# Persistence: I/O devices

**Questions answered in this lecture:**

How does the OS **interact** with I/O devices (check status, send data+control)?

What is a **device driver**?

What are the components of a **hard disk drive**?

How can you calculate **sequential** and **random throughput** of a disk?

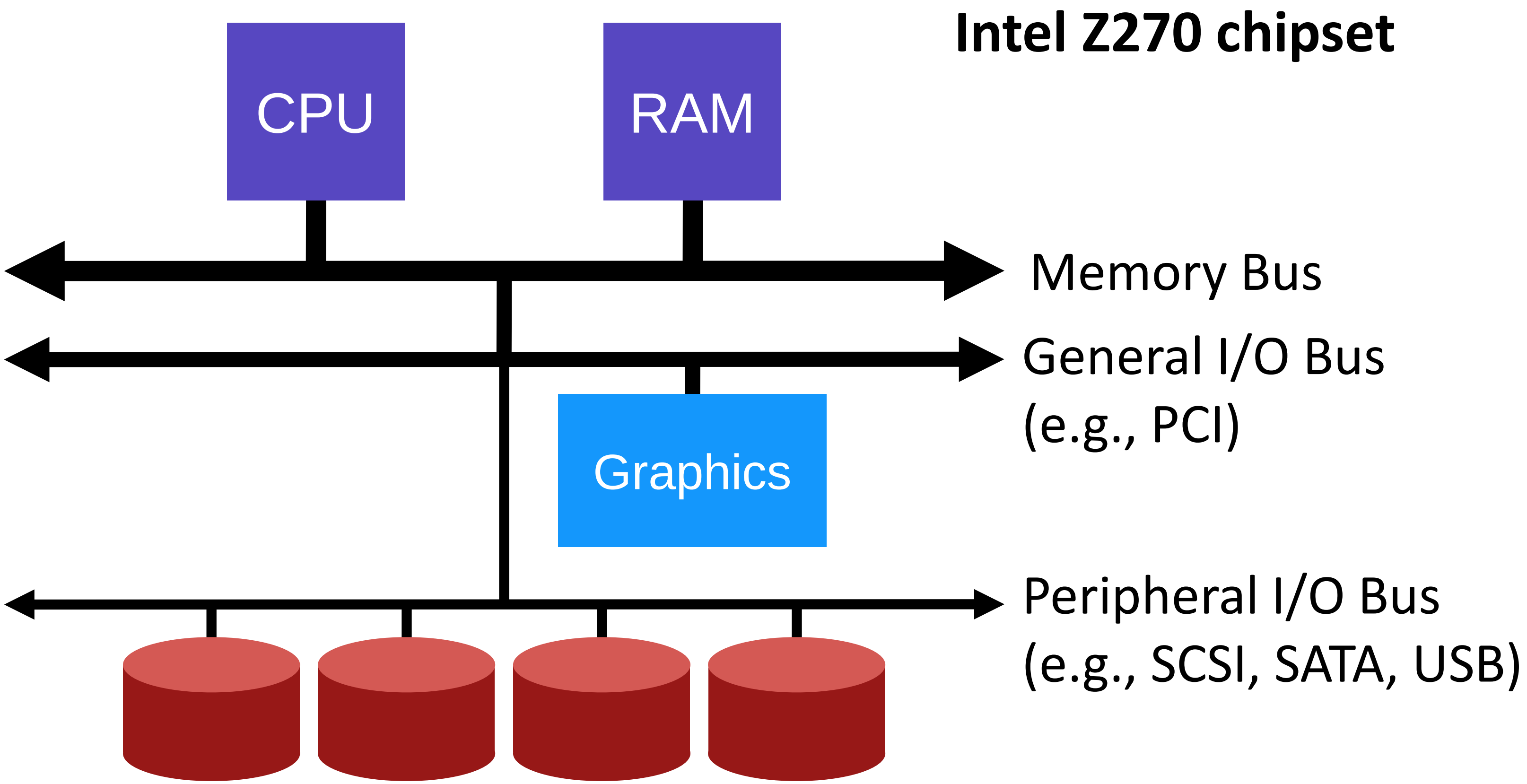
What algorithms are used to **schedule I/O** requests?

# Motivation

- **What good is a computer without any I/O devices?**
  - keyboard, display, disks

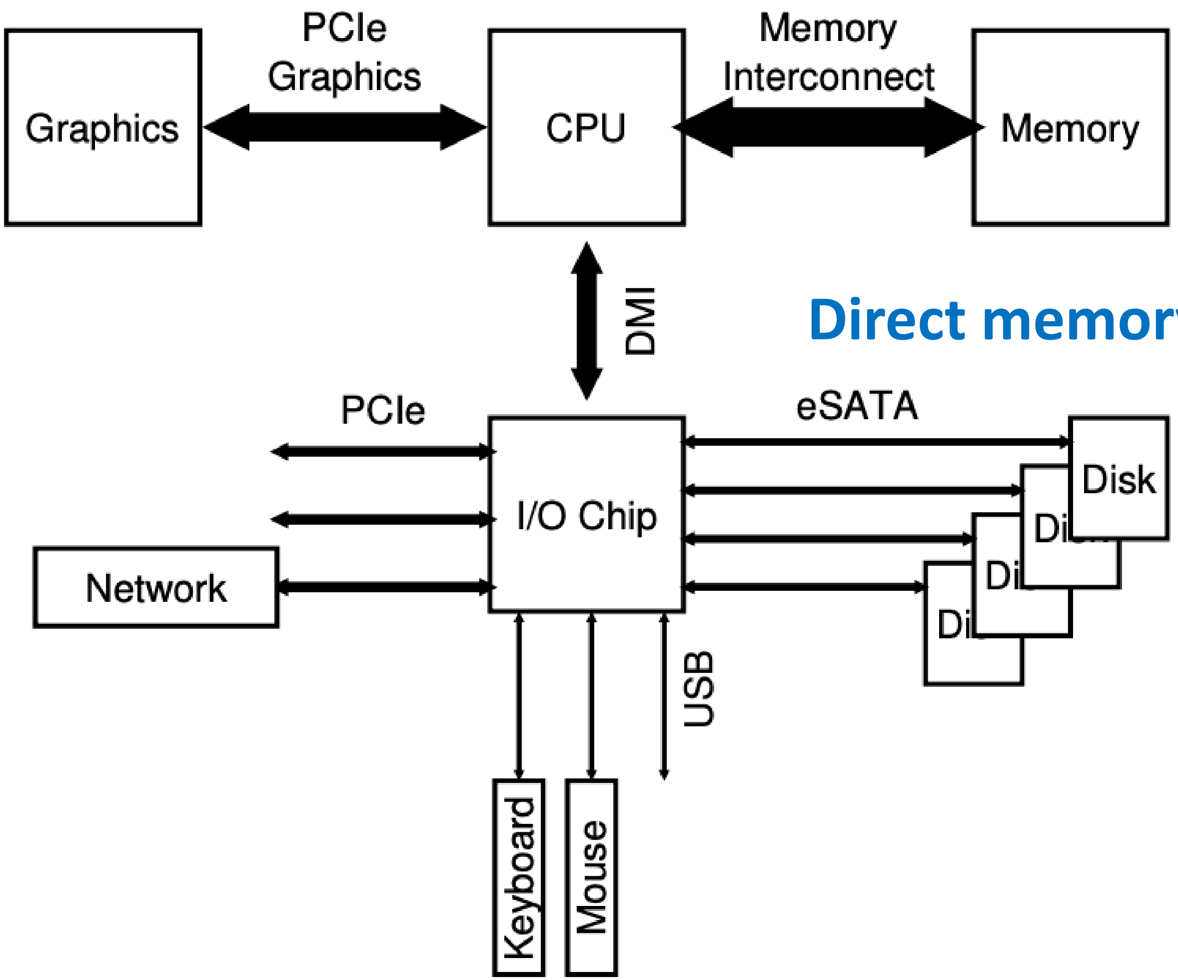
- **We want:**
  - H/W that will let us plug in different devices
  - OS that can interact with different combinations

# Hardware support for I/O




Why use hierarchical buses?

# Hardware support for I/O




Direct memory interface

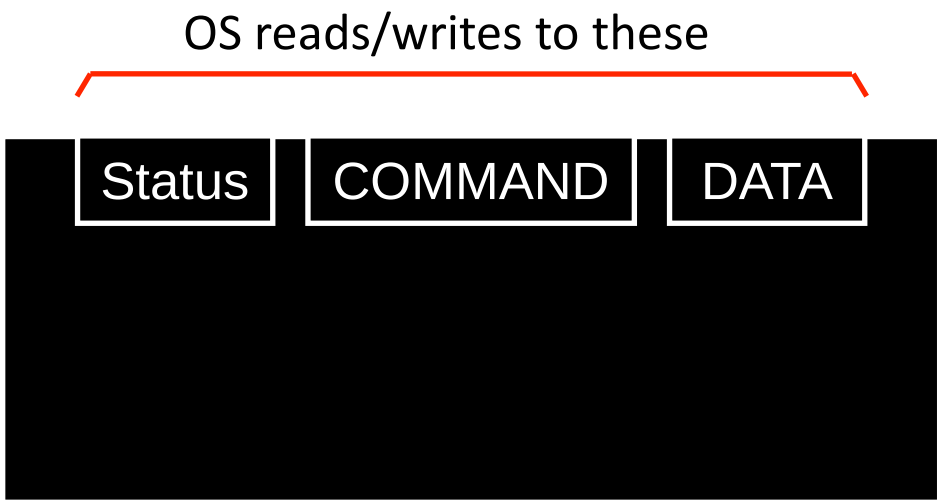
Why use hierarchical buses?

