Operating Systems

IO Devices and HDD (Chapter 36 ~ 37)

Dr. Young-Woo Kwon

36. I/O Devices

Motivation

What good is a computer without any I/O devices?

- keyboard, display, disks

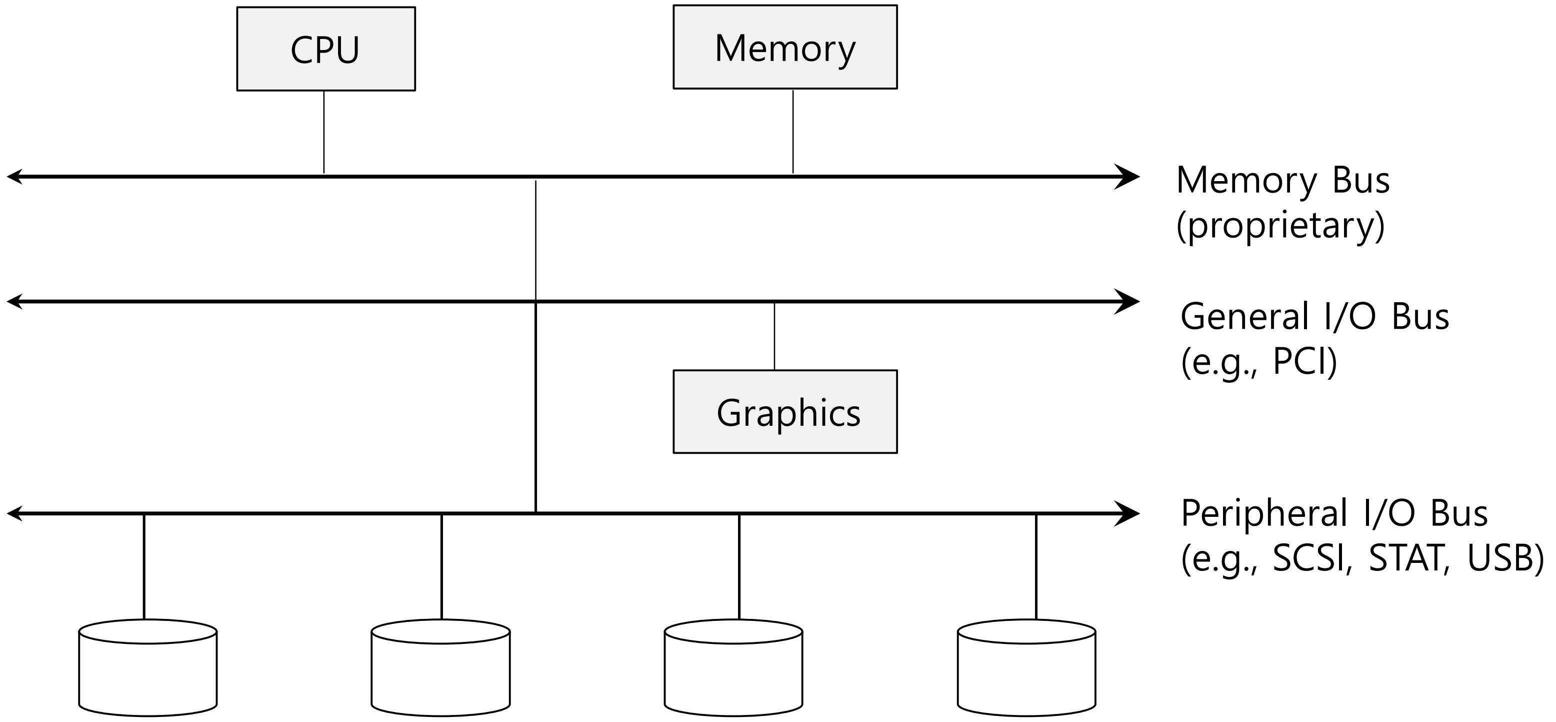
We want:

- **H/W** that will let us plug in different devices
- **OS** that can interact with different combinations

I/O Devices

- I/O is critical to computer system to interact with systems.
- Issue :
  - 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



Prototypical System Architecture

CPU is attached to the main memory of the system via some kind of memory bus.

Some devices are connected to the system via a general I/O bus.

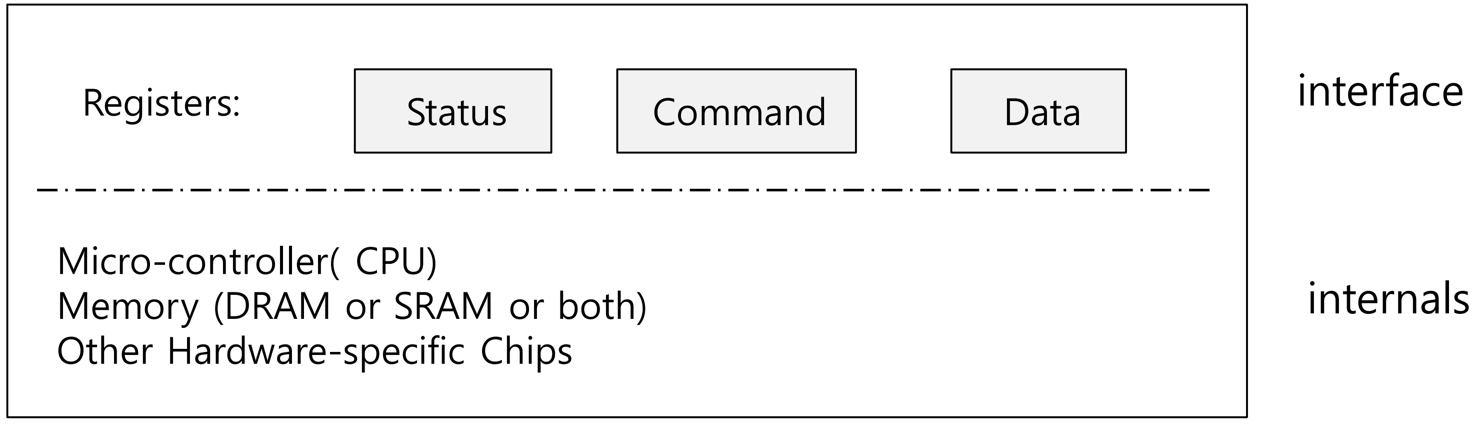
I/O Architecture

- Buses
  - **Data paths** that provided to enable information between **CPU(s), RAM, and I/O devices.**

- I/O bus
  - **Data path** that connects **a CPU to an I/O device**.
  - I/O bus is connected to I/O device by three hardware components: **I/O ports, interfaces and device controllers.**

Canonical Device

- Canonical Devices has two important components.
  - **Hardware interface** allows the system software to control its operation.
  - **Internals** which is implementation specific.



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 and writing above three registers,
the operating system can control device behavior.

Hardware interface of Canonical Device

- Typical interaction example

```
while ( STATUS == BUSY)
    SPIN; //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)
    SPIN; //wait until device is done with your request
```

Polling

- Operating system 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.

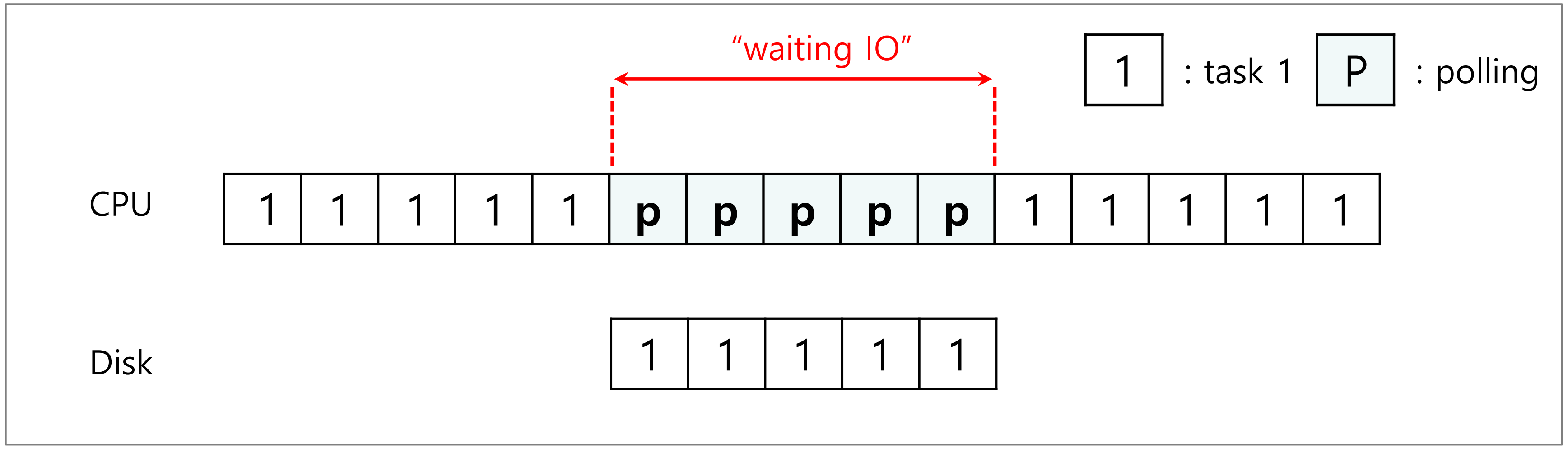


Diagram of CPU utilization by polling

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.

