ANNOUNCEMENTS

Project 2a: Graded – see Learn@UW; contact your TA if questions
Part 2b will be graded end of next week

Exam 2: Wednesday 11/9 7:15 – 9:15 Humanities 3650.

- Assume you have notified us of academic conflicts
- Covers all of Concurrency Piece (lecture and book) + I/O (and RAID?)
  - Light on chapter 29, nothing from chapter 33; chapters 36, 37, 38
  - Few review questions from Virtualization Piece
- Look at homework simulators
- Questions from Project 2
- See previous practice exam

Project 3: Extension til Friday 11/04 9:00 pm

- Use name "stats_" and not "stat_"

Today's Reading: Chapter 36 + 37

UNIVERSITY of WISCONSIN-MADISON
Computer Sciences Department

CS 537
Introduction to Operating Systems

Andrea C. Arpaci-Dusseau
Remzi H. Arpaci-Dusseau

PERSISTENCE: I/O DEVICES

Questions answered in this lecture:

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

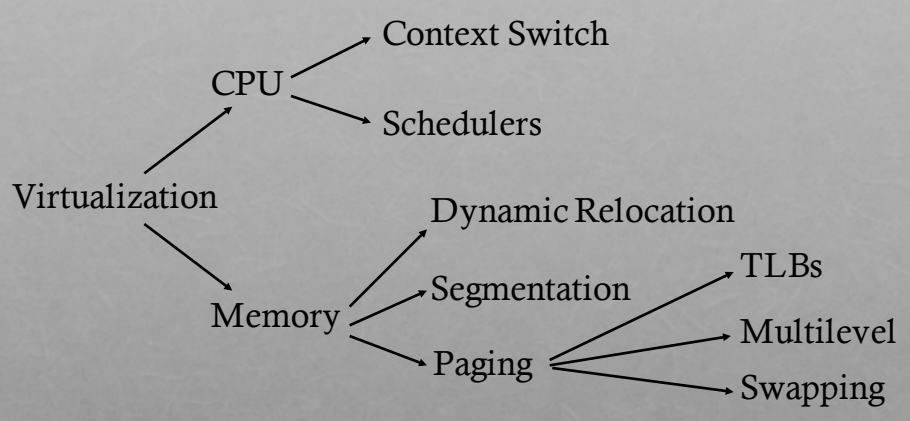
What is a **device driver**?

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

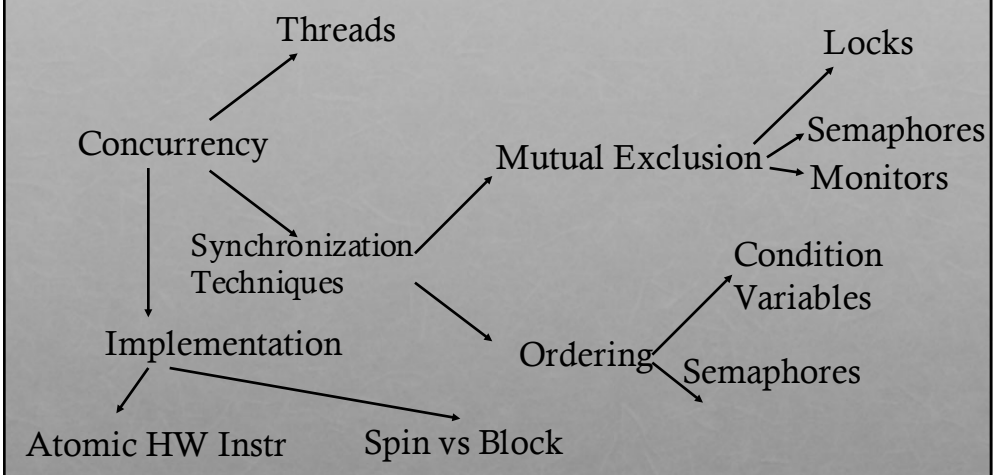
How can you calculate **sequential** and **random throughput** of a disk?

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

REVIEW: EASY PIECE 1



REVIEW: EASY PIECE 2



3RD EASY PIECE

How to provide **persistence**?

How to ensure data is available even after power outages, hardware failure, system crashes?