# Why hierarchical buses?

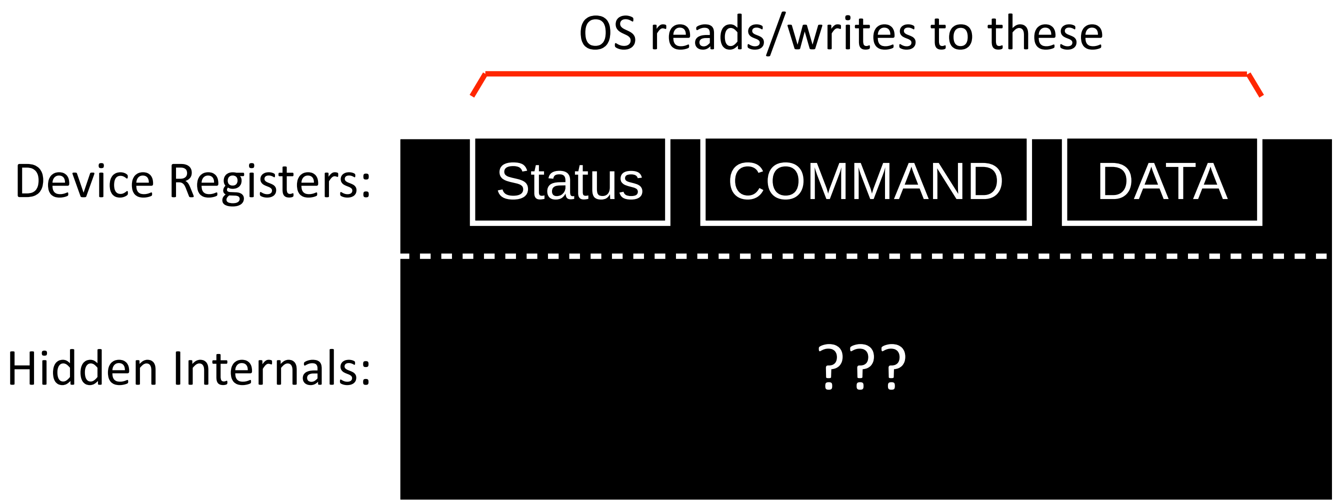
- **Physics and cost**
- **Faster bus**
  - The faster a bus is, the shorter it must be
  - no room to plug more devices
  - costly
- **Slower bus**
  - no demand for higher speed
  - more devices

# Canonical Device

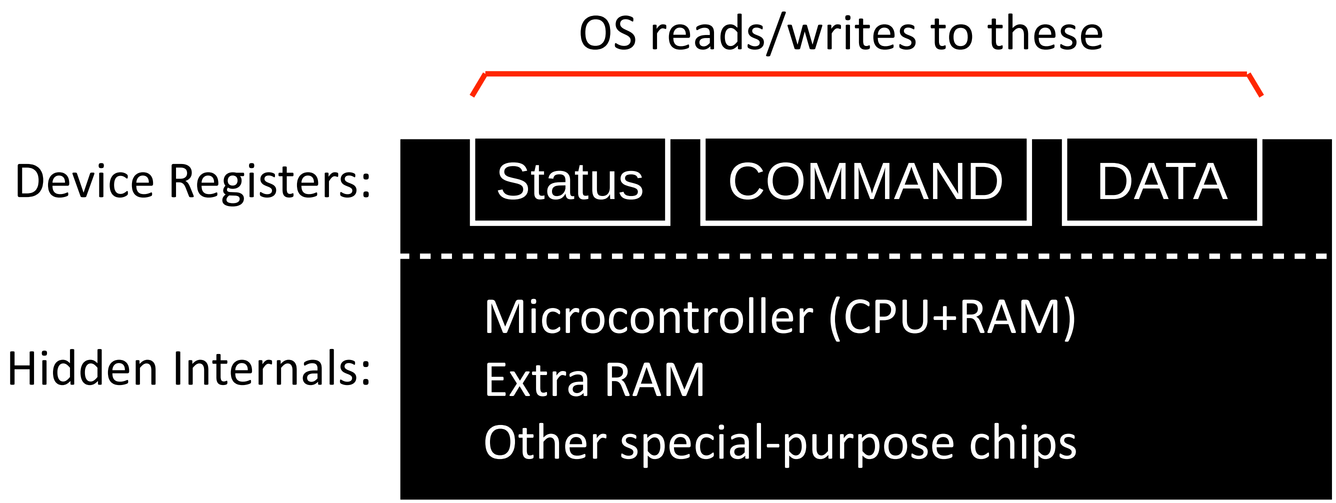
Device Registers:



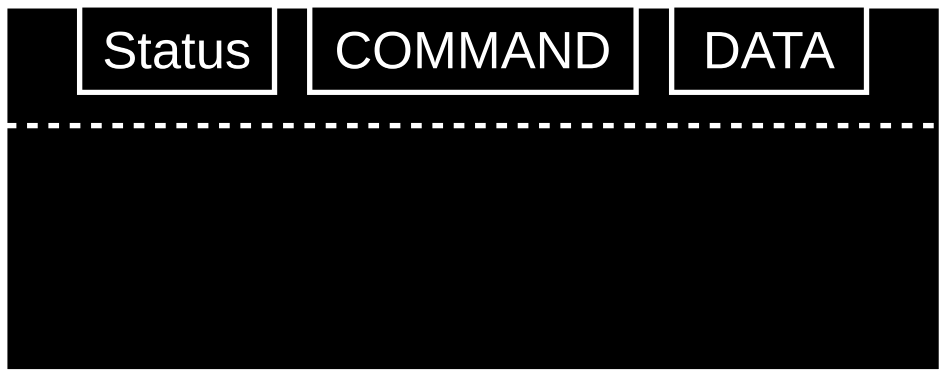
# Canonical Device



# Canonical Device



# Example Write Protocol




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

CPU:

Disk:

```
while (STATUS == BUSY)            // 1
    ;
Write data to DATA register       // 2
Write command to COMMAND register  // 3
while (STATUS == BUSY)            // 4
    ;
```

CPU: A

Disk: C

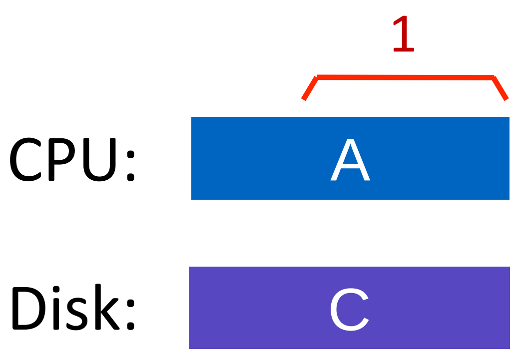
```
while (STATUS == BUSY)          // 1
    ;
Write data to DATA register      // 2
Write command to COMMAND register  // 3
while (STATUS == BUSY)          // 4
    ;
```

A wants to do I/O

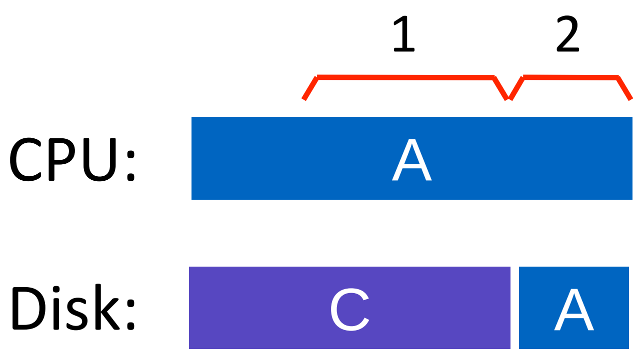
CPU: A

Disk: C

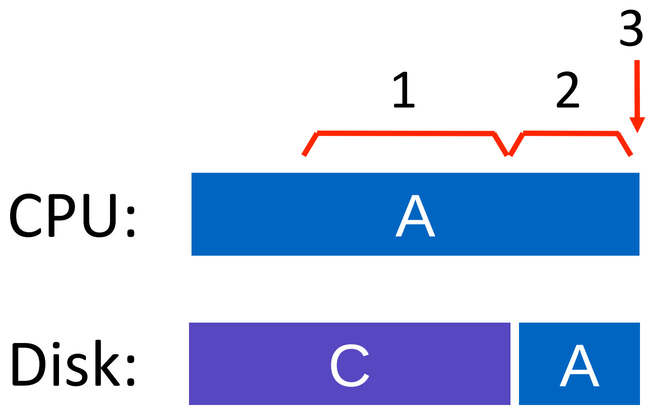
```
while (STATUS == BUSY)          // 1
    ;
Write data to DATA register       // 2
Write command to COMMAND register  // 3
while (STATUS == BUSY)          // 4
    ;
```



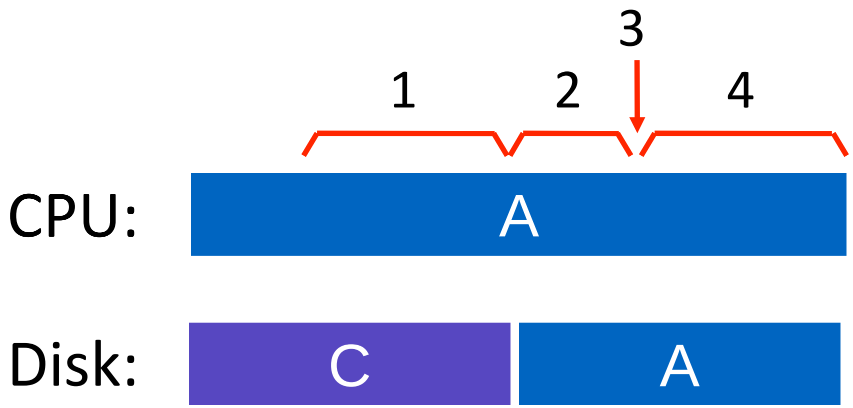

```
while (STATUS == BUSY)           // 1
    ;
Write data to DATA register       // 2
Write command to COMMAND register  // 3
while (STATUS == BUSY)           // 4
    ;
```



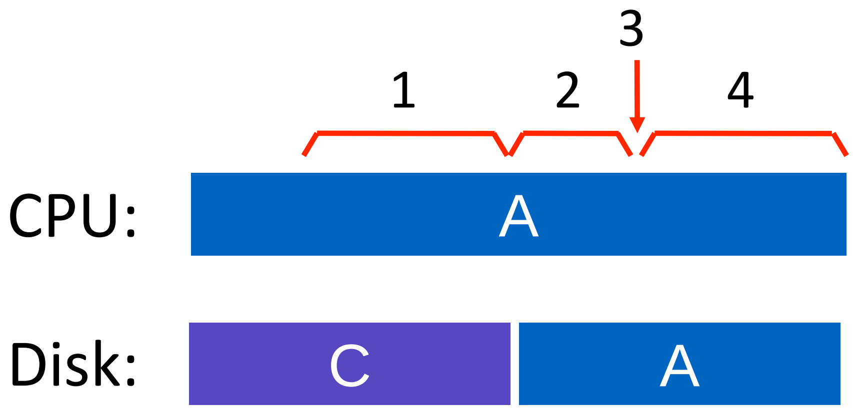

```
while (STATUS == BUSY)          // 1
    ;
Write data to DATA register     // 2
Write command to COMMAND register  // 3
while (STATUS == BUSY)          // 4
    ;
```



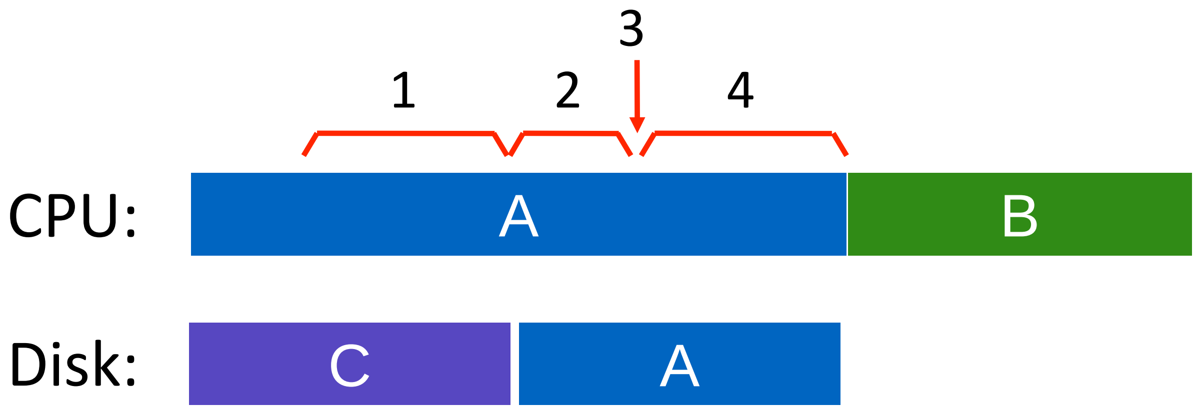

```
while (STATUS == BUSY)          // 1
    ;
Write data to DATA register      // 2
Write command to COMMAND register  // 3
while (STATUS == BUSY)          // 4
    ;
```




```
while (STATUS == BUSY)          // 1
    ;
Write data to DATA register       // 2
Write command to COMMAND register  // 3
while (STATUS == BUSY)          // 4
    ;
```



