OSTEP
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

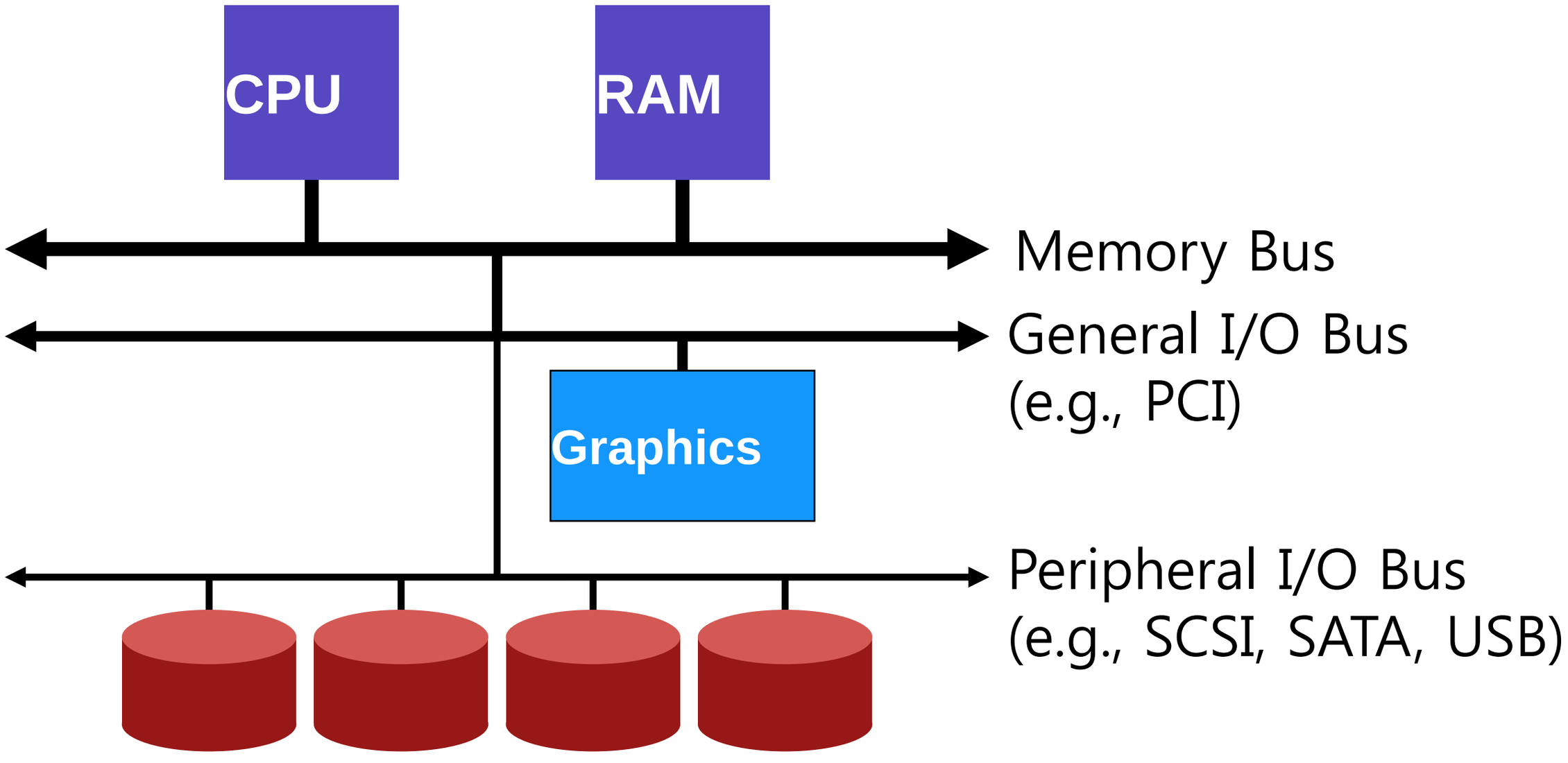
What good is a computer without any I/O devices?
- keyboard, display, disks

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

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?

Hardware support for I/O



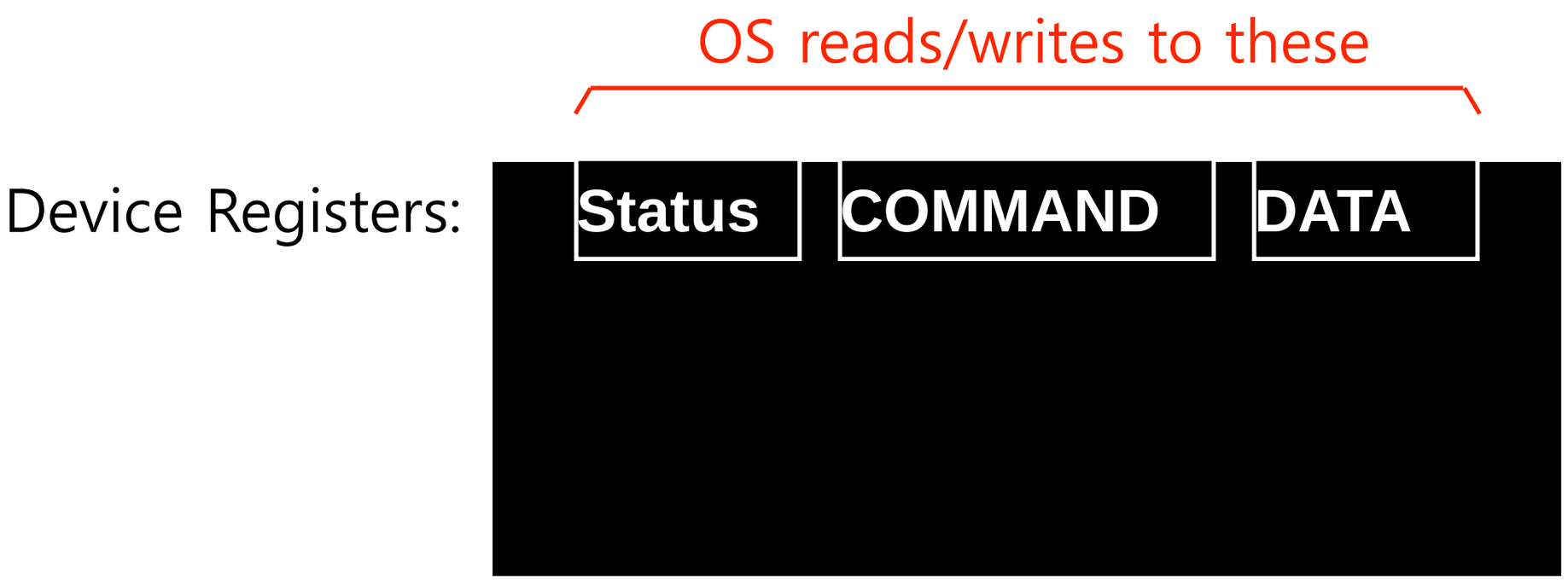
Why use hierarchical buses?

I/O Architecture

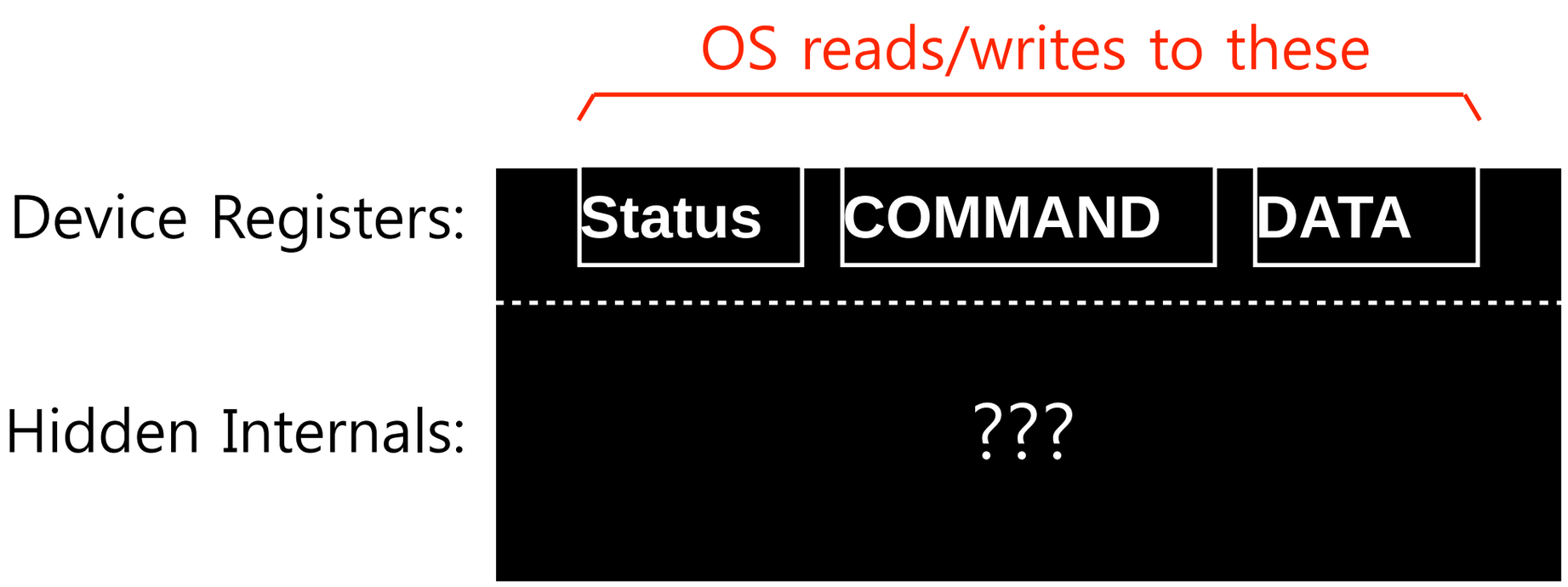
- Buses
  - Data paths that provided to enable information between CPU(s), RAM, and I/O devices.

- 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 Device

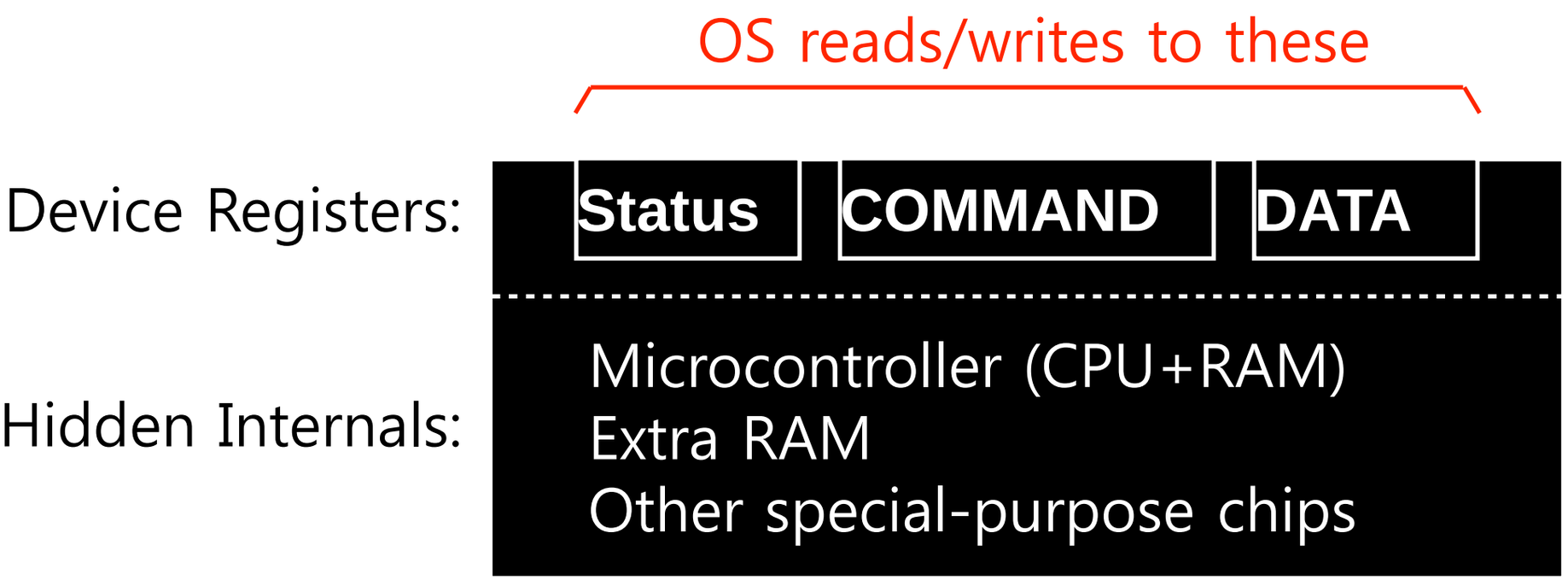


Canonical Devices has two important components.

Hardware interface allows the system software to control its operation.

Internals which is implementation specific.

Canonical Device

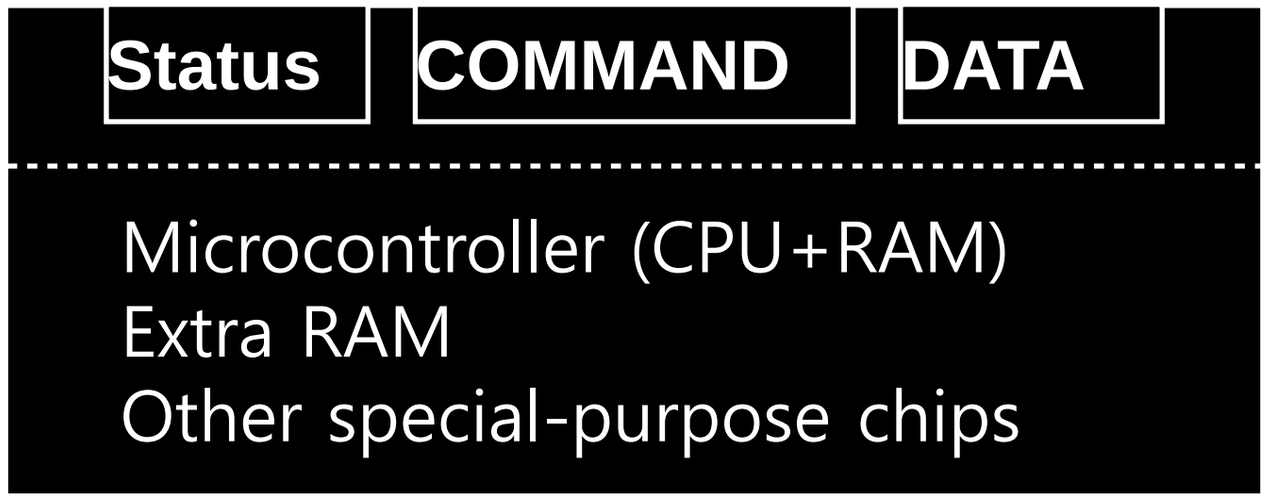


Canonical Devices has two important components.

Hardware interface allows the system software to control its operation.

Internals which is implementation specific.