MOTIVATION FOR I/O

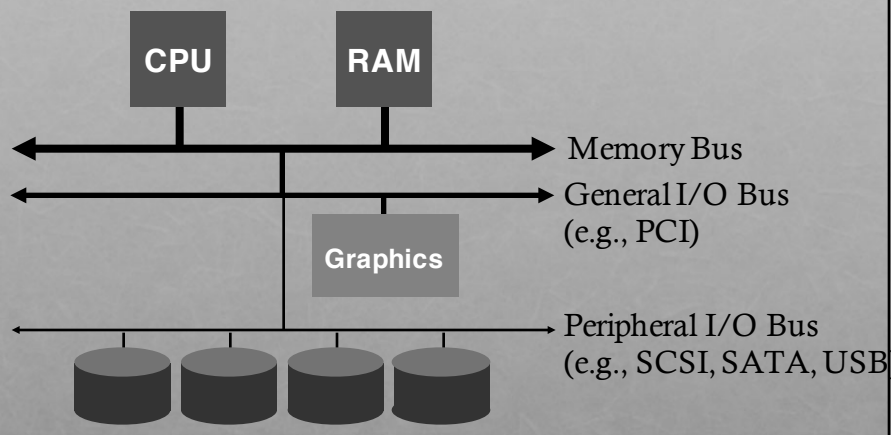
What good is a computer without any I/O devices?

- Examples: keyboard, display, disks, network
- What if no input?
- What if no output?

We want:

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

Hardware Support for I/O



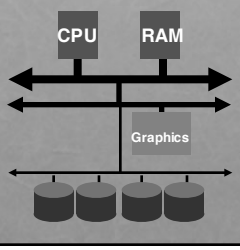
Why use hierarchical buses?

Canonical Device

OS reads/writes to these

Device Registers: Status COMMAND DATA

Hidden Internals:



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

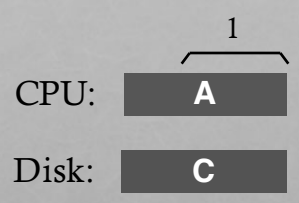
```
while (STATUS == BUSY)            // 1
  ;
Write data to DATA register       // 2
Write command to COMMAND register  // 3
while (STATUS == BUSY)            // 4
  ;
```

A wants to do I/O

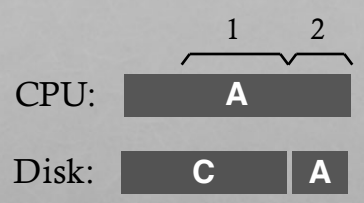
CPU: A

Disk: C

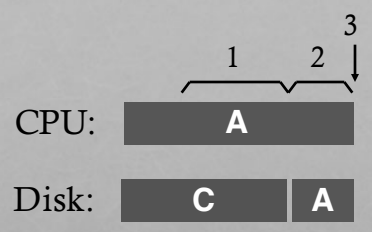
```
while (STATUS == BUSY)            // 1
  ;
Write data to DATA register       // 2
Write command to COMMAND register  // 3
while (STATUS == BUSY)            // 4
  ;
```



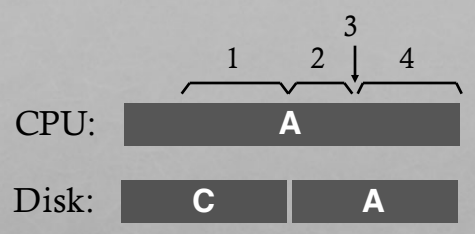

```
while (STATUS == BUSY)            // 1
  ;
Write data to DATA register       // 2
Write command to COMMAND register  // 3
while (STATUS == BUSY)            // 4
  ;
```



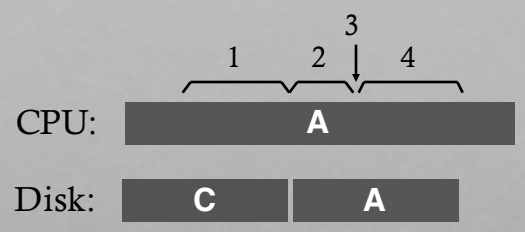

```
while (STATUS == BUSY)            // 1
  ;
Write data to DATA register       // 2
Write command to COMMAND register  // 3
while (STATUS == BUSY)            // 4
  ;
```