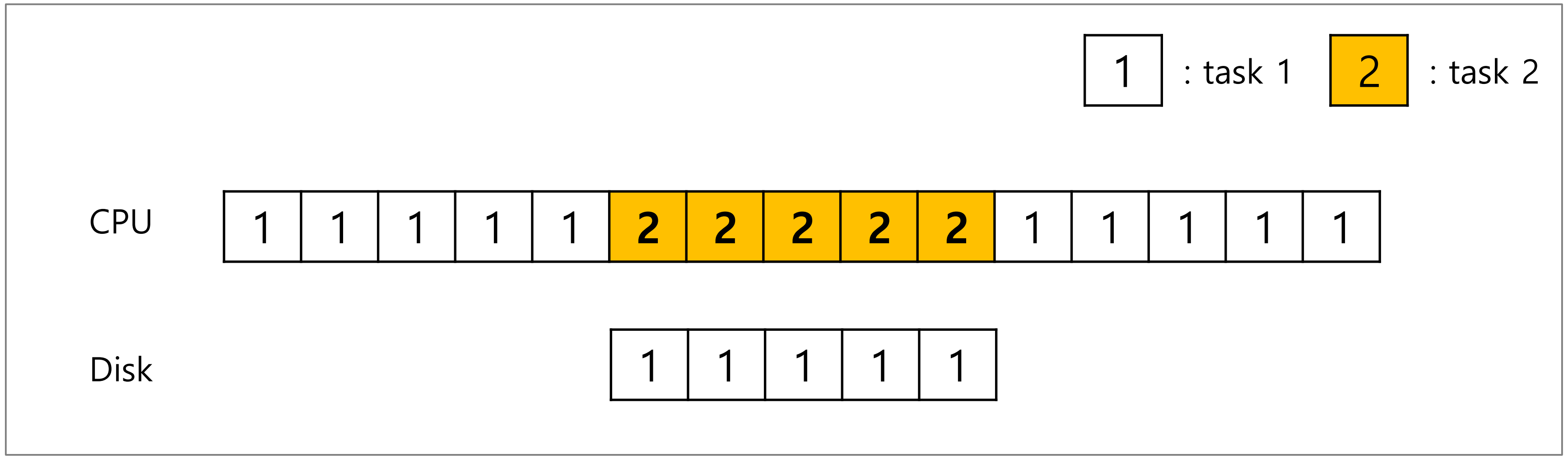


Diagram of CPU utilization by interrupt

Polling vs interrupts

- *However,* **“interrupts is not always the best solution”**
  - If, device performs very quickly, interrupt will “slow down” the system.
  - Because **context switch is expensive (switching to another process)**

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

CPU is once again over-burdened

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

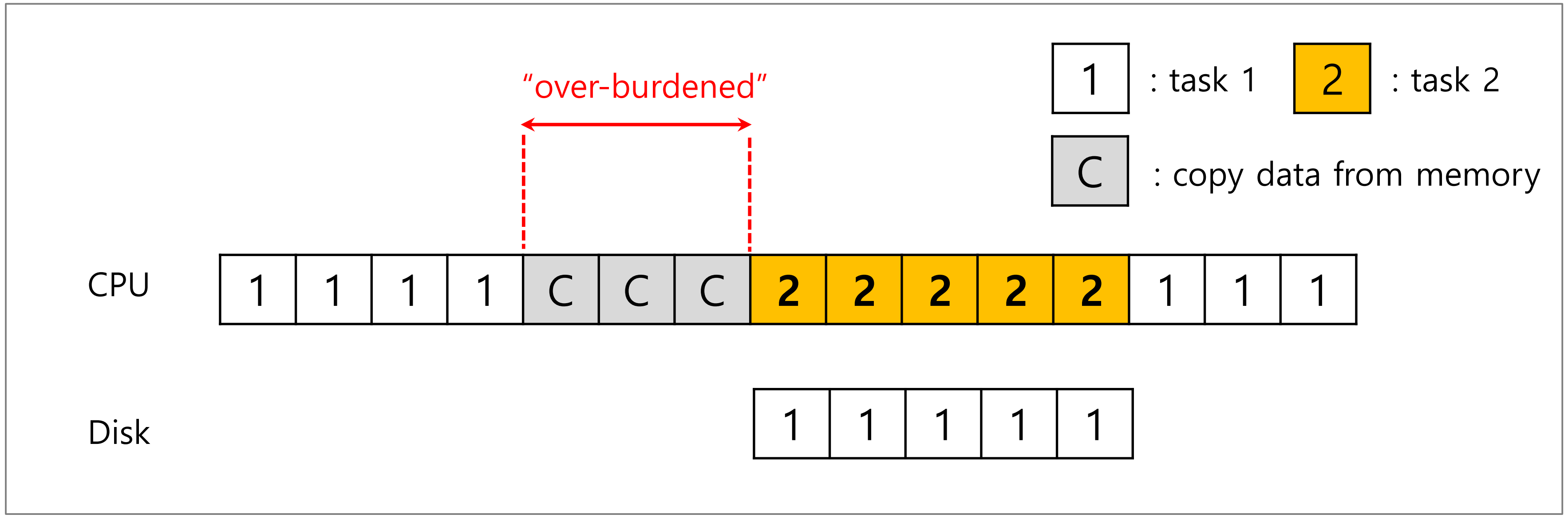


Diagram of CPU utilization

Programmed I/O vs. Direct Memory Access

- PIO (Programmed I/O):
  - CPU **directly tells device** what the data is

- DMA (Direct Memory Access):
  - CPU **leaves data in memory**
  - Device reads data directly from memory

DMA (Direct Memory Access)

- **Copy data** in memory by knowing "where the data lives in memory, how much data to copy"
- When completed, DMA raises an interrupt, I/O begins on Disk.

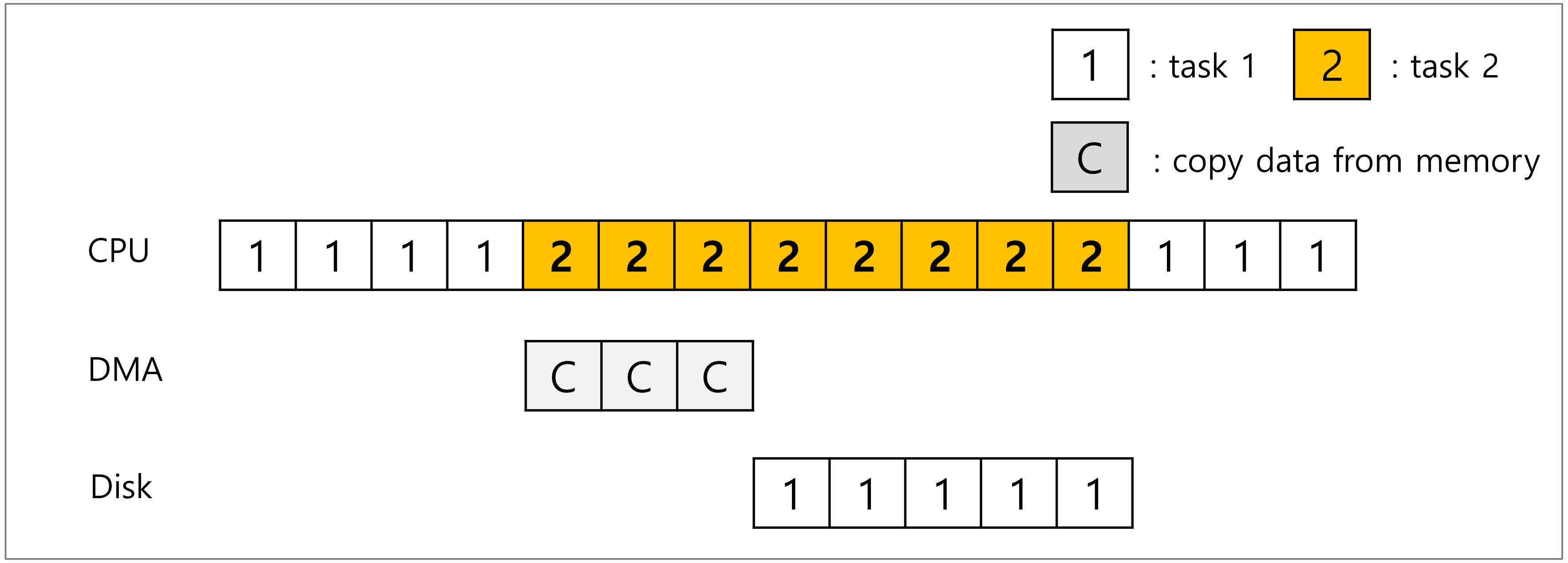
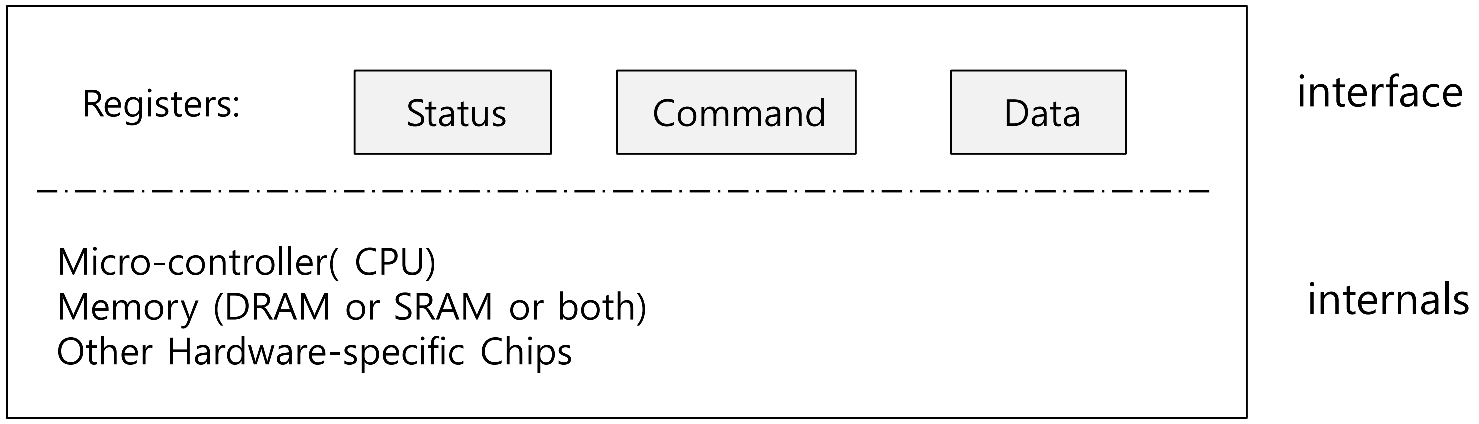


