We acknowledge and pay our respects to the Kaurna people,
the traditional custodians whose ancestral lands we gather on.

We acknowledge the deep feelings of attachment and relationship of the Kaurna people to country and we respect and value their past, present and ongoing connection to the land and cultural beliefs.

Operating Systems

COMP SCI 3004 / COMP SCI 7064

Week 8

I/O Devices, Hard Disk Drives

make history.

THE UNIVERSITY of ADELAIDE

Quiz 1

Which one of the following cases requires synchronisation?

	Answer
✗	Two threads modifying different data structures
✗	Two threads traversing the same data structure
✓	Two threads modifying the same data structure
✗	One thread modifying two different data structures
✗	Two threads traversing different data structures

THE UNIVERSITY of ADELAIDE

Quiz 2

Under what circumstances is busy waiting acceptable?

	Answer
✗	When each thread always holds multiple locks at a time
✓	When locks are only ever held for short periods of time
✗	When there is a large number of threads
✗	When each thread holds only a single lock at a time
✗	When there is a small number of threads

Quiz 3

Under which of the following circumstances can interrupt disabling be used as a mechanism for ensuring mutual exclusion?

	Answer
✗	The instruction set includes an atomic "test and set" instruction
✗	Interrupt handlers always run with a lower priority than the kernel
✓	The system contains only a single processor
✗	All interrupts are handled by a single device driver

Implementing Locks: W/ Interrupts

Disable interrupts for critical sections

- Prevent dispatcher from running another thread
- Code between interrupts executes atomically

```
void lock(lockT *l) {
    disableInterrupts();
}
```

```
void unlock(lockT *l) {
    enableInterrupts();
}
```

Disadvantages

- Only works on uniprocessors
- Process can keep control of CPU for arbitrary length
- Cannot perform other necessary work

Quiz 4

We have a variable *x*, shared amongst threads A, B, C and D. We want A to initialise *x*, then A, B and C may update its value in any order, and thread D should print its final value. Select the correct implementation to achieve this goal.

	Answer
✓	2 semaphores: a mutex between all threads, unlocked by A, and another semaphore to force D to wait for the last update
✗	3 semaphores: one to wait for A, a mutex between B and C and another semaphore to force D to wait for the last update
✗	1 semaphore: we run thread A first, then run both threads B and C using a mutex , and D will run last
✗	3 semaphores: two sync-semaphores for B and C and a counting semaphore for D.

Semaphores Like Integers Except…

Semaphores are like integers, except:

No negative values

Only operations allowed are *wait* and *post* – can't read or write value, except initially

Join with Semaphores:

```
sem_t s;
sem_init(&s, ???);
```

Initialize to 0 (so sem_wait() must wait…)

```
void thread_join() {
        sem_wait(&s);
}
```

```
void thread_exit() {
        sem_post(&s)
}
```

Quiz 5

Most deadlocks in operating systems develop because of the normal contention for:

	Answer
✓	dedicated resources
✗	device drivers
✗	processors
✗	main memory

Quiz 6

If a cycle exists in the resource allocation graph

	Answer
✓	the system could go in a deadlock
✗	the system is already deadlocked
✗	the request will be denied by the bankers algorithm for deadlock avoidance
✗	the system will be in a safe state

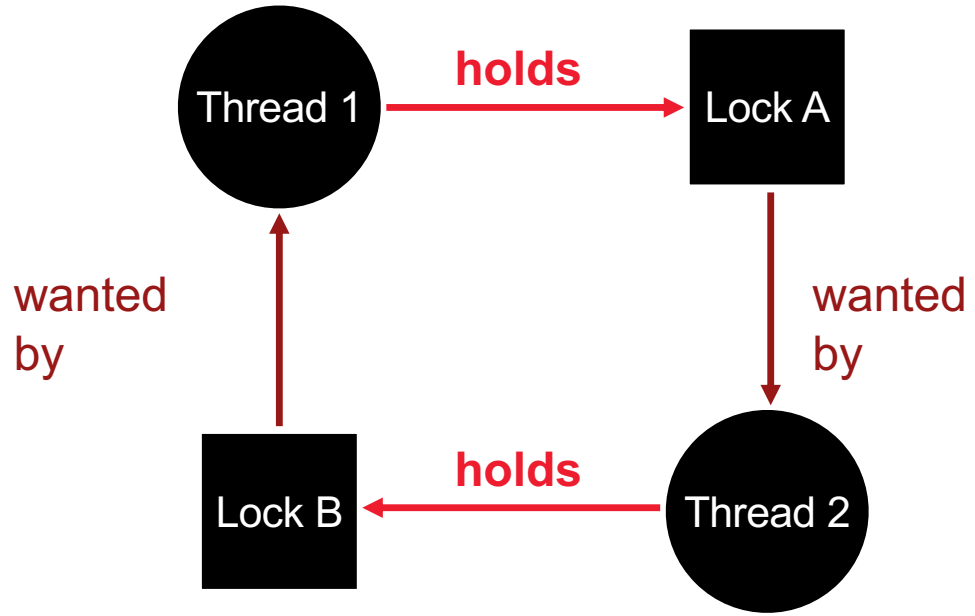
Example: Circular Dependency

```
Thread 1:

lock(&A);
lock(&B);
```

```
Thread 2:

lock(&B);
lock(&A);
```



Quiz 7

A deadlock prevention method that imposes a total ordering of all resource types, and requires that each process requests resources in an increasing order of enumeration. Which condition of deadlock is eliminated?

	Answer
×	No Preemption
×	Hold and Wait
×	Mutual exclusion
✓	Circular Wait

Quiz 8

The Banker's algorithm is used

	Answer
✗	to detect if the system is deadlocked
✓	to eliminate the possibility of a deadlock
✗	to recover from a deadlocked state
✗	to reduce the probability of deadlock

THE UNIVERSITY of ADELAIDE

Banker's Algorithm for Avoiding Deadlock

Toward right idea:

State maximum (max) resource needs in advance

Allow particular thread to proceed if:

(available resources - #requested) ≥ max remaining that might be needed by any thread

Banker's algorithm:

Allocate resources dynamically

- Evaluate each request and grant if some ordering of threads is still deadlock free afterward
- Technique: pretend each request is granted, then run deadlock detection algorithm, substituting:
  ([Max_{node}]-[$Alloc_{node}$] <= [Avail]) for ([$Request_{node}$] <= [Avail])
  Grant request if result is deadlock free (conservative!)

Quiz 9

If a semaphore is to be used for mutual exclusion (implement a lock), what should its initial value be? (Assume no processes are initially in the critical section).