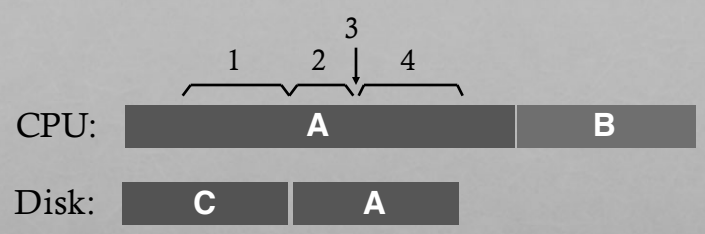

```
while (STATUS == BUSY)              // 1
  ;
Write data to DATA register         // 2
Write command to COMMAND register   // 3
while (STATUS == BUSY)              // 4
  ;
```




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



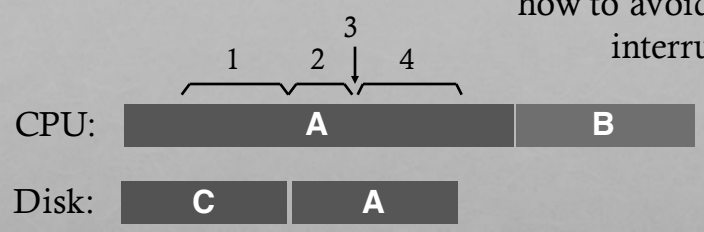

```
while (STATUS == BUSY)              // 1
  ;
Write data to DATA register         // 2
Write command to COMMAND register   // 3
while (STATUS == BUSY)              // 4
  ;
```




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

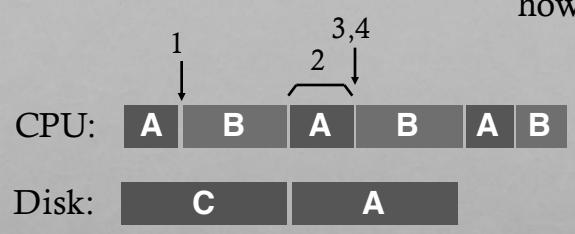
how to avoid spinning? interrupts!

how to avoid spinning?
interrupts!




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

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

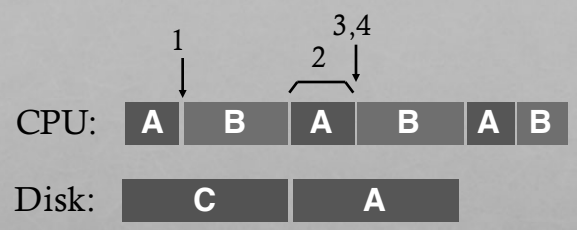
- Might not finish one interrupt before next arrives
- Better to **ignore** interrupts while make some progress handling them

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

what else can we optimize? data transfer in step 2!

PROGRAMMED I/O VS. DIRECT MEMORY ACCESS

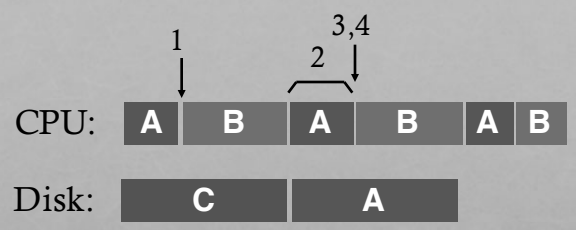
PIO (Programmed I/O):

- CPU directly tells device what the data is

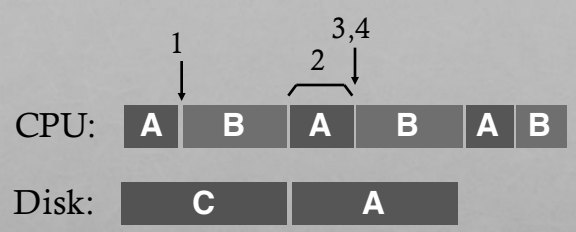
DMA (Direct Memory Access):

- CPU leaves data in memory
- OS tells device where data lives in memory
- Device reads data directly from memory

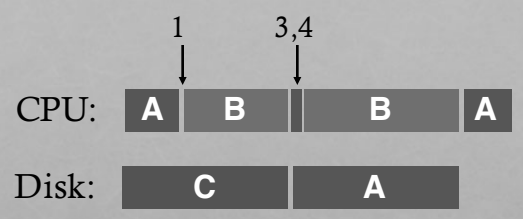
Complications with DMA?




```
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;
```




```
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;
```