Diagram of CPU utilization by DMA

Device interaction

- How the OS communicates with the **device**?
- Solutions
  - I/O instructions: a way for the OS to send data to specific device registers.
    - Ex) `in` and `out` instructions on x86
  - memory-mapped I/O
    - Device registers available as if they were memory locations.
    - The OS `load` (to read) or `store` (to write) to the device instead of main memory.

Summary



Canonical Device

Status checks: polling *vs.* interrupts

Data: PIO *vs.* DMA

Control: special instructions *vs.* memory-mapped I/O

Variety is a Challenge

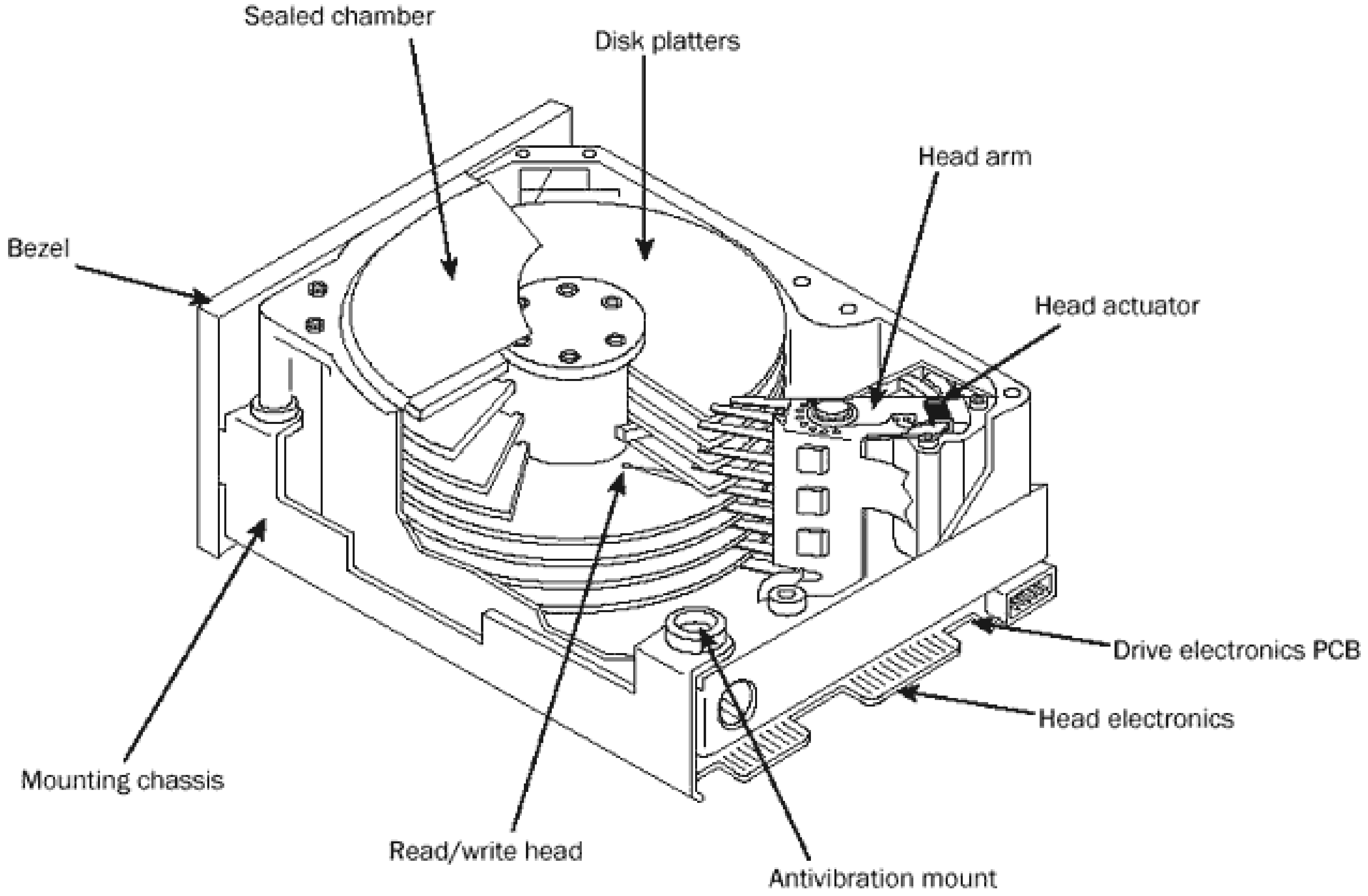
- Problem:
  - many, many devices
  - each has its own protocol
- How the OS interact with **different types of interfaces**?
  - Ex) We'd like to build a file system that worked on top of SCSI disks, IDE disks, USB keychain drivers, and so on.
- Solutions: **Abstraction (device driver)**
  - Abstraction encapsulate any specifics of device interaction.

Summary

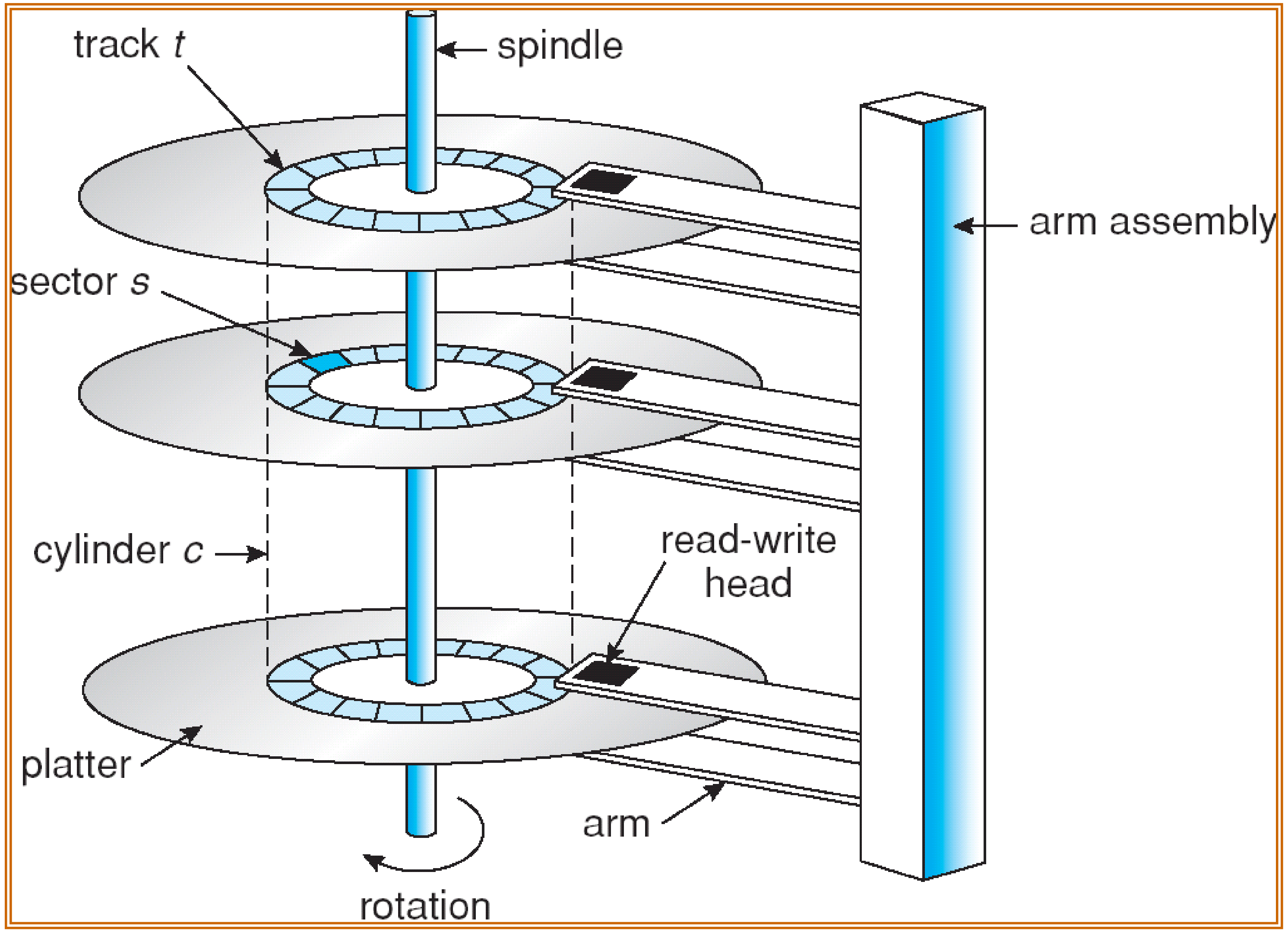
- To save the CPU cycles for IO
  - Use Interrupt
  - Use DMA
- To access the device registers
  - Memory mapped IO
  - Explicit IO instruction