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
    ;
```

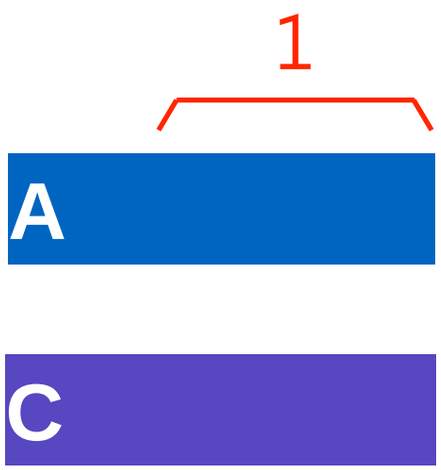
A

C

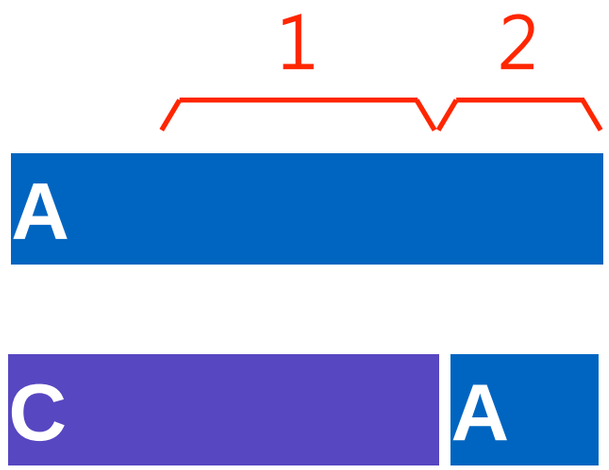
```
while (STATUS == BUSY)              // 1
    ;
Write data to DATA register         // 2
Write command to COMMAND register   // 3
while (STATUS == BUSY)              // 4
    ;
```

A wants to do I/O

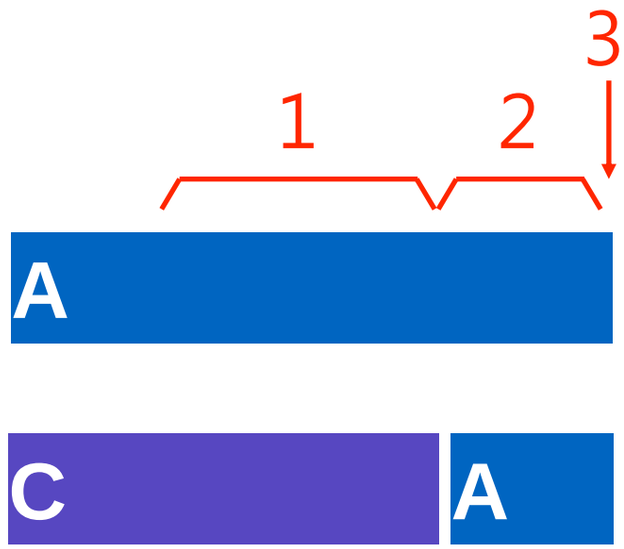

```
while (STATUS == BUSY)                  // 1
    ;
Write data to DATA register             // 2
Write command to COMMAND register       // 3
while (STATUS == BUSY)                  // 4
    ;
```



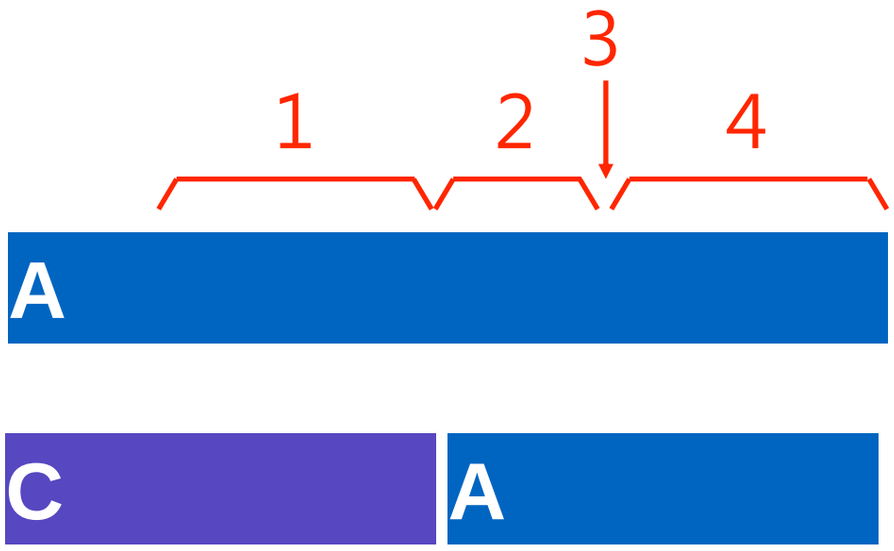

```
while (STATUS == BUSY)                  // 1
    ;

Write data to DATA register          // 2

Write command to COMMAND register   // 3

while (STATUS == BUSY)                  // 4
    ;
```



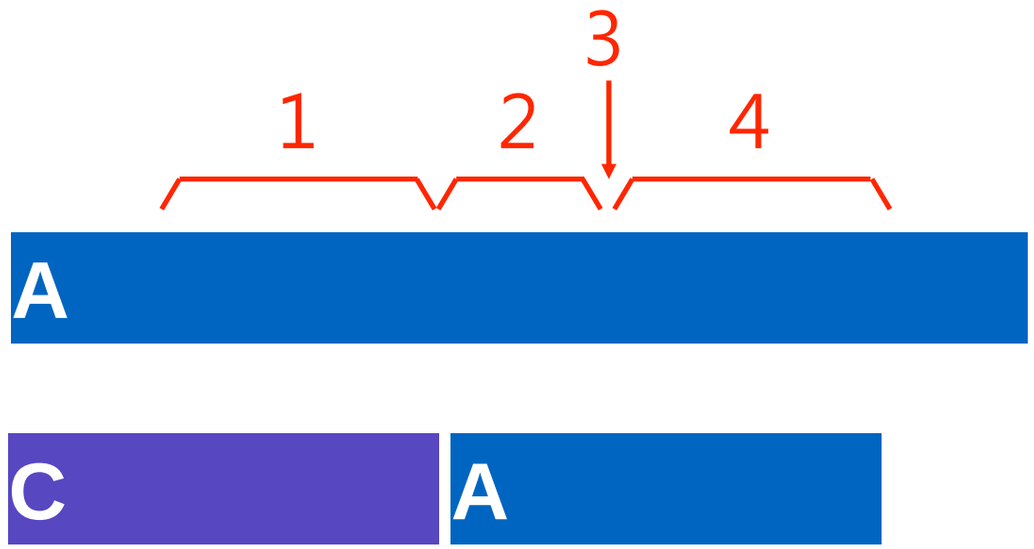

```
while (STATUS == BUSY)                  // 1
    ;

Write data to DATA register             // 2

Write command to COMMAND register   // 3

while (STATUS == BUSY)                  // 4
    ;
```



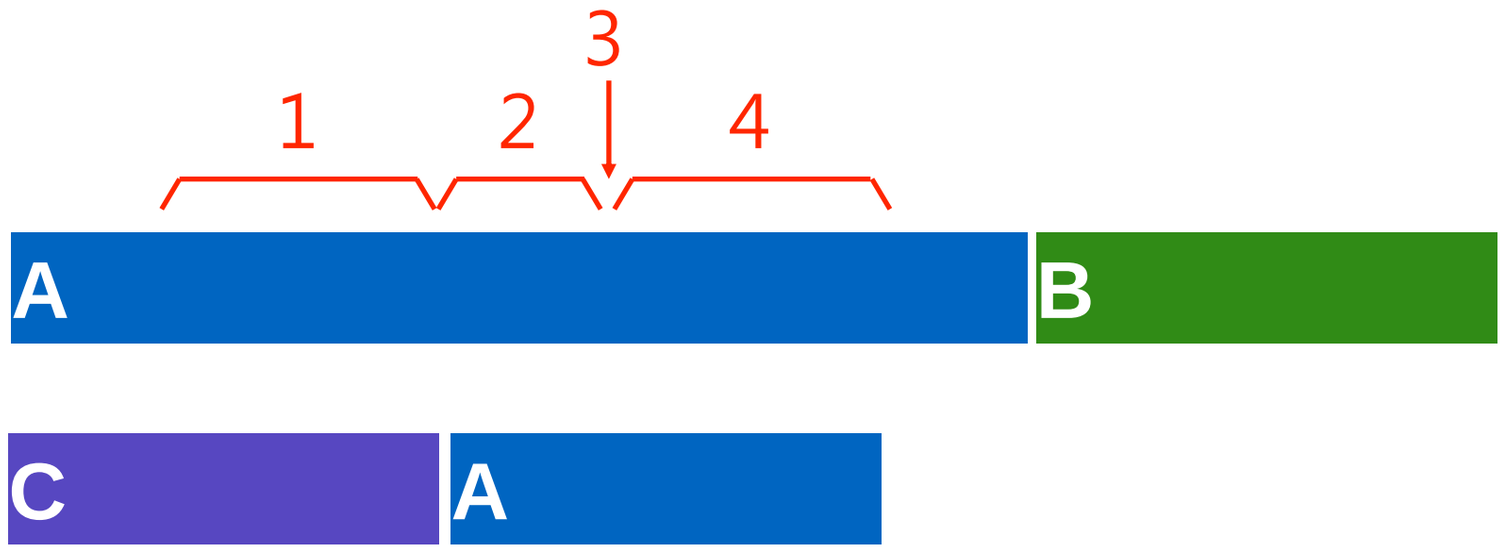

```
while (STATUS == BUSY)                  // 1
    ;

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
    ;
```



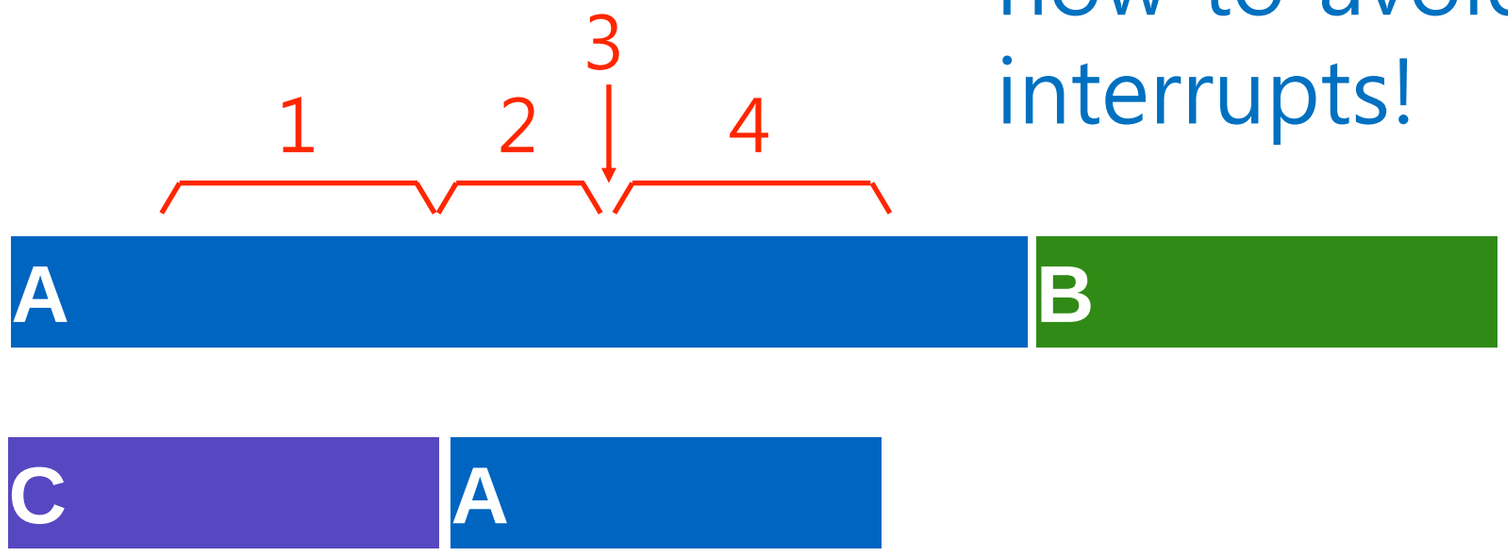

```
while (STATUS == BUSY)                  // 1
    ;