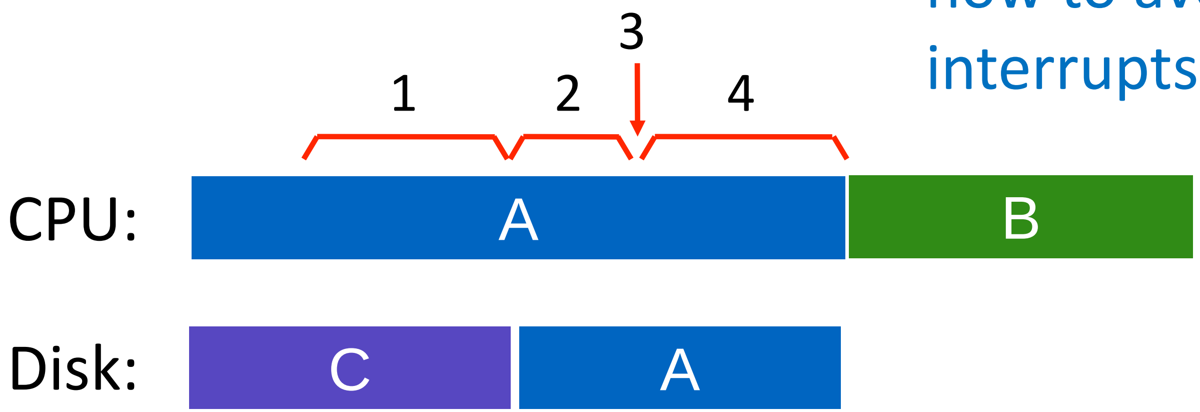

```
while (STATUS == BUSY)          // 1
    ;
Write data to DATA register      // 2
Write command to COMMAND register  // 3
while (STATUS == BUSY)          // 4
    ;
```




```
while (STATUS == BUSY)          // 1
    ;
Write data to DATA register       // 2
Write command to COMMAND register  // 3
while (STATUS == BUSY)          // 4
    ;
```

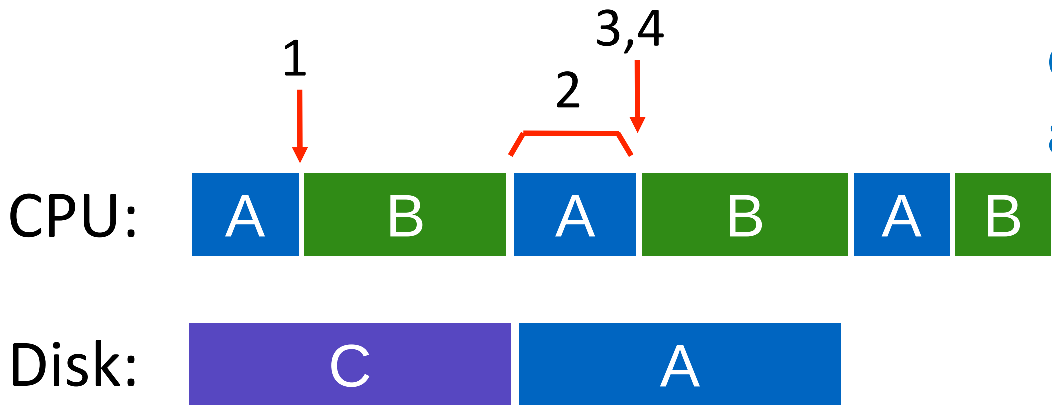
how to avoid spinning? interrupts!

how to avoid spinning? interrupts!




```
while (STATUS == BUSY)          // 1
        sleep & wait for interrupt;
Write data to DATA register       // 2
Write command to COMMAND register  // 3
while (STATUS == BUSY)          // 4
        sleep & wait for interrupt;
```

## A CPU-Disk pipieline that enables computation and I/O overlapping

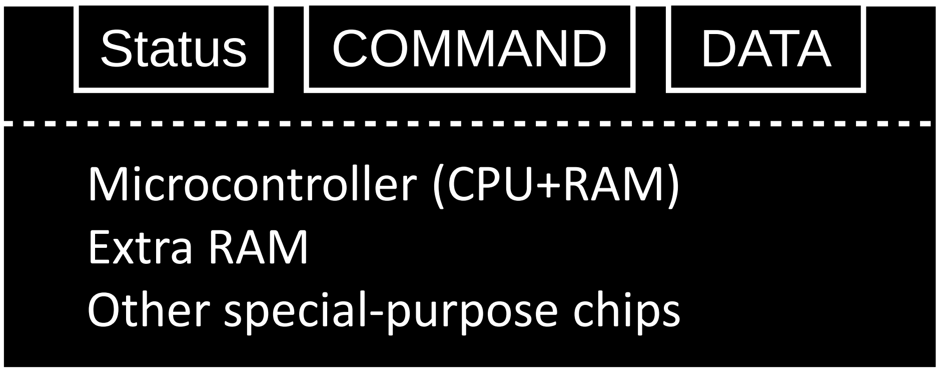



```
while (STATUS == BUSY)          // 1
        sleep & wait for interrupt;
Write data to DATA register       // 2
Write command to COMMAND register  // 3
while (STATUS == BUSY)          // 4
        sleep & wait for interrupt;
```

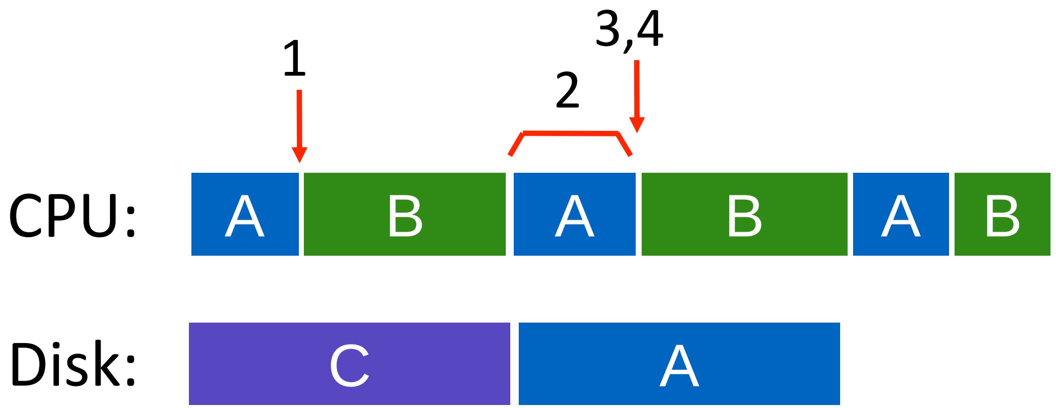
# Interrupts vs. Polling

- **Are interrupts ever worse than polling?**
  - The overhead of task switching is expensive
- **Fast device: Better to spin than take interrupt overhead**
  - Device time unknown? Hybrid approach (spin then use interrupts)
- **Flood of interrupts arrive**
  - Can lead to livelock (always handling interrupts)
  - Better to ignore interrupts while make some progress handling them
- **Other improvement**
  - Interrupt coalescing (batch together several interrupts)

# Protocol Variants




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




```
while (STATUS == BUSY)          // 1
        sleep & wait for interrupt;
Write data to DATA register      // 2
Write command to COMMAND register  // 3
while (STATUS == BUSY)          // 4
        sleep & wait for interrupt;
```

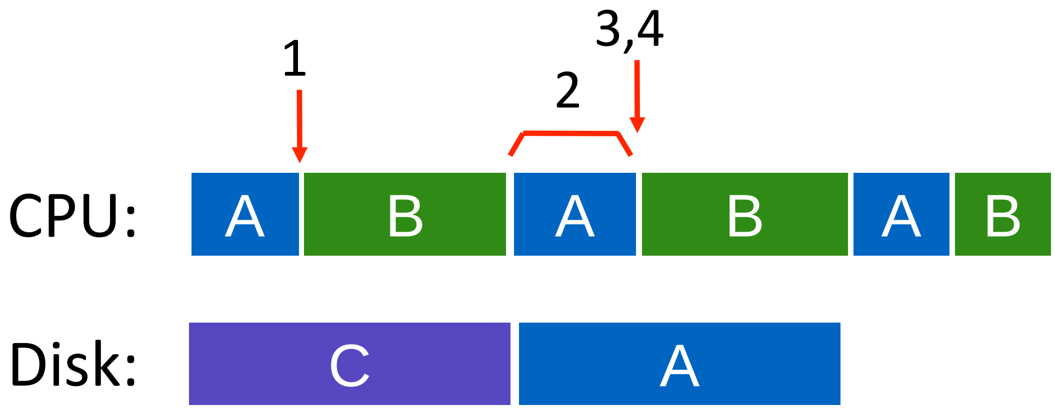
What else can we optimize?

Data Transfer!