HARD DISKS

Disk Schematics

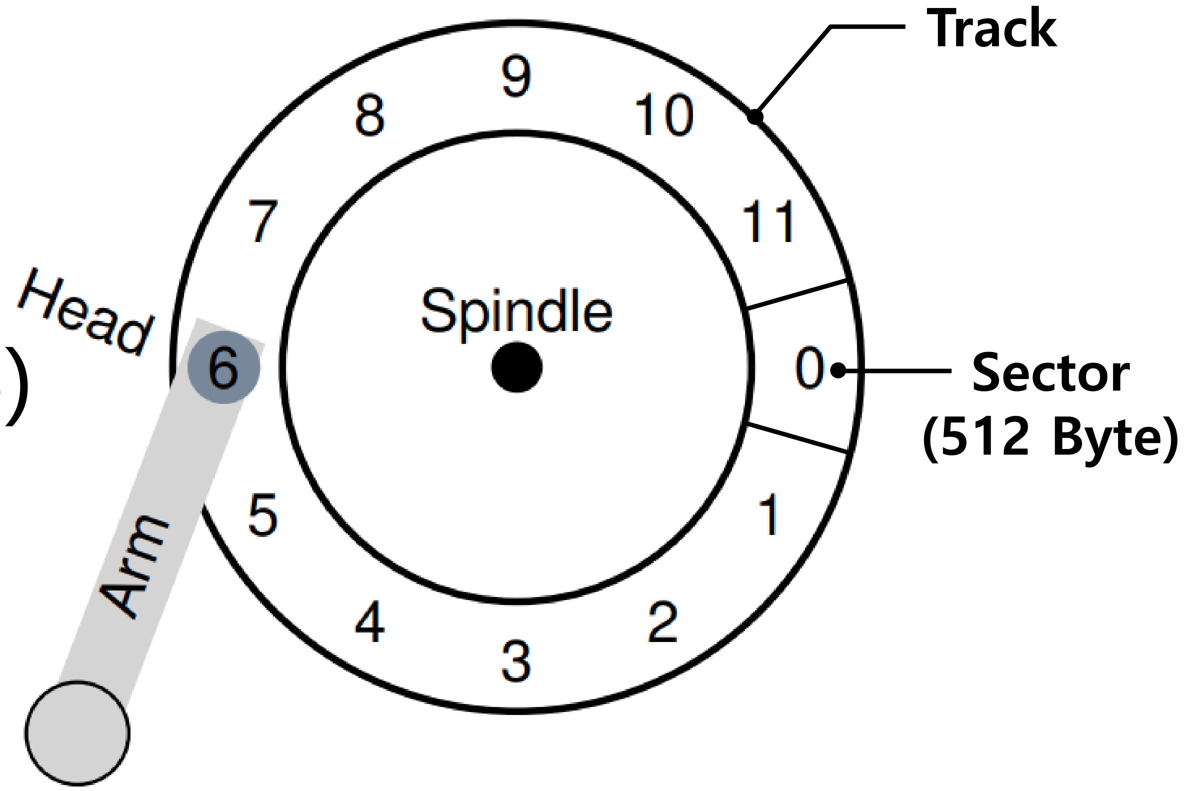


Tracks, Sectors, Cylinders



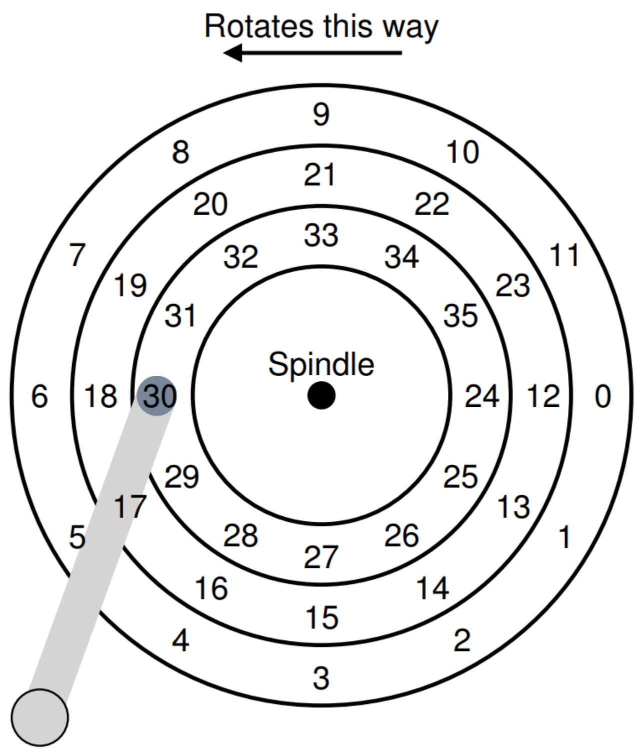
Base Geometry

- 2-30 heads (2 per platter)
  - 1 ~ 5 platters per a HDD
- Diameter: 2.5” – 14”
- Capacity: ~ 22TB (SSD: ~ 100TB)
- Sector size: 64 bytes to 8K bytes
  - Most PC disks: 512 byte sectors
- 700-20,480 tracks per surface
- 16-1,600 sectors per track



A Simple Disk Drive

- Rotation Delay

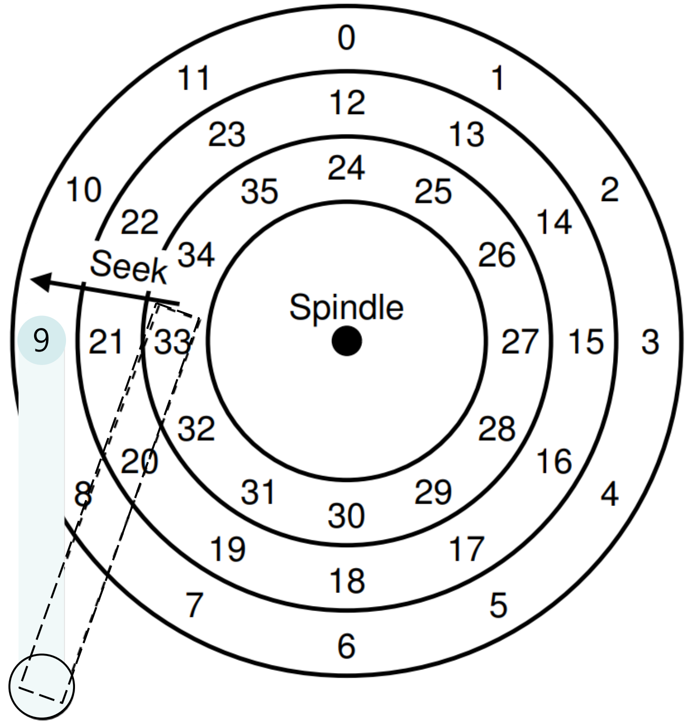


If the full rotation delay is R,

the rotation delay of (30→24) is $\frac{R}{2}$

A Simple Disk Drive (Cont.)