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
    ;
```

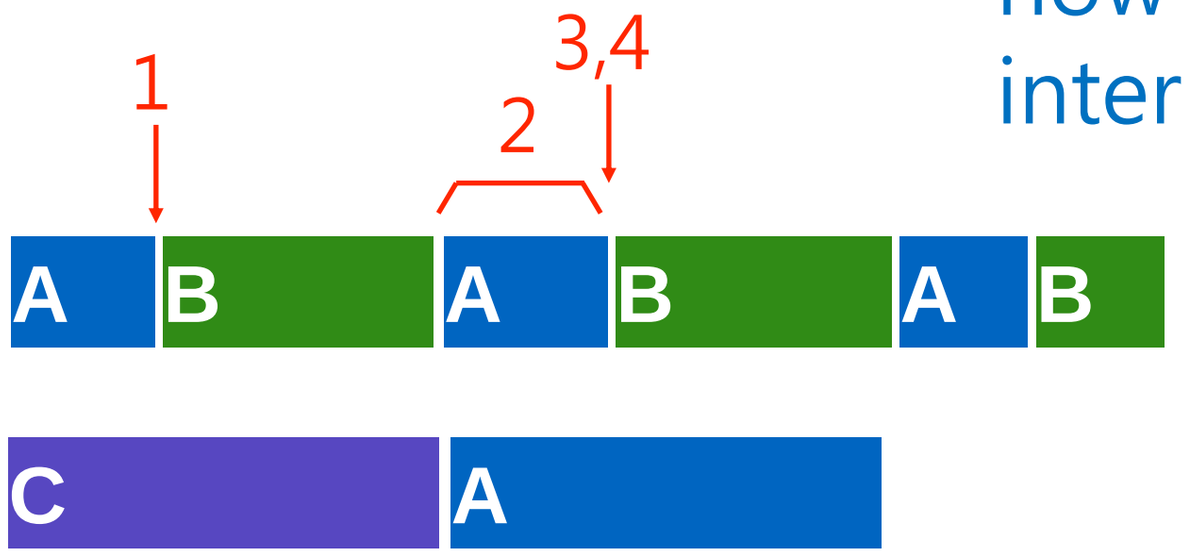
how to avoid spinning? interrupts!

how to avoid spinning?
interrupts!




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

how to avoid spinning?
interrupts!




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

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.

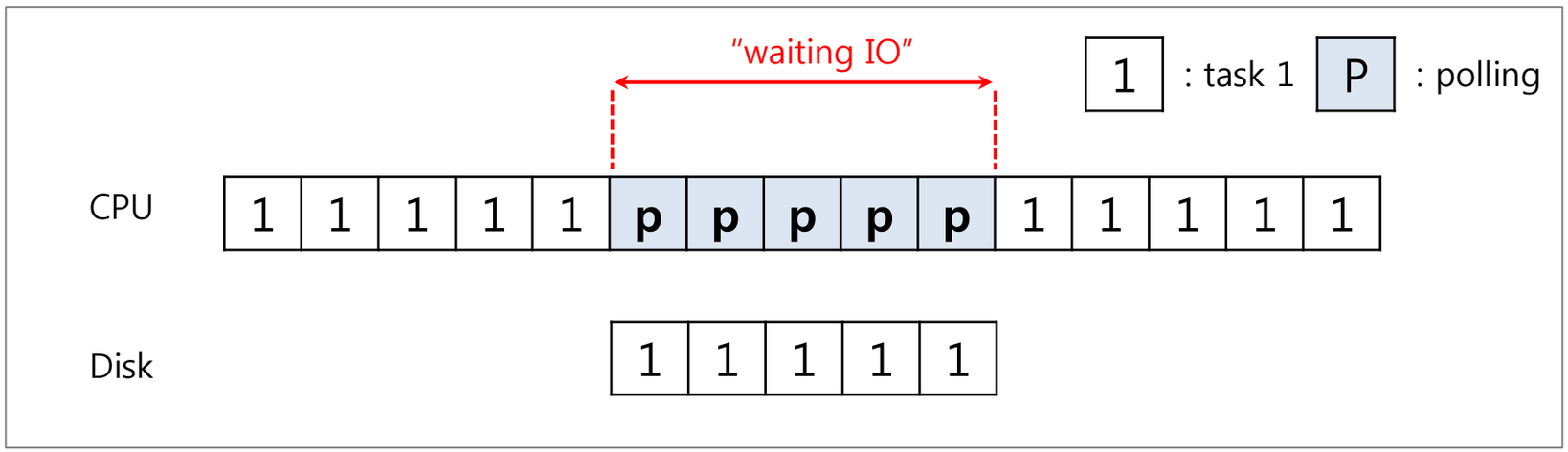


Diagram of CPU utilization by polling

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.**

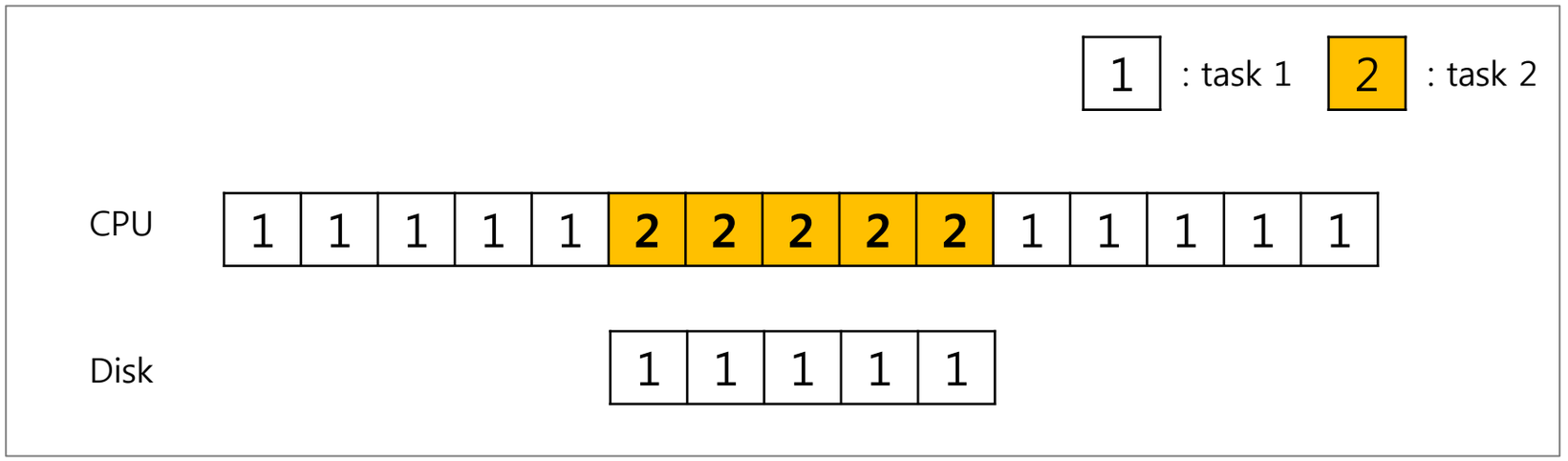


Diagram of CPU utilization by interrupt

Interrupts vs. Polling

Are interrupts ever worse than polling?

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)

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.

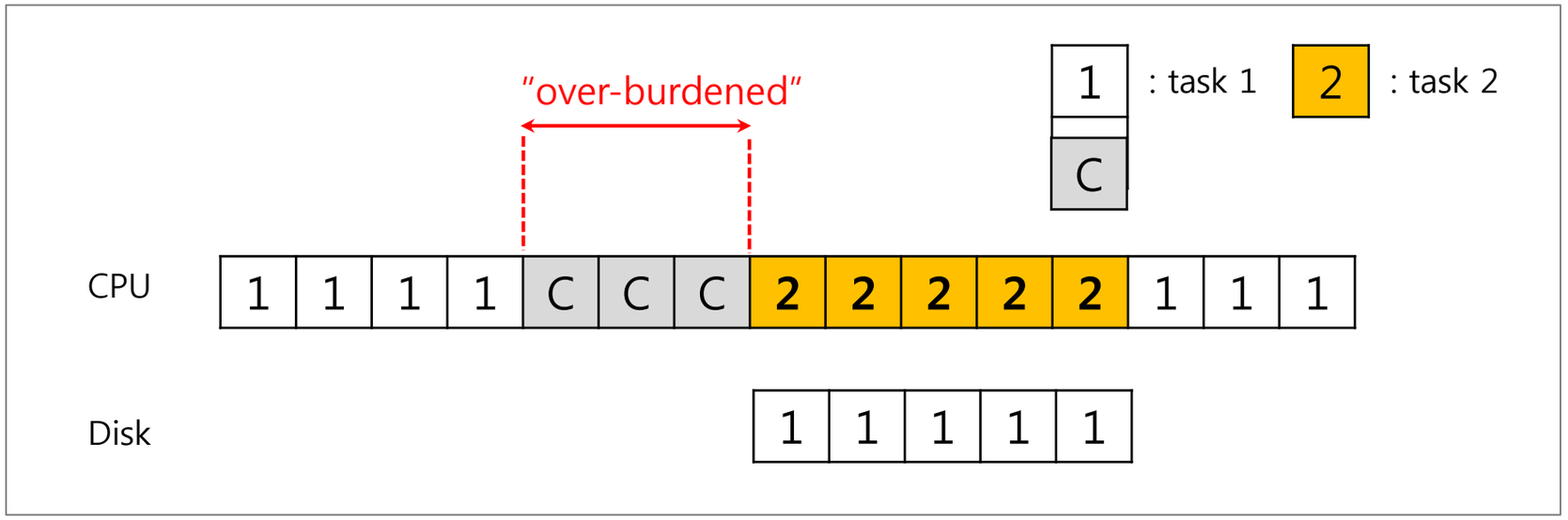


Diagram of CPU utilization

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.

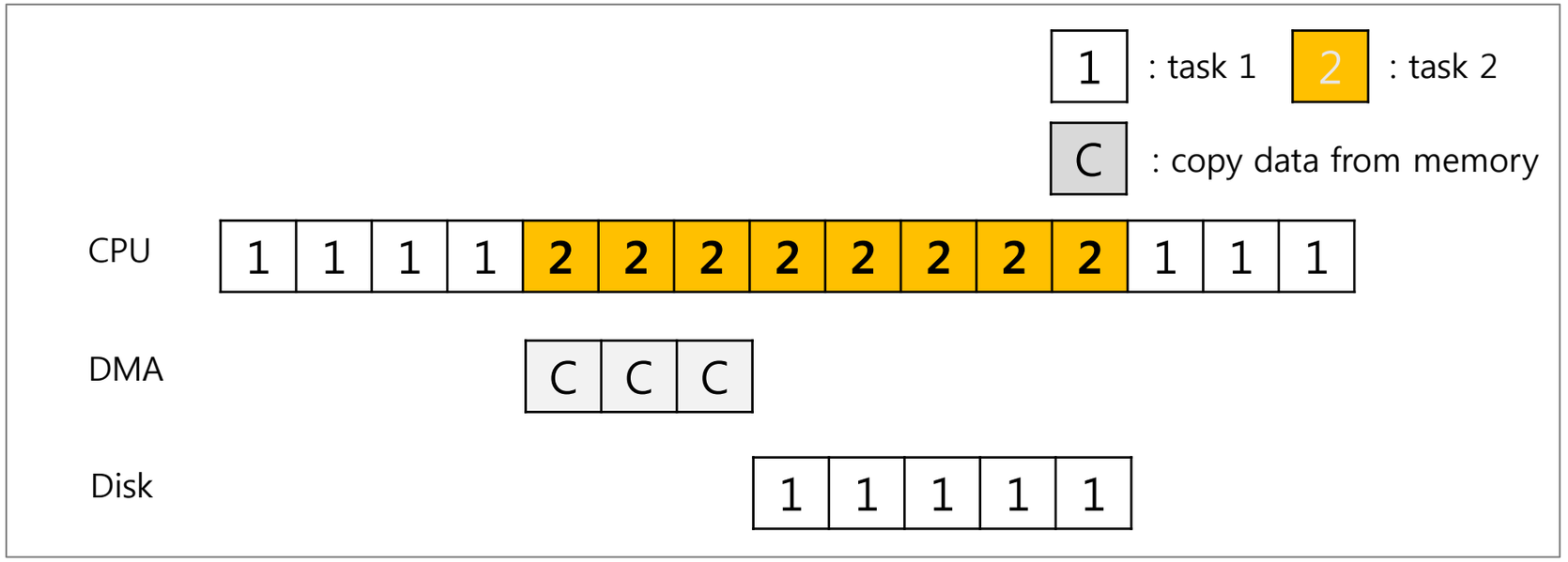


Diagram of CPU utilization by DMA

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.

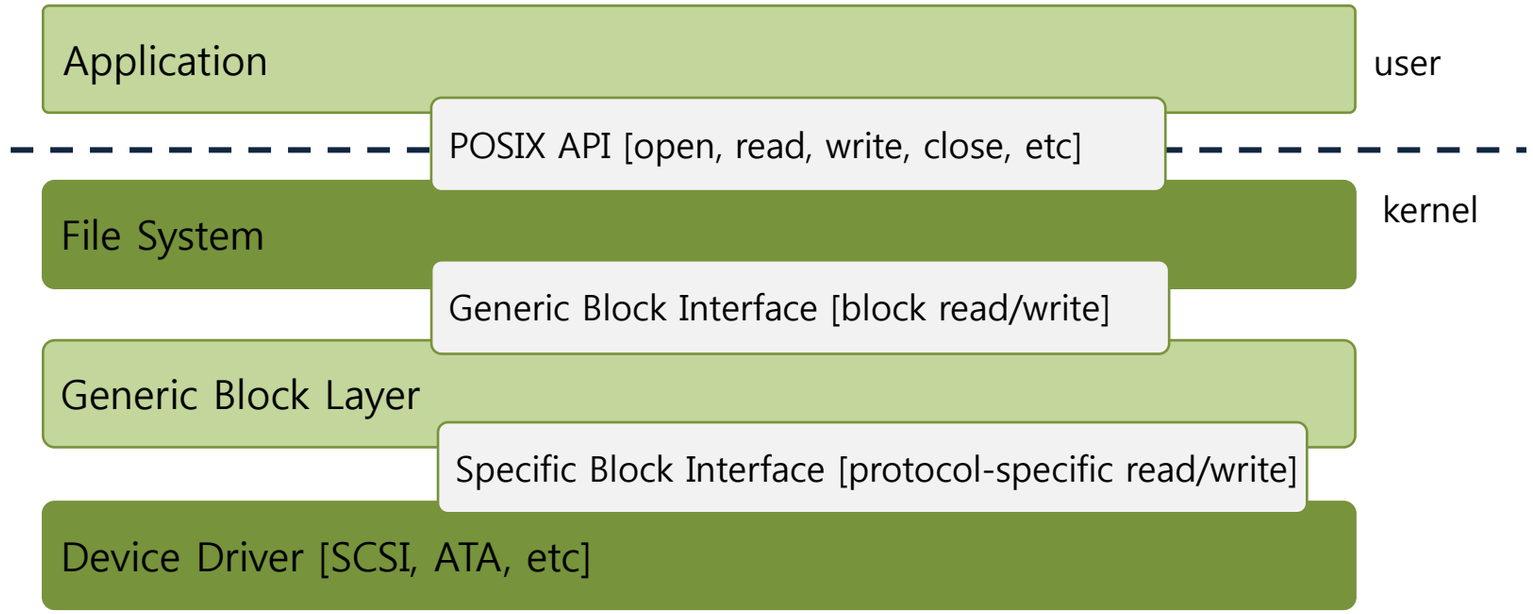
Device interaction (Cont.)

- How the OS interact with **different specific 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
  - Abstraction encapsulate any specifics of device interaction.

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.

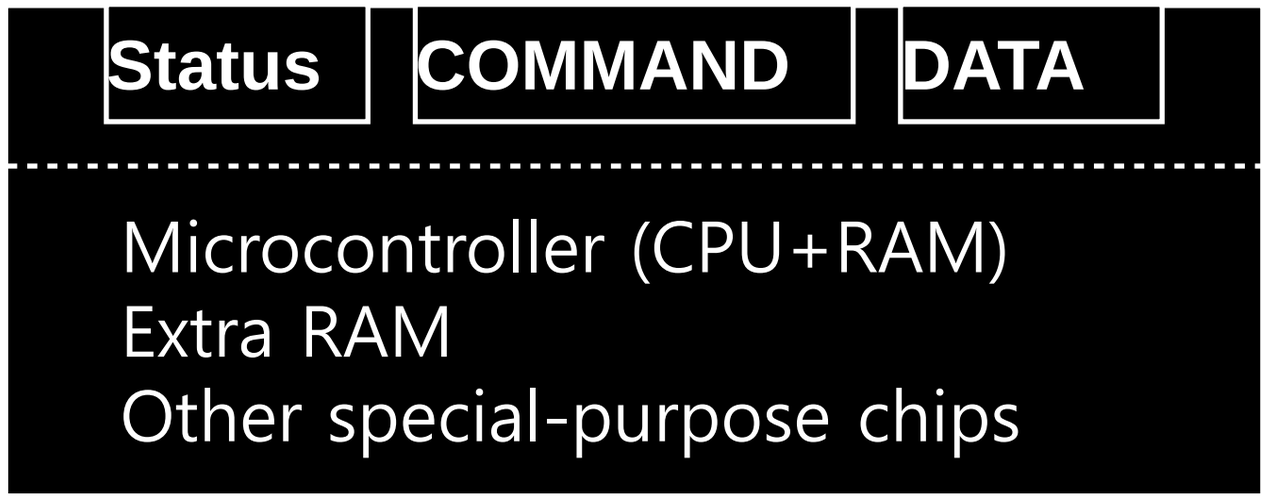


The File System Stack

A Simple IDE Disk Driver

- Four types of register
  - Control, command block, status and error
  - Memory mapped IO
  - in and out I/O instruction

Protocol Variants



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

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

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)

Variety is a Challenge

Problem:

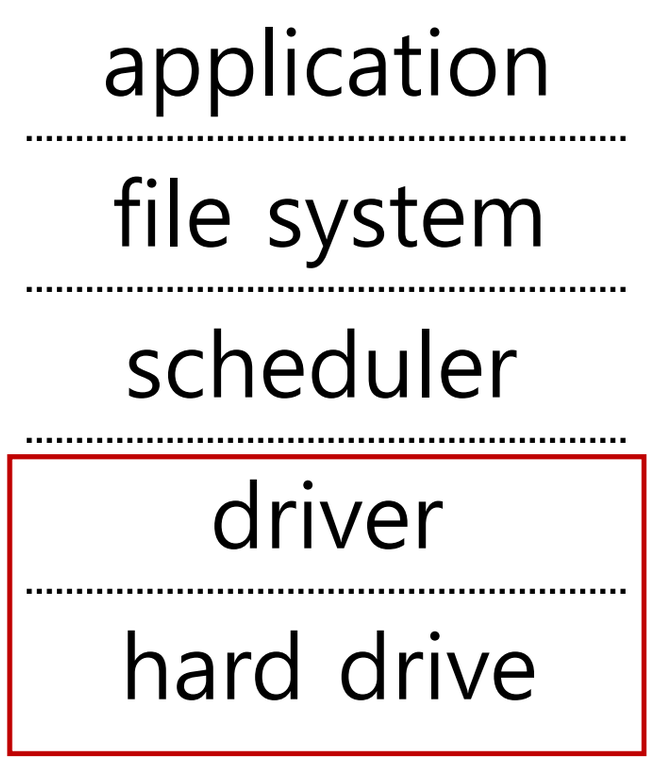
- many, many devices
- each has its own protocol

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

Solution

- Write device driver for each device
- Drivers are **70%** of Linux source code