	Answer
✓	1
×	The same as the number as concurrent threads are started.
×	0
×	-1

Quiz 10

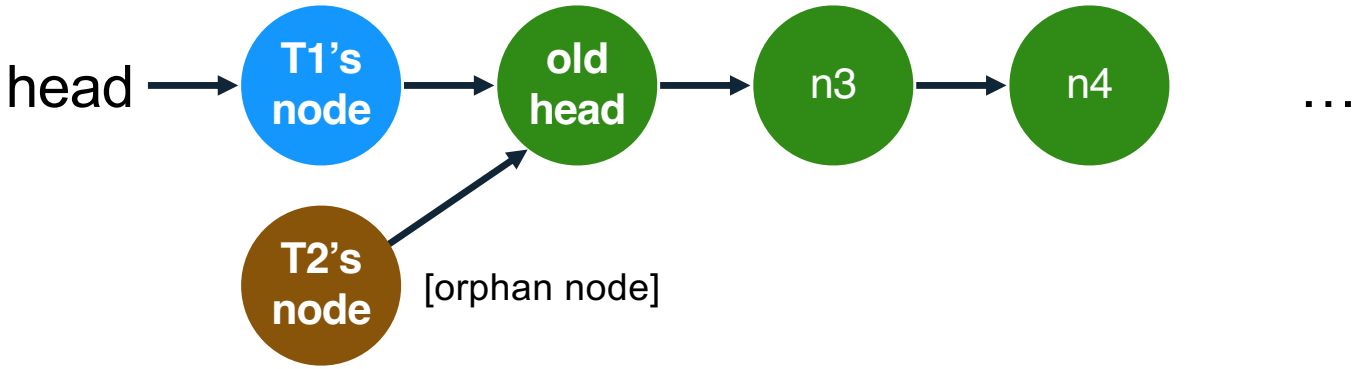
Consider the following code, which adds a node to the start of a linked list:

```
1     Node n = new Node();
2     n.next = list;
3     list = n;
```

Which of these lines constitute the critical section?

	Answer
✓	Lines 2 and 3
×	Lines 1 and 2
×	Line 2 only
×	Lines 1, 2, and 3

Resulting Linked List




```
1     Node n = new Node();
2     n.next = list;
3     list = n;
```

Operating Systems

COMP SCI 3004 / COMP SCI 7064

Week 8

I/O Devices, Hard Disk Drives

make history.

THE UNIVERSITY of ADELAIDE

Motivation

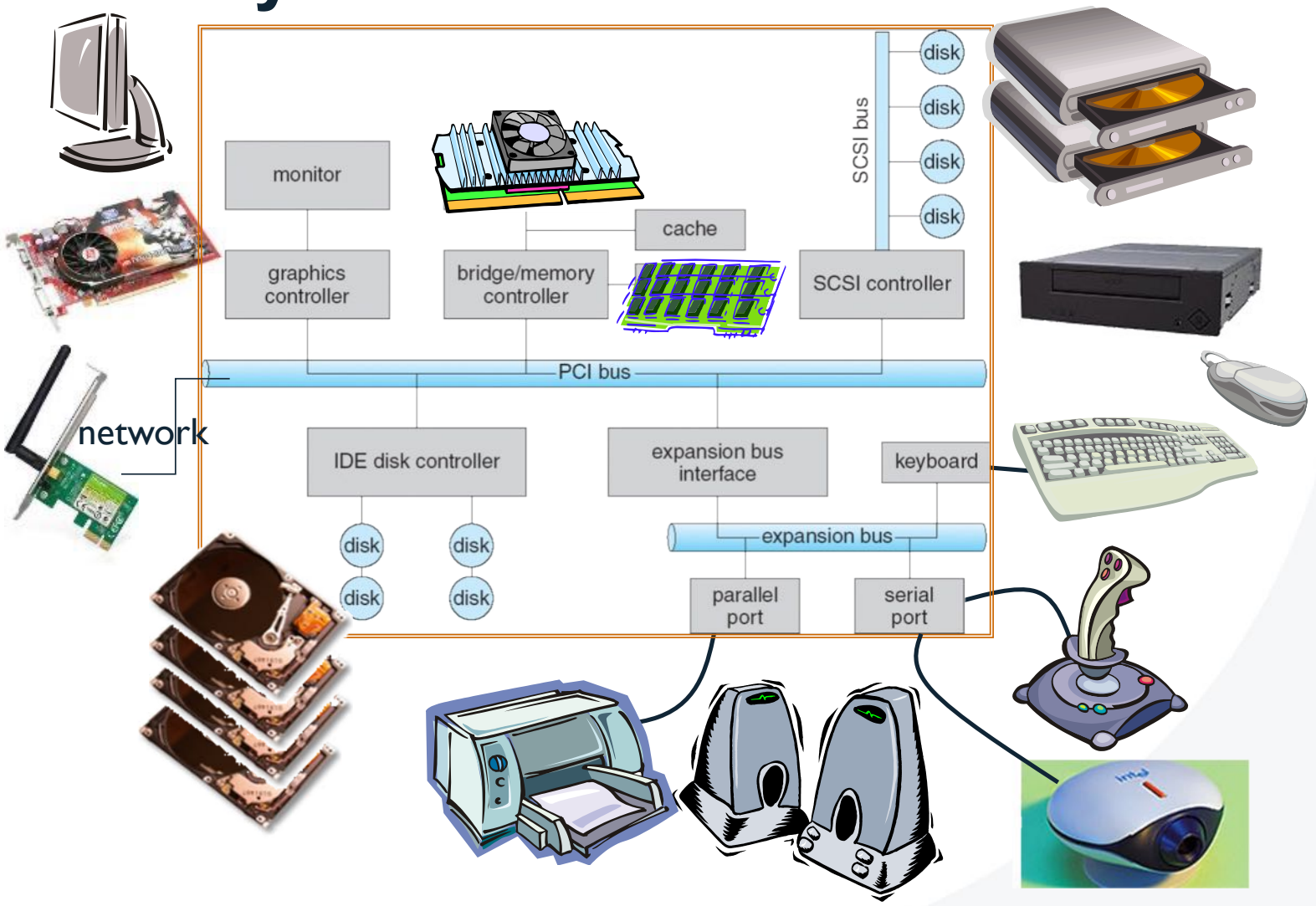
What good is a computer without any I/O devices?