- Seek Time

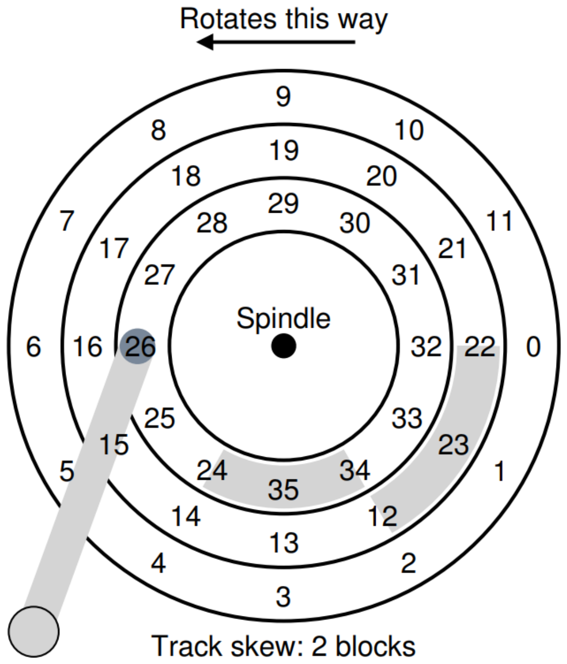


Phases of seek

Acceleration → *Coasting* → *Deceleration* → *Setting time*
(about 0.5~2 ms)

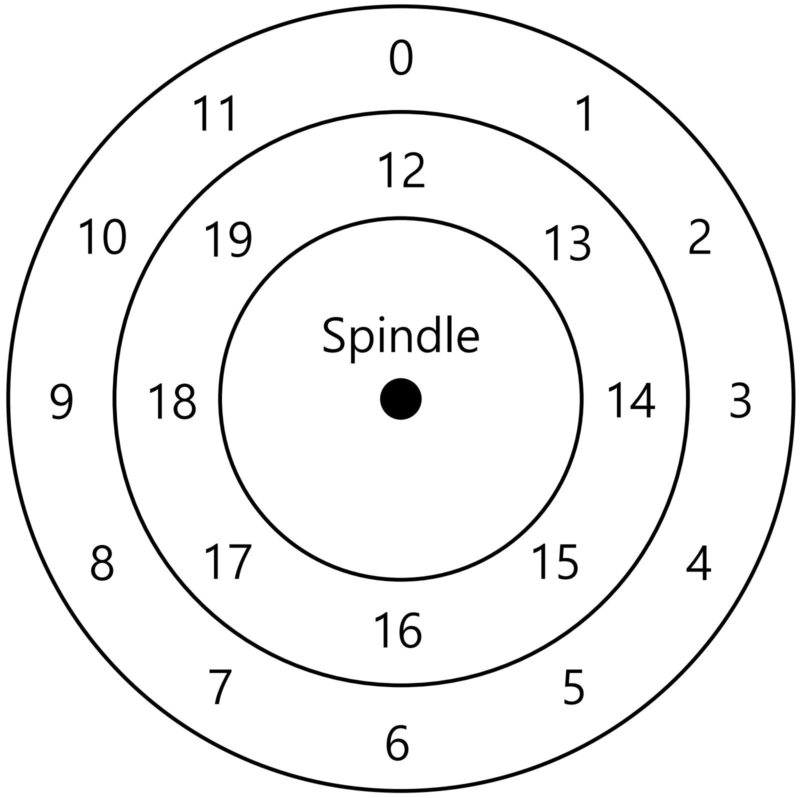
A Simple Disk Drive (Cont.)

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



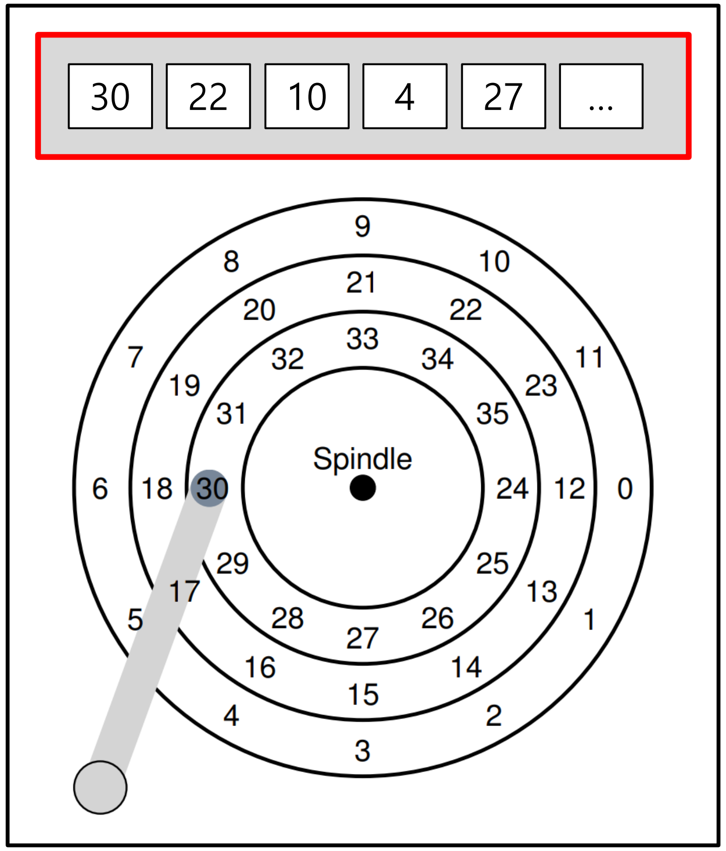
A Simple Disk Drive (Cont.)

- Multi-zoned
  - Outer tracks tend to have more sectors than inner tracks, because of result of geometry



A Simple Disk Drive (Cont.)

- Cache (Track buffer)



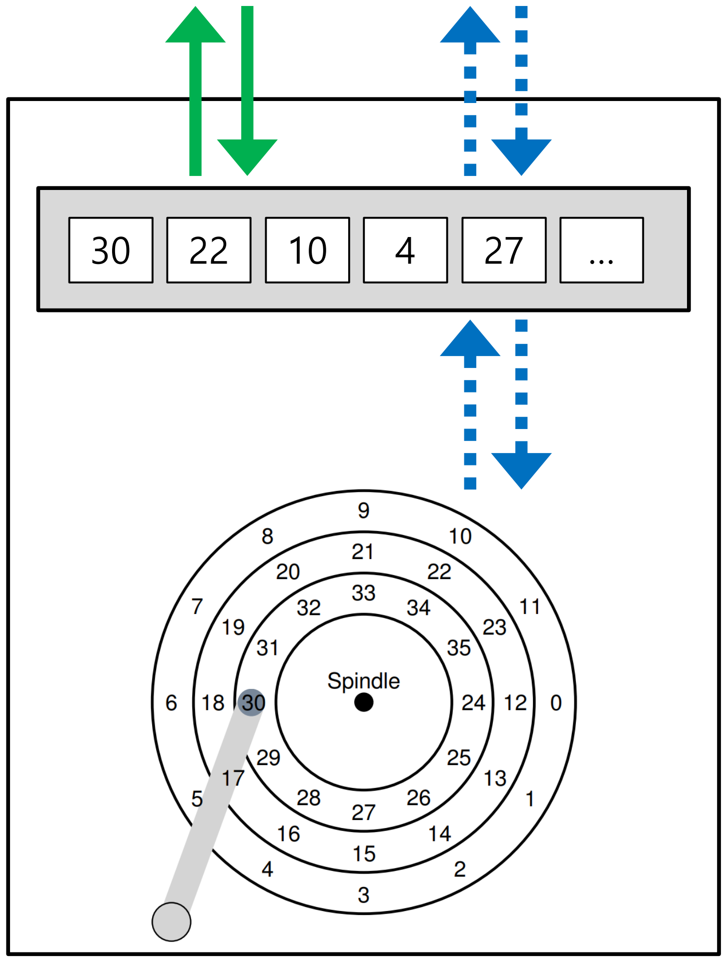
Small amount of memory
(usually around 8 or 16MB)

Hold data read from or written to the disk

Allow the drive to quickly respond to requests

A Simple Disk Drive (Cont.)