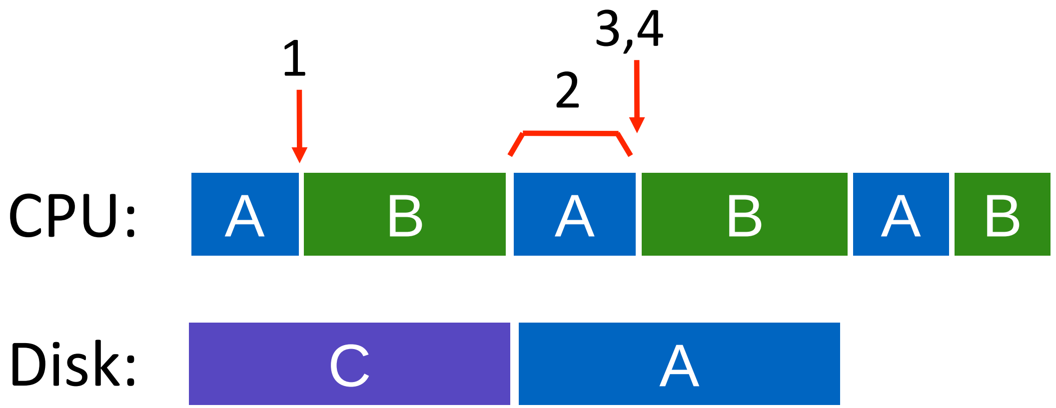
# Programmed I/O vs. Direct Memory Access

- **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



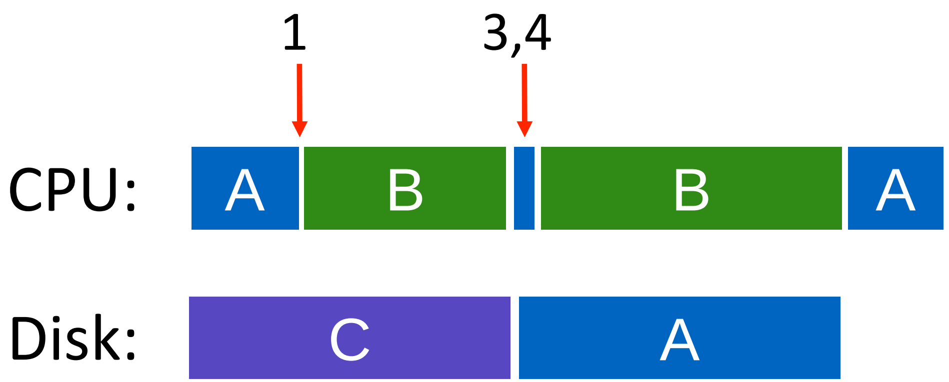

```
while (STATUS == BUSY)          // 1
        sleep & wait for interrupt;
Write data to DATA register      // 2
Write command to COMMAND register  // 3
while (STATUS == BUSY)          // 4
        sleep & wait for interrupt;
```




```
while (STATUS == BUSY)          // 1
        sleep & wait for interrupt;
```

~~Write data to DATA register        // 2~~

```
Write command to COMMAND register  // 3
while (STATUS == BUSY)          // 4
        sleep & wait for interrupt;
```




while (STATUS == BUSY) // 1

sleep & wait for interrupt;

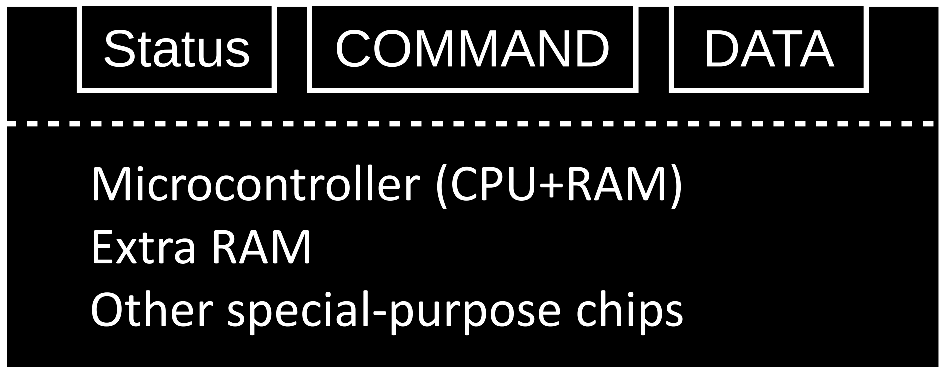
~~Write data to DATA register // 2~~

Write command to COMMAND register // 3

while (STATUS == BUSY) // 4

sleep & wait for interrupt;

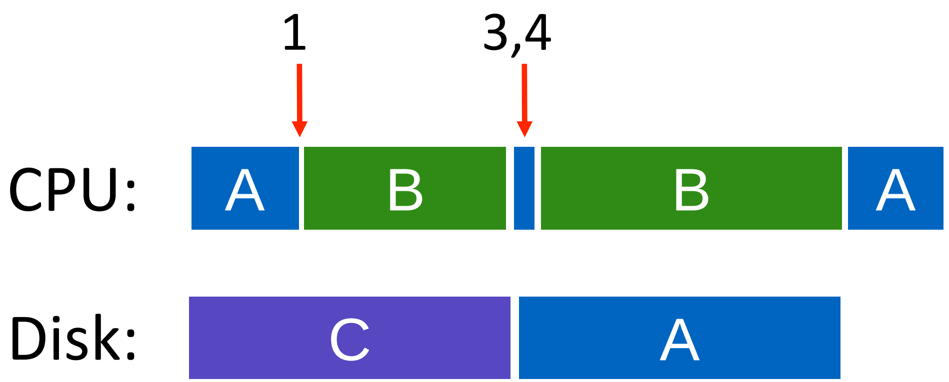
# Protocol Variants




**Status checks: polling *vs.* interrupts**

**Data: PIO *vs.* DMA**

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




```
while (STATUS == BUSY)          // 1
        sleep & wait for interrupt;
```

~~Write data to DATA register       // 2~~

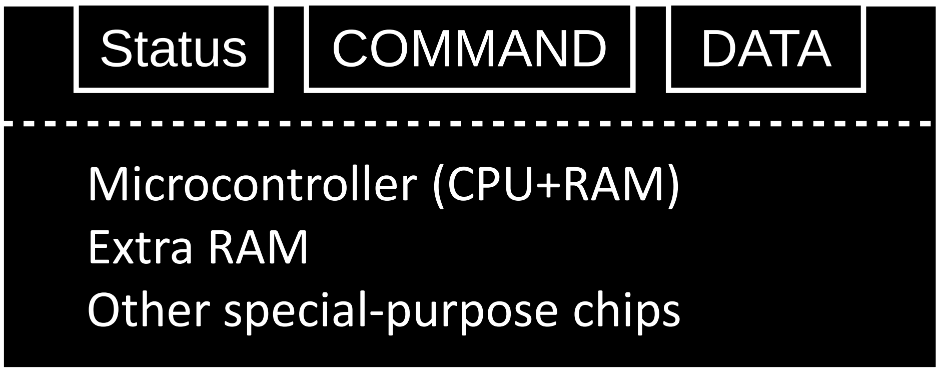
```
Write command to COMMAND register  // 3
while (STATUS == BUSY)          // 4
        sleep & wait for interrupt;
```

how does OS read and write registers?

# Special Instructions vs. Mem-Mapped I/O

- **Special instructions**
  - each device has a port
  - in/out instructions (x86) communicate with device
- **Memory-Mapped I/O**
  - H/W maps registers into address space
  - loads/stores sent to device
- **Doesn't matter much (both are used)**

# Protocol Variants




**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 can we avoid writing a slightly different OS for each H/W combination?**

- **Abstract a device with a driver**
  - Write device driver for each device

- **Drivers are 70% of Linux source code**

# Storage Stack

application

file system

scheduler

driver

hard drive

build common interface
on top of all HDDs

# A Simple Device Driver

```
Control Register:
  Address 0x3F6 = 0x08 (0000 1RE0): R=reset,
                  E=0 means "enable interrupt"

Command Block Registers:
  Address 0x1F0 = Data Port
  Address 0x1F1 = Error
  Address 0x1F2 = Sector Count
  Address 0x1F3 = LBA low byte
  Address 0x1F4 = LBA mid byte
  Address 0x1F5 = LBA hi  byte
  Address 0x1F6 = 1B1D TOP4LBA: B=LBA, D=drive
  Address 0x1F7 = Command/status

Status Register (Address 0x1F7):
    7      6     5      4     3     2     1      0
   BUSY  READY FAULT SEEK  DRQ   CORR IDDEX ERROR

Error Register (Address 0x1F1): (check when 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
```

Figure 36.5: **The IDE Interface**

# A Simple Device Driver

Spin for ready

```
static int ide_wait_ready() {
  while (((int r = inb(0x1f7)) & IDE_BSY) || !(r & IDE_DRDY))
    ;  // loop until drive isn't busy
}

static void ide_start_request(struct buf *b) {
  ide_wait_ready();
  outb(0x3f6, 0);                      // enable interrupt
  outb(0x1f2, 1);                      // how many sectors?
  outb(0x1f3, b->sector & 0xff);       // LBA goes here ...
  outb(0x1f4, (b->sector >> 8) & 0xff);   // ... and here
  outb(0x1f5, (b->sector >> 16) & 0xff);  // ... and here!
  outb(0x1f6, 0xe0 | ((b->dev&1)<<4) | ((b->sector>>24)&0x0f));
  if(b->flags & B_DIRTY){
    outb(0x1f7, IDE_CMD_WRITE);        // this is a WRITE
    outsl(0x1f0, b->data, 512/4);      // transfer data too!
  } else {
    outb(0x1f7, IDE_CMD_READ);         // this is a READ (no data)
  }
}
```

LBA = logical block address -> sector

# A Simple Device Driver

**Read or write**

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

**Interrupt procedure**

```
void ide_intr() {
  struct buf *b;
  acquire(&ide_lock);
  if (!(b->flags & B_DIRTY) && ide_wait_ready() >= 0)
    insl(0x1f0, b->data, 512/4);      // if READ: get data
  b->flags |= B_VALID;
  b->flags &= ˜B_DIRTY;
  wakeup(b);                          // wake waiting process
  if ((ide_queue = b->qnext) != 0)    // start next request
    ide_start_request(ide_queue);     // (if one exists)
  release(&ide_lock);
}
```

Figure 36.6: **The xv6 IDE Disk Driver (Simplified)**

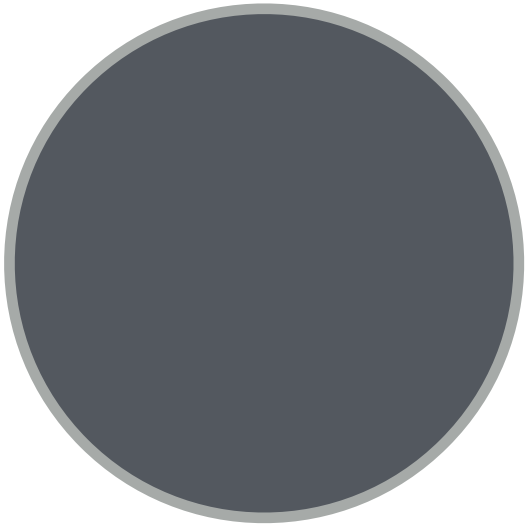
# Hard Disks

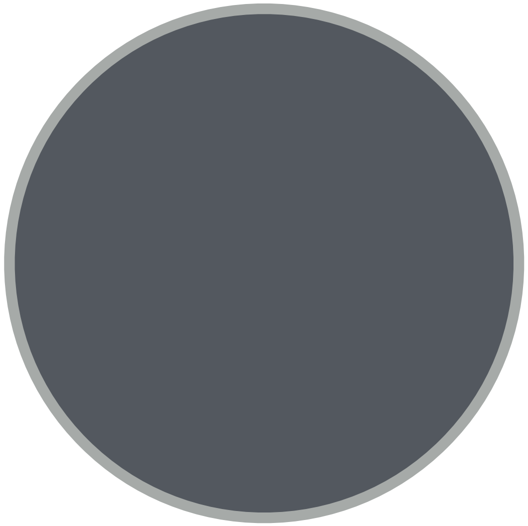
# Basic Interface

- **Disk has a sector-addressable address space**
  - Appears as an array of sectors
- **Sectors are typically <u>512 bytes</u> or 4096 bytes.**
- **Main operations: reads + writes to sectors**
- **Mechanical (slow) nature makes management "interesting"**