- Encapsulation also enables us to mix-and-match devices, schedulers, and file systems.

Storage Stack



what about special capabilities?

build common interface on top of all HDDs

Hard Disks

Hard Disk Driver

- Hard disk driver have been **the main form of persistent data storage** in computer systems for decades.
  - The drive consists of a large number of **sectors** (512-byte blocks).
  - **Address Space :**
    - We can view the disk with `n` sectors as an array of sectors; `0` to `n-1`.

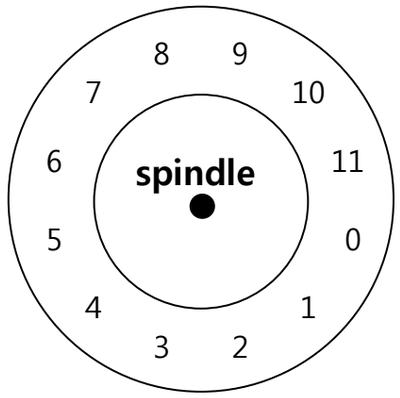
Interface

- The only guarantee is that a single 512-byte write is **atomic**.

- Multi-sector operations are possible.
  - Many file systems will read or write 4KB at a time.
  - **Torn write**:
    - If an untimely power loss occurs, only a portion of a larger write may complete.

- Accessing blocks in **a contiguous chunk** is the fastest access mode.
  - A sequential read or write
  - Much faster than any more random access pattern.

Basic Geometry



A Disk with Just A Single Track (12 sectors)

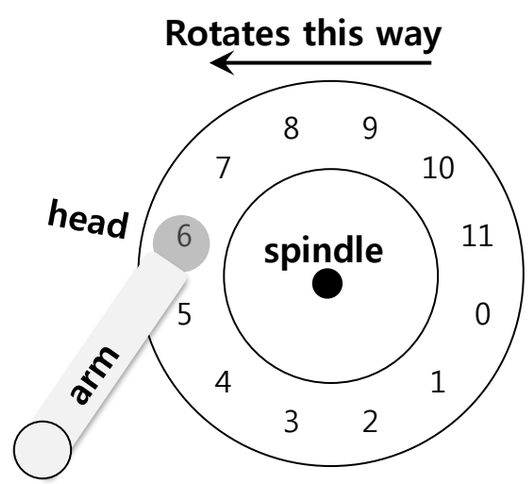
- **Platter** (Aluminum coated with a thin magnetic layer)
  - A circular hard surface
  - Data is stored persistently by inducing magnetic changes to it.
  - Each platter has 2 sides, each of which is called a **surface**.

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.
    - E.g., 10000 RPM : A single rotation takes about 6 ms.

- **Track**
  - Concentric circles of sectors
  - Data is encoded on each surface in a track.
  - A single surface contains many thousands and thousands of tracks.

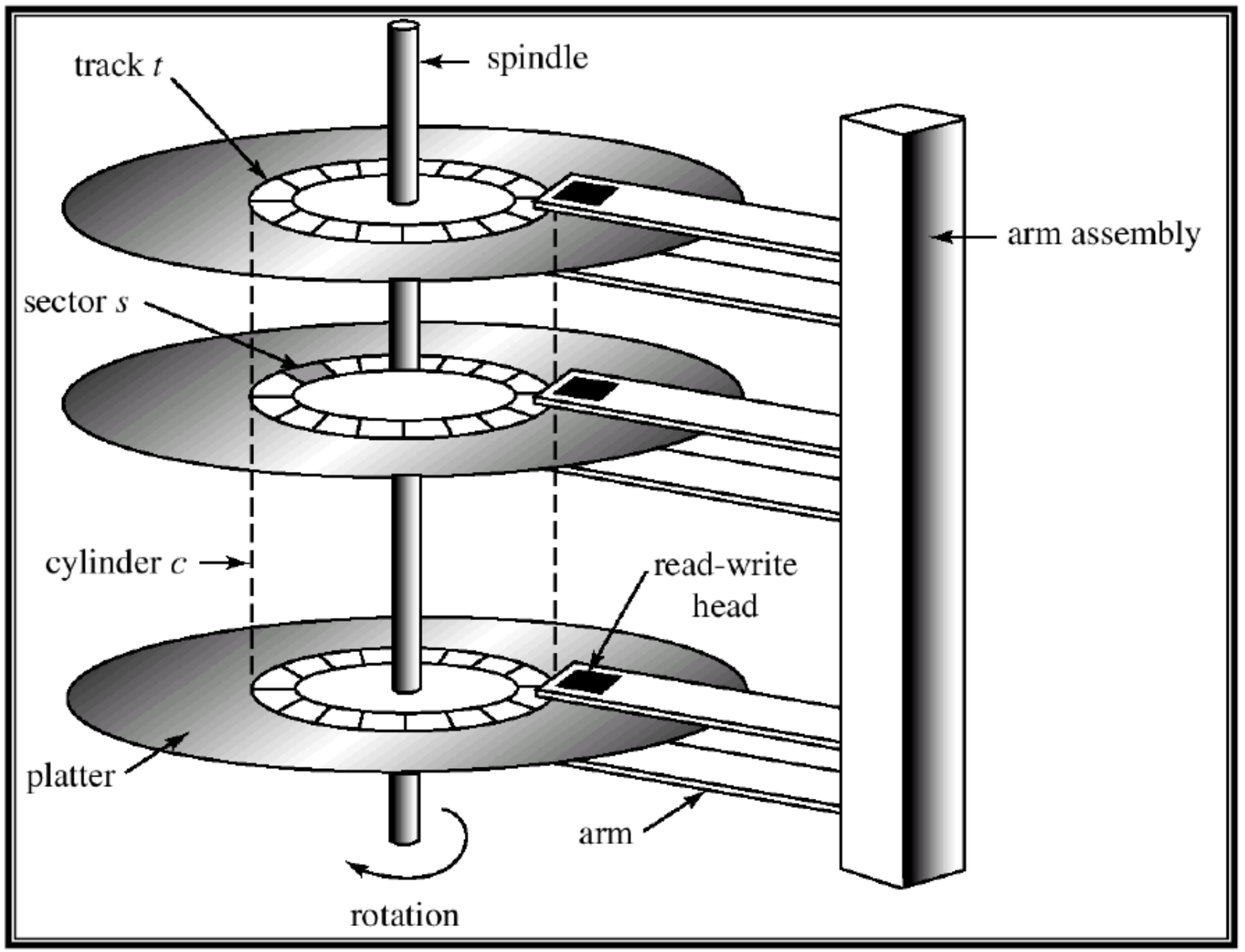
A Simple Disk Drive



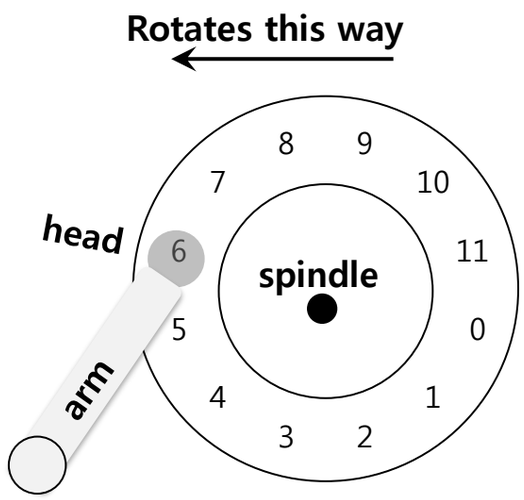
A Single Track Plus A Head

- **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.

Example of a Disk



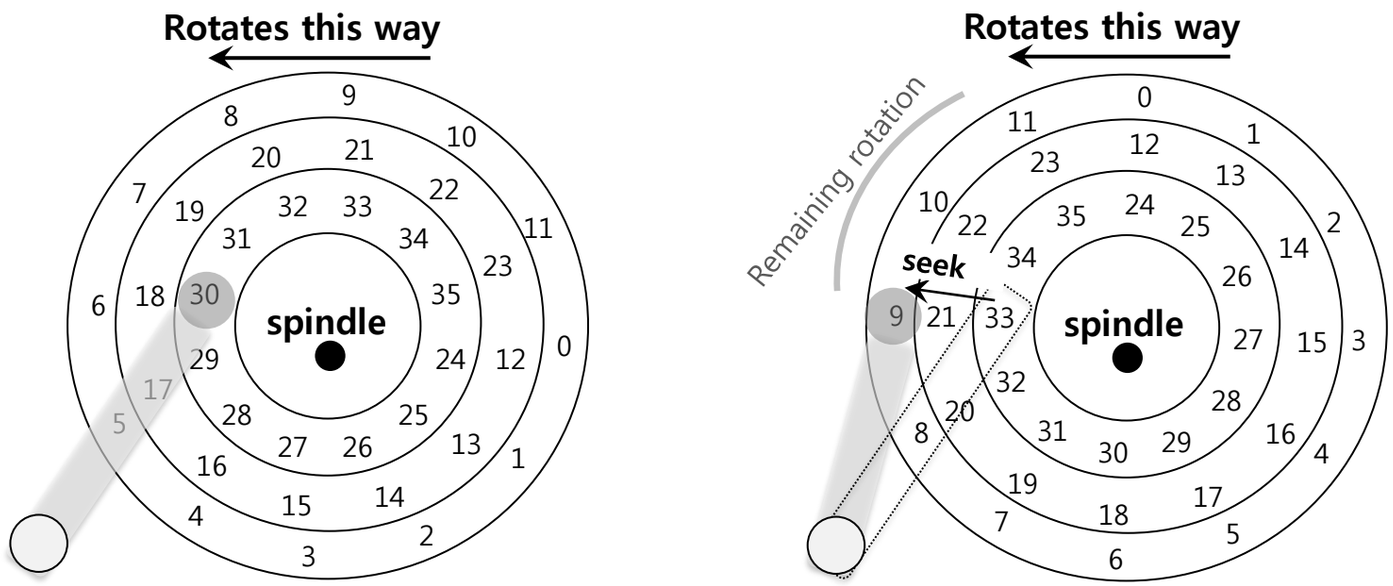
Single-track Latency: The Rotational Delay



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: Seek Time



Three Tracks Plus A Head (Right: With Seek)
(e.g., read to sector 11)

- **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.