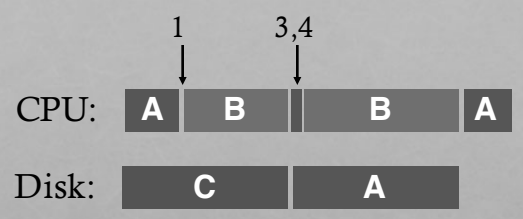

```
while (STATUS == BUSY)              // 1
      wait for interrupt;
Write data to DATA register         // 2
Write command to COMMAND register   // 3
while (STATUS == BUSY)              // 4
      wait for interrupt;
```

PROTOCOL VARIANTS

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

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

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




```
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 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~~

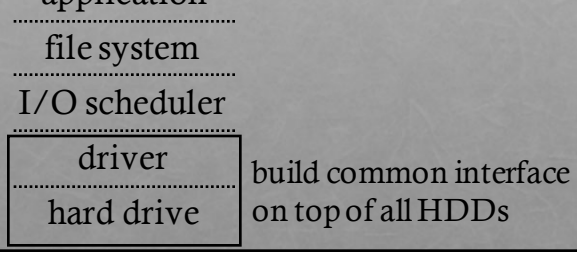
VARIETY IS A CHALLENGE

Problem: Many devices, each with own protocol

How to avoid writing different OS for each H/W combination?

Write **device driver** for each device

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



HARD DISKS

BASIC INTERFACE

Mechanical (slow) nature makes management "interesting"

Disk has a sector-addressable address space

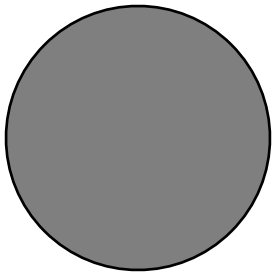
- Appears to OS as **array** of sectors
- Sectors typically 512 bytes or 4096 bytes.

Main operations

- reads + writes to sectors

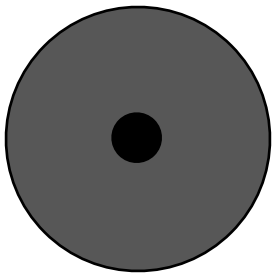
Disk Internals

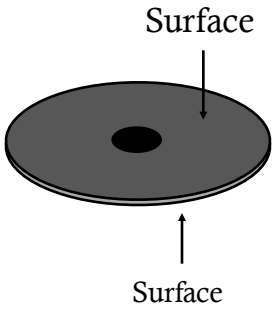
Platter

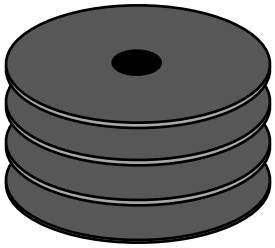


Platter is covered with a magnetic film

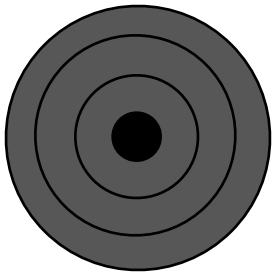
Spindle





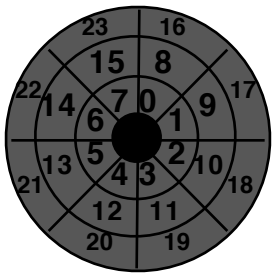


Many platters may be bound to spindle

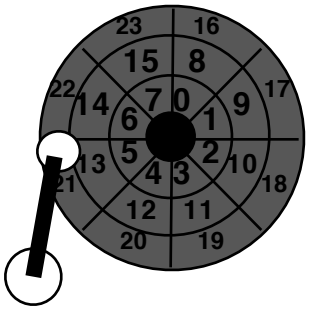


Each surface is divided into rings called tracks

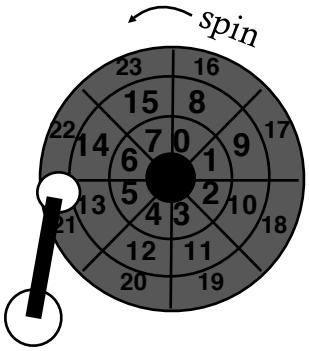
Stack of tracks (across platters) is called a cylinder



Tracks are divided into numbered sectors
