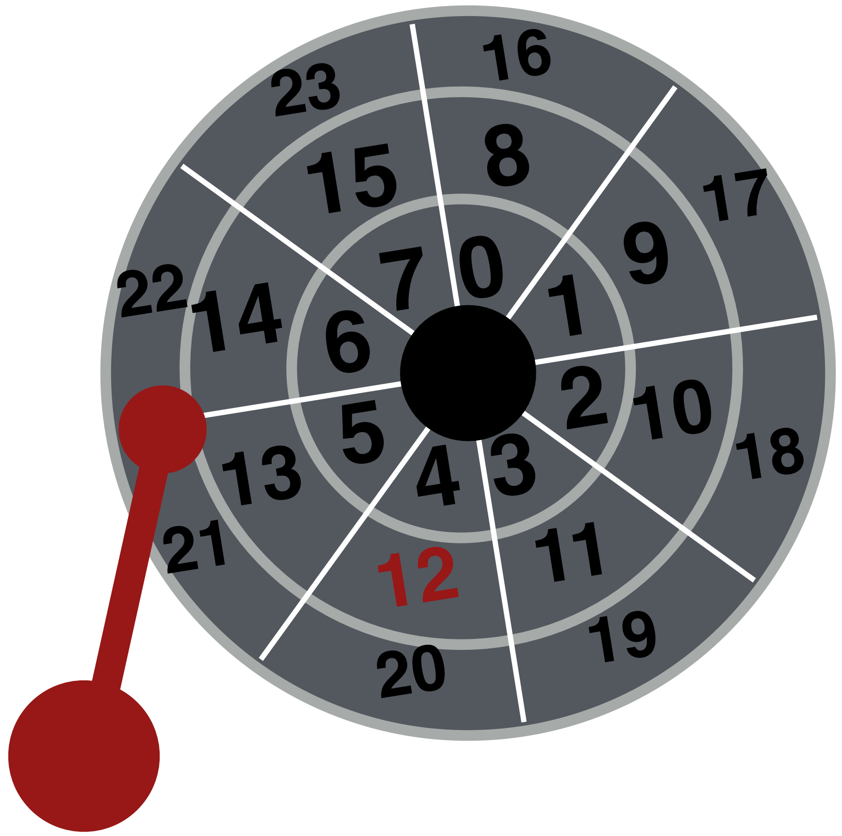
Multiple Tracks

- **Let's Read 12!**



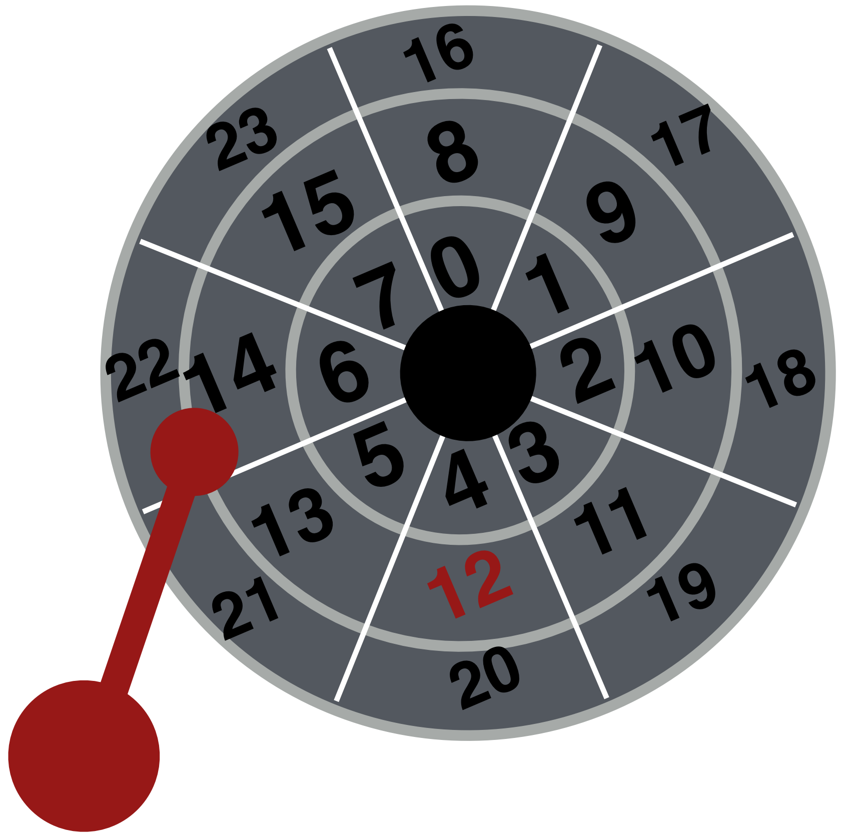
Multiple Tracks: Seek To Right Track

- **Let's Read 12!**



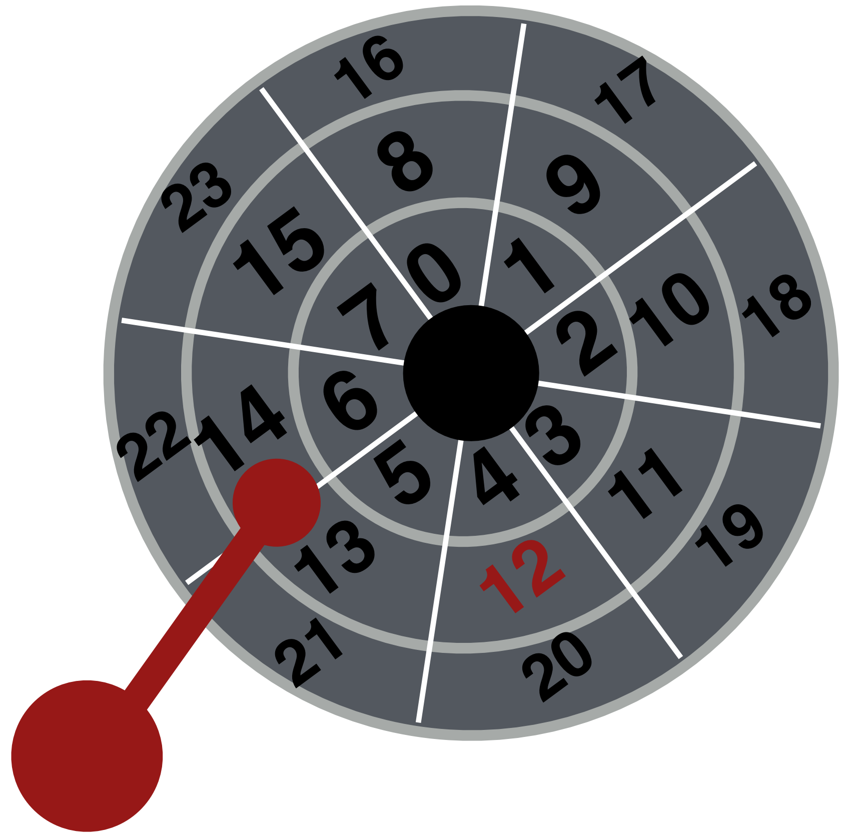
Multiple Tracks: Seek To Right Track

- **Let’s Read 12!**



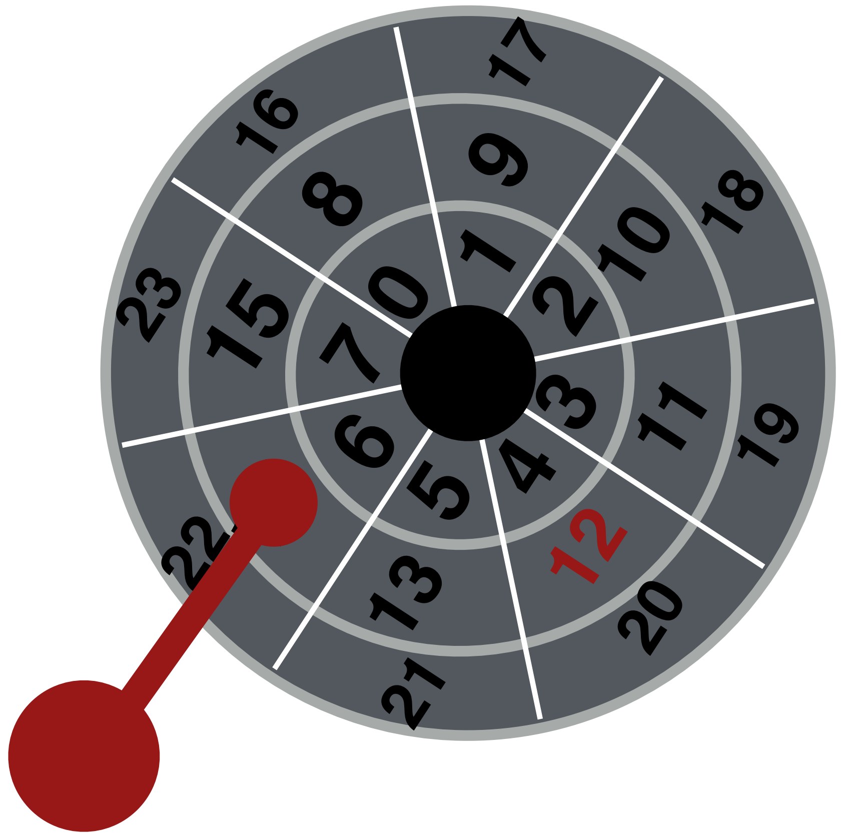
Multiple Tracks: Seek To Right Track

- **Let's Read 12!**



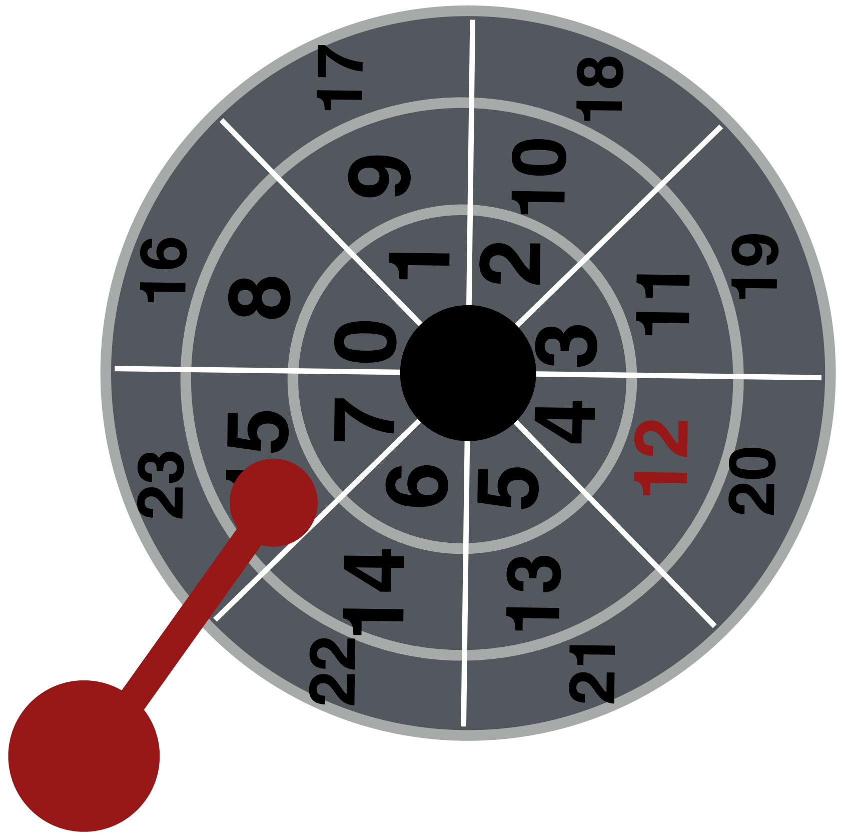
Multiple Tracks: Wait for Rotation

- **Let's Read 12!**



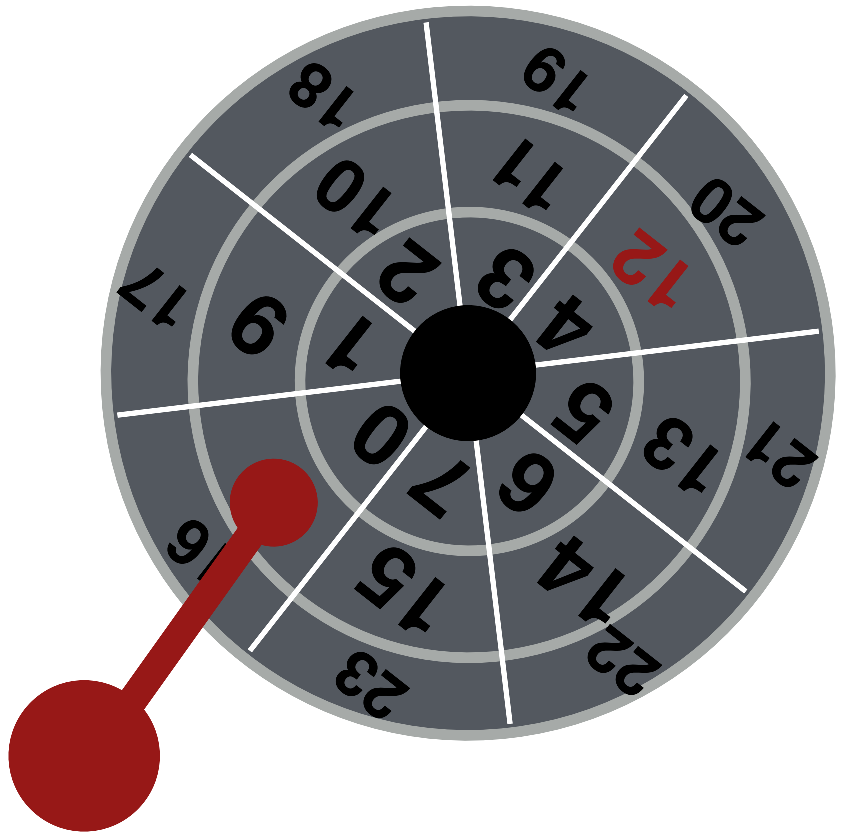