Phases of Seek

- 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.
    - The settling time is often quite significant, e.g., 0.5 to 2ms.

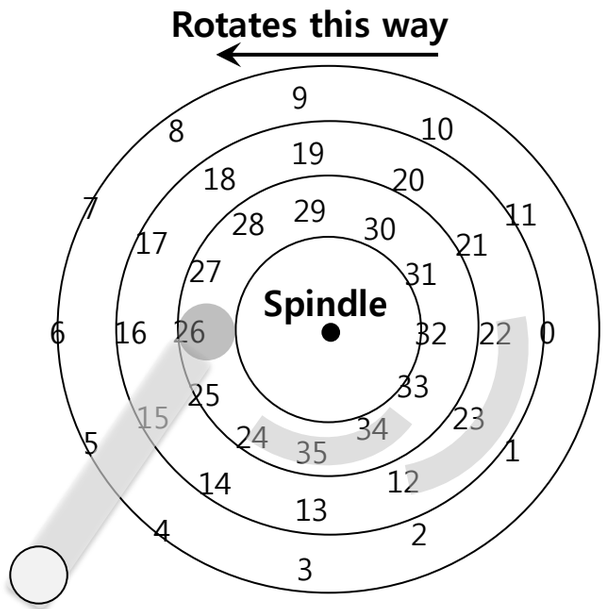
Transfer

- The final phase of I/O
  - Data is either *read from* or *written* to the surface.

- Complete I/O time:
  - **Seek**
  - Waiting for the **rotational delay**
  - **Transfer**

Track Skew

- Make sure that sequential reads can be properly serviced **even when crossing track boundaries**.



Three Tracks: Track Skew Of 2

- *Without track skew*, the head would be moved to the next track but the desired next block would have already rotated under the head.

Cache (Track Buffer)

- **Hold data** read from or written to the disk
  - Allow the drive to quickly respond to requests.
  - Small amount of memory (usually around 8 or 16 MB)

Write on cache

- **Writeback** (Immediate reporting)
  - Acknowledge a write has completed when it has **put the data in its memory**.
  - faster but dangerous
- **Write through**
  - Acknowledge a write has completed after the write has **actually been written to disk**.

I/O Time: Doing The Math

- I/O time ($T_{I/O}$): $$T_{I/O} = T_{seek} + T_{rotation} + T_{transfer}$$

- The rate of I/O ($R_{I/O}$):

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

	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

Disk Drive Specs: SCSI Versus SATA

I/O Time Example

- **Random workload**: Issue 4KB read to random locations on the disk
- **Sequential workload**: Read 100MB consecutively from the disk

		Cheetah 15K.5	Barracuda
T_{seek}		4 ms	9 ms
$T_{rotation}$		2 ms	4.2 ms
Random	$T_{transfer}$	30 microsecs	38 microsecs
	$T_{I/O}$	6 ms	13.2 ms
	$R_{I/O}$	0.66 MB/s	0.31 MB/s
Sequential	$T_{transfer}$	800 ms	950 ms
	$T_{I/O}$	806 ms	963.2 ms
	$R_{I/O}$	125 MB/s	105 MB/s

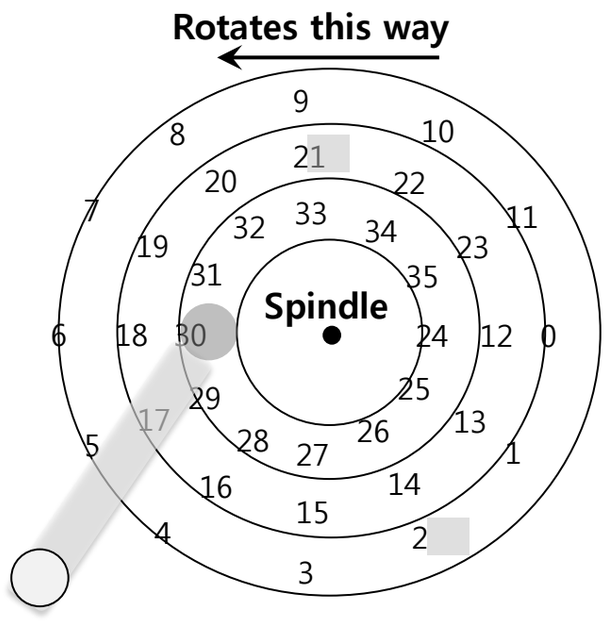
Disk Drive Performance: SCSI Versus SATA

There is a huge gap in drive performance between random and sequential workloads

Disk Scheduling

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