# Disk Internals

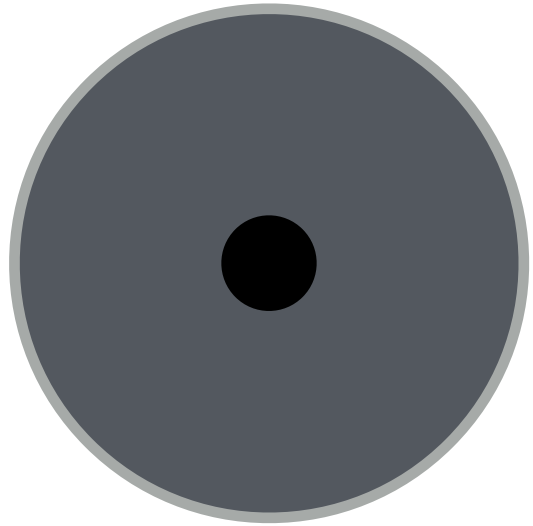
Platter



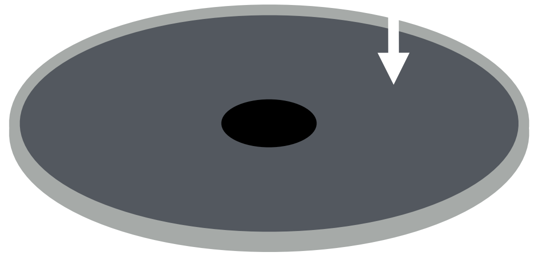


Platter is covered with a magnetic film.

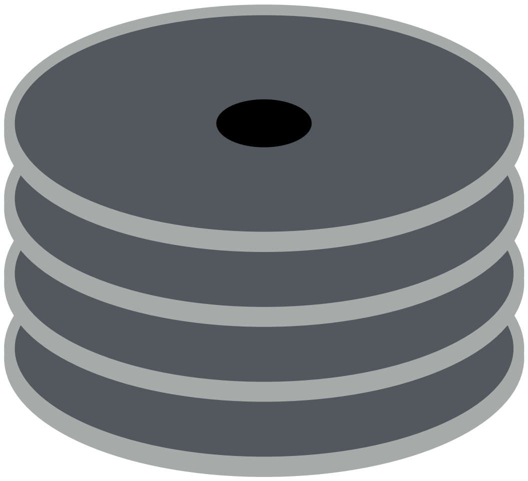
Spindle



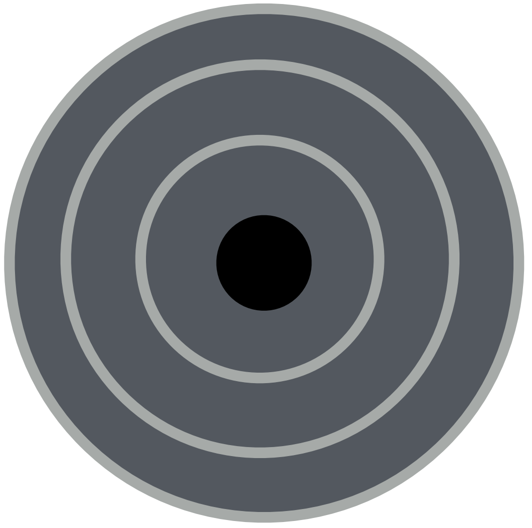
Surface



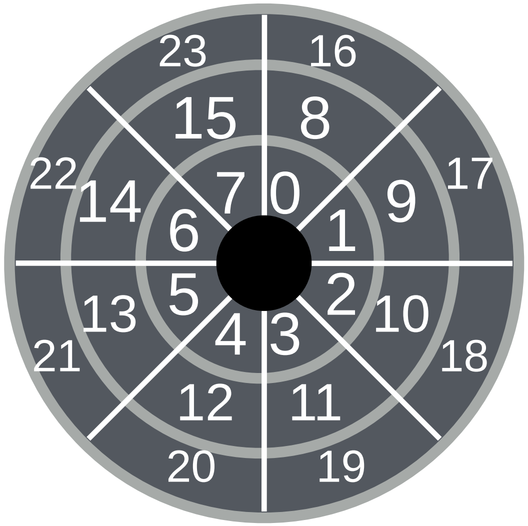
Surface



Many platters may be bound to the spindle.

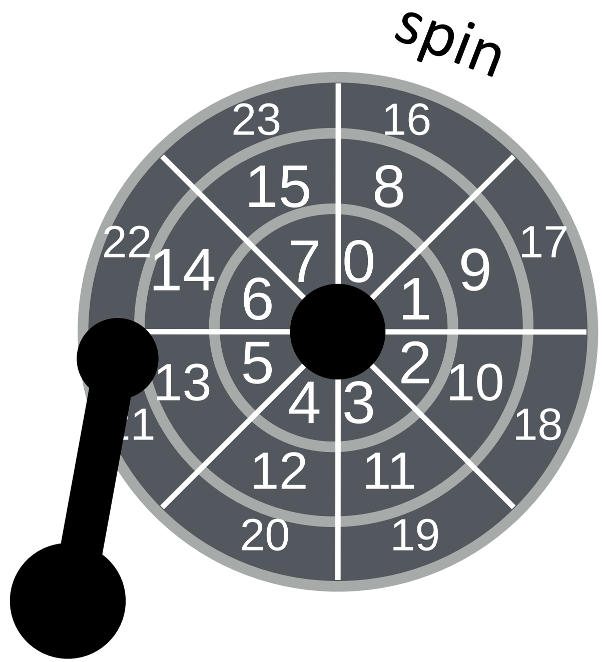


Each surface is divided into rings called tracks.
A stack of tracks (across platters) is called a cylinder.



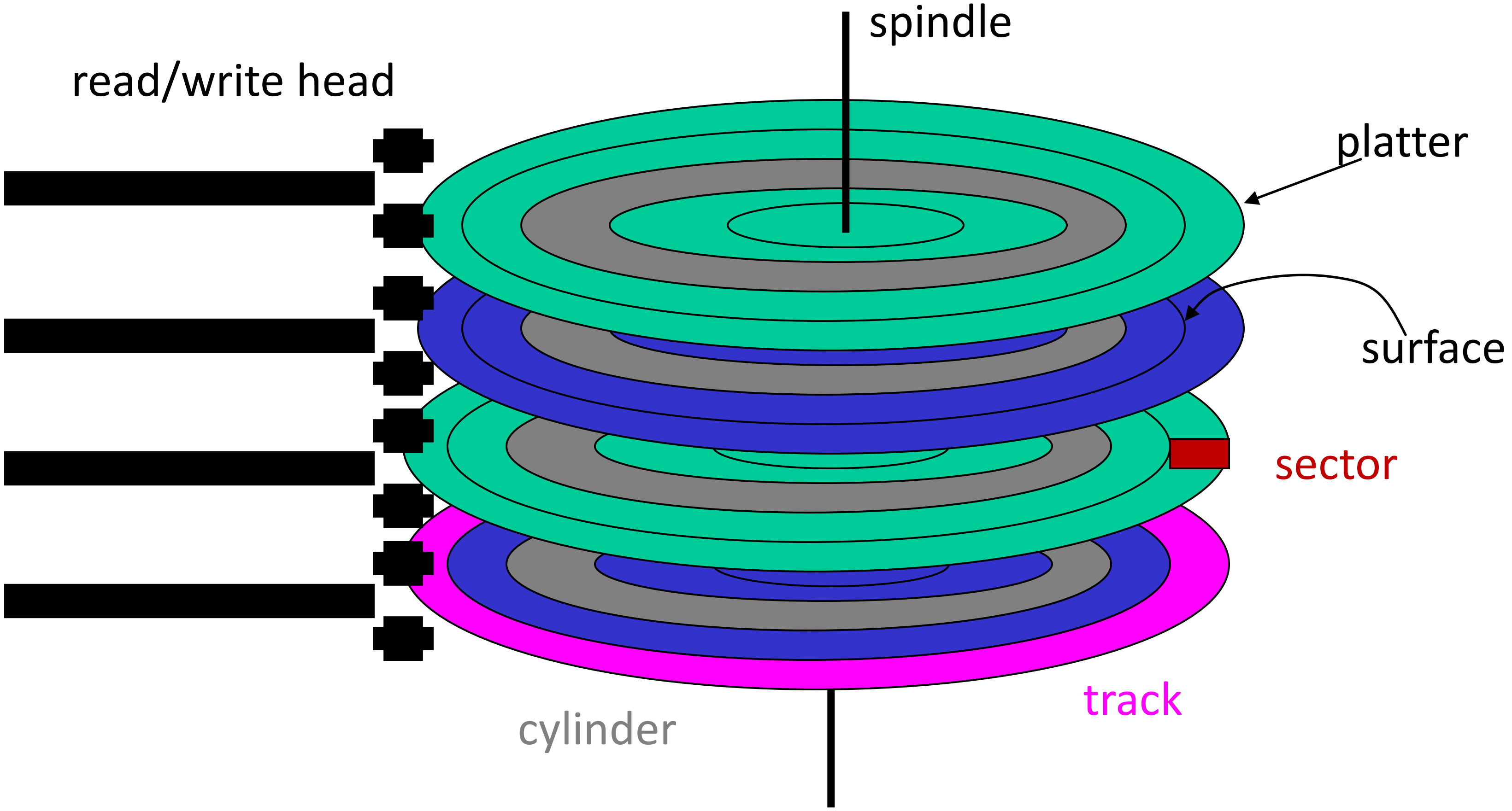

The tracks are divided into numbered sectors.


Heads on a moving arm can read from each surface.




Spindle/platters rapidly spin.

# Disk Terminology



# Let’s Read 12!

# Positioning

- **Drive servo system keeps head on track**
  - How does the disk head know where it is?
  - Platters not perfectly aligned, tracks not perfectly concentric (runout) -- difficult to stay on track
  - More difficult as density of disk increase
    - More bits per inch (BPI), more tracks per inch (TPI)
- **Use servo burst:**
  - Record placement information every few (3-5) sectors
  - When head cross servo burst, figure out location and adjust as needed

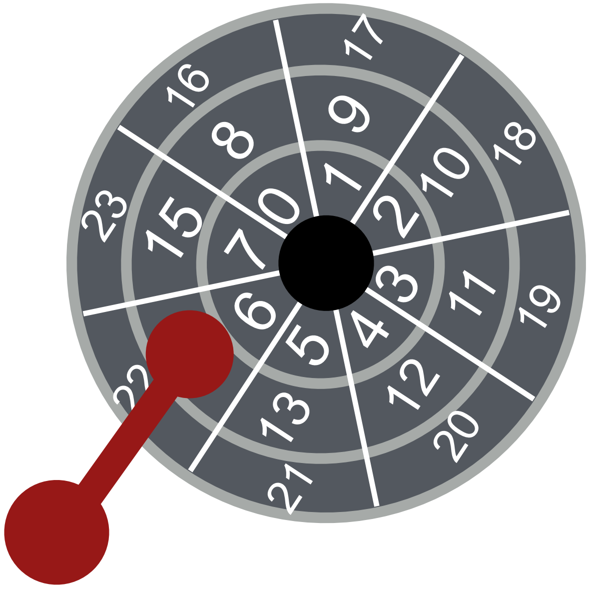
# Let's Read 12!