- keyboard, display, disks

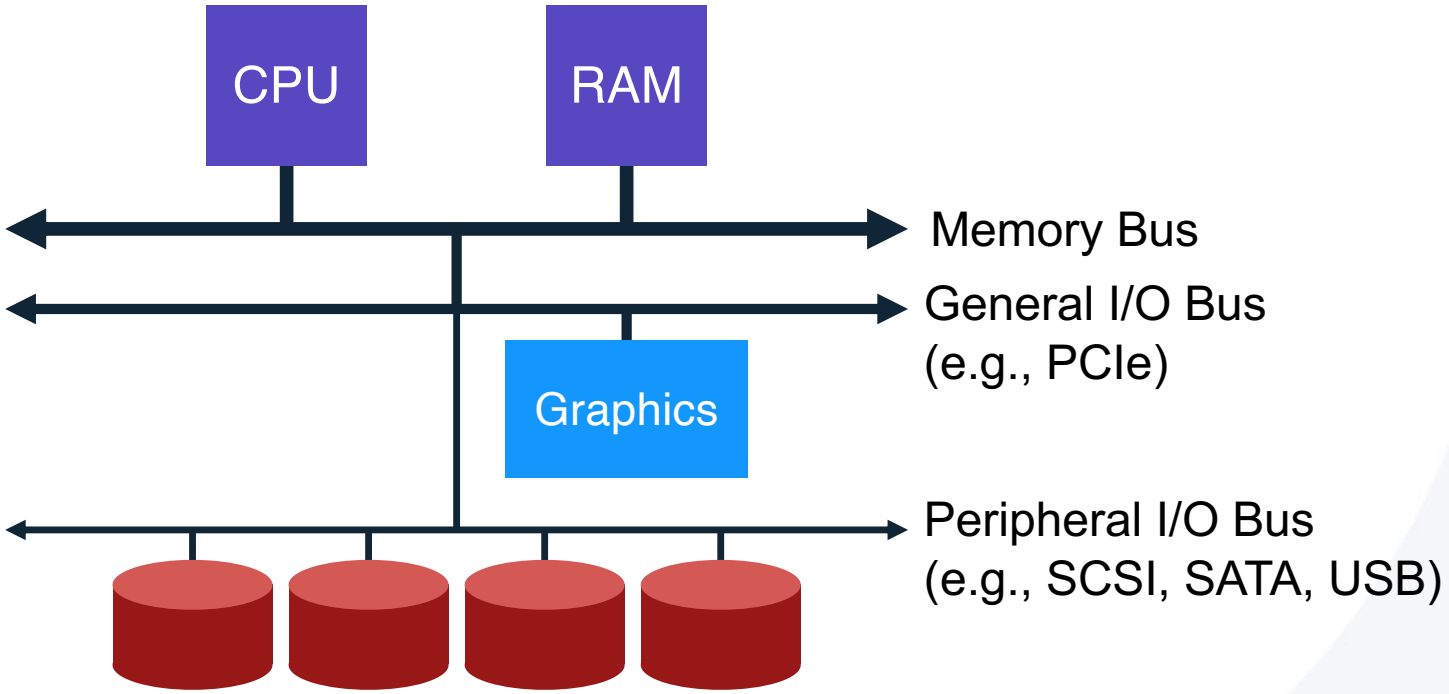
We want:

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

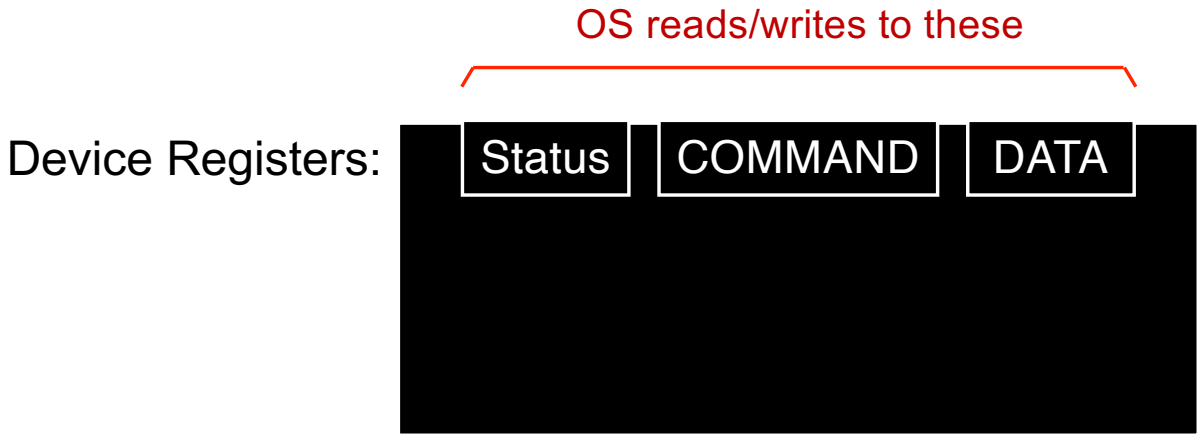
Modern I/O Systems



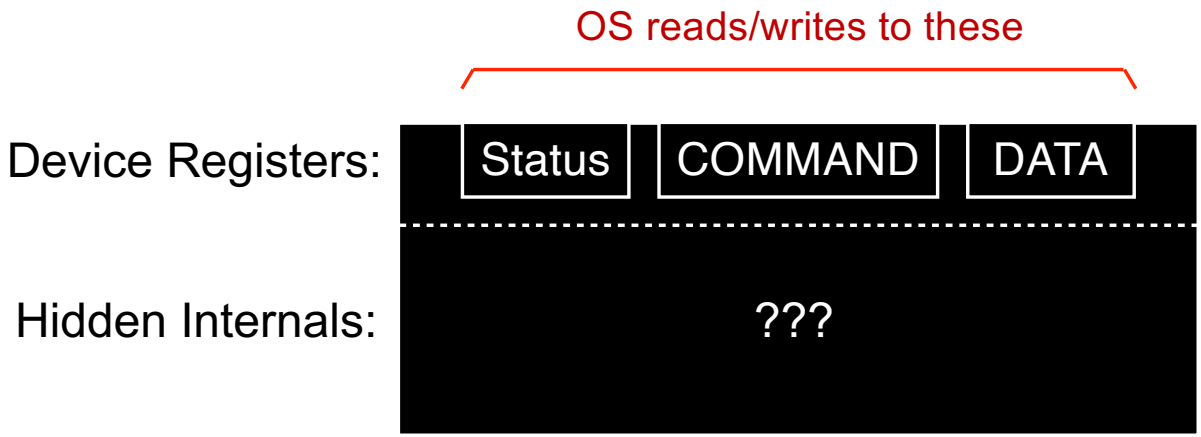
Hardware support for I/O



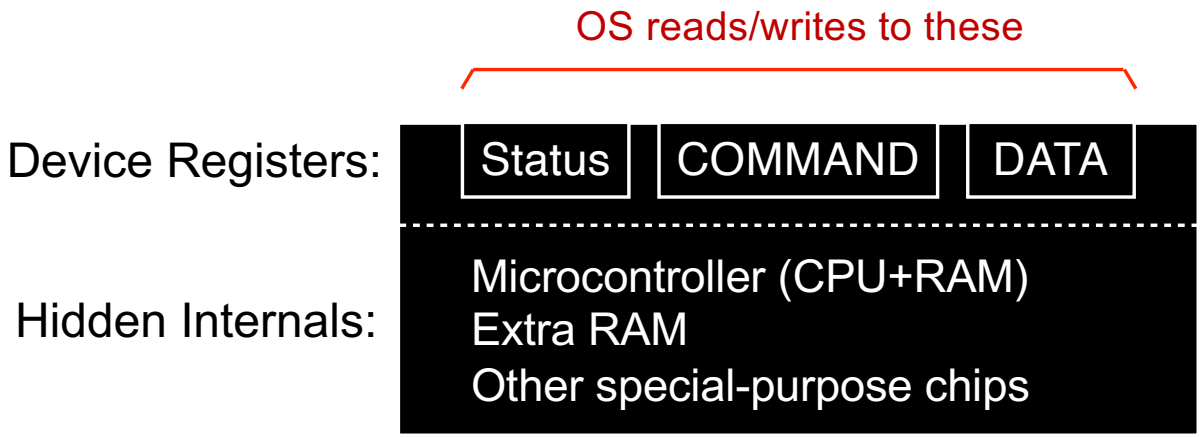
Canonical Device



Canonical Device

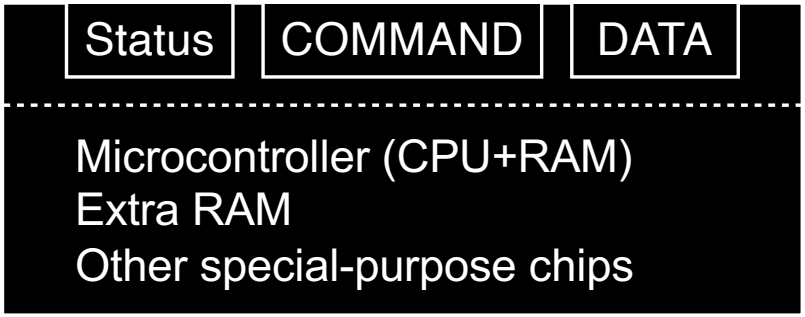


Canonical Device



Example Write Protocol

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



CPU:

Disk:

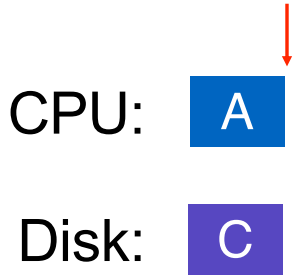
```
while (STATUS == BUSY)                  // 1
  ;
Write data to DATA register             // 2
Write command to COMMAND register       // 3
while (STATUS == BUSY)                  // 4
  ;
```

CPU: A

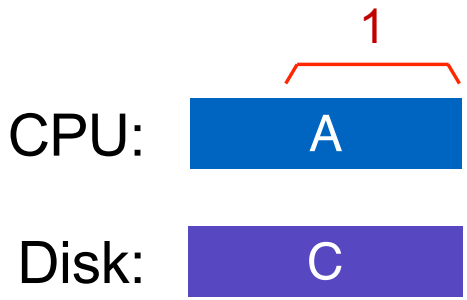
Disk: C

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

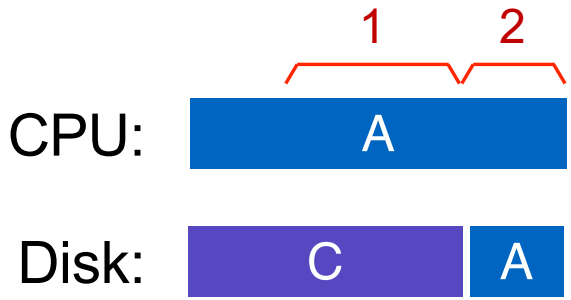
A wants to do I/O



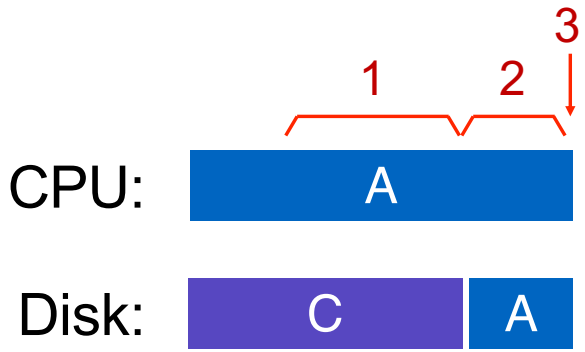

```
while (STATUS == BUSY)                  // 1
  ;
Write data to DATA register             // 2
Write command to COMMAND register       // 3
while (STATUS == BUSY)                  // 4
  ;
```



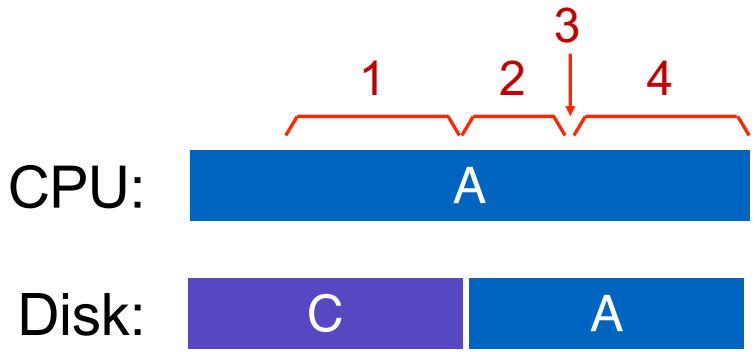

```
while (STATUS == BUSY)                  // 1
  ;
Write data to DATA register             // 2
Write command to COMMAND register       // 3
while (STATUS == BUSY)                  // 4
  ;
```



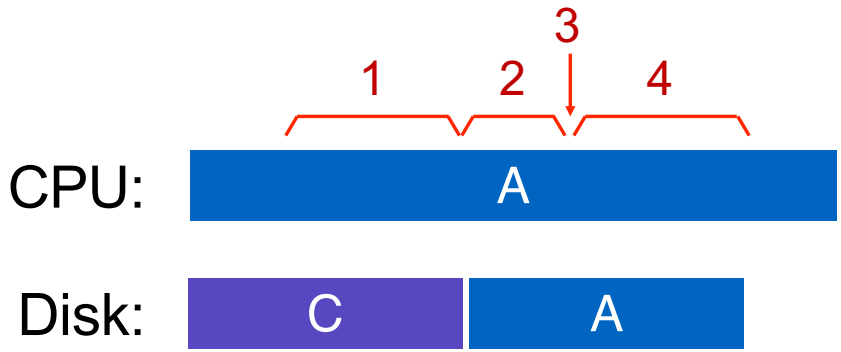

```
while (STATUS == BUSY)                  // 1
  ;
Write data to DATA register             // 2
Write command to COMMAND register       // 3
while (STATUS == BUSY)                  // 4
  ;
```