(view as linear array)

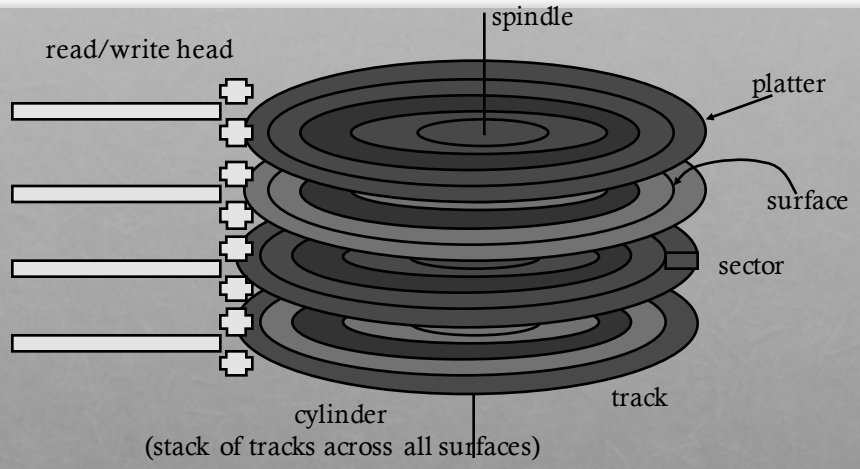


Heads on moving arm can read from each surface



Spindle/platters rapidly spin.

Disk Terminology

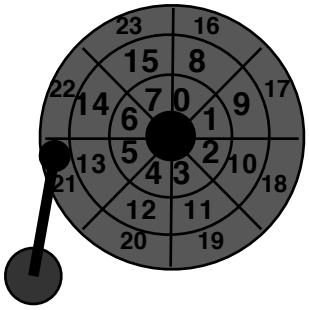


Hard Drive Demo

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

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

LET'S READ 12!



POSITIONING

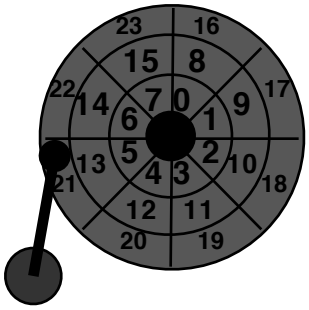
Drive servo system keeps head on track

- 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)

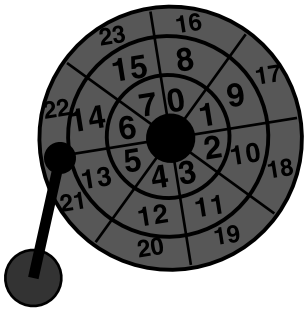
Use servo burst:

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

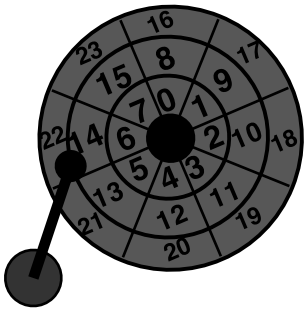
LET'S READ 12!



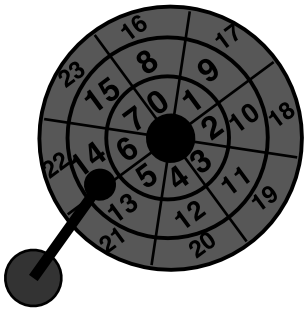
SEEK TO RIGHT TRACK.



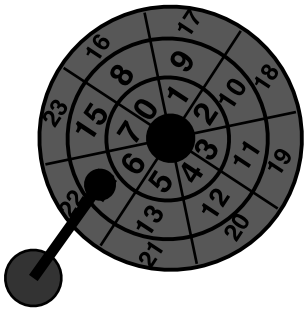
SEEK TO RIGHT TRACK.



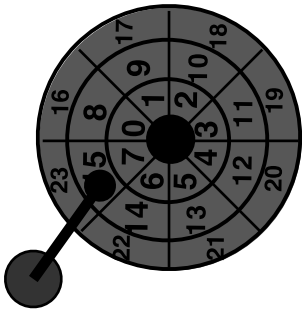
SEEK TO RIGHT TRACK.



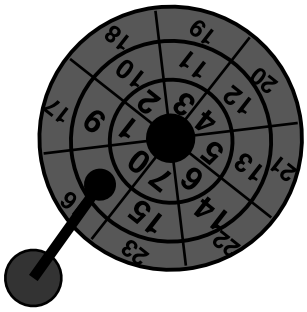
WAIT FOR ROTATION.



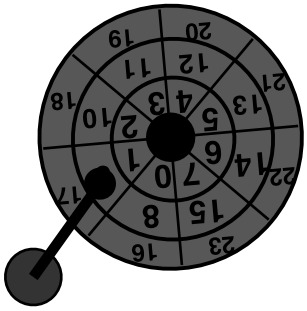
WAIT FOR ROTATION.



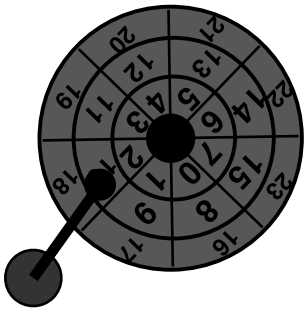
WAIT FOR ROTATION.



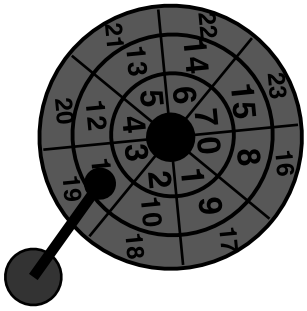
WAIT FOR ROTATION.



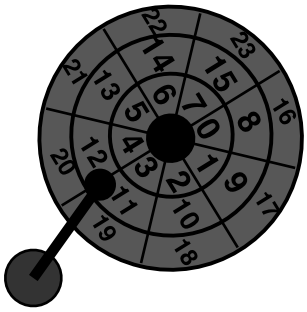
WAIT FOR ROTATION.



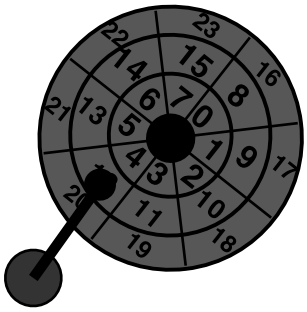
WAIT FOR ROTATION.



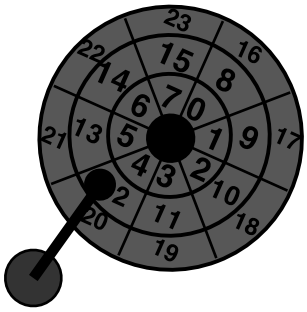
TRANSFER DATA.



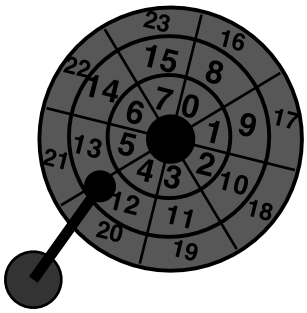
TRANSFER DATA.



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

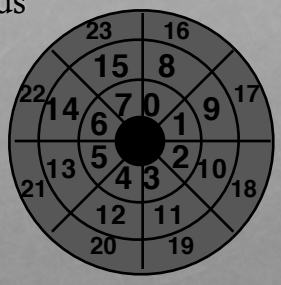
Entire seeks often takes several milliseconds

- 4 - 10 ms

Approximate average seek distance?

= 1/3 max seek distance

(derivation in text book)



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 distance?
½