Multiple Tracks: Wait for Rotation

- **Let's Read 12!**



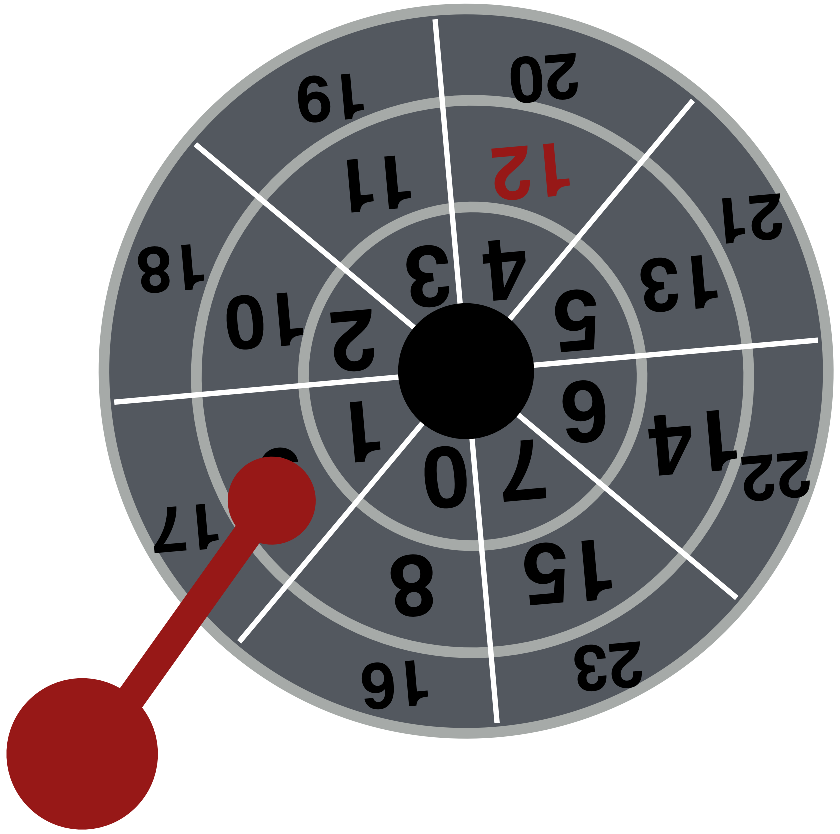
Multiple Tracks: Wait for Rotation

- **Let's Read 12!**



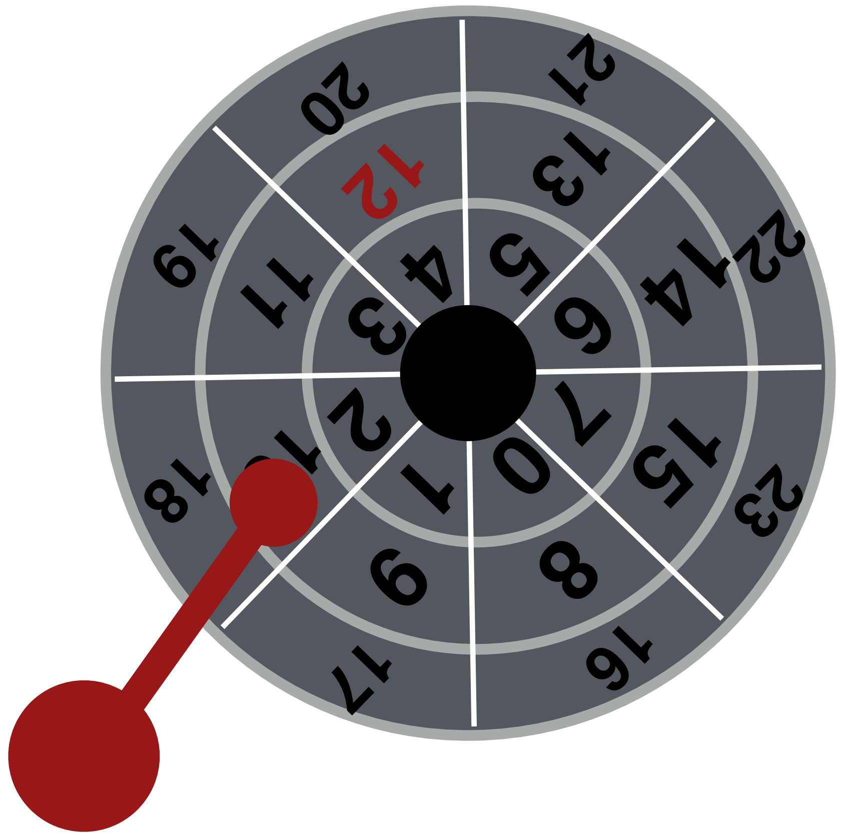
Multiple Tracks: Wait for Rotation

- **Let's Read 12!**



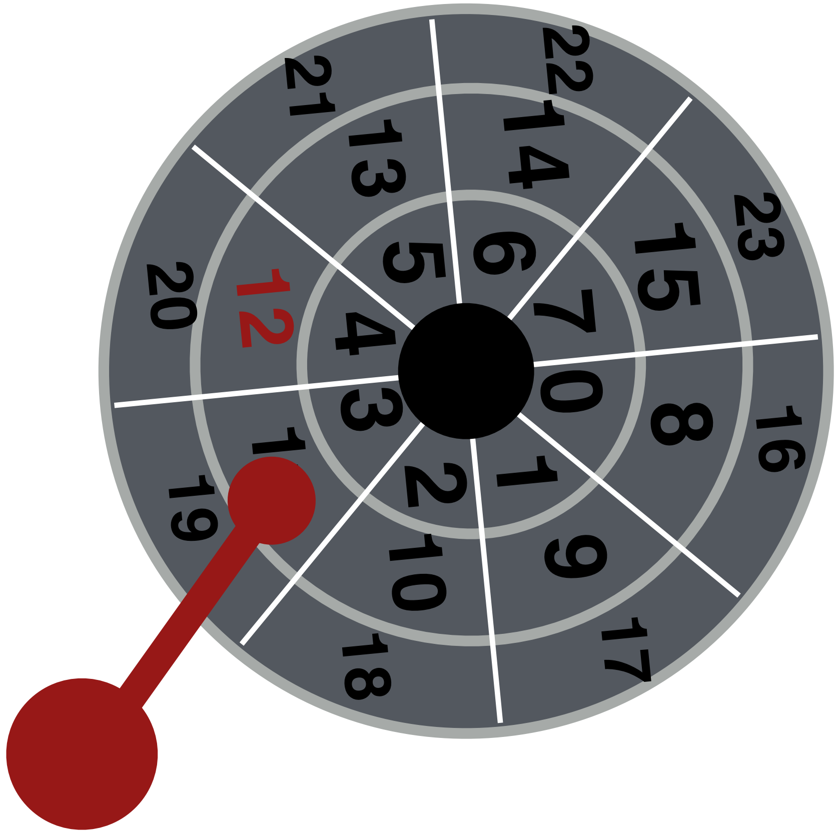
Multiple Tracks: Wait for Rotation

- **Let's Read 12!**



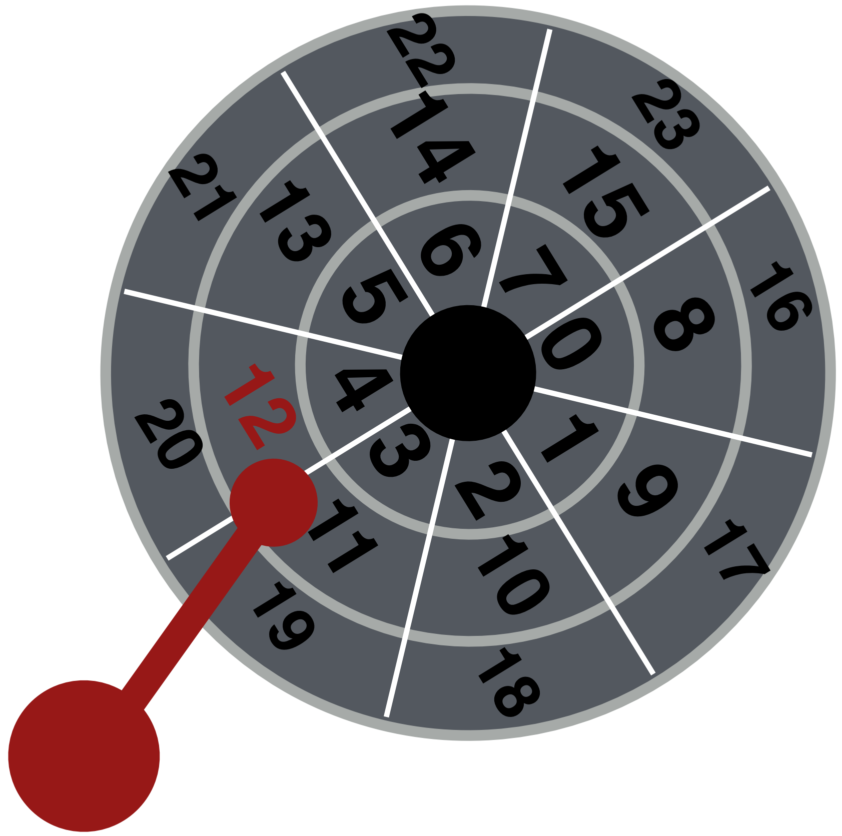
Multiple Tracks: Wait for Rotation

- **Let's Read 12!**



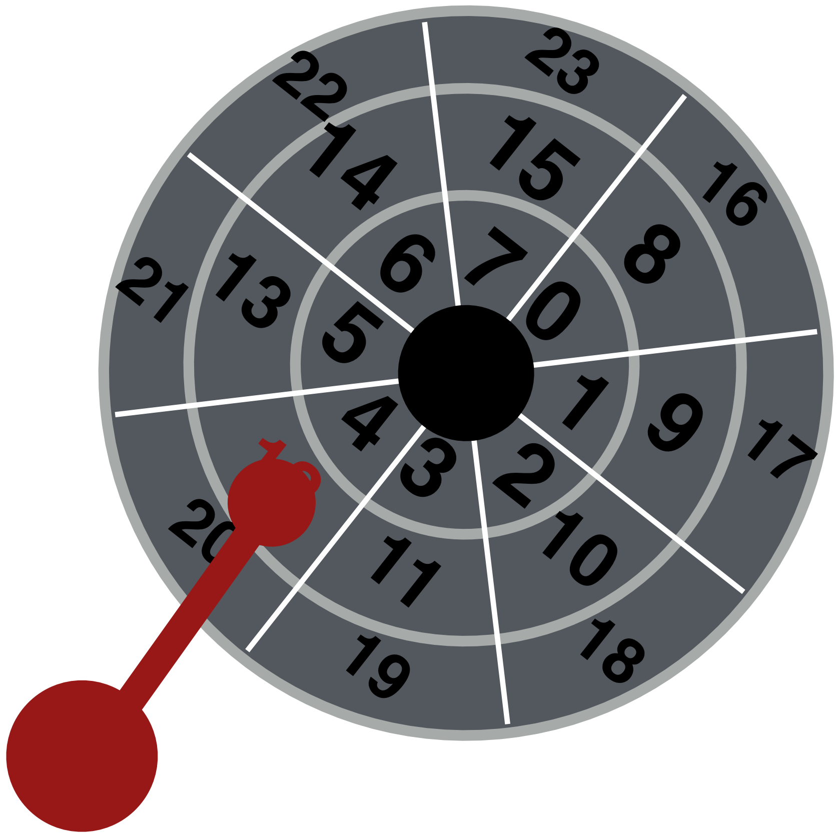
Multiple Tracks: Transfer Data