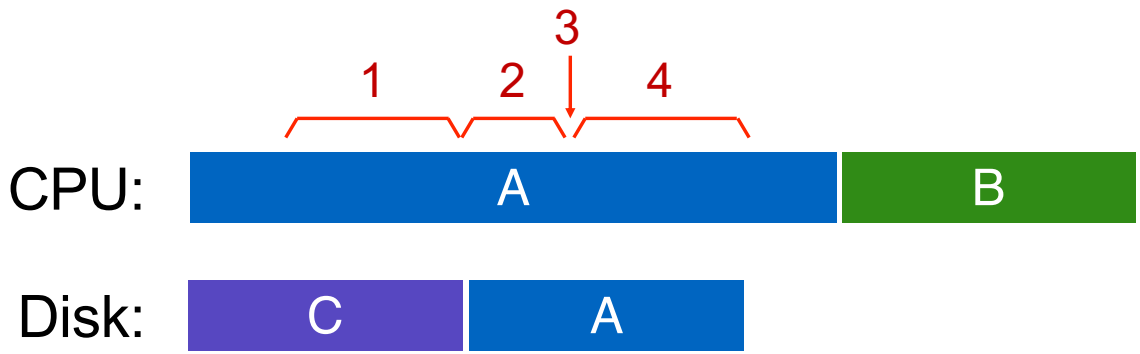

```
while (STATUS == BUSY)                  // 1
  ;
Write data to DATA register             // 2
Write command to COMMAND register       // 3
while (STATUS == BUSY)                  // 4
  ;
```




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

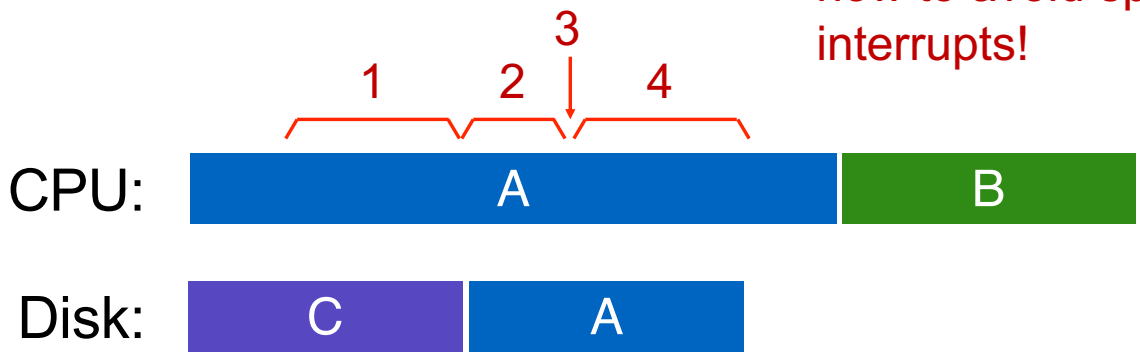



```
while (STATUS == BUSY)                  // 1
  ;
Write data to DATA register             // 2
Write command to COMMAND register       // 3
while (STATUS == BUSY)                  // 4
  ;
```



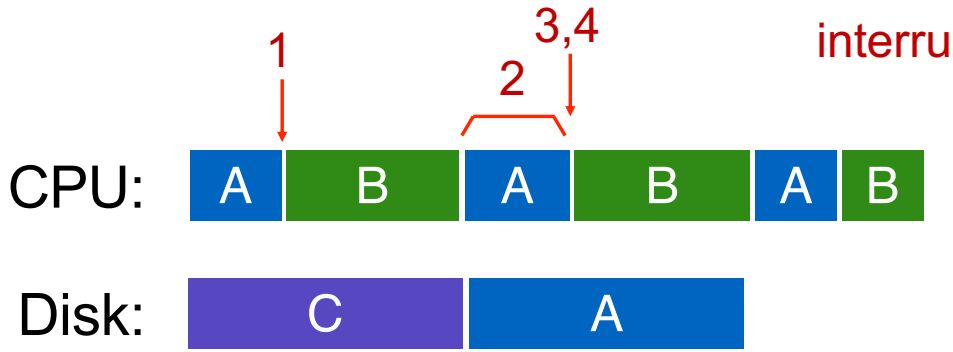

```
while (STATUS == BUSY)                  // 1
  ;
Write data to DATA register             // 2
Write command to COMMAND register       // 3
while (STATUS == BUSY)                  // 4
  ;
```