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 (ms)
- rotations are slow (ms)
- transfers are fast (us)

What kind of workload is fastest for disks?

Sequential: access sectors in order (transfer dominated)

Random: access sectors arbitrarily (seek+rotation dominated)

Disk Spec: Seq vs Random Throughput

	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: Random Throughput

	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
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

Throughput for average random 16-KB read w/ Cheetah?

Time = Seek + rotation + transfer

Average seek? Seek = 4 ms

Average rotation in ms?

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

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
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

Throughput for average random 16-KB read w/ Cheetah?

Time = Seek + rotation + transfer

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

Random Throughput? (MB/s)

$$\text{throughput} = \frac{16\text{ KB}}{6.1\text{ms}} \times \frac{1\text{ MB}}{1024\text{ KB}} \times \frac{1000\text{ ms}}{1\text{ sec}} = 2.5\text{ MB/s}$$

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

Throughput for average random 16-KB read on Barracuda?

Time = seek + rotation + transfer

Avg seek = 9ms

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

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

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

Throughput for average random 16-KB read on Barracuda?

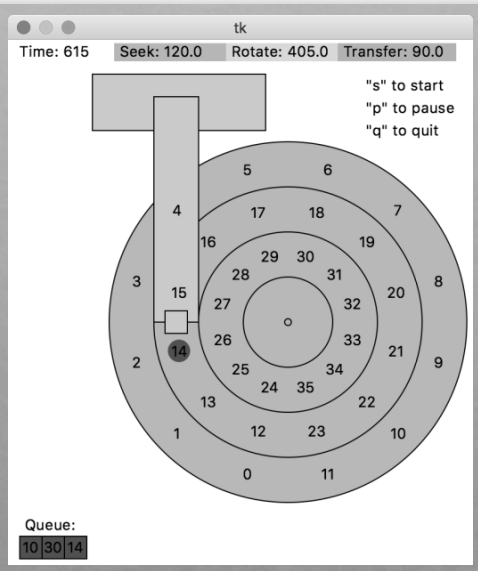
Barracuda time = 9ms + 4.1ms + 149us = 13.2ms

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

	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

Disk Simulator