- **Let's Read 12!**



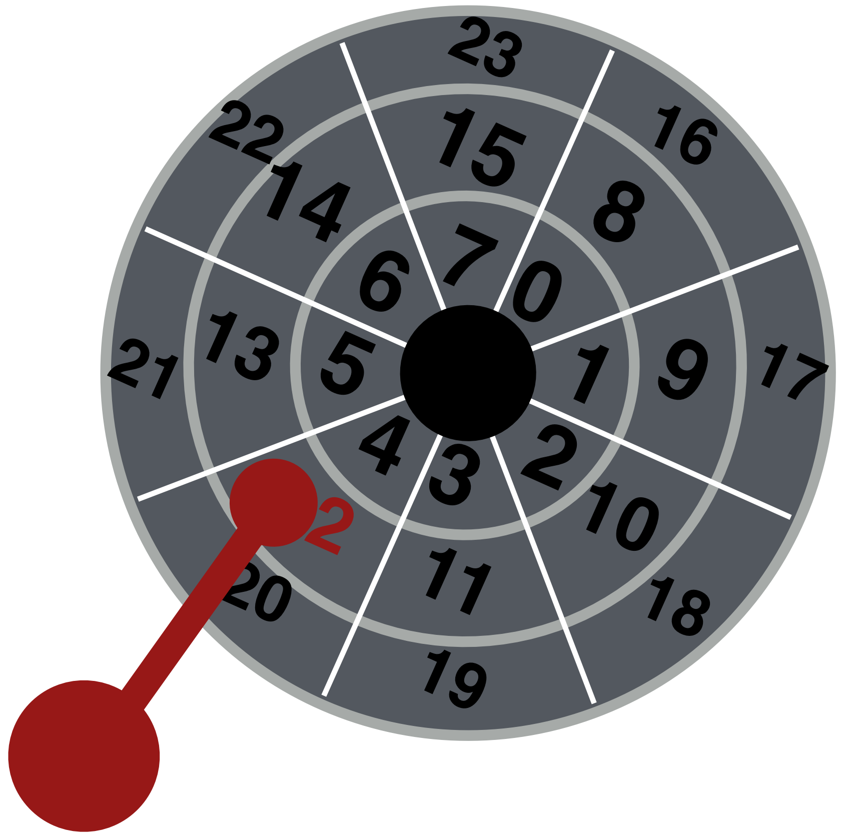
Multiple Tracks: Transfer Data

- **Let's Read 12!**

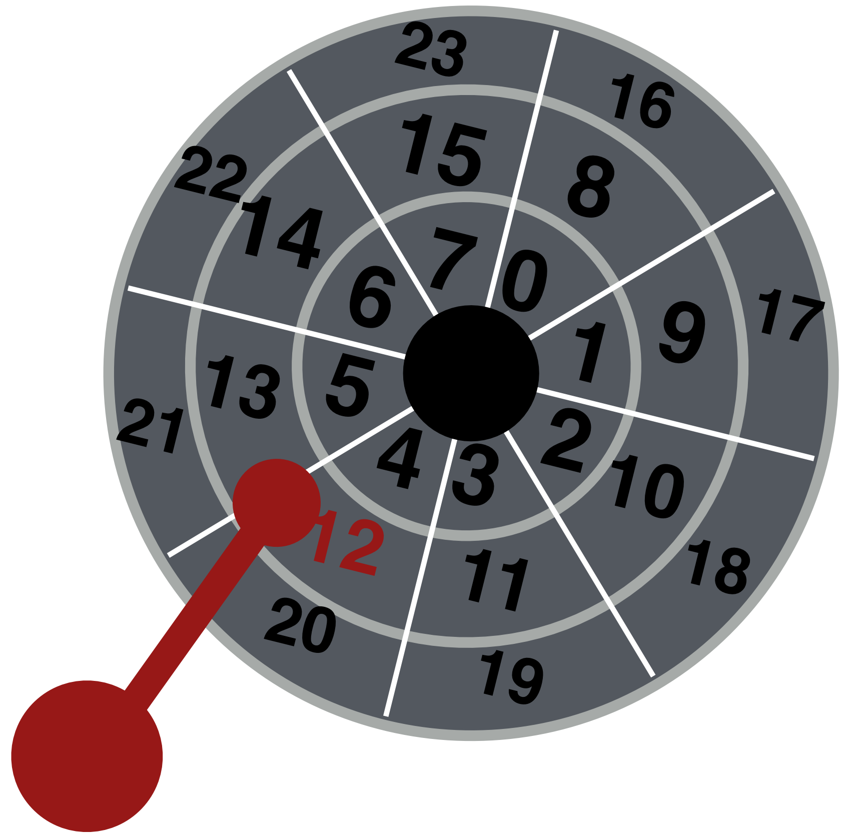


Multiple Tracks: Transfer Data

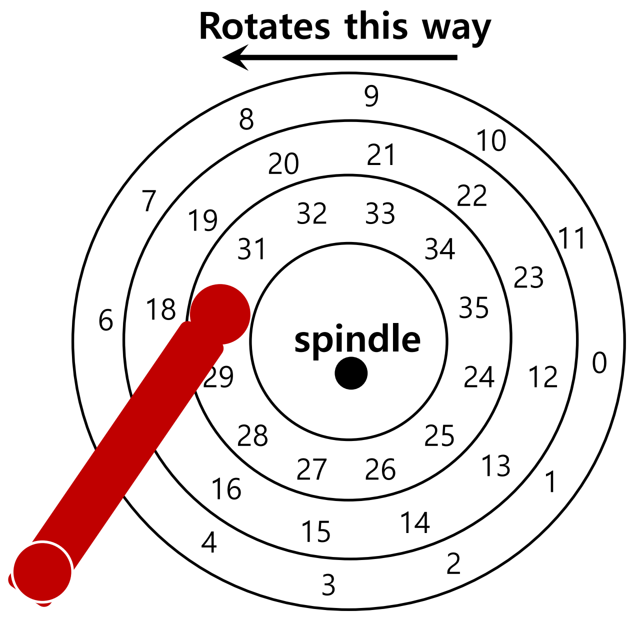
- **Let's Read 12!**

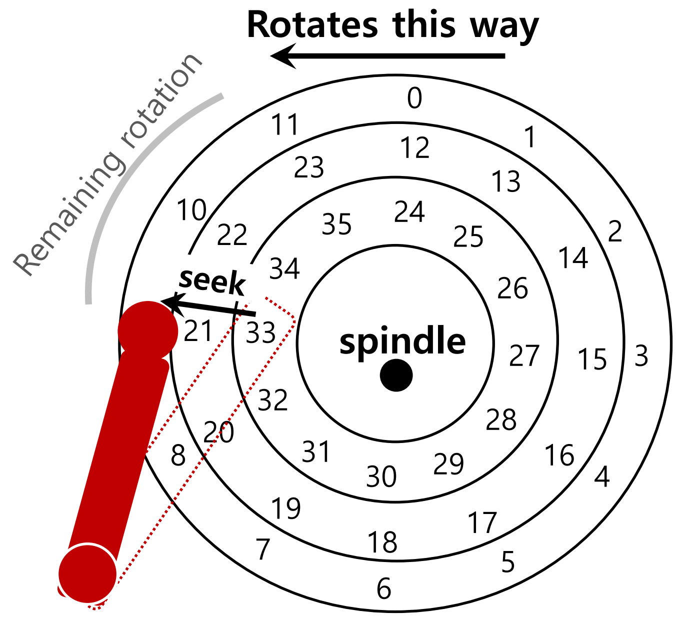


Yay!



Multiple Tracks: Seek Time





- **Seek: Move the disk arm to the correct track**
  - Seek time: Time to move head to the track contain the desired sector.
  - One of the most costly disk operations.

Seek, Rotate, Transfer