how to avoid spinning?
interrupts!




```
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

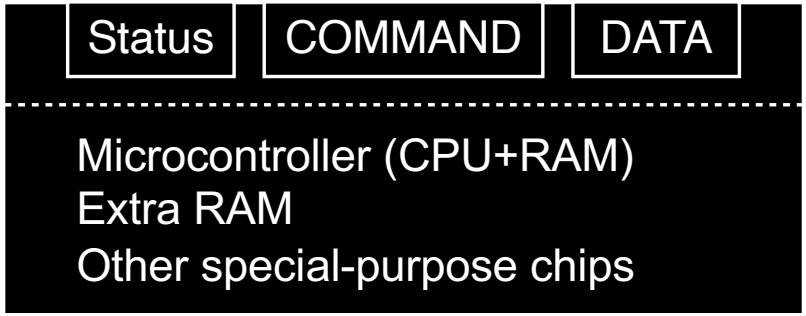
Can lead to livelock (always handling interrupts)

Better to ignore interrupts while make some progress handling them

Other improvement

Interrupt coalescing (batch together several interrupts)

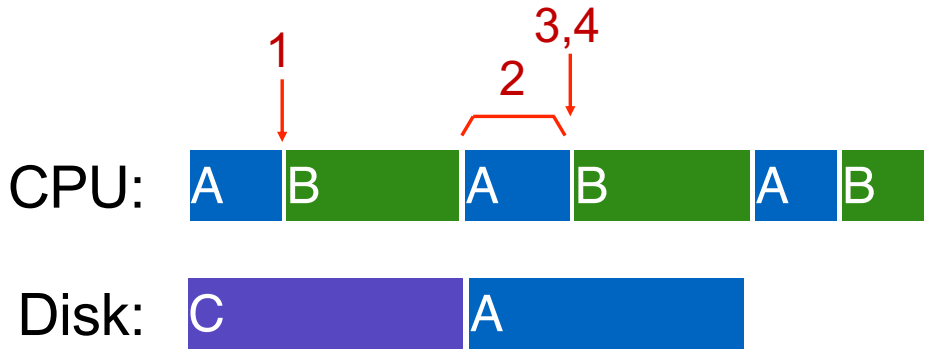
Protocol Variants



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

Data: PIO *vs.* DMA

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




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

Transferring Data To/From Controller

Programmed I/O:

Each byte transferred via processor in/out or load/store

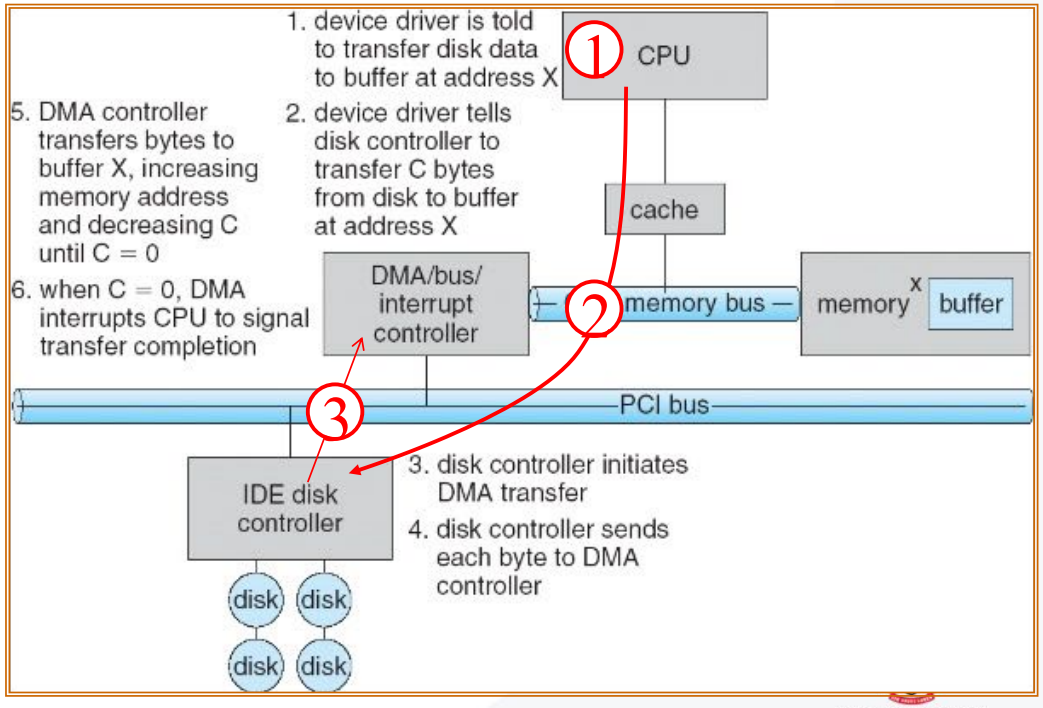
Pro: Simple hardware, easy to program

Con: Consumes processor cycles proportional to data size

Direct Memory Access:

Give controller access to memory bus

Ask it to transfer data blocks to/from memory directly



Transferring Data To/From Controller

Programmed I/O:

Each byte transferred via processor in/out or load/store

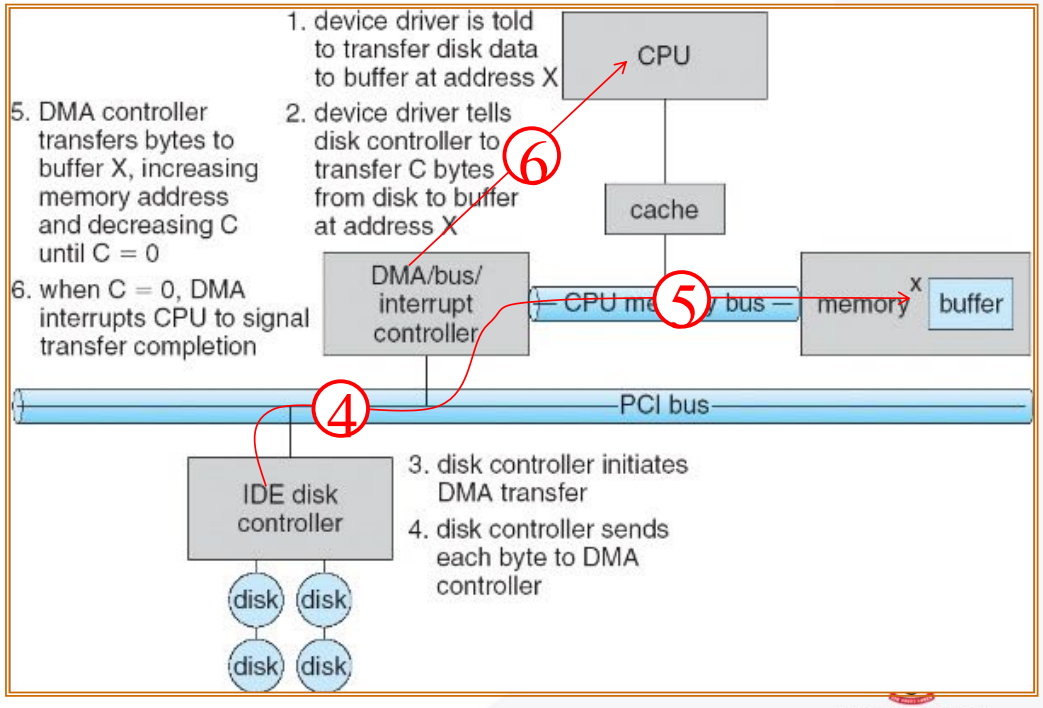
Pro: Simple hardware, easy to program

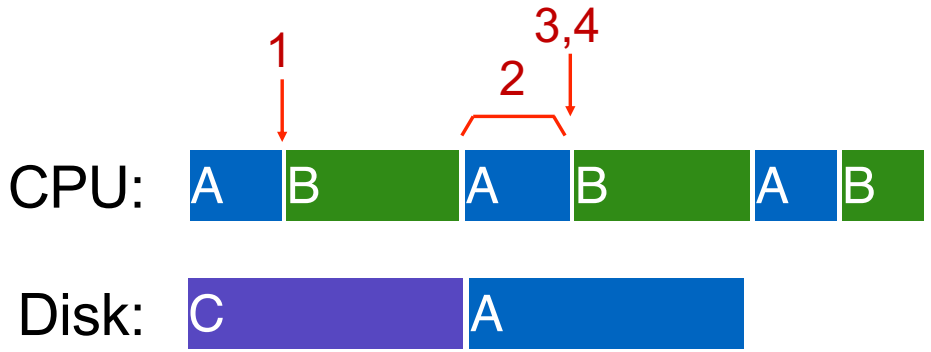
Con: Consumes processor cycles proportional to data size

Direct Memory Access:

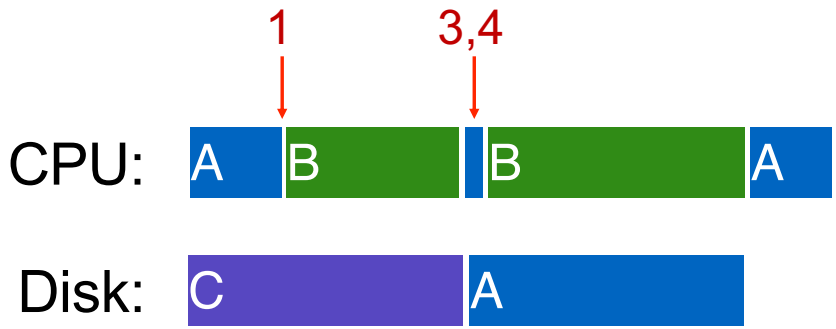
Give controller access to memory bus

Ask it to transfer data blocks to/from memory directly



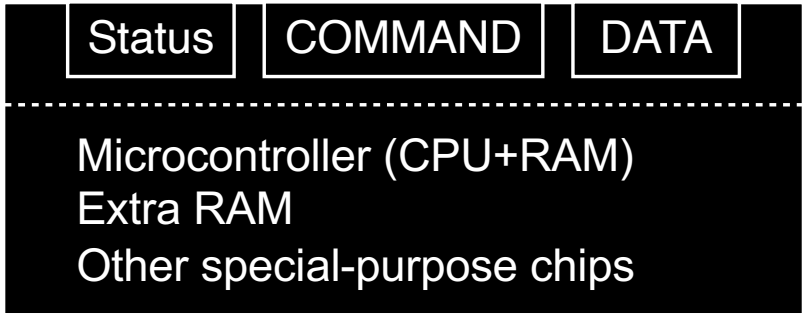