- **SSTF** (Shortest Seek Time First)
  - Order the queue of I/O request by track
  - Pick requests on the nearest track to complete first



SSTF: Scheduling Request 21 and 2

Issue the request to 21 → issue the request to 2

SSTF is not a panacea.

- **Problem 1**: The drive geometry is not available to the host OS
  - Solution: OS can simply implement Nearest-block-first (NBF)

- **Problem 2**: Starvation
  - If there were a steady stream of request to the inner track, request to other tracks would then be ignored completely.

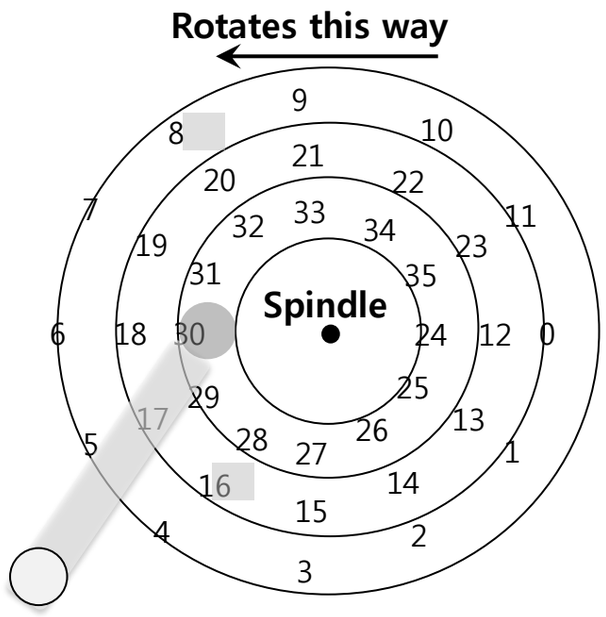
Elevator (a.k.a. SCAN or C-SCAN)

- Move across the disk servicing requests in order across the tracks.
  - **Sweep**: A single pass across the disk
    - If a request comes for a block on a track that has already been services on this sweep of the disk, it is queued until the next sweep.

- **F-SCAN**
  - Freeze the queue to be serviced when it is doing a sweep
  - Avoid starvation of far-away requests

- **C-SCAN** (Circular SCAN)
  - Sweep from outer-to-inner, and then inner-to-outer, etc.

How to account for Disk rotation costs?



SSTF: Sometimes Not Good Enough

- If rotation is faster than seek : request 16 → request 8
- If seek is faster than rotation : request 8 → request 16

On modern drives, both seek and rotation are roughly equivalent:
Thus, SPTF (Shortest Positioning Time First) is useful.

I/O merging

- **Reduce the number of request** sent to the disk and lowers overhead
  - E.g., read blocks 33, then 8, then 34:
    - The scheduler merge the request for blocks 33 and 34 *into a single two-block request*.

Basic Interface

Disk has a sector-addressable address space

- Appears as an array of sectors

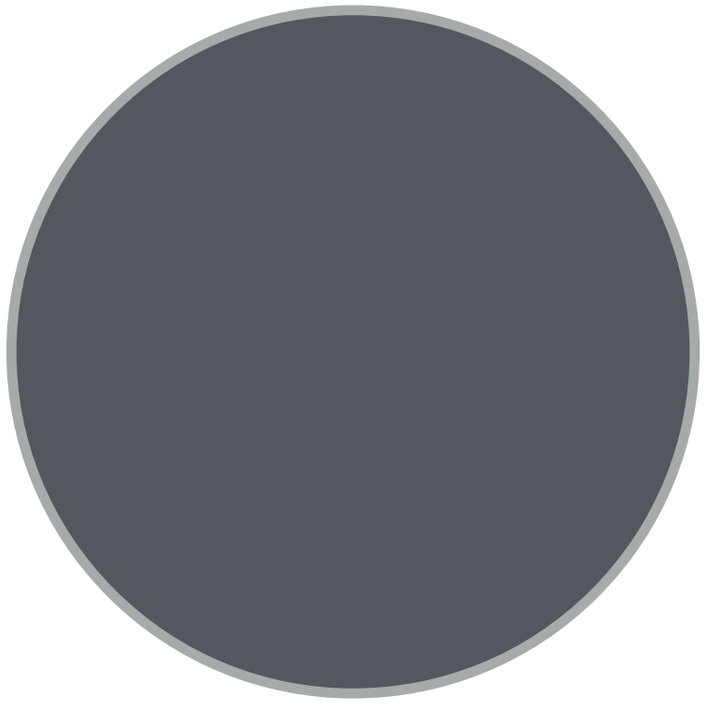
Sectors are typically 512 bytes or 4096 bytes.

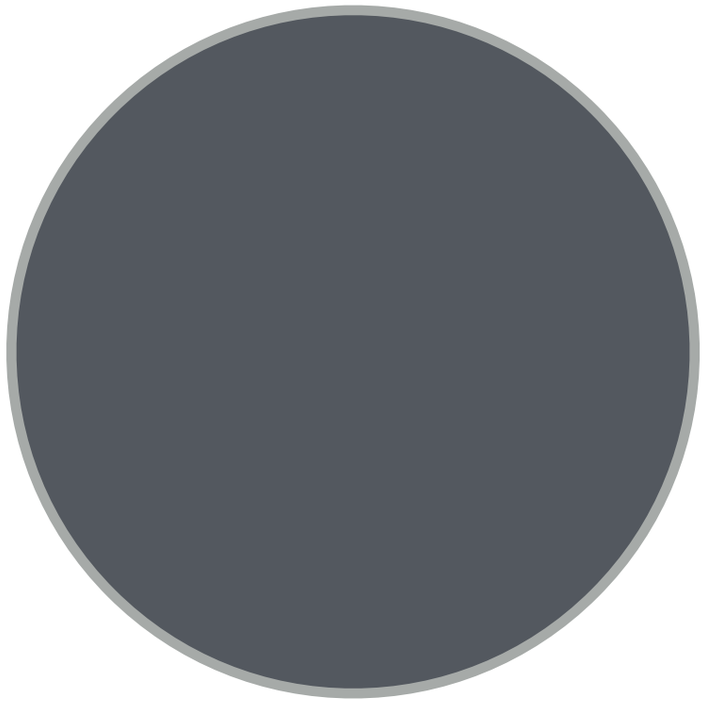
Main operations: reads + writes to sectors

Mechanical (slow) nature makes management “int eresting”

Disk Internals

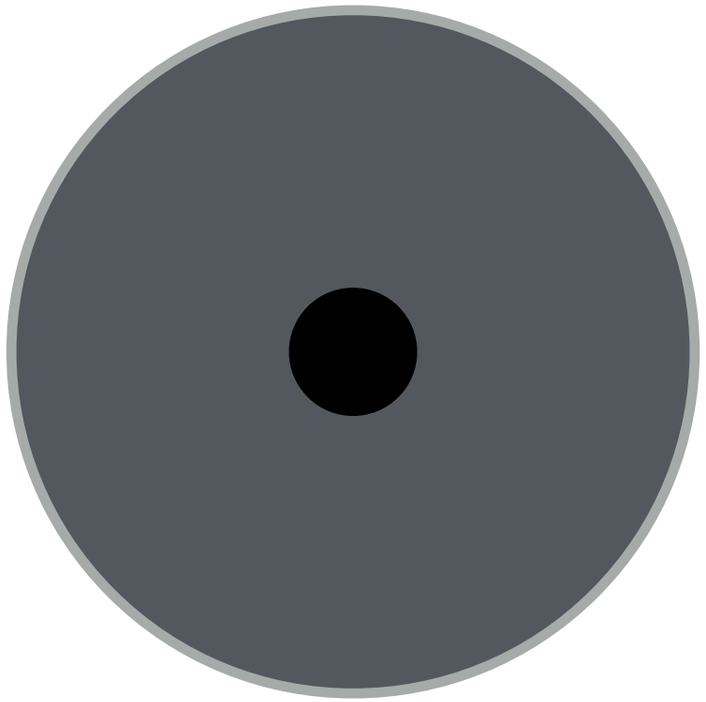
Platter

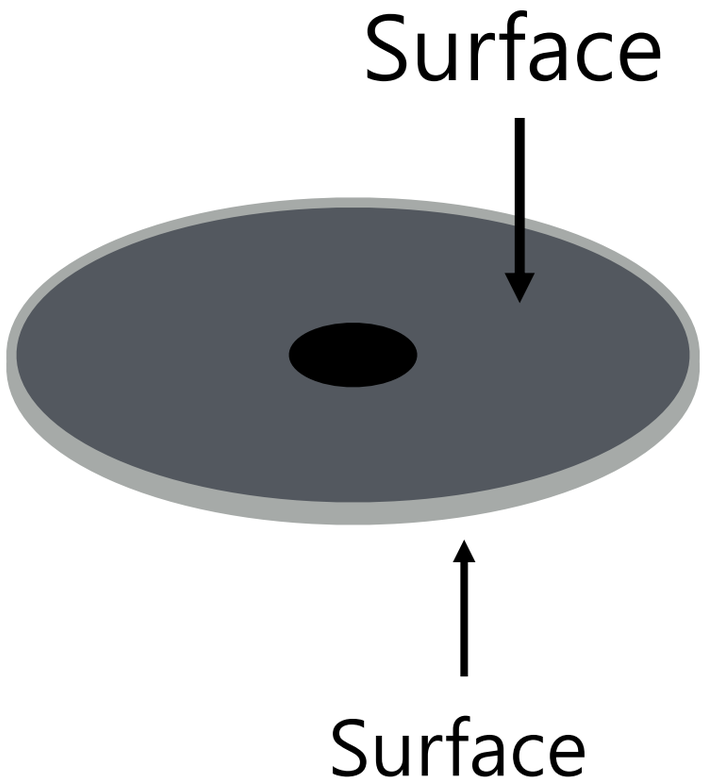