- Cache (Track buffer)



→ Write-Back

Acknowledge the write has completed when it has put the data in its memory

⇢ Write-Through

Acknowledge after the write has actually been written to disk

I/O Time: Doing The Math

I/O Time

$$T_{I/O} = T_{seek} + T_{rotation} + T_{transfer}$$

I/O Rate

$$R_{I/O} = \frac{Size_{Transfer}}{T_{I/O}}$$

Seek, Rotate, Transfer

Seek cost: Function of cylinder distance

- Not purely linear cost

Must accelerate, coast, decelerate, settle

Settling alone can take 0.5 - 2 ms

Entire seeks often takes several milliseconds

- **4 - 10 ms**

Approximate average seek distance = 1/3 max seek distance

Seek, **Rotate**, Transfer

Depends on rotations per minute (RPM)

– 7200 RPM is common, 15000 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**

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 us

Workload Performance

So…

- seeks are slow
- rotations are slow
- transfers are fast

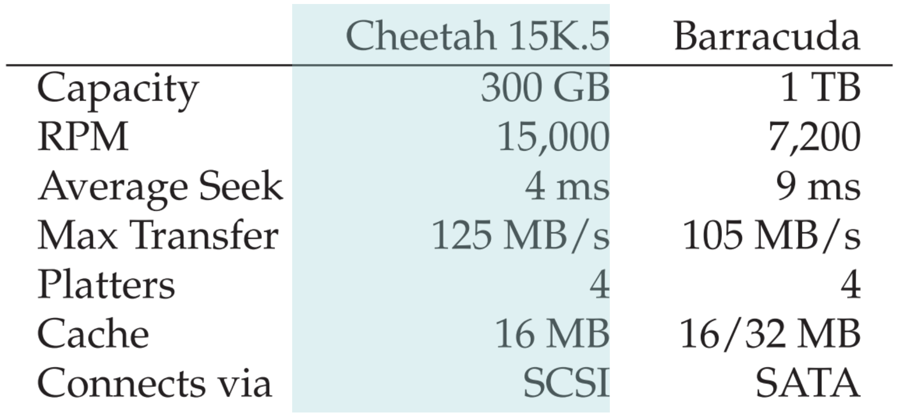
What kind of workload is fastest for disks?

Sequential: access sectors in order (transfer dominated)

Random: access sectors arbitrarily (seek+rotation dominated)

I/O Time: Doing The Math (Cont.)

- 4KB Random Write Example



	Cheetah 15K.5	Barracuda
Capacity	300 GB	1 TB
RPM	15,000	7,200
Average Seek	4 ms	9 ms
Max Transfer	125 MB/s	105 MB/s
Platters	4	4
Cache	16 MB	16/32 MB
Connects via	SCSI	SATA

T_{seek} = 4ms

$T_{rotation}$ = 15,000 RPM(= 250RPS = 4ms / 1 rotation) / 2
= 2ms

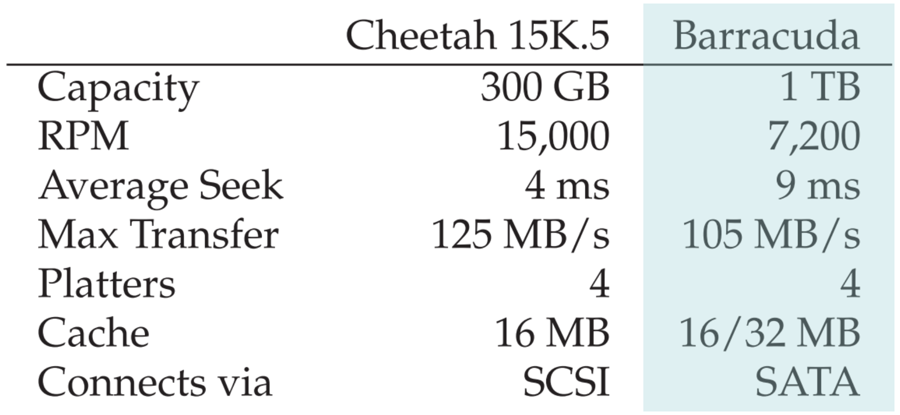
$T_{transfer}$ = 4KB / 125(MB/s)
= 30us

$T_{I/O}$ = 4ms + 2ms + 30us ≒ 6ms

$R_{I/O}$ = 4KB / 6ms = 0.66MB/s

I/O Time: Doing The Math (Cont.)

- 4KB Random Write Example (Cont.)



	Cheetah 15K.5	Barracuda
Capacity	300 GB	1 TB
RPM	15,000	7,200
Average Seek	4 ms	9 ms
Max Transfer	125 MB/s	105 MB/s
Platters	4	4
Cache	16 MB	16/32 MB
Connects via	SCSI	SATA

T_{seek} = 9ms

$T_{rotation}$ = 7,200 RPM(= 120RPS = 8ms / 1 rotation) / 2
= 4ms

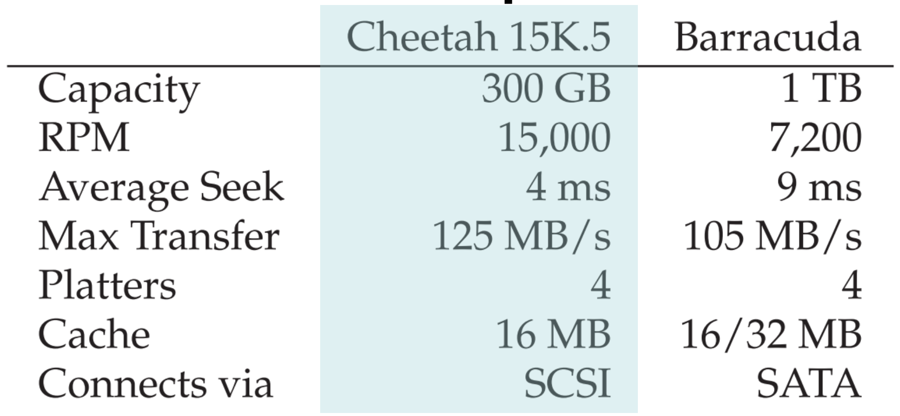
$T_{transfer}$ = 4KB / 105(MB/s)
= 38us

$T_{I/O}$ = 9ms + 4ms + 38us ≒ 13ms

$R_{I/O}$ = 4KB / 13ms = 0.31MB/s

I/O Time: Doing The Math (Cont.)

- Sequential Write Example



	Cheetah 15K.5	Barracuda
Capacity	300 GB	1 TB
RPM	15,000	7,200
Average Seek	4 ms	9 ms
Max Transfer	125 MB/s	105 MB/s
Platters	4	4
Cache	16 MB	16/32 MB
Connects via	SCSI	SATA

T_{seek} = 4ms

$T_{rotation}$ = 15,000 RPM(= 250RPS = 4ms / 1 rotation) / 2
= 2ms

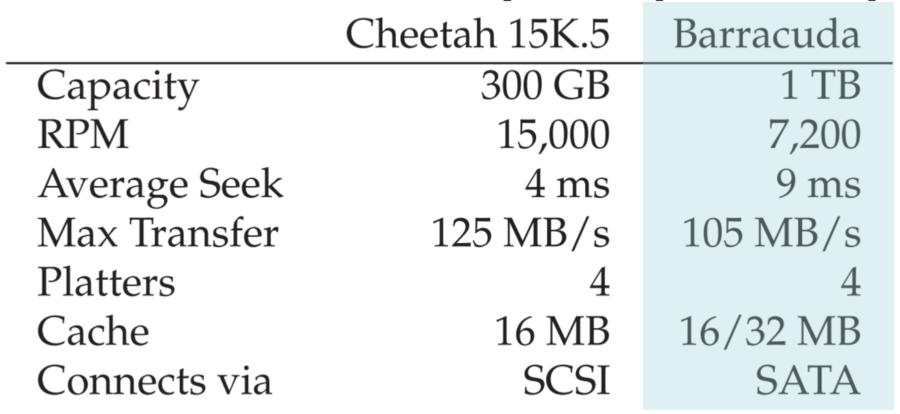
$T_{transfer}$ = 100MB / 125(MB/s)
= 800ms

$T_{I/O}$ = 4ms + 2ms + 800ms = 806ms ≒ 800ms

$R_{I/O}$ = 100MB / 800ms = 125MB/s

I/O Time: Doing The Math (Cont.)

- Sequential Write Example (Cont.)



	Cheetah 15K.5	Barracuda
Capacity	300 GB	1 TB
RPM	15,000	7,200
Average Seek	4 ms	9 ms
Max Transfer	125 MB/s	105 MB/s
Platters	4	4
Cache	16 MB	16/32 MB
Connects via	SCSI	SATA

T_{seek} = 9ms

$T_{rotation}$ = 7,200 RPM(= 120RPS = 8ms / 1 rotation) / 2
= 4ms

$T_{transfer}$ = 100MB / 105(MB/s)
= 950ms

$T_{I/O}$ = 9ms + 4ms + 950ms = 963ms ≒ 950ms

$R_{I/O}$ = 100MB / 950ms = 105MB/s

I/O Time: Doing The Math (Cont.)