```
while (STATUS == BUSY)                  // 1
      wait for interrupt;
Write data to DATA register             // 2
Write command to COMMAND register       // 3
while (STATUS == BUSY)                  // 4
      wait for interrupt;
```




```
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

Special Instructions vs. Mem-Mapped I/O

Special instructions

each device has a port

in/out instructions (x86) communicate with device

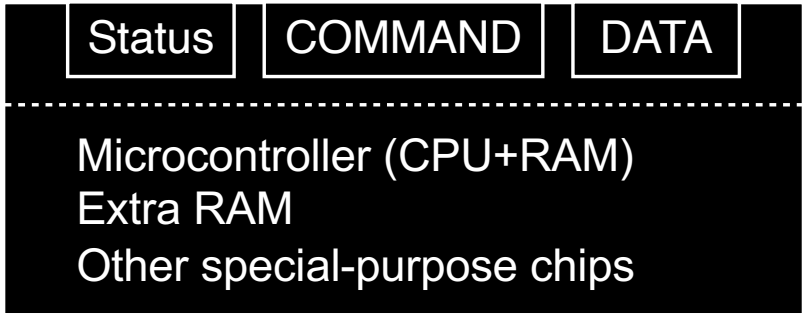
Memory-Mapped I/O

H/W maps registers into address space

loads/stores sent to device

Does not 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

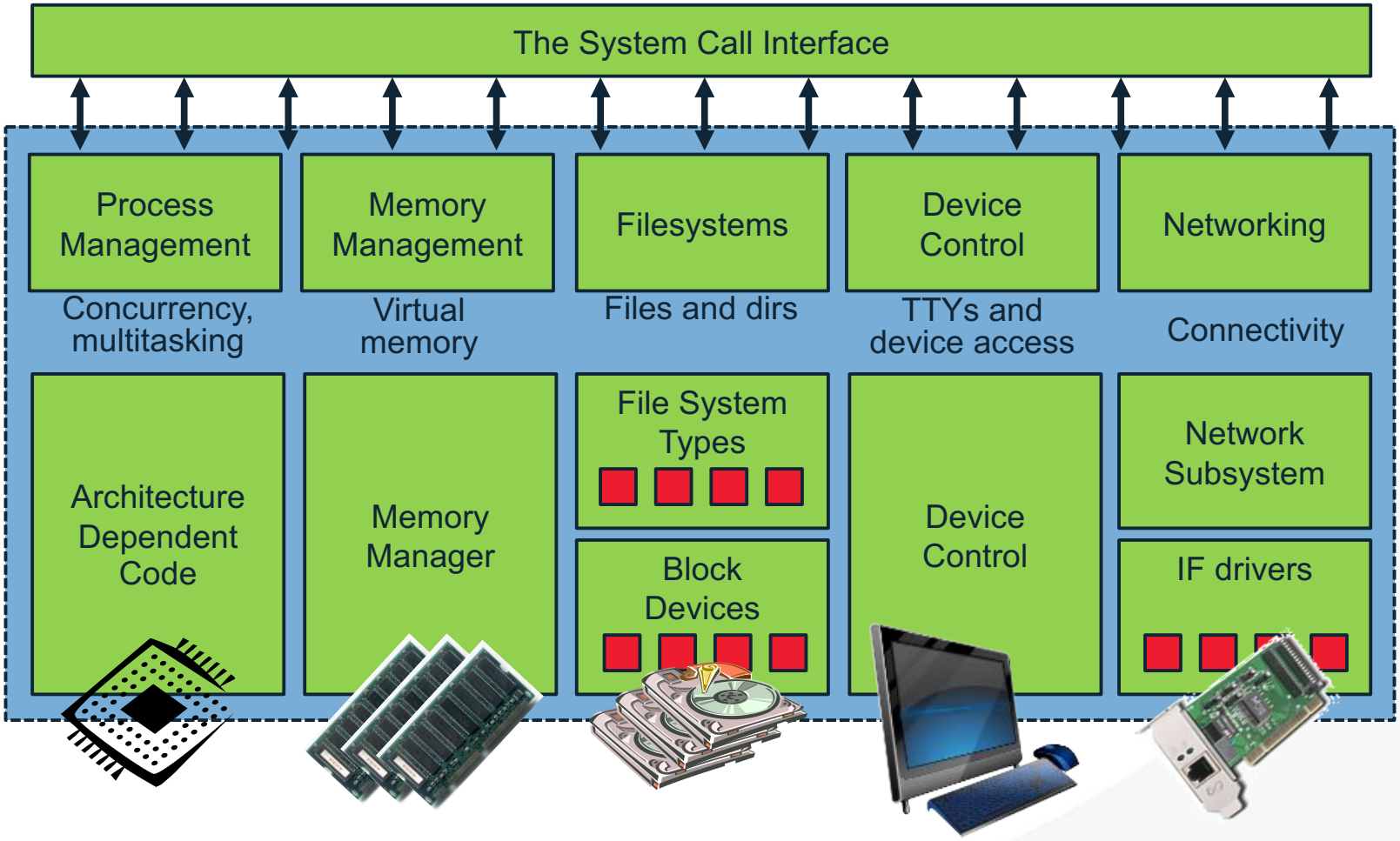
each has its own protocol

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

Write device driver for each device

Drivers are 70% of Linux source code

Kernel Device Structure



Device Drivers

Device Driver: Device-specific code in the kernel that interacts directly with the device hardware