# Seek to right track.

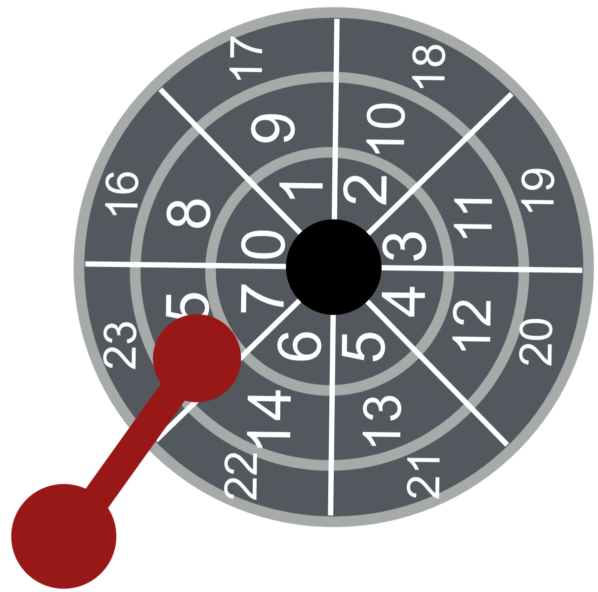
# Seek to right track.

# Seek to right track.

# Wait for rotation.

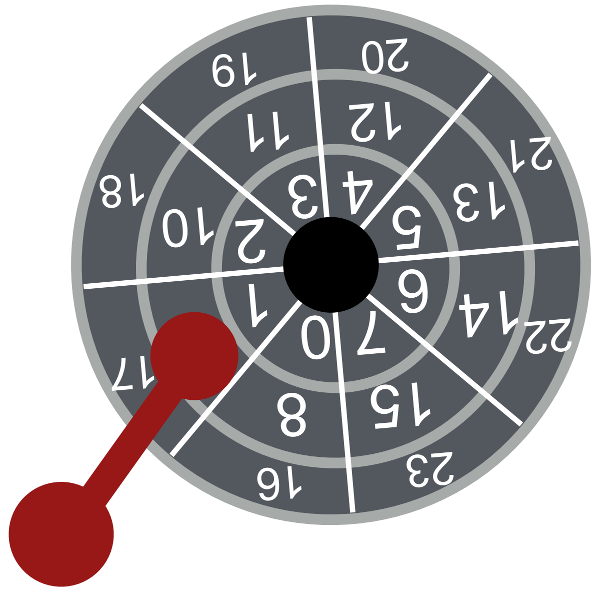


# Wait for rotation.

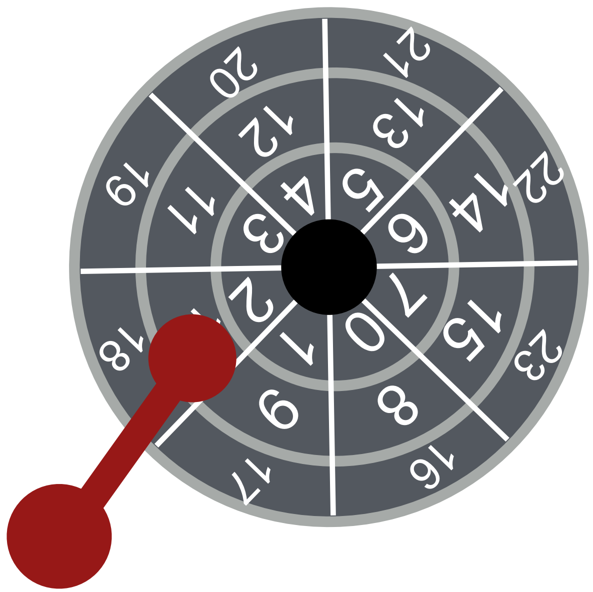


# Wait for rotation.

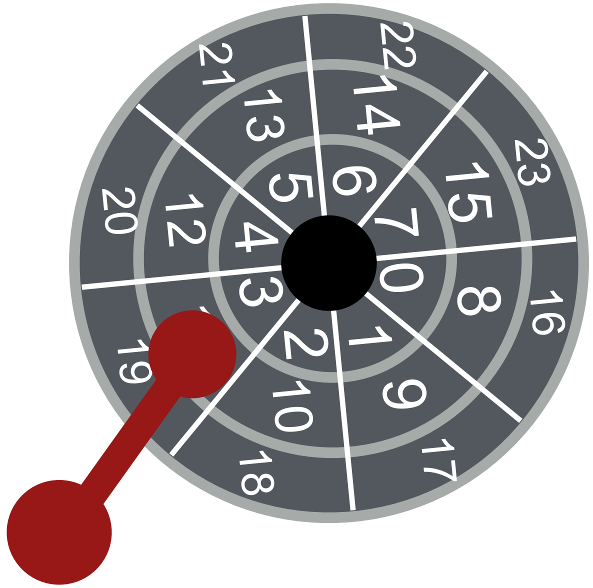
# Wait for rotation.



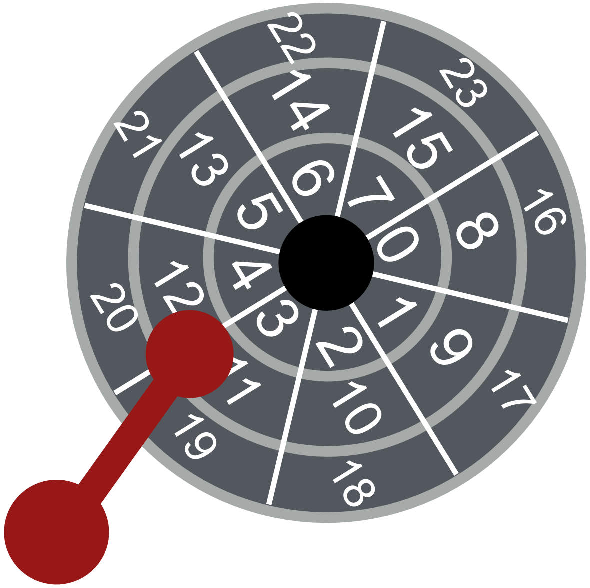
# Wait for rotation.



# Wait for rotation.



# Transfer data.



# Transfer data.

# Transfer data.

# Yay!

# Time to Read/write

- **Three components:**
- **Time = seek + rotation + transfer time**

# 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)

# Disk Spec

| | Cheetah | Barracuda |
|---|---|---|
| Capacity | 300 GB | 1 TB |
| RPM | 15,000 | 7,200 |
| Avg Seek | 4 ms | 9 ms |
| Max Transfer | 125 MB/s | 105 MB/s |
| Platters | 4 | 4 |
| Cache | 16 MB | 32 MB |

Sequential workload: what is throughput for each?

Cheeta: 125 MB/s.
Barracuda: 105 MB/s.

# Disk Spec

| | Cheetah | Barracuda |
|---|---|---|
| Capacity | 300 GB | 1 TB |
| RPM | 15,000 | 7,200 |
| Avg Seek | 4 ms | 9 ms |
| Max Transfer | 125 MB/s | 105 MB/s |
| Platters | 4 | 4 |
| Cache | 16 MB | 32 MB |

Random workload: what is throughput for each?
(what else do you need to know?)

What is size of each random read?
Assume 16-KB reads

| | Cheetah | Barracuda |
|---|---|---|
| RPM | 15,000 | 7,200 |
| Avg Seek | 4 ms | 9 ms |
| Max Transfer | 125 MB/s | 105 MB/s |

How long does an average random 16-KB read take w/ Cheetah?

**Seek + rotation + transfer**

Seek = 4 ms

| | Cheetah | Barracuda |
|---|---|---|
| RPM | 15,000 | 7,200 |
| Avg Seek | 4 ms | 9 ms |
| Max Transfer | 125 MB/s | 105 MB/s |

How long does an average random 16-KB read take w/ Cheetah?

Average rotation in ms?

$$\text{avg rotation} = \frac{1}{2} \times \frac{1 \text{ min}}{15000} \times \frac{60 \text{ sec}}{1 \text{ min}} \times \frac{1000 \text{ ms}}{1 \text{ sec}} = 2 \text{ ms}$$

| | Cheetah | Barracuda |
|---|---|---|
| RPM | 15,000 | 7,200 |
| Avg Seek | 4 ms | 9 ms |
| Max Transfer | 125 MB/s | 105 MB/s |

How long does an average random 16-KB read take w/ Cheetah?

Transfer of 16 KB?

= 16KB / 125 MB/s

= 128 us

| | Cheetah | Barracuda |
|---|---|---|
| RPM | 15,000 | 7,200 |
| Avg Seek | 4 ms | 9 ms |
| Max Transfer | 125 MB/s | 105 MB/s |

How long does an average random 16-KB read take w/ Cheetah?

Cheetah time = 4ms + 2ms + 128us = 6.1ms

Throughput?

| | Cheetah | Barracuda |
|---|---|---|
| RPM | 15,000 | 7,200 |
| Avg Seek | 4 ms | 9 ms |
| Max Transfer | 125 MB/s | 105 MB/s |

How long does an average random 16-KB read take w/ Cheetah?

Cheetah time = 4ms + 2ms + 128us = 6.1ms

$$\text{throughput} = \frac{16\text{ KB}}{6.1\text{ms}} \times \frac{1\text{ MB}}{1024\text{ KB}} \times \frac{100\text{ ms}}{1\text{ sec}} = 2.5\text{ MB/s}$$

| | Cheetah | Barracuda |
|---|---|---|
| RPM | 15,000 | 7,200 |
| Avg Seek | 4 ms | 9 ms |
| Max Transfer | 125 MB/s | 105 MB/s |

How long does an average random 16-KB read take w/ Barracuda?

Time = seek + rotation + transfer

Seek = 9ms

| | Cheetah | Barracuda |
|---|---|---|
| RPM | 15,000 | 7,200 |
| Avg Seek | 4 ms | 9 ms |
| Max Transfer | 125 MB/s | 105 MB/s |

How long does an average random 16-KB read take w/ Barracuda?

$$\text{avg rotation} = \frac{1}{2} \times \frac{1\ \text{min}}{7200} \times \frac{60\ \text{sec}}{1\ \text{min}} \times \frac{1000\ \text{ms}}{1\ \text{sec}} = 4.1\ \text{ms}$$

| | Cheetah | Barracuda |
|---|---|---|
| RPM | 15,000 | 7,200 |
| Avg Seek | 4 ms | 9 ms |
| Max Transfer | 125 MB/s | 105 MB/s |

How long does an average random 16-KB read take w/ Barracuda?

$$\text{transfer} = \frac{1\ \text{sec}}{105\ \text{MB}} \times 16\ \text{KB} \times \frac{1{,}000{,}000\ \text{us}}{1\ \text{sec}} = 149\ \text{us}$$

| | Cheetah | Barracuda |
|---|---|---|
| RPM | 15,000 | 7,200 |
| Avg Seek | 4 ms | 9 ms |
| Max Transfer | 125 MB/s | 105 MB/s |

How long does an average random 16-KB read take w/ Barracuda?