	Cheetah	Barracuda
$R_{I/O}$ Random	0.66 MB/s	0.31 MB/s
$R_{I/O}$ Sequential	125 MB/s	105 MB/s

Performance vs Capacity

	Cheetah	Barracuda
$R_{I/O}$ Random	0.66 MB/s	0.31 MB/s
$R_{I/O}$ Sequential	125 MB/s	105 MB/s

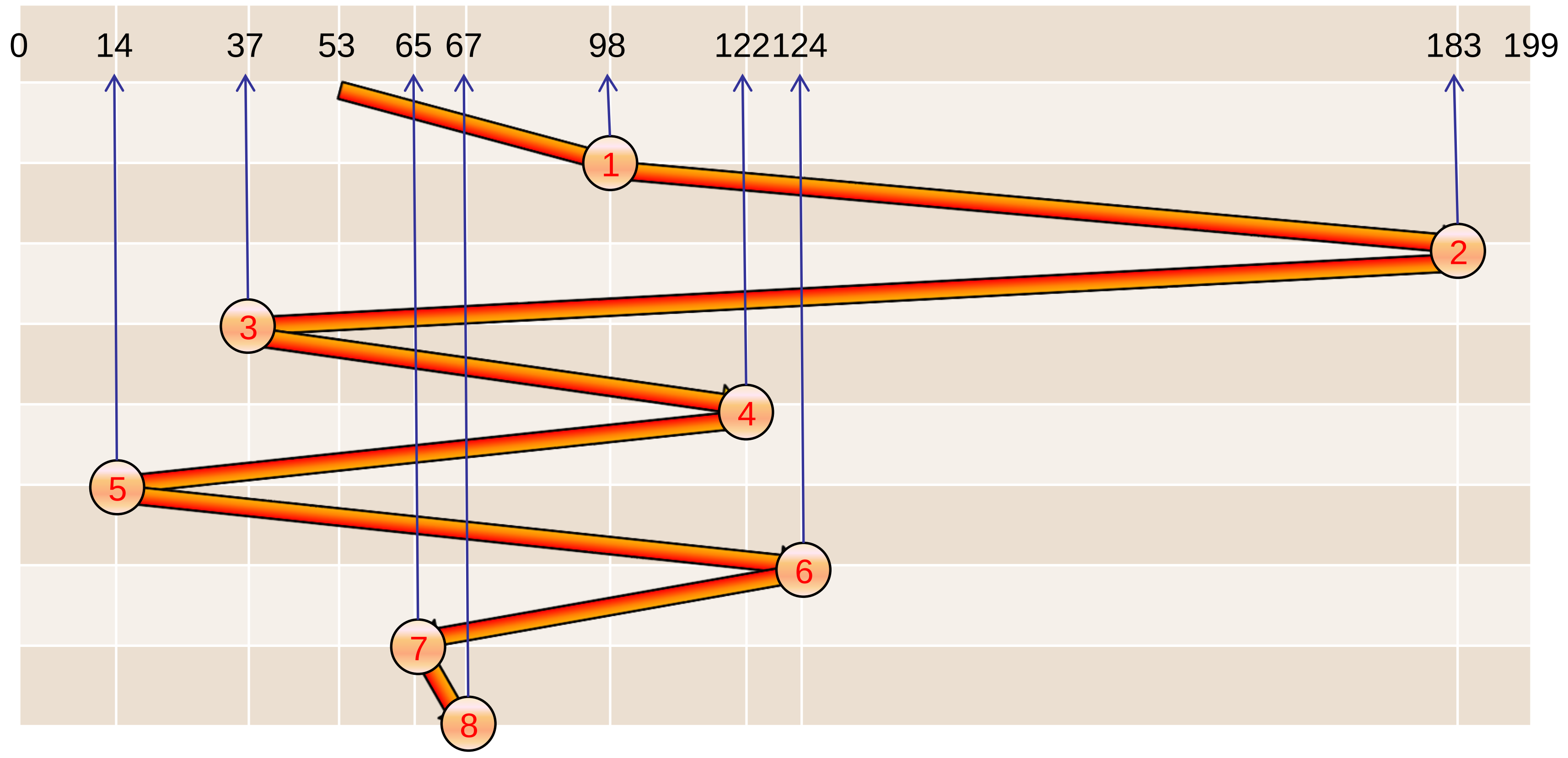
Random Write vs Sequential Write

Disk Scheduling: FCFS

How to order the services for the requests in the queue?

Illustration shows total head movement of **640** cylinders.

Queue = 98, 183, 37, 122, 14, 124, 65, 67
Head starts at 53

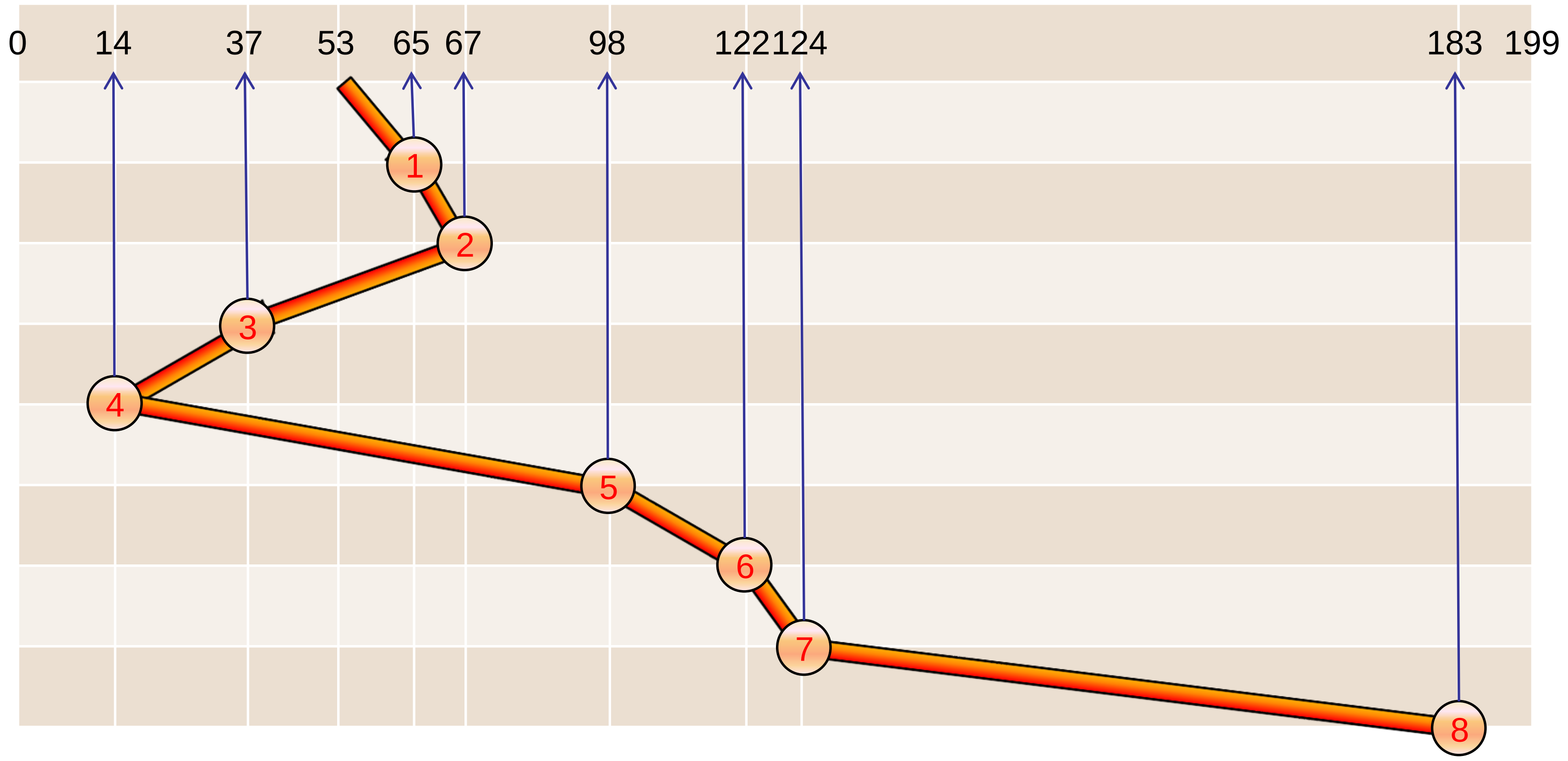


SSTF(Shortest Seek Time First)

- Selects the request with the minimum seek time from the current head position.
- SSTF scheduling is a form of SJF scheduling; may cause starvation of some requests.
- Illustration shows total head movement of **236** cylinders.

SSTF (Cont.)