- **Acceleration → Coasting → Deceleration → Settling**
  - **Acceleration**: The disk arm gets moving.
  - **Coasting**: The arm is moving at full speed.
  - **Deceleration**: The arm slows down.
  - **Settling**: The head is *carefully positioned* over the correct track.
- **Seeks often take several milliseconds!**
  - settling alone can take 0.5 to 2ms.
  - entire seek often takes 4 - 10 ms.

Seek, Rotate, Transfer

- **Depends on rotations per minute (RPM)**
  - 7200 RPM is common, 1500 RPM is high end.
- **With 7200 RPM, how long to rotate around?**
  - 1 / 7200 RPM = 1 minute / 7200 rotations = 1 second / 120 rotations = 8.3 ms / rotation
- **Average rotation?**
  - 8.3 ms / 2 = 4.15 ms

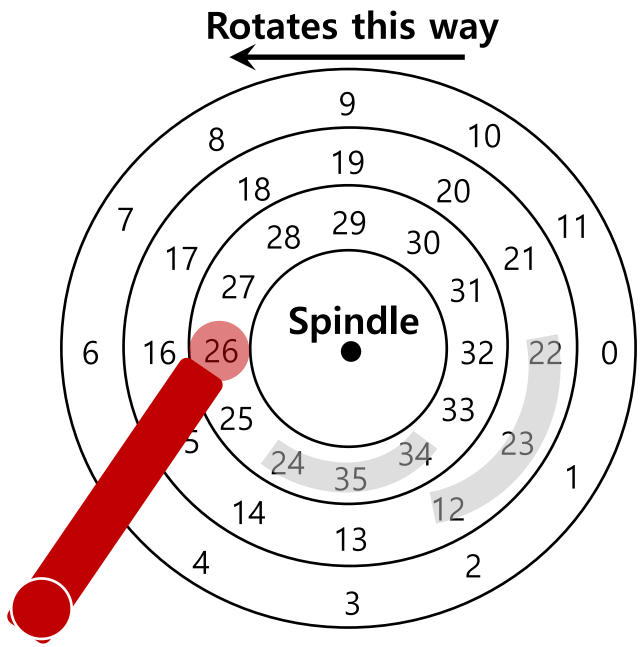
Seek, Rotate, Transfer

- **The final phase of I/O**
  - Data is either *read from* or *written* to the surface.
- **Pretty fast — depends on RPM and sector density**
- **100+ MB/s is typical for maximum transfer rate**
- **How long to transfer 512-bytes?**
  - 512 bytes * (1s / 100 MB) = 5 μs

Workload

- **So…**
  - seeks are slow
  - rotations are slow