Barracuda time = 9ms + 4.1ms + 149us = 13.2ms

| | Cheetah | Barracuda |
|---|---|---|
| RPM | 15,000 | 7,200 |
| Avg Seek | 4 ms | 9 ms |
| Max Transfer | 125 MB/s | 105 MB/s |

How long does an average random 16-KB read take w/ Barracuda?

Barracuda time = 9ms + 4.1ms + 149us = 13.2ms

$$\text{throughput} = \frac{16\text{ KB}}{13.2\text{ms}} \times \frac{1\text{ MB}}{1024\text{ KB}} \times \frac{1000\text{ ms}}{1\text{ sec}} = 1.2\text{ MB/s}$$

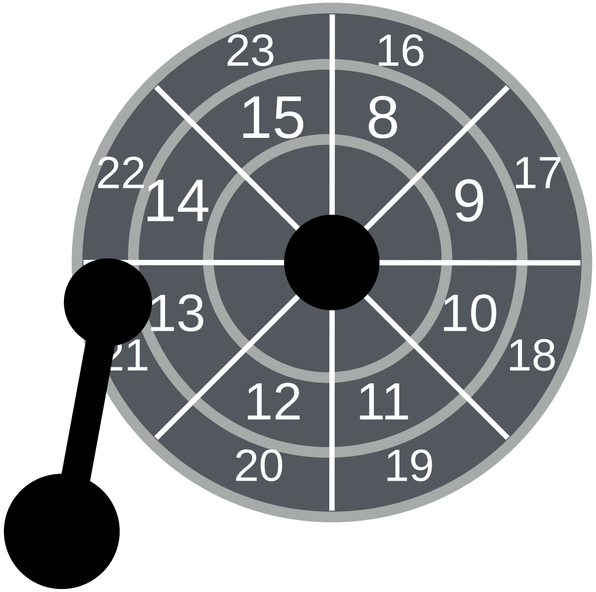
| | Cheetah | Barracuda |
|---|---|---|
| Capacity | 300 GB | 1 TB |
| RPM | 15,000 | 7,200 |
| Avg Seek | 4 ms | 9 ms |
| Max Transfer | 125 MB/s | 105 MB/s |
| Platters | 4 | 4 |
| Cache | 16 MB | 32 MB |

| | Cheetah | Barracuda |
|---|---|---|
| Sequential | 125 MB/s | 105 MB/s |
| Random | 2.5 MB/s | 1.2 MB/s |

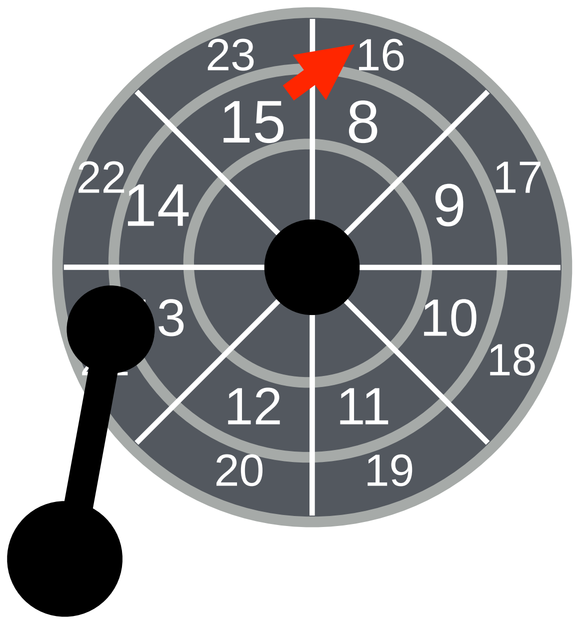
# Other Improvements

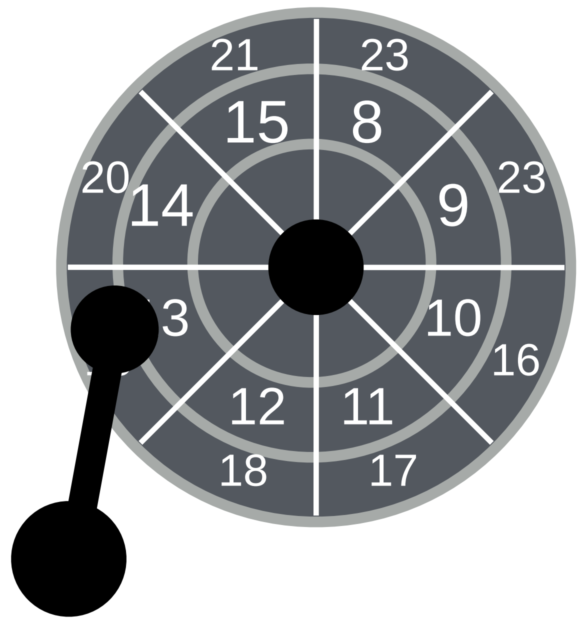
- Track Skew
- Zones
- Cache

Imagine sequential reading,
How should sectors numbers be laid out on disk?

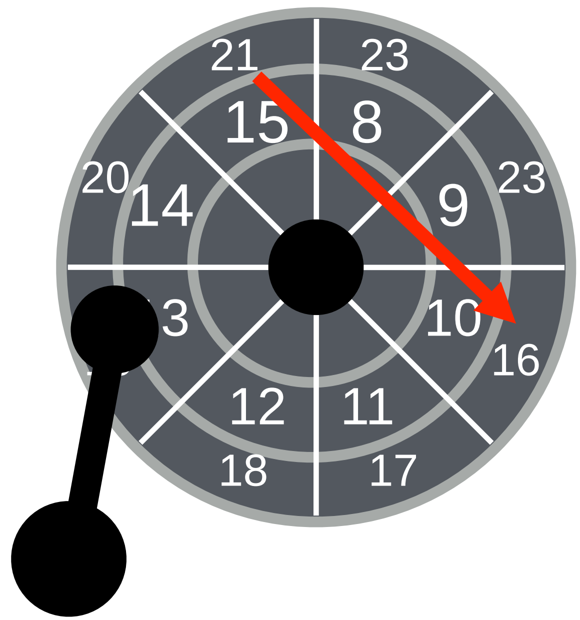


When reading 16 after 15, the head won't settle quick enough, so we need to do a rotation.