Queue = 98, 183, 37, 122, 14, 124, 65, 67
Head starts at 53

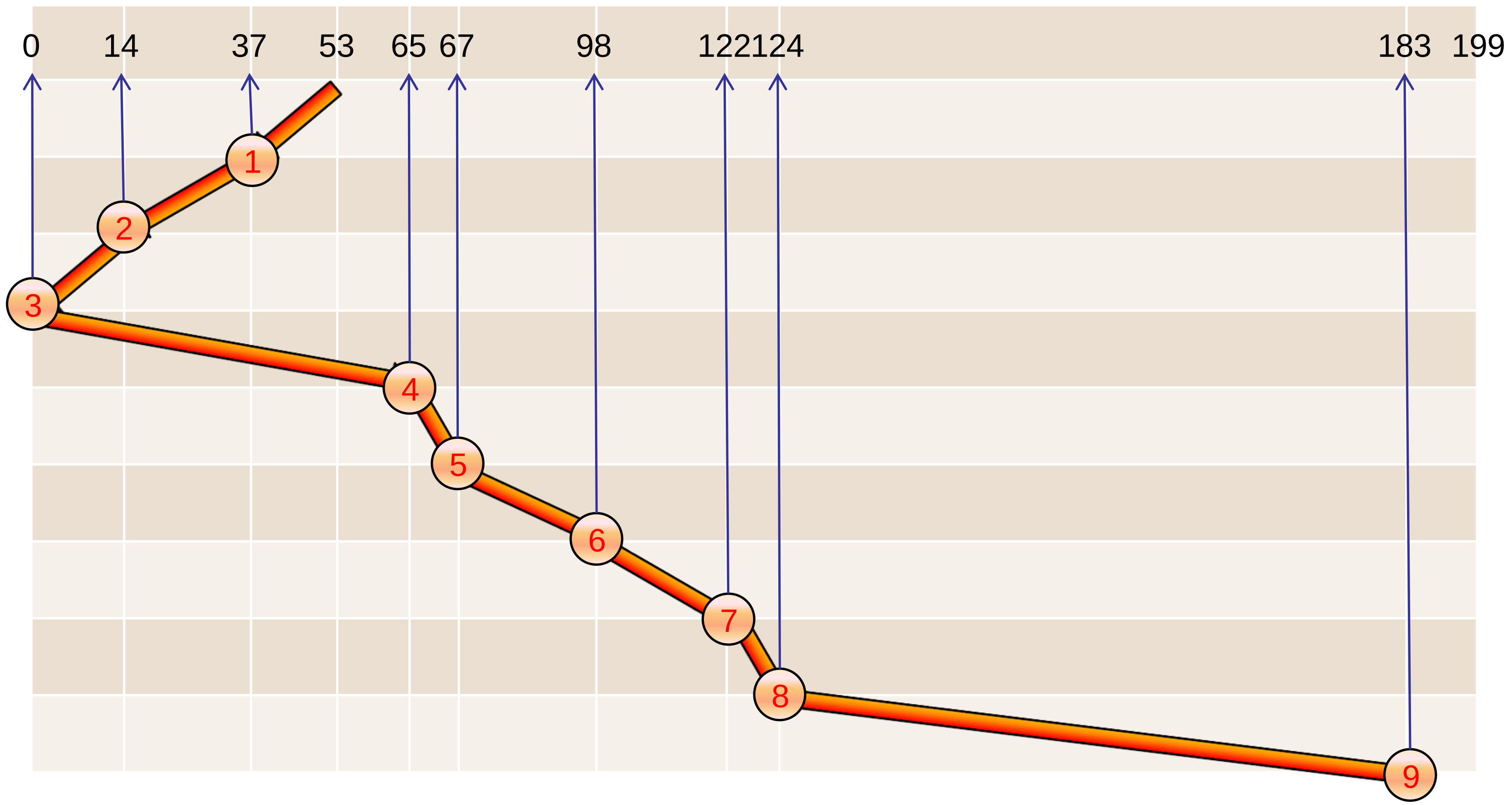


SCAN

- The disk arm **starts at one end of the disk, and moves toward the other end**, servicing requests until it gets to the other end of the disk, where the head movement is reversed and servicing continues.
- Sometimes called the ***elevator algorithm***.
- Illustration shows total head movement of **208** cylinders.

SCAN (Cont.)

Queue = 98, 183, 37, 122, 14, 124, 65, 67
Head starts at 53



HDD DEVICE DRIVER

Basic Interface

- Disk has a sector-addressable address space
  - Appears as an array of sectors
- Main operations: reads + writes to sectors
- Mechanical (slow) nature makes management “interesting”

Files vs Disks

File Abstraction

- Byte oriented
- Names
- Access protection
- Consistency guarantees

Disk Abstraction

- Block oriented
- Block #s
- No protection
- No guarantees beyond block write

Storage Stack

application

file system

scheduler

driver

hard drive

build common interface
on top of all HDDs

IDE Device Driver

- Registers
  - Control register (8 bit)
  - Command block registers
  - Status register (8 bit)
  - Error register (8 bit)

The IDE Interface

- Control Register (1 Byte):

  Address 0x3F6 = 0x08 (0000 1RE0): R=reset, E="Enable Interrupt", E=0 means "enable interrupt"
- Command Block Registers:

  Address 0x1F0 = Data Port (128 Byte)

  Address 0x1F1 = Error (1 Byte)

  Address 0x1F2 = Sector Count (1 Byte)

  Address 0x1F3 = LBA low byte (1 Byte)

  Address 0x1F4 = LBA mid byte (1 Byte)

  Address 0x1F5 = LBA hi byte (1 Byte)

  Address 0x1F6 = 1B1D TOP4LBA: B=LBA, D=drive

  Address 0x1F7 = Command/status(1 Byte)
- Status Register (Address 0x1F7): (1 Byte)

7	6	5	4	3	2	1	0
BUSY	READY	FAULT	SEEK	DRQ	CORR	IDDEX	ERROR

The IDE Interface (Cont.)

- Error Register (Address 0x1F1): (check when Status ERROR==1)

7	6	5	4	3	2	1	0
BBK	UNC	MC	IDNF	MCR	ABRT	T0NF	AMNF

BBK	=	Bad Block,
UNC	=	Uncorrectable data error,
MC	=	Media Changed,
IDNF	=	ID mark Not Found,
MCR	=	Media Change Requested,
ABRT	=	Command aborted,
T0NF	=	Track 0 Not Found,
AMNF	=	Address Mark Not Found

The basic protocol of IDE device driver

- Wait for the drive to be ready.
- Write parameters to command registers.
  - Sector count
  - logical block address (LBA) of sector
  - Driver number
- Issue read/write to command register.
  - Read or Write
  - For write command, transfer data
- Handle interrupt.
- Handle Error.

IDE device driver in xv6

```
void ide_rw(struct buf *b) {
        acquire(&ide_lock);
1       for (struct buf **pp = &ide_queue; *pp; pp=&(*pp)->qnext) ;
        *pp = b;
2       if (ide_queue == b)              // if q is empty,
                ide_start_request(b);    // send request to disk.
3       while ((b->flags & (B_VALID|B_DIRTY)) != B_VALID)
                sleep(b, &ide_lock);    // wait for completion
        release(&ide_lock);
}
```

Mechanism: add the buffer to the `ide_queue` and perform IO.

1. Enqueu the buffer to the `ide_queue.`
2. If the buffer is the only entry in the `ide_queue`, start the IO right away.
3. Wait for the completion of the IO.
    1. If the IO is READ, interrupt handler resets the `B_VALID` flag.
    2. If the IO is write, interrupt handler resets the `B_DIRTY` flag.

Device driver

```
static void ide_start_request(struct buf *b) {
   ide_wait_ready();
   outb(0x3f6, 0); // generate interrupt
   outb(0x1f2, 1); // the count of sectors
   outb(0x1f3, b->sector & 0xff);
   outb(0x1f4, (b->sector >> 8) & 0xff);
   outb(0x1f5, (b->sector >> 16) & 0xff);
   outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((b-
   >sector>>24)&0x0f));
   if (b->flags & B_DIRTY){
           outb(0x1f7, IDE_CMD_WRITE); // WRITE
           outsl(0x1f0, b->data, 512/4); // transfer data
   } else {
      outb(0x1f7, IDE_CMD_READ); // READ
   }
}
```

Device driver

```
static void ide_start_request(struct buf *b) {
        ide_wait_ready();
        ...
}
```

Call

```
static int ide_wait_ready() {
    int r;
    while (((r = inb(0x1f7)) & IDE_BSY) || !(r & IDE_DRDY))
        ;
 }
```

1. Read Status Register(0x1F7) and save it to r.
2. Check if the drive is the busy. (IDE_BSY).
3. Check if the drive is ready .
4. If the device is not busy and ready, get out of the while loop.

# Status Register is also used as a command register. The device uses it to store the status of the register. The host uses it to store the command.

Write Command and the data transfer

```
static void ide_start_request(struct buf *b) {
    ...
    outb(0x1f3, b->sector & 0xff);
    outb(0x1f4, (b->sector >> 8) & 0xff);
    outb(0x1f5, (b->sector >> 16) & 0xff);
    outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((b->sector>>24)&0x0f));
    if (b->flags & B_DIRTY){
            outb(0x1f7, IDE_CMD_WRITE); // WRITE
            outsl(0x1f0, b->data, 512/4); // transfer data
    }
```

1. If the DIRTY bit is set, perform write. (In xv6, device driver performs write only when the buffer is dirty.)
2. Set the write command at the Command Register (0x1F7).
3. Write the data to the Data Register (0x1F0).

Read command

```
static void ide_start_request(struct buf *b) {
    ...
    outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((b->sector>>24)&0x0f));
    if (b->flags & B_DIRTY){
            outb(0x1f7, IDE_CMD_WRITE); // WRITE
            outsl(0x1f0, b->data, 512/4); // transfer data
    } else {
        outb(0x1f7, IDE_CMD_READ); // READ
    }
}
```

1. If the buffer is not dirty, perform read.
2. Set the command READ at Command Register (0x1F7).
3. Perform read.