  - transfers are fast
- **What kind of workload is fastest for disks?**
  - **Sequential**: access sectors in order (transfer dominated)
  - **Random**: access sectors arbitrarily (seek+rotation dominated)

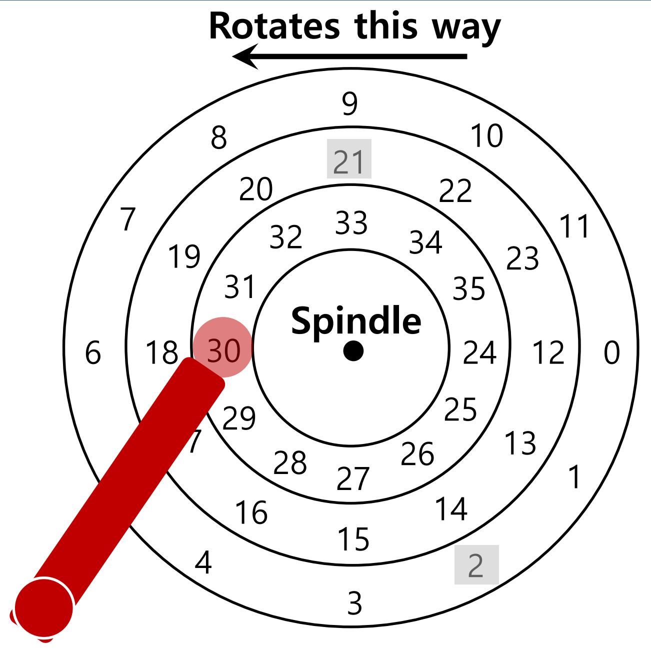
Track Skew



Three Tracks: Track Skew Of 2

- **Make sure that sequential reads can be properly serviced even when crossing track boundaries**
  - *Without track skew*, the head would be moved to the next track but the desired next block would have already rotated under the head.