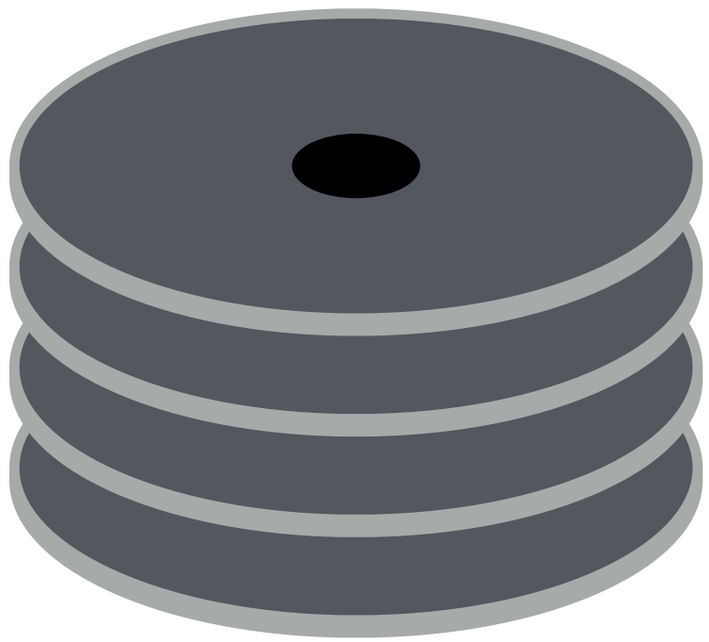


Platter is covered with a magnetic film.

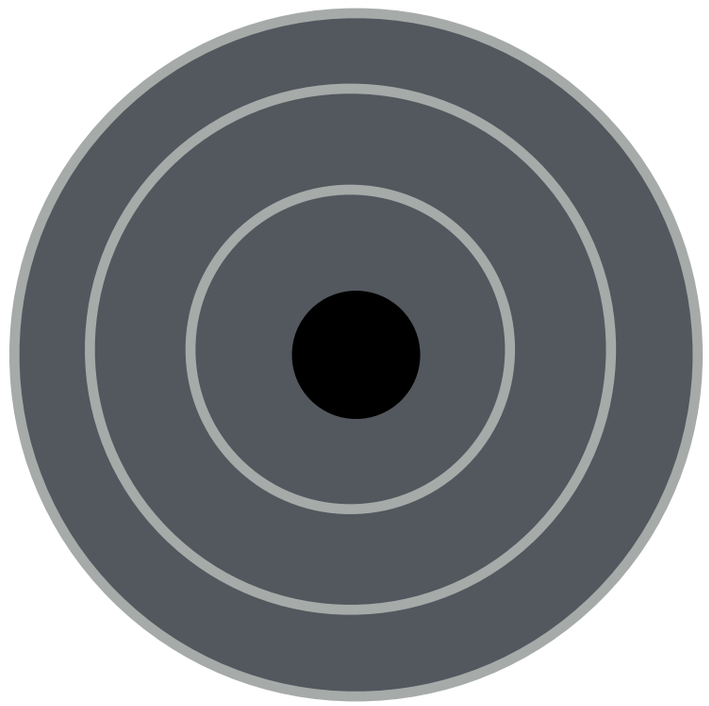
Spindle



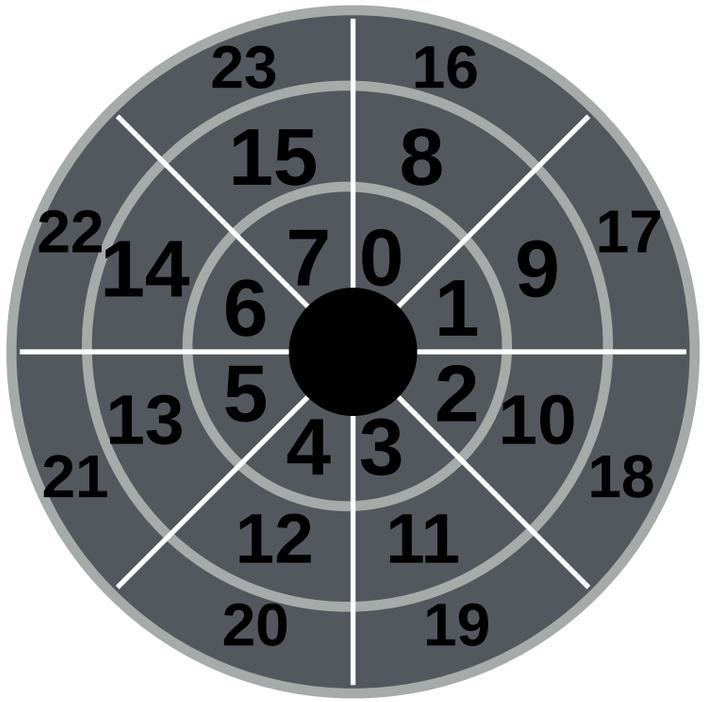

Surface
Surface



Many platters may be bound to the spindle.

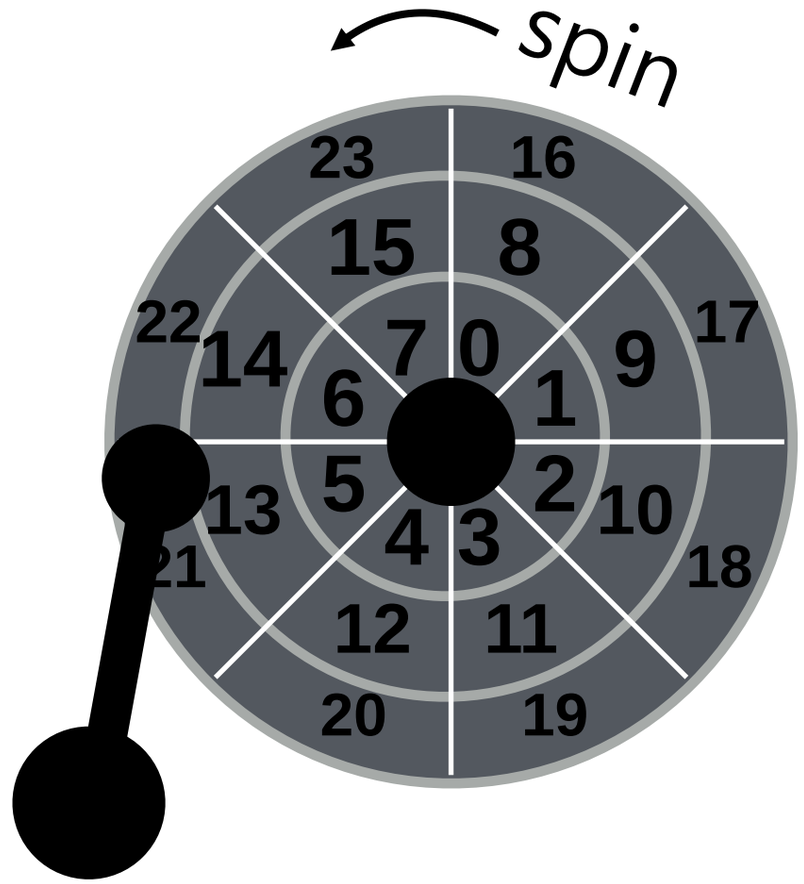


Each surface is divided into rings called tracks.
A stack of tracks (across platters) is called a cylinder.



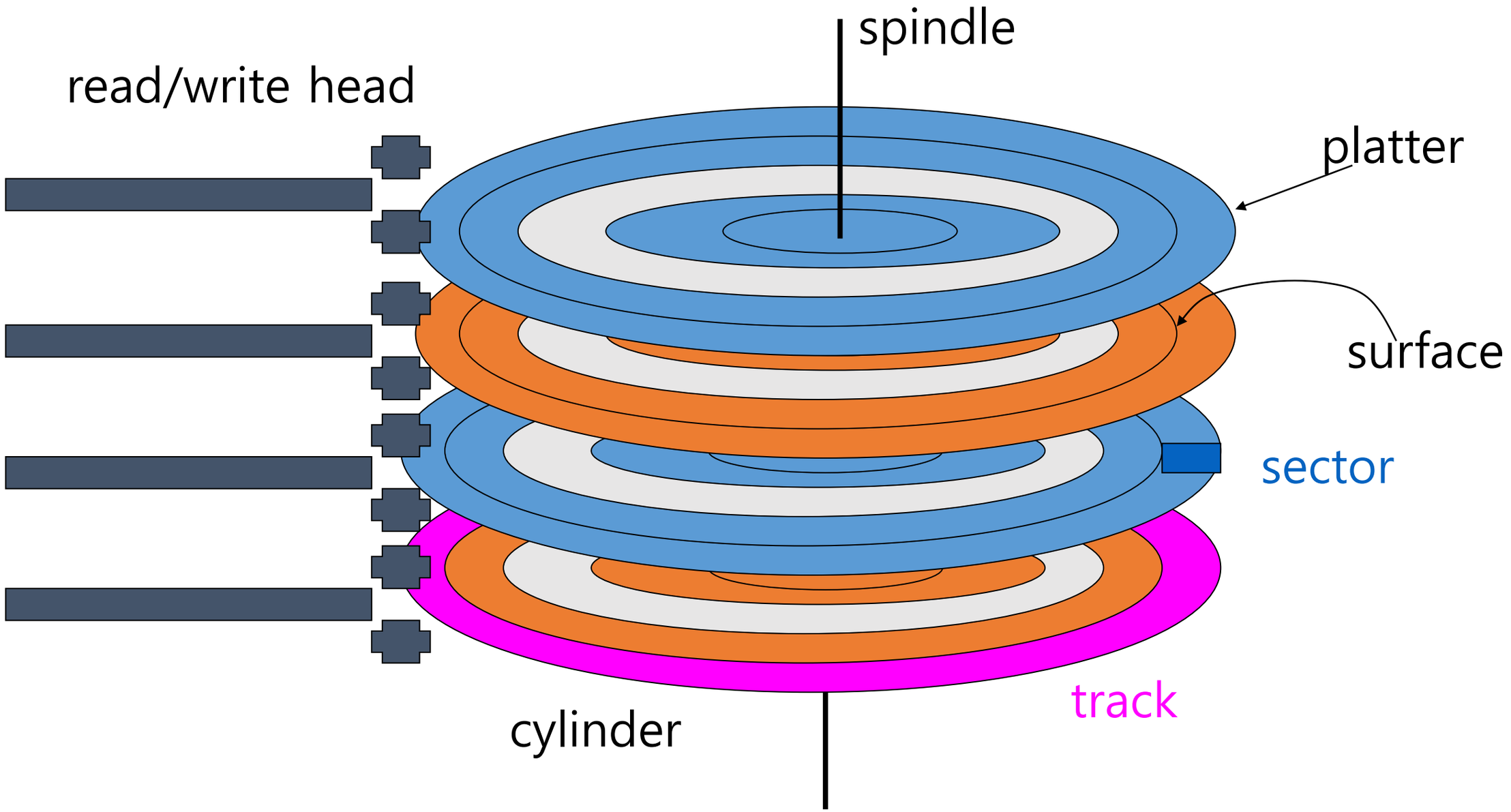
The tracks are divided into numbered sectors.

Heads on a moving arm can read from each surface.



Spindle/platters rapidly spin.

Disk Terminology



Hard Drive Demo

http://youtu.be/9eMWG3fwiEU?t=30s

https://www.youtube.com/watch?v=L0nbo1VOF4M

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)

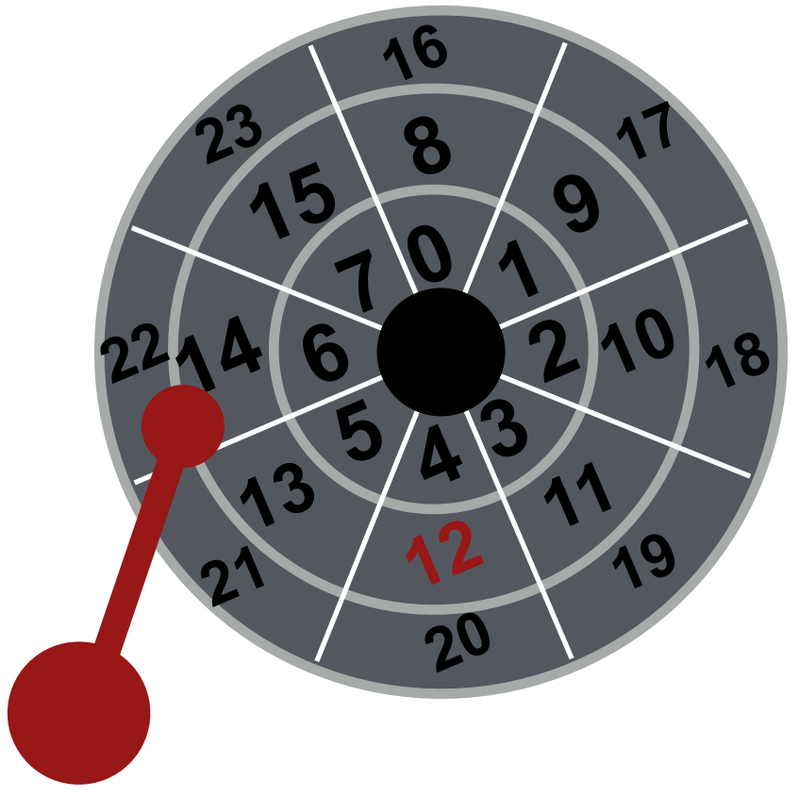
Use servo burst:

- Record placement information every few (3-5) sectors
- When head cross servo burst, figure out location and adjust a s needed

Let’s Read 12!

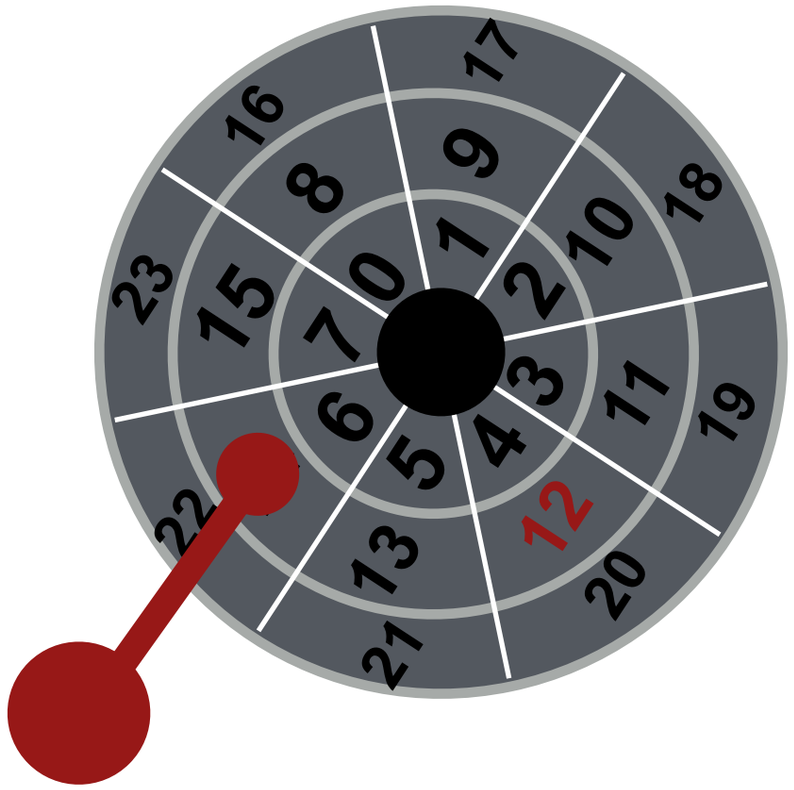
Seek to right track.

Seek to right track.

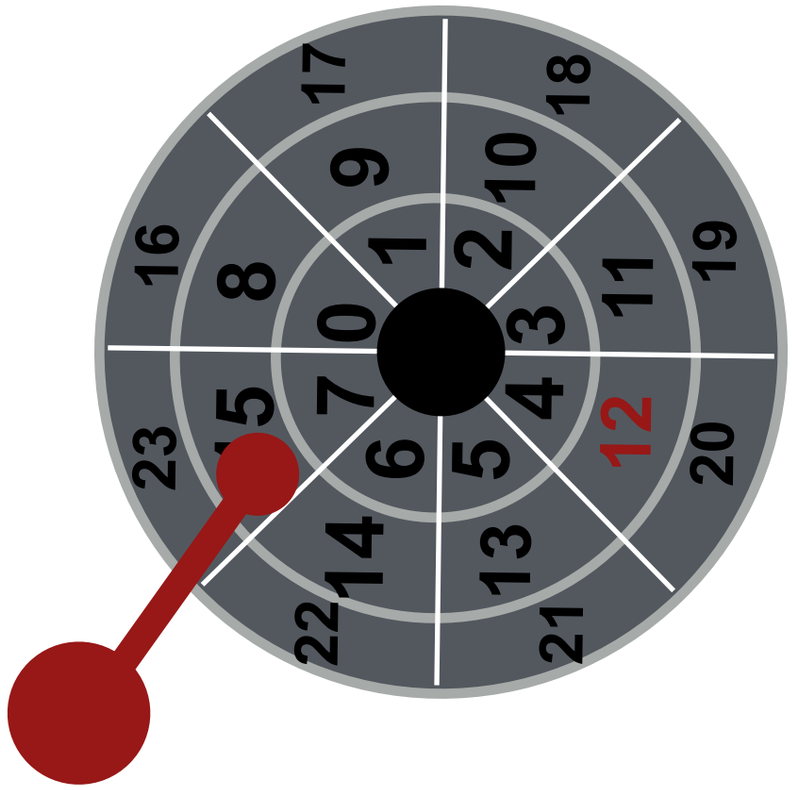


Seek to right track.

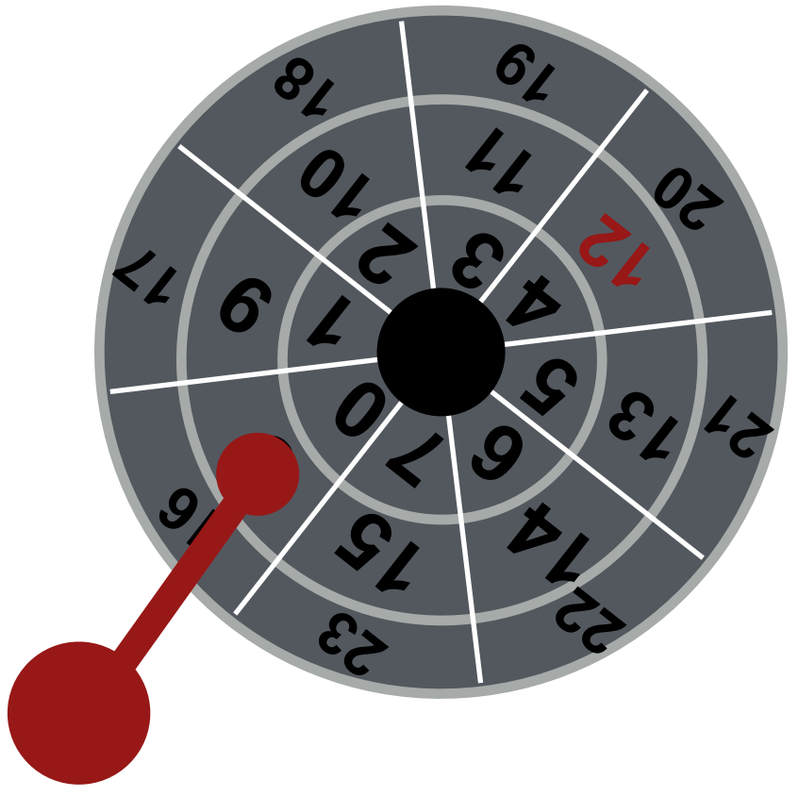
Wait for rotation.



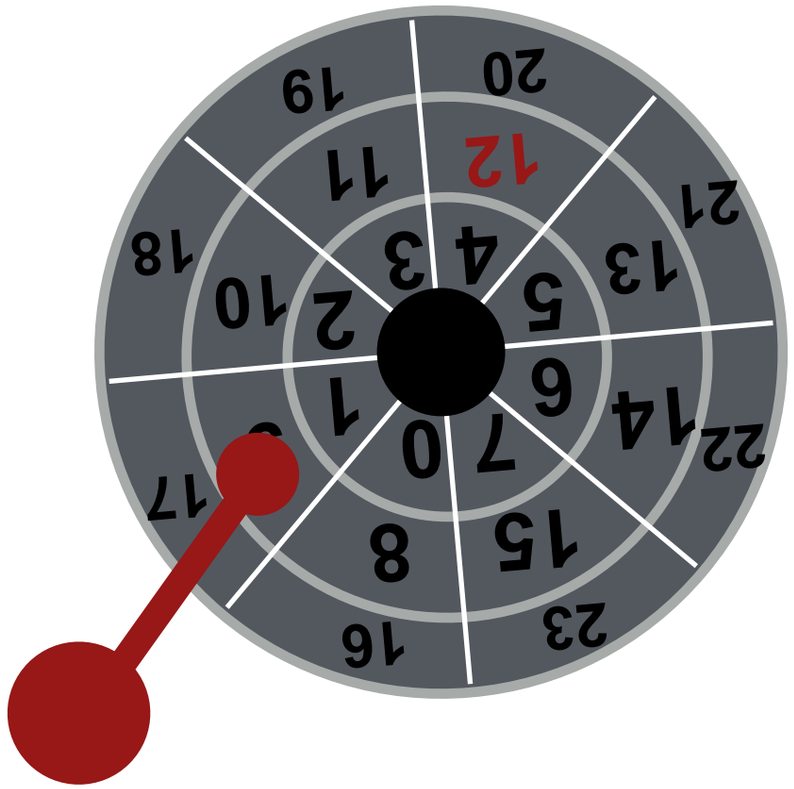
Wait for rotation.



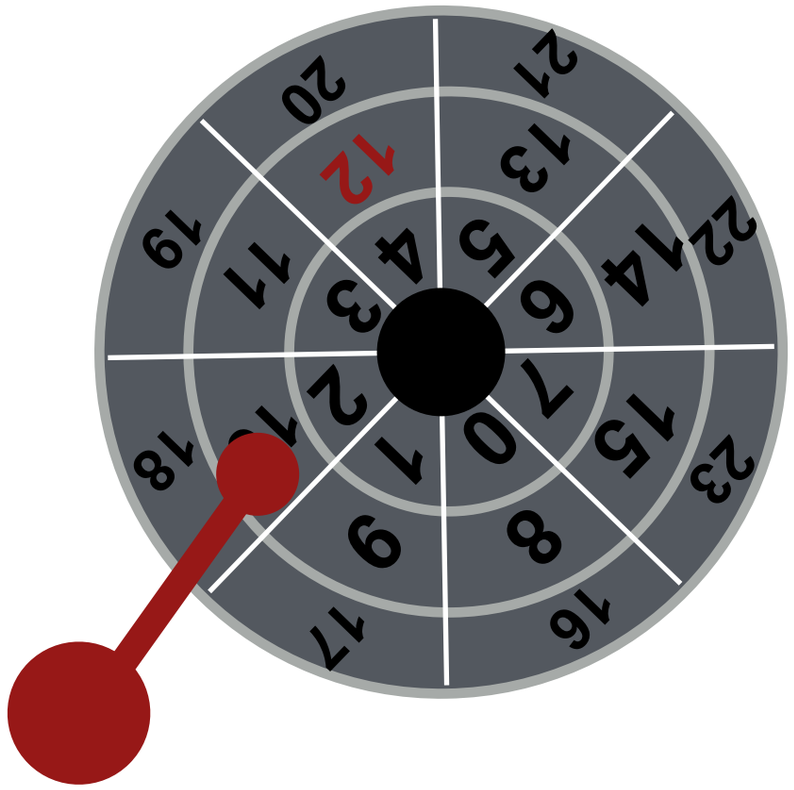
Wait for rotation.



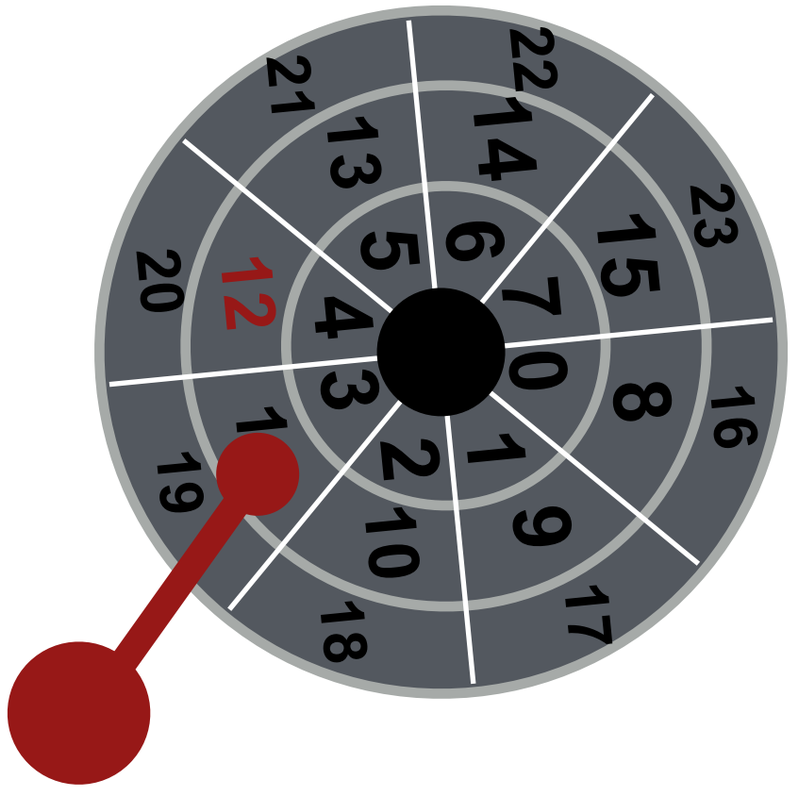
Wait for rotation.



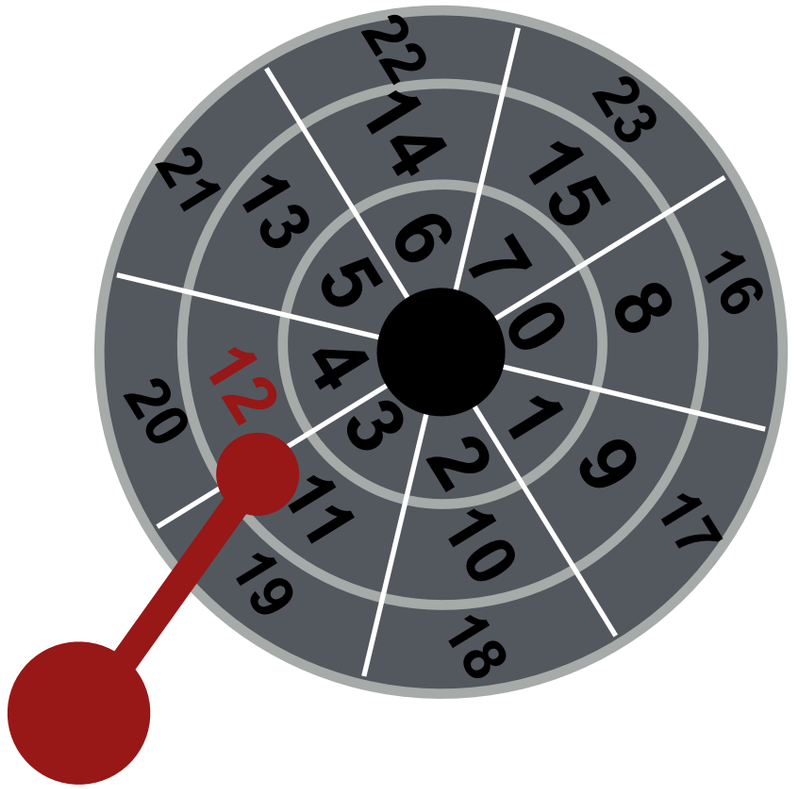
Wait for rotation.



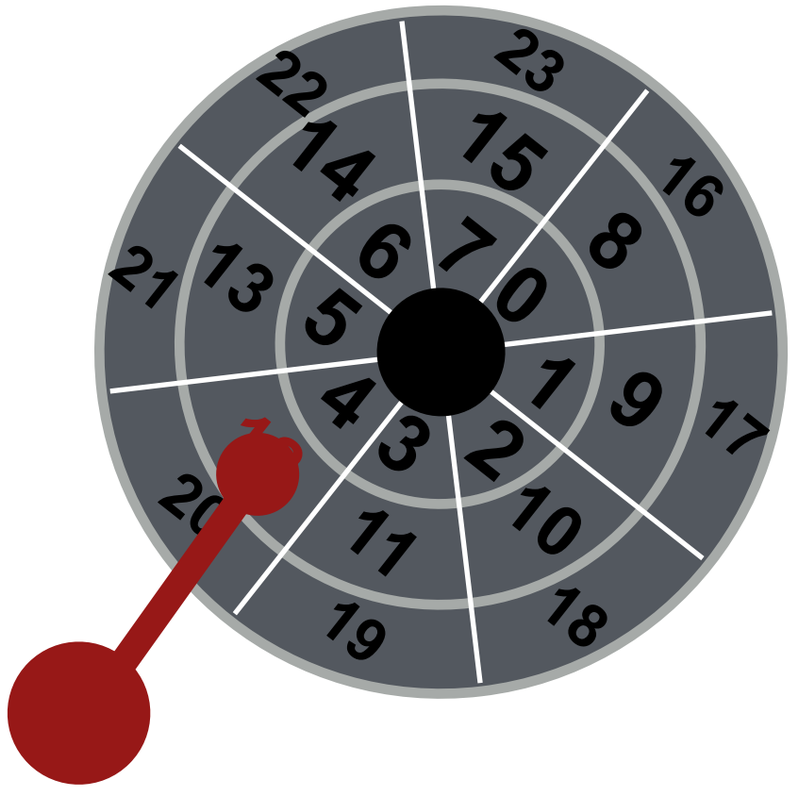
Wait for rotation.



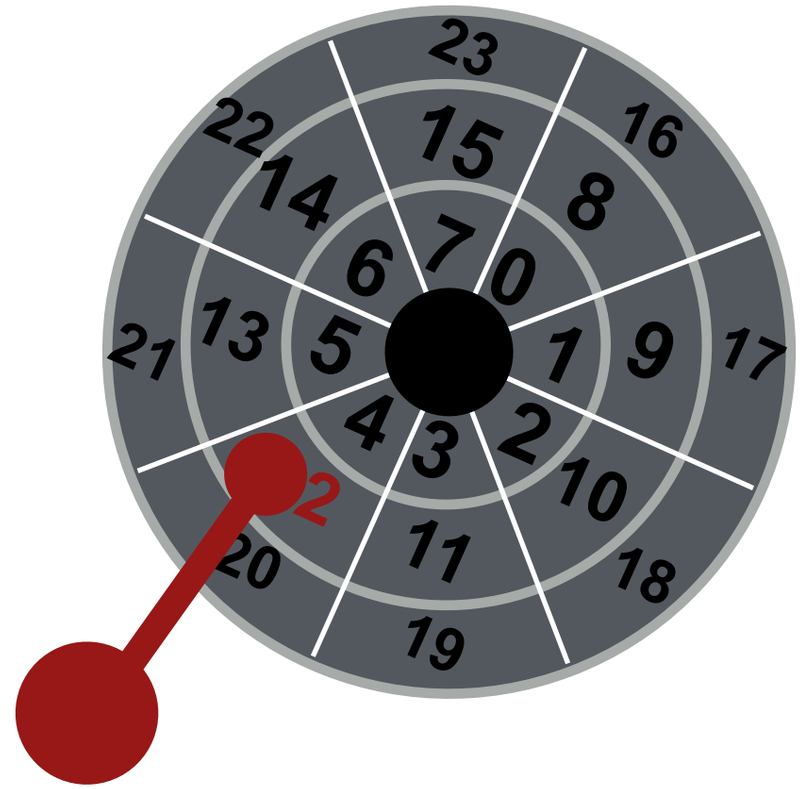
Transfer data.



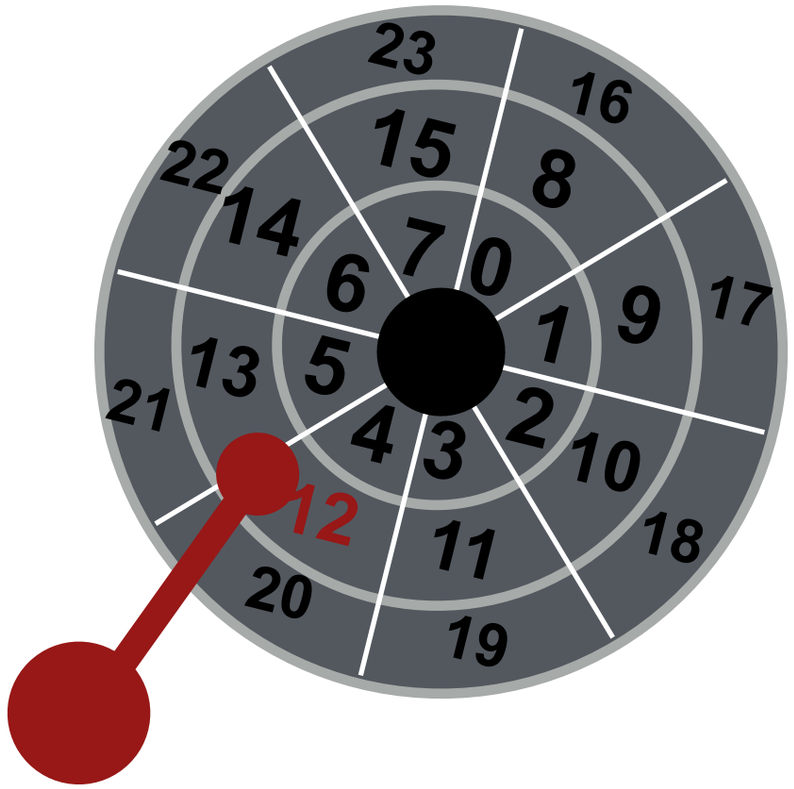
Transfer data.



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)

: 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?

$$\text{transfer} = \frac{1 \text{ sec}}{125 \text{ MB}} \times 16 \text{ KB} \times \frac{1{,}000{,}000 \text{ us}}{1 \text{ sec}} = 125 \text{ us}$$

	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/ Cheetah?

Cheetah time = 4ms + 2ms + 125us = 6.1ms

Throughput?

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