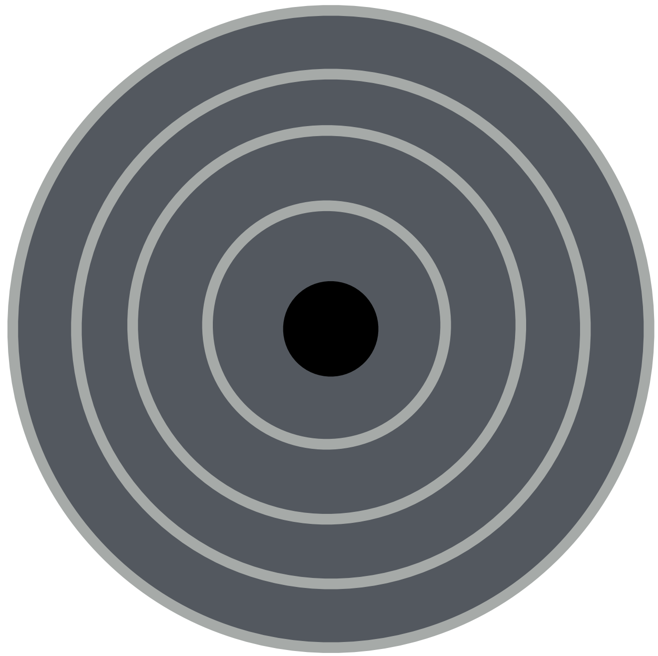

21
23
15
8
20
14
9
23
10
16
12
11
18
17

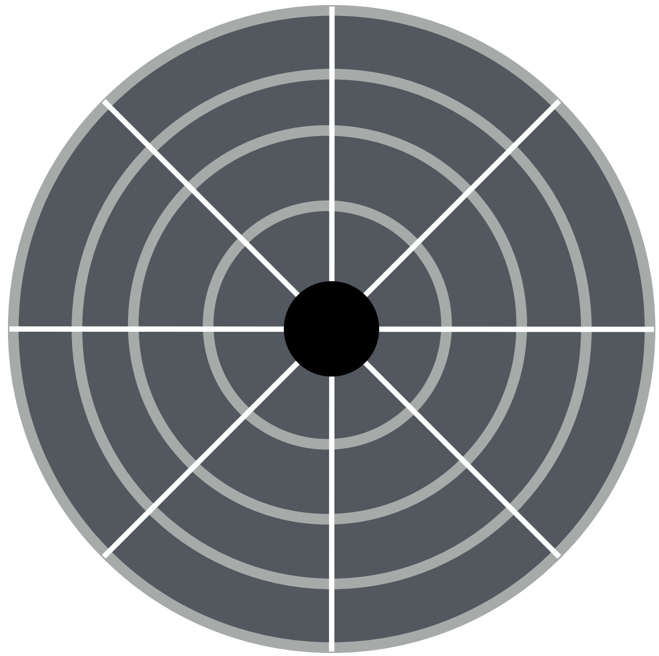
# enough time to settle now



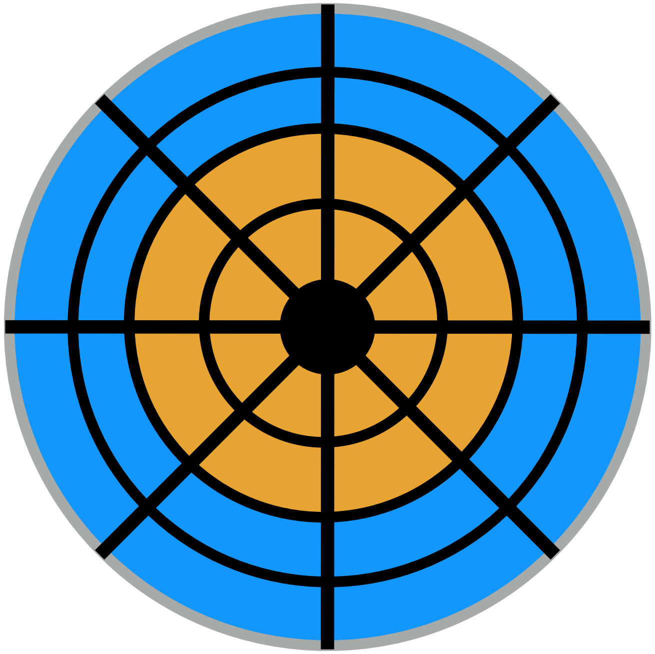
# Other Improvements

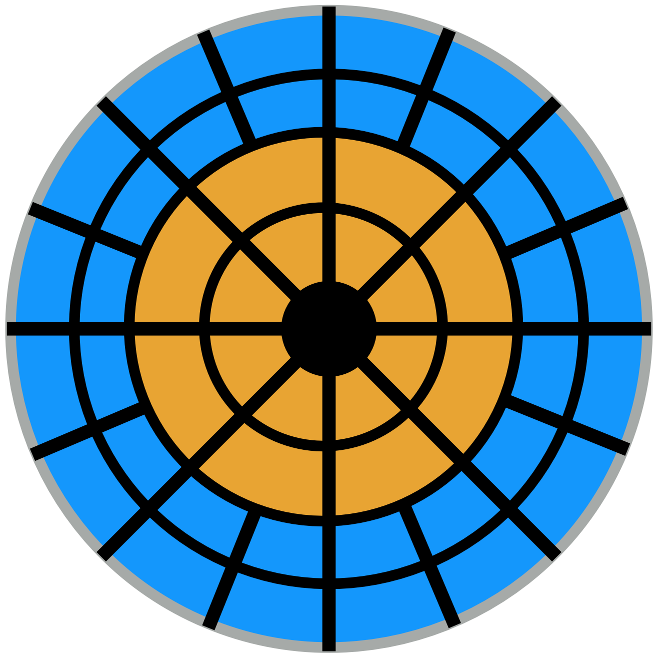
- Track Skew
- Zones
- Cache





**More space in outer track, but same sector size?**





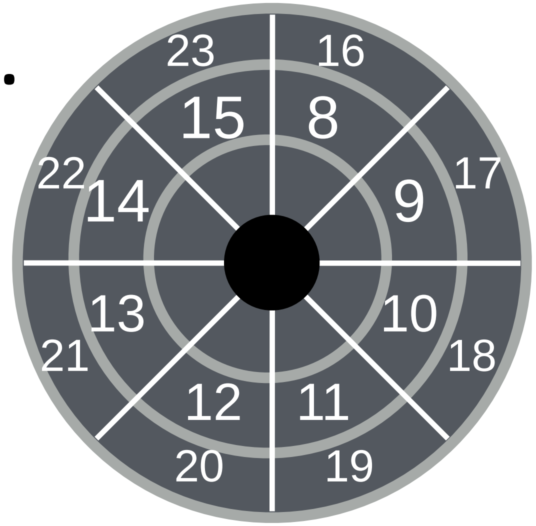
ZBR (Zoned bit recording): More sectors on outer tracks

# Other Improvements

- Track Skew
- Zones
- Cache

# Drive Cache

- Drives may cache both reads and writes.




- What advantage does caching in drive have for reads?
- What advantage does caching in drive have for writes?

# Buffering

- **Disks contain internal memory (2MB-16MB) used as cache**
- **Read-ahead: “Track buffer” (prefetching)**
  - Read contents of entire track into memory during rotational delay
  - read following sectors 14 and 15 with the start sector as 13
- **Write caching with volatile memory**
  - Immediate reporting: Claim written to disk when not
  - Data could be lost on power failure
- **Tagged command queueing**
  - Have multiple outstanding requests to the disk
  - Disk can reorder (schedule) requests for better performance

# I/O Schedulers

# I/O Schedulers

- Given a stream of I/O requests, in what order should they be served?
- Much different than CPU scheduling
- Position of disk head relative to request position matters more than length of job

# FCFS (First-Come-First-Serve)

- **Assume seek+rotate = 10 ms for random request**
- **How long (roughly) does the below workload take?**
  - Requests are given in sector numbers
- **300001, 700001, 300002, 700002, 300003, 700003**

~60ms

# FCFS (First-Come-First-Serve)

**Assume seek+rotate = 10 ms for random request**

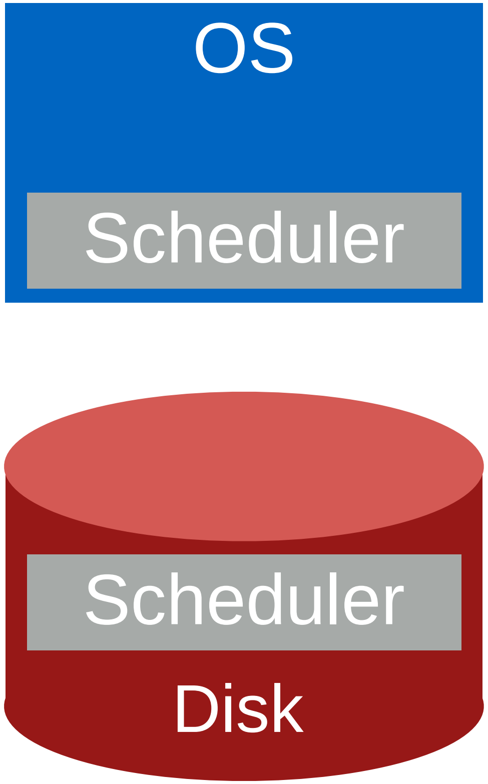
**How long (roughly) does the below workload take?**

- Requests are given in sector numbers

**300001, 700001, 300002, 700002, 300003, 700003** ~60ms

**300001, 300002, 300003, 700001, 700002, 700003** ~20ms

# Schedulers



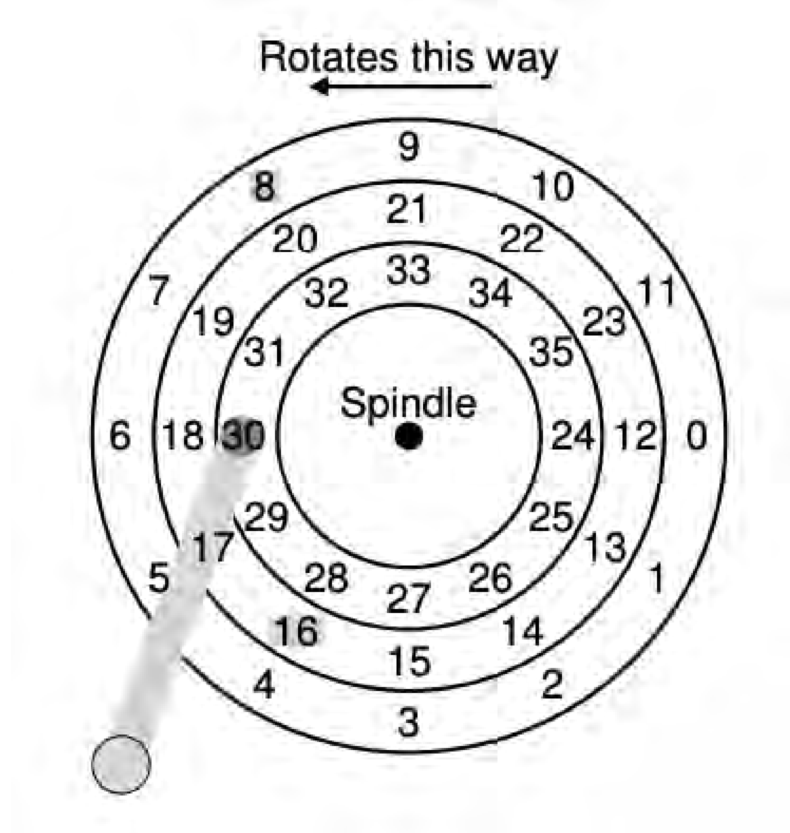

Where should the scheduler go?

# SPTF (Shortest Positioning Time First)

- **Strategy: always choose request that requires least positioning time (time for seeking and rotating)**
  - Greedy algorithm (just looks for best NEXT decision)
- **How to implement in disk?**
  - Shortest Seek Time First (SSTF)
- **How to implement in OS?**
  - Drive geometry is not available to the host OS
  - OS sees an array of blocks
  - Nearest block first (NBF)

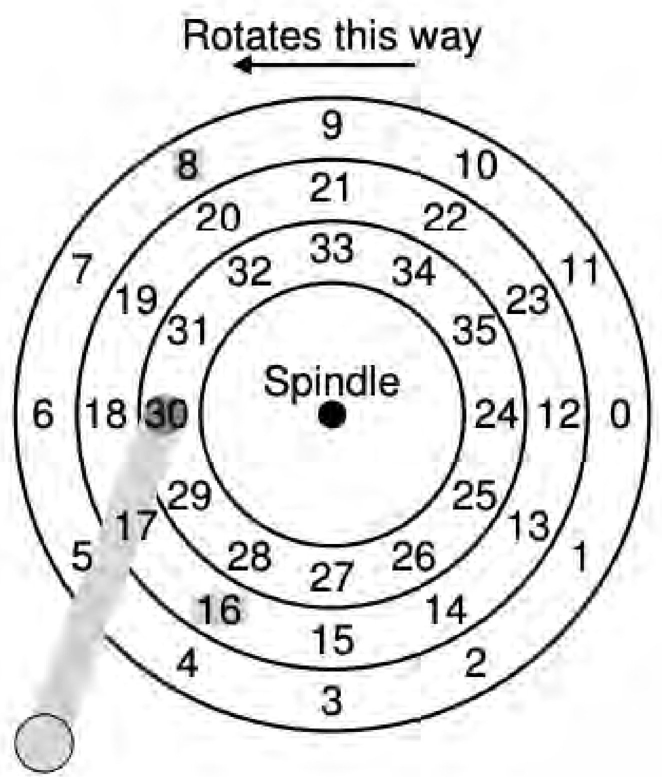
# SPTF (Shortest Positioning Time First)

- Which to serve, 8 or 16?
- Depends on whether seek or rotate is faster



# Disadvantage of SPTF?

- Easy for far away requests to starve



# SCAN

- **Elevator Algorithm:**
  - Sweep back and forth, from one end of disk other, serving requests as pass that cylinder
  - Sorts by cylinder number; ignores rotation delays
- **Pros/Cons?**
- **Better: C-SCAN (circular scan)**
  - Only sweep in one direction

# What happens?

- Assume 2 processes each calling read() with C-SCAN

```
void reader(int fd) {
        char buf[1024];
        int rv;
        while((rv = read(buf)) != 0) {
                assert(rv);
                // takes short time, e.g., 1ms
                process(buf, rv);
        }
}
```

# Work Conservation

- Work conserving schedulers always try to do work if there's work to be done

- Sometimes, it's better to wait instead if system anticipates another request will arrive

- Such non-work-conserving schedulers are called anticipatory schedulers

# CFQ (Linux Default)

- **Completely Fair Queueing**
  - Queue for each process
  - Weighted round-robin between queues, with slice time proportional to priority
  - Yield slice only if idle for a given time (anticipation)

- **Optimize order within queue**

# I/O Device Summary

- **Overlap I/O and CPU whenever possible!**
  - use interrupts, DMA
- **Storage devices provide common block interface**
- **On a disk: Never do random I/O unless you must!**
  - e.g., Quicksort is a terrible algorithm on disk
- **Spend time to schedule on slow, stateful devices**