Disk Scheduling

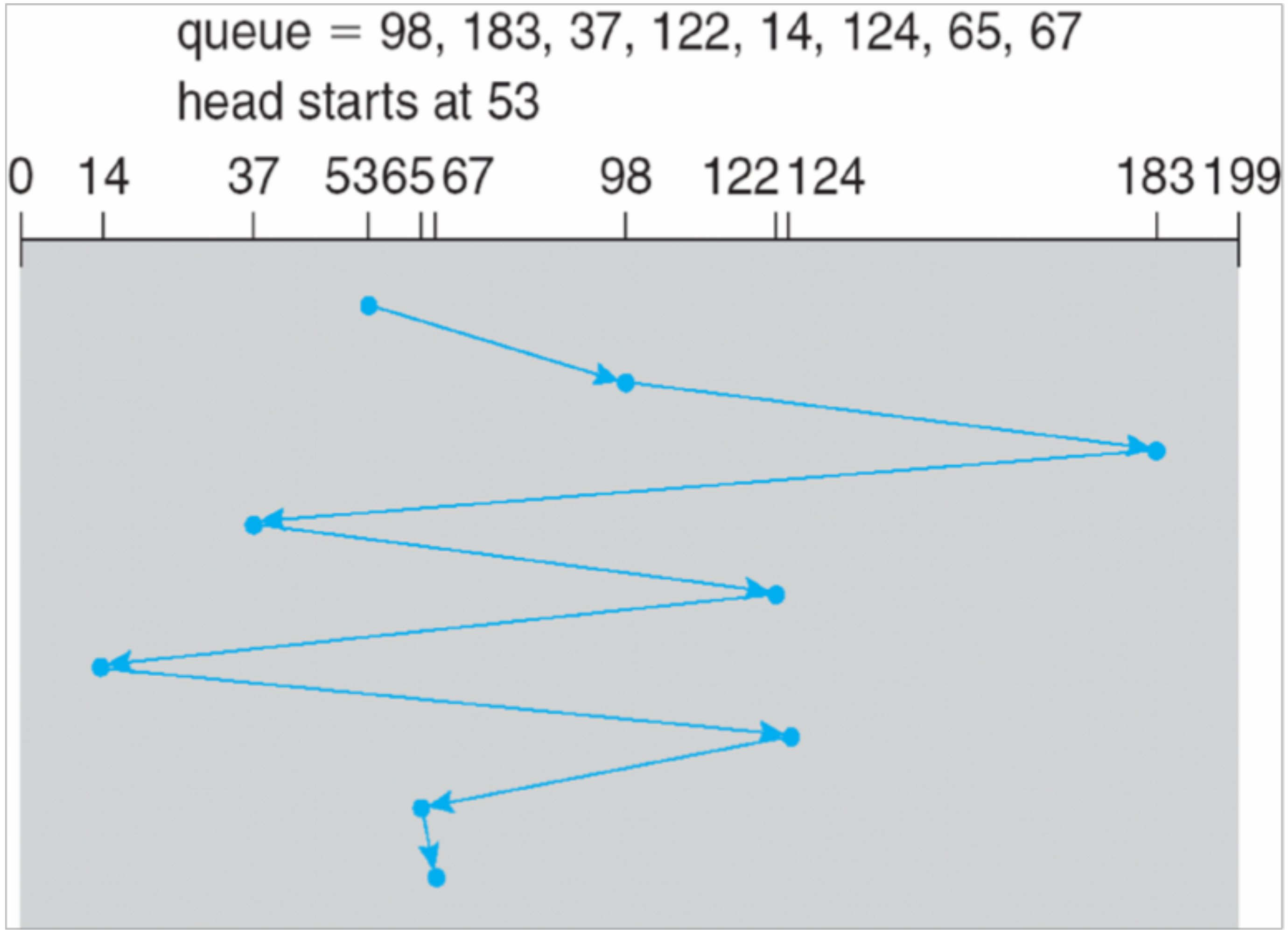


- **Disk Scheduler decides which I/O request to schedule next**

Disk Scheduling: FCFS

- **“First Come First Served”**
  - Process disk requests in the order they are received
- **Advantages**
  - Easy to implement
  - Good fairness
- **Disadvantages**
  - Cannot exploit request locality
  - Increases average latency, decreasing throughput