./example-rand.csh
./example-seq.csh

Other Improvements

Track Skew

Zones

Cache

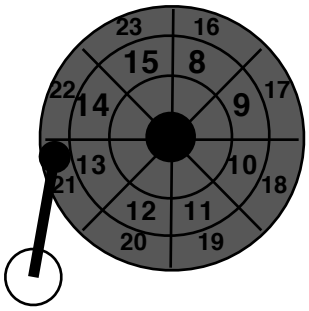
PROBLEM

What if sequential request spans multiple tracks?

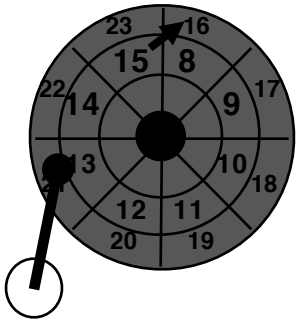
./example-skew.csh

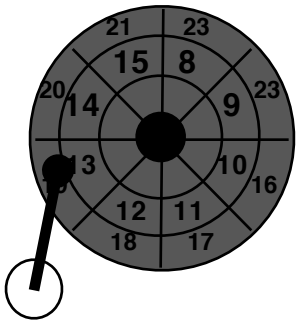
./example-skew-fixed.csh

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

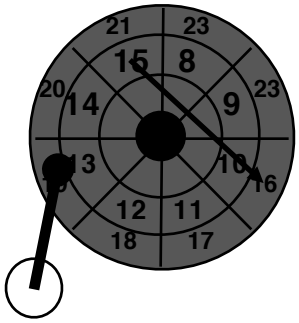


When reading 16 after 15, the head won't settle quick enough, so we need to do a rotation





enough time to settle now

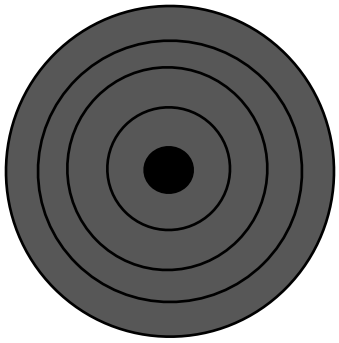


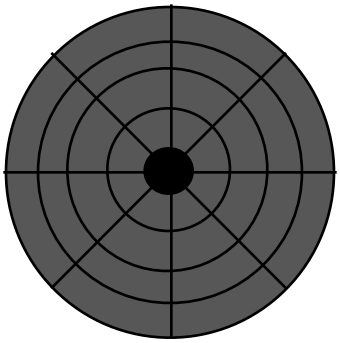
OTHER IMPROVEMENTS

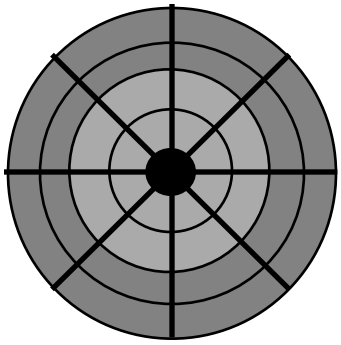
Track Skew

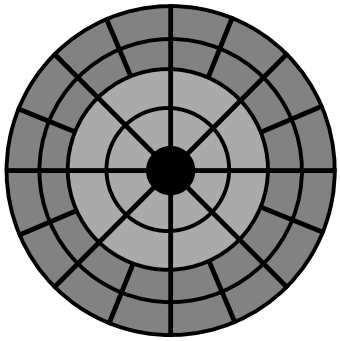
Zones

Cache









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

SIMULATOR

Pros and cons of ZBR?

./example-zones-outer.csh

./example-zones-inner.csh

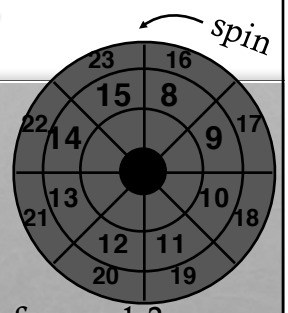
OTHER IMPROVEMENTS

Track Skew

Zones

Cache

DRIVE CACHE



Drives may cache both reads and writes
- OS caches data too (later lecture)
- Disks contain 2MB-16MB used as cache

What advantage does caching in **drive** have for reads?

Spatial locality -- Read-ahead: "Track buffer"
- Read contents of entire track into track buffer during rotational delay

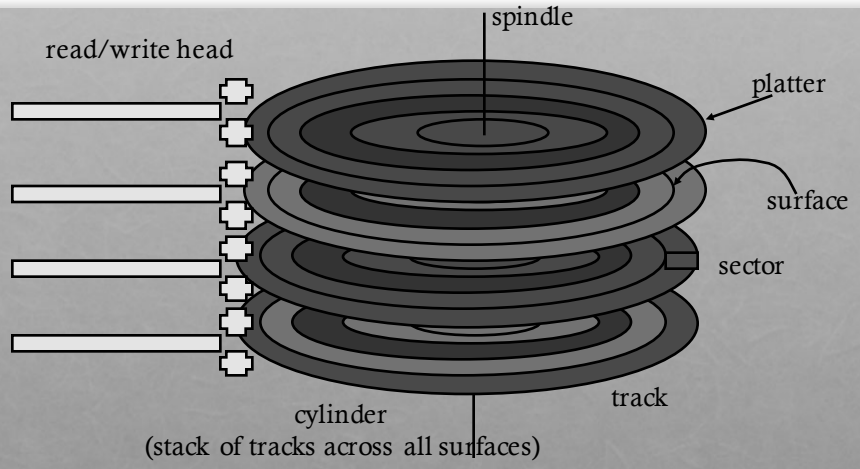
What advantage does caching in **drive** have for writes?

Write caching with volatile memory
- Immediate reporting: Claim written to disk when not
- Data could be lost on power failure

BUFFERING

Tagged command queueing
- Have multiple outstanding requests to the disk
- Disk can reorder (schedule) requests for better performance

REVIEW: DISK TERMINOLOGY