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

	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

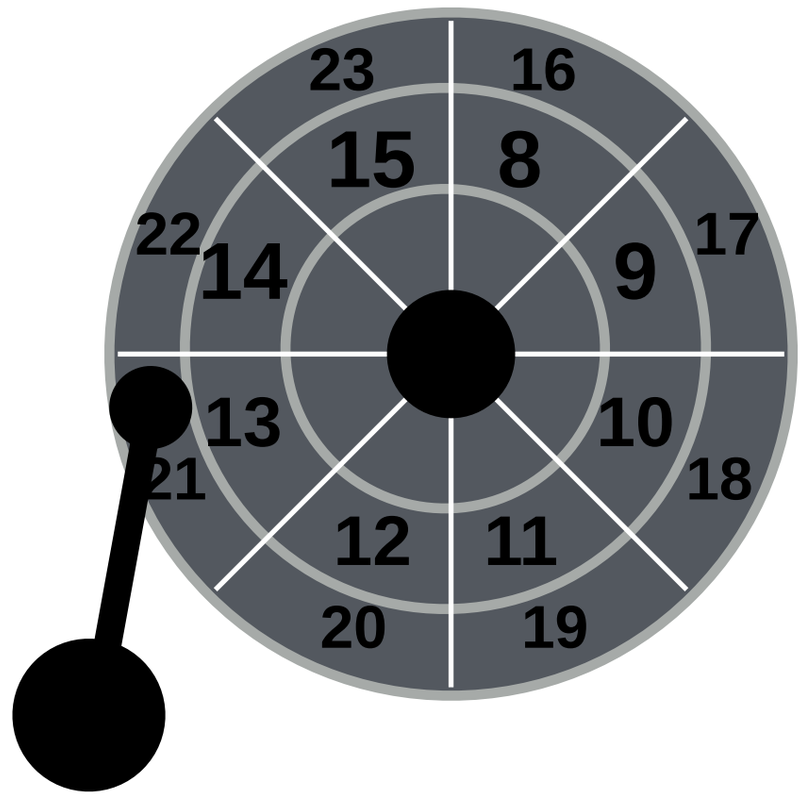
Other Improvements

Track Skew

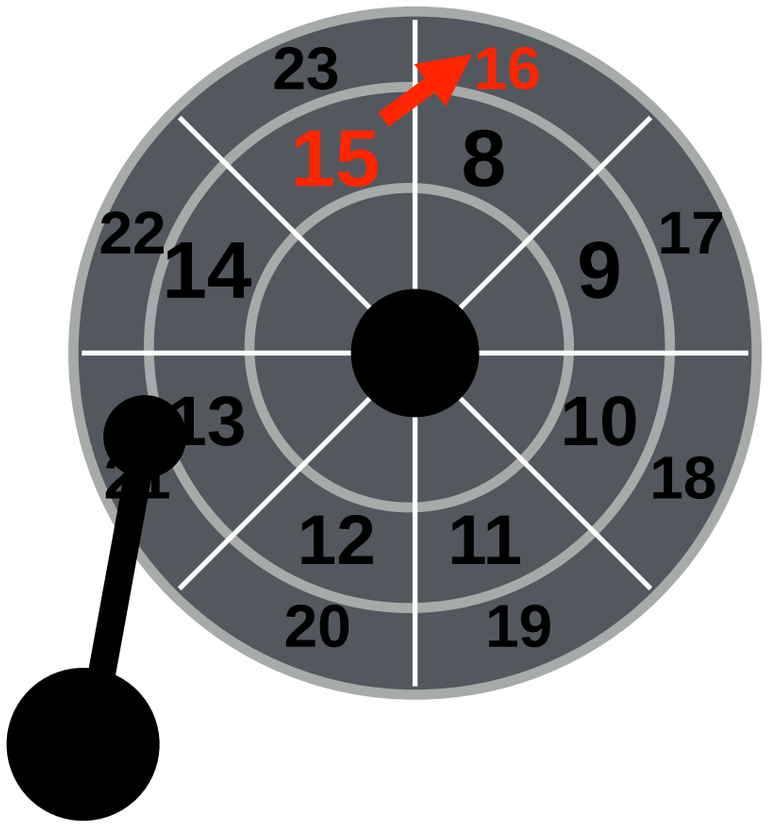
Zones

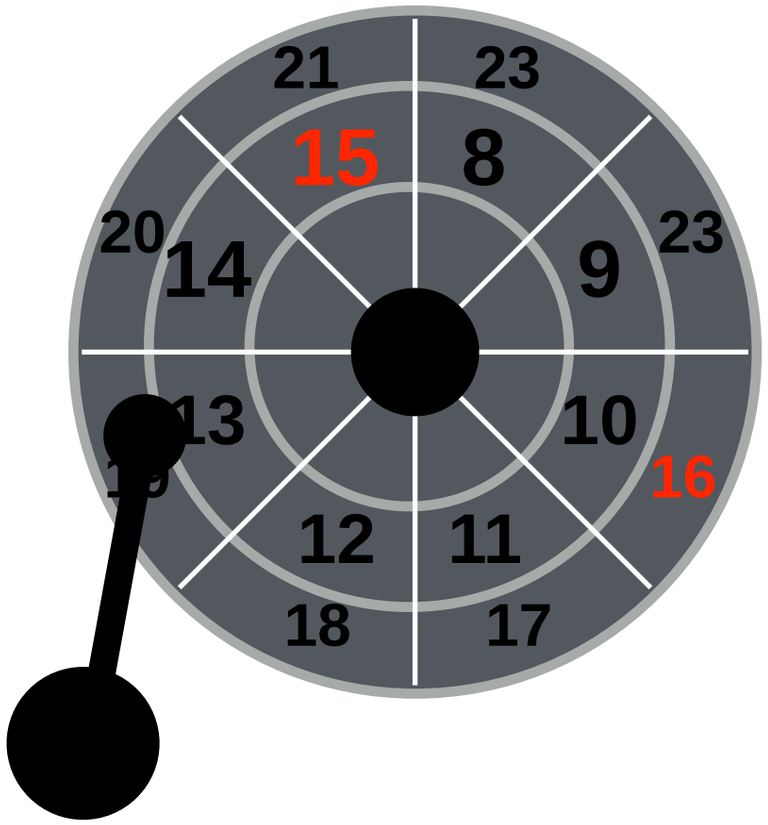
Cache

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

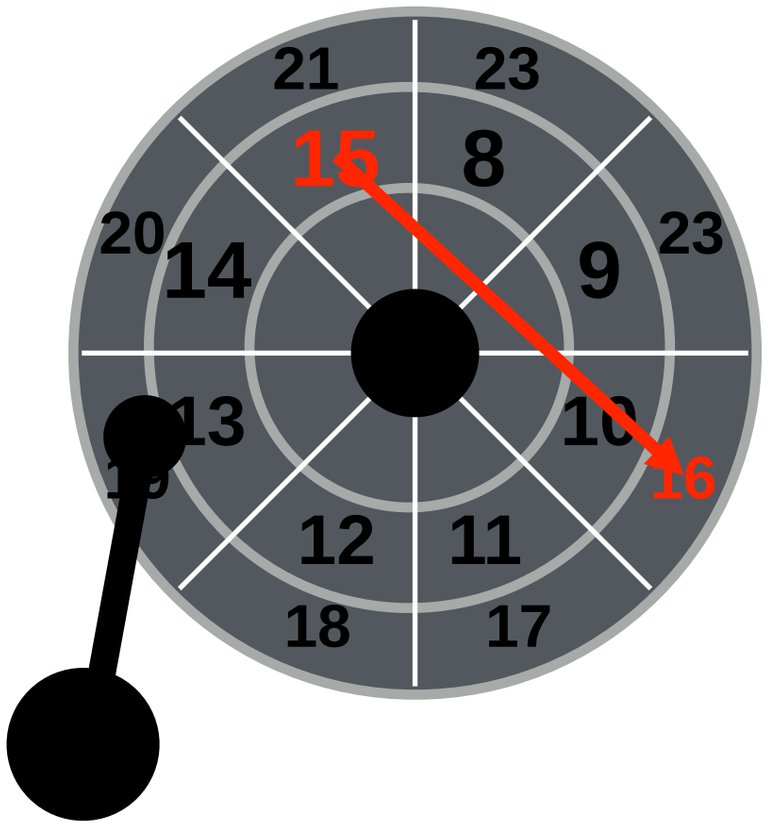


When reading 16 after 15, the head won't settle quick enough, so we need to do a rotation.




21
23
15
8
20
14
9
23
13
10
16
12
11
18
17

enough time to settle now

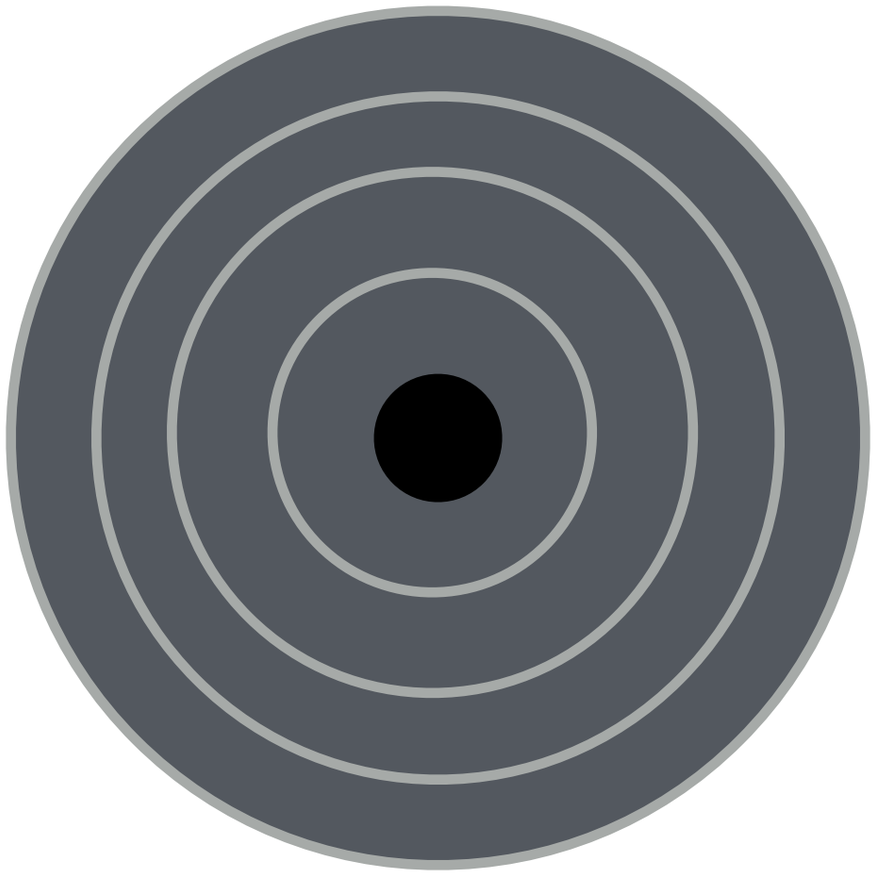


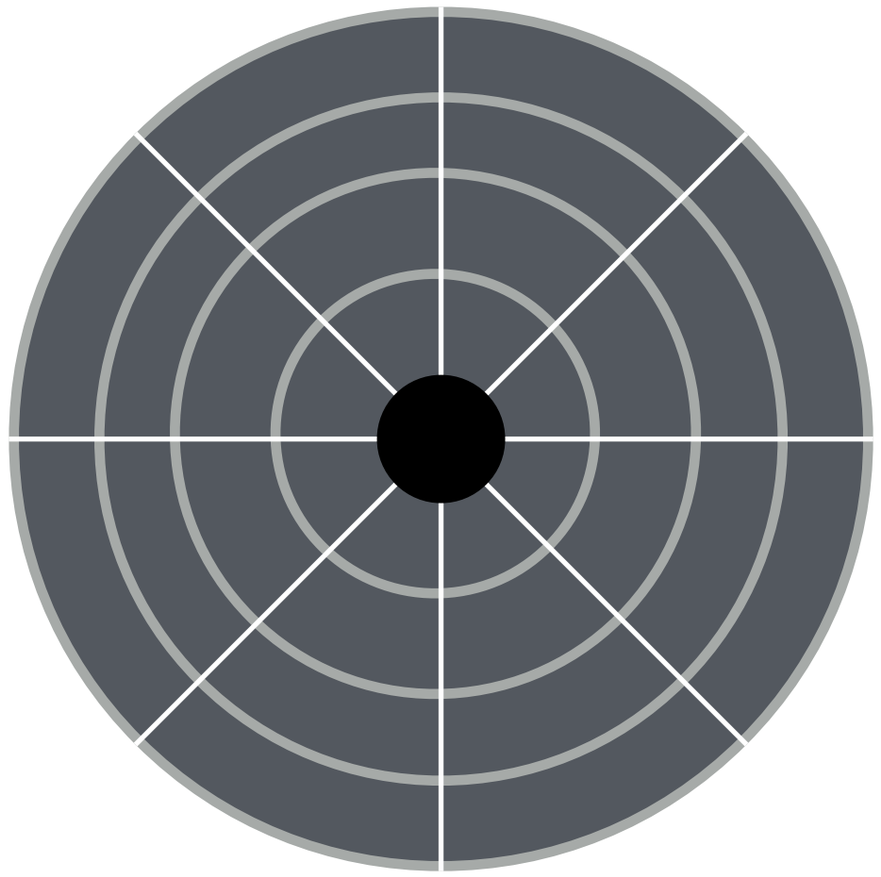
Other Improvements

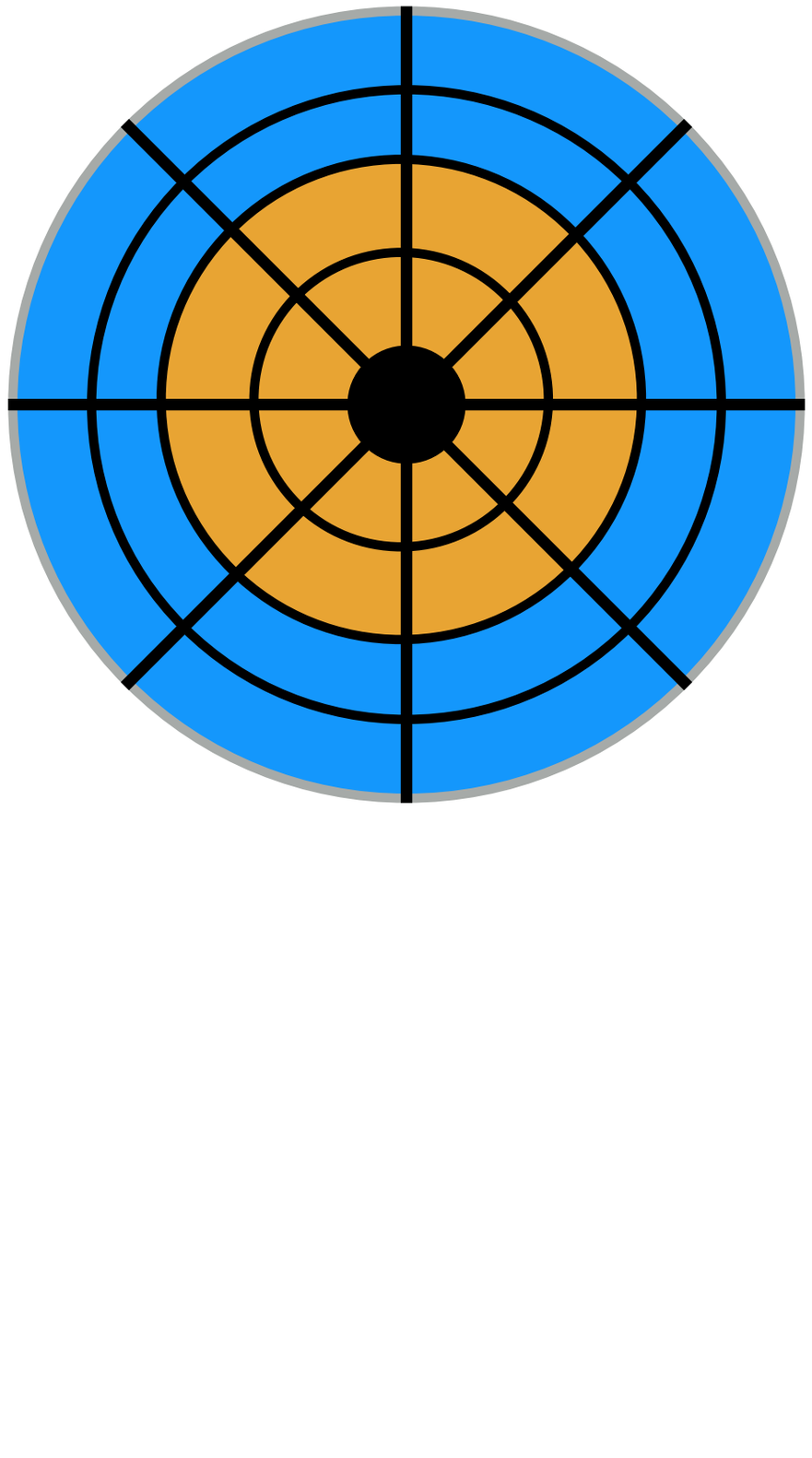
Track Skew

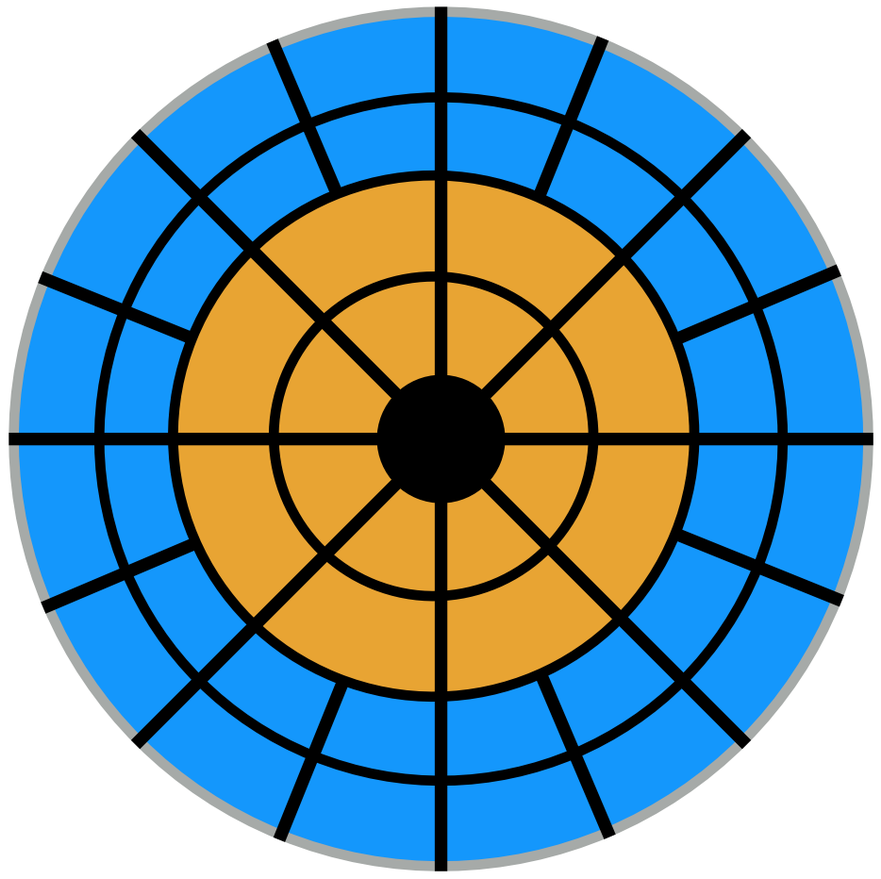
Zones

Cache









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

Other Improvements

Track Skew

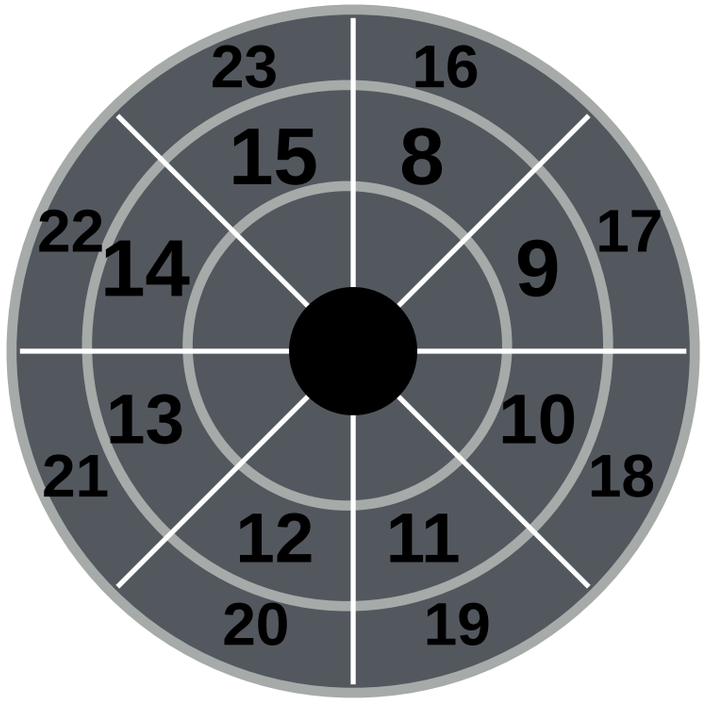
Zones

Cache

Drive Cache

Drives may cache both reads and writes.

- OS caches data too



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"

- Read contents of entire track into memory during rotational delay

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
Assume transfer = 100 MB/s

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
Assume transfer = 100 MB/s

How long (roughly) does the below workload take?

- Requests are given in sector numbers

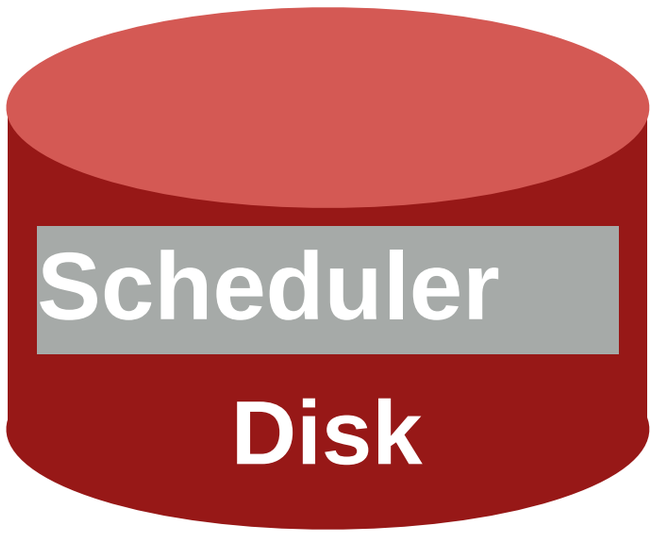
~60ms

300001, 700001, 300002, 700002, 300003, 700003

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 positio ning time (time for seeking and rotating)

- Greedy algorithm (just looks for best NEXT decision)

How to implement in **disk**?

How to implement in **OS**?

Use Shortest Seek Time First (SSTF) instead

Disadvantages?

Easy for far away requests to **starve**

SCAN

Elevator Algorithm:

- Sweep back and forth, from one end of disk other, ser ving requests as pass that cylinder
- Sorts by cylinder number; ignores rotation dela ys

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 pr oportional 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