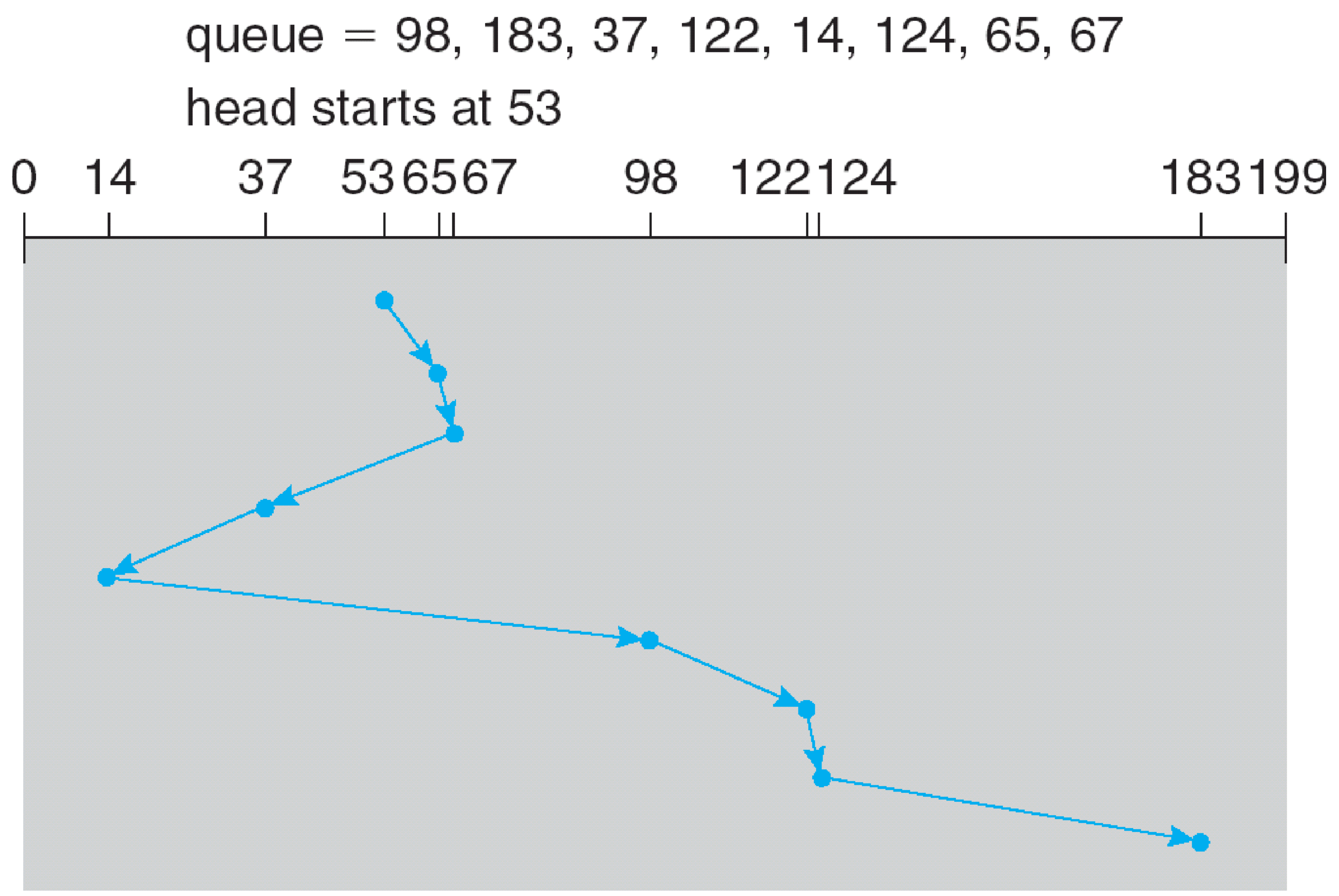
FCFS Example



SSTF (Shortest Seek Time First)

- **Order the queue of I/O request by track**
- **Pick requests on the nearest track to complete first**
  - Also called shortest positioning time first (SPTF)
- **Advantages**
  - Exploits locality of disk requests
  - Higher throughput
- **Disadvantages**
  - Starvation
  - Don’t always know what request will be fastest

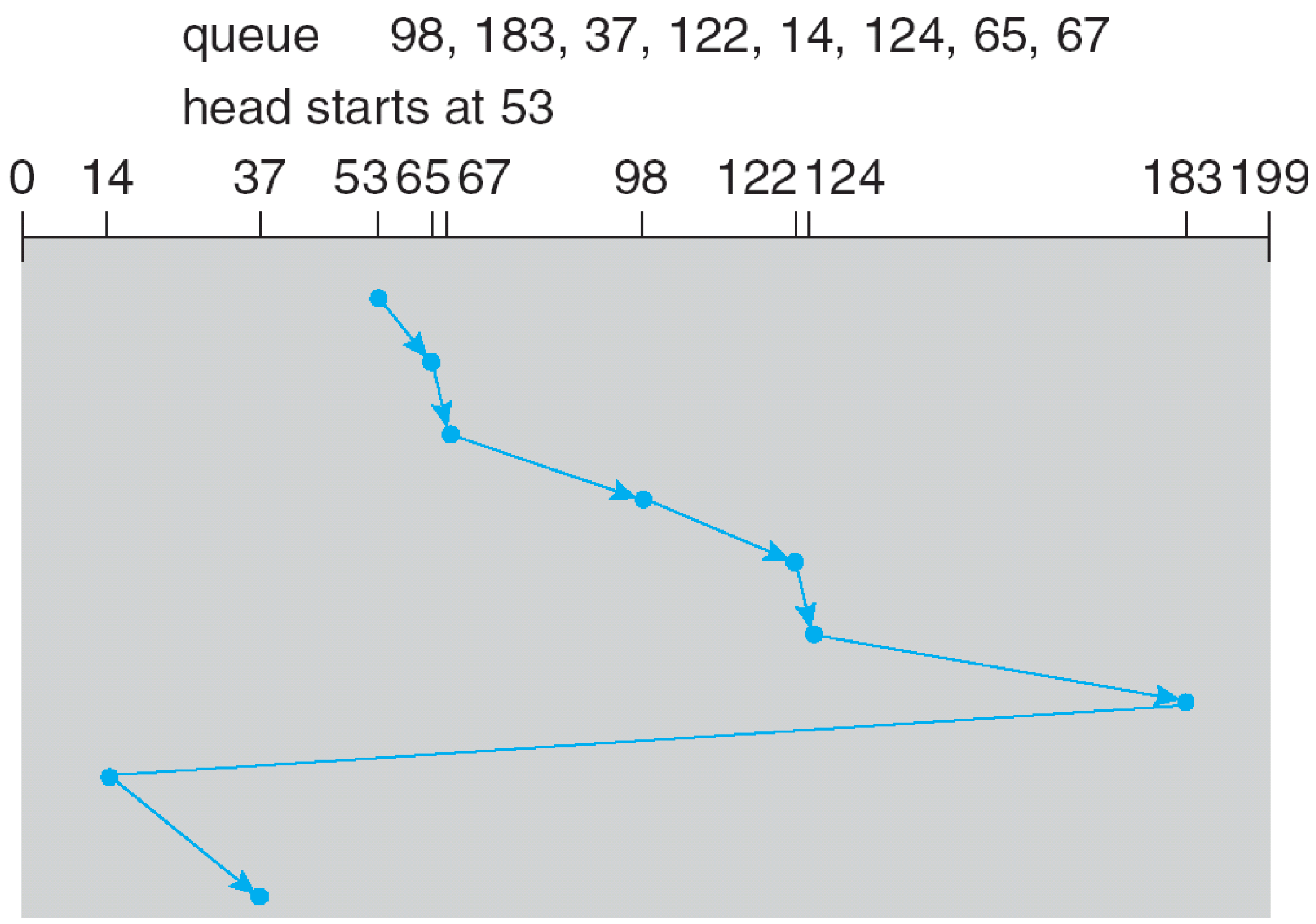
SSTF Example



“Elevator” Scheduling (SCAN)

- **Sweep across disk, servicing all requests passed**
  - Like SSTF, but next seek must be in same direction
  - Switch directions only if no further requests
- **Advantages**
  - Takes advantage of locality
  - Bounded waiting
- **Disadvantages**
  - Cylinders in the middle get better service
  - Might miss locality SSTF could exploit
- **CSCAN: Only sweep in one direction**
  - **Very commonly used algorithm in Unix**

CSCAN example



Flash Memory

- **Today, people increasingly using flash memory**
- **Completely solid state (no moving parts)**
  - Remembers data by storing charge
  - Lower power consumption and heat
  - No mechanical seek times to worry about
- **Limited # overwrites possible**
  - Blocks wear out after 10,000 (MLC) – 100,000 (SLC) erases
  - Requires flash translation layer (FTL) to provide wear leveling, so repeated writes to logical block don't wear out physical block
  - FTL can seriously impact performance
- **Limited durability**
  - Charge wears out over time
  - Turn off device for a year, you can potentially lose data!

Next Time…

- **Read Chapter 39, 40**