One r/w head activated at a time

I/O SCHEDULERS

I/O SCHEDULERS

Given 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

SIMULATOR

./example-sched-fifo.csh

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

~60ms

300001, 700001, 300002, 700002, 300003, 700003

300001, 300002, 300003, 700001, 700002, 700003

~20ms

SIMULATOR

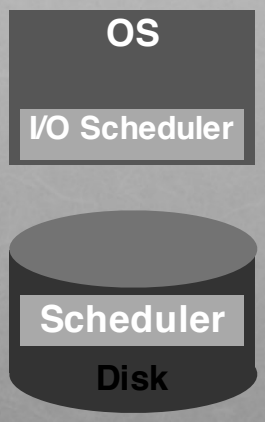
What would be a better algorithm than FIFO?

./example-sched-sstf.csh

./example-rotate-question.csh

./example-rotate-satf.csh

I/O SCHEDULERS



Where should the I/O scheduler go?

SPTF (SHORTEST POSITIONING TIME FIRST)

Strategy: always choose request that requires least positioning time (time for seeking and rotating)

How to implement in **disk**?

Greedy algorithm (just looks for best NEXT decision)

How to implement in **OS**?

Use Shortest Seek Time First (SSTF) instead

Approximate by scheduling by sector number

Disadvantages?

Easy for far away requests to **starve**

STARVATION

./example-starve.csh

Avoid Starvation: 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

Bounded Window

./example-starve-bsatf.csh

./example-bsatf-w1.csh

./example-bsatf-w3.csh

./example-bsatf-wall.csh

IS THIS BEST POSSIBLE?

./example-greedy-satf.csh

./example-greedy-optimal.csh

WHAT HAPPENS AT OS LEVEL?

Assume 2 processes each call 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 do work if work exists

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

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

CFQ (Linux Default)

Implemented within OS

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!

- -Quicksort is a terrible algorithm on disk

Spend time to schedule on slow, stateful devices