Supports a standard, internal interface

Same kernel I/O system can interact easily with different device drivers

Special device-specific configuration supported with the `ioctl()` system call

Device Drivers typically divided into two pieces:

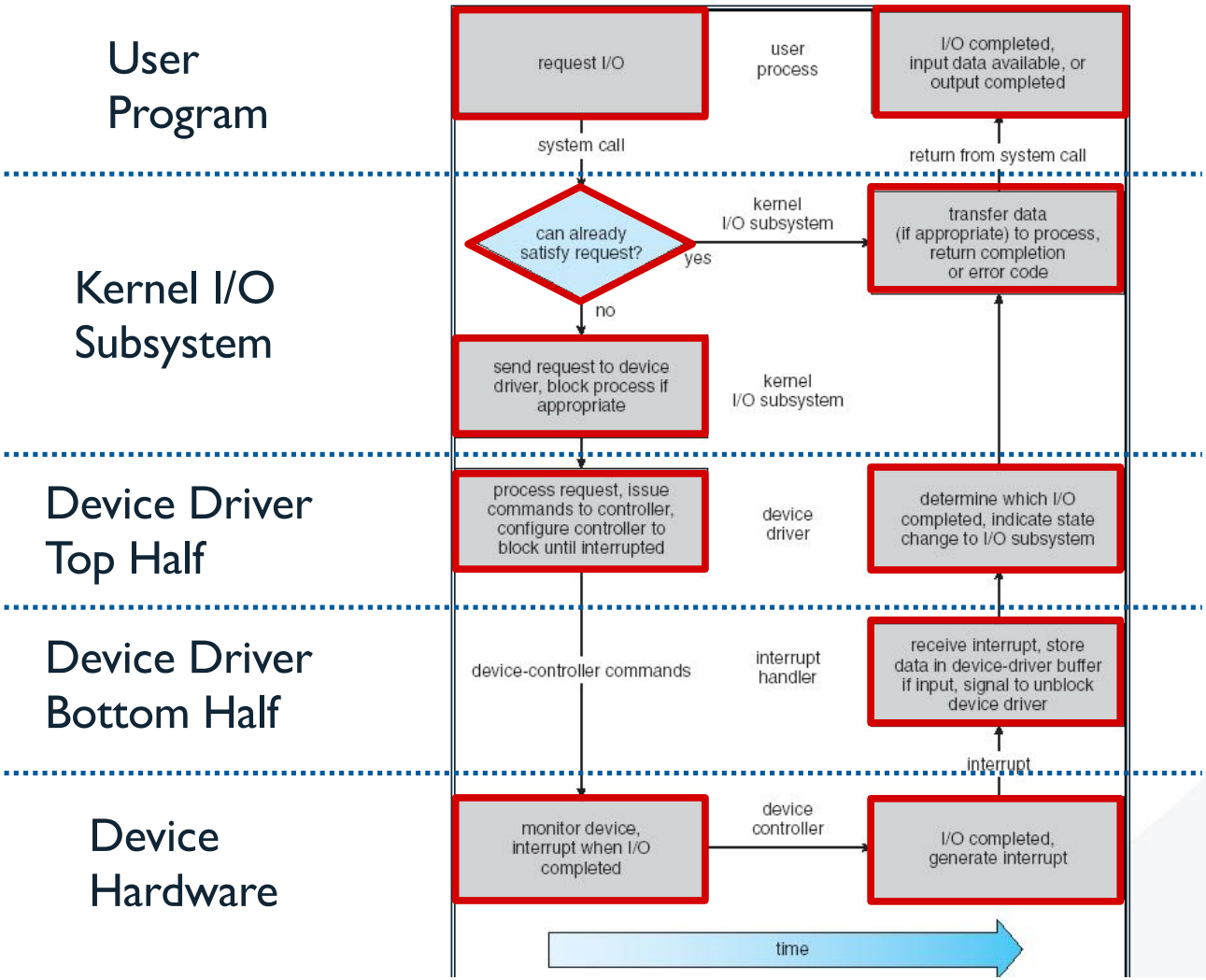
Top half: accessed in call path from system calls

- implements a set of standard, cross-device calls like `open()`, `close()`, `read()`, `write()`, `ioctl()`
- This is the kernel's interface to the device driver
- Top half will *start* I/O to device, may put thread to sleep until finished

Bottom half: run as interrupt routine

- Gets input or transfers next block of output
- May wake sleeping threads if I/O now complete

Life Cycle of An I/O Request



The Goal of the I/O Subsystem

Provide Uniform Interfaces, Despite Wide Range of Different Devices

This code works on many different devices:

```
FILE fd = fopen("/dev/something", "rw");
for (int i = 0; i < 10; i++) {
  fprintf(fd, "Count %d\n", i);
}
close(fd);
```

Why? Because code that controls devices (“device driver”) implements standard interface.

Want Standard Interfaces to Devices

Block Devices: *e.g.* disk drives, tape drives, DVD-ROM

Access blocks of data

Commands include `open()`, `read()`, `write()`, `seek()`

Raw I/O or file-system access

Memory-mapped file access possible

Character Devices: *e.g.* keyboards, mice, serial ports, some USB devices

Single characters at a time

Commands include `get()`, `put()`

Libraries layered on top allow line editing

Network Devices: *e.g.* Ethernet, Wireless, Bluetooth

Different enough from block/character to have own interface

Unix and Windows include socket interface

- Separates network protocol from network operation
- Includes `select()` functionality

Usage: pipes, FIFOs, streams, queues, mailboxes

How Does User Deal with Timing?

Blocking Interface: "Wait"

When request data (e.g. `read()` system call), put process to sleep until data is ready

When write data (e.g. `write()` system call), put process to sleep until device is ready for data

Non-blocking Interface: "Don't Wait"

Returns quickly from read or write request with count of bytes successfully transferred

Read may return nothing, write may write nothing

Asynchronous Interface: "Tell Me Later"

When request data, take pointer to user's buffer, return immediately; later kernel fills buffer and notifies user

When send data, take pointer to user's buffer, return immediately; later kernel takes data and notifies user

I/O Devices Types:

Many different speeds (0.1 bytes/sec to GBytes/sec)

Different Access Patterns:

- Block Devices, Character Devices, Network Devices

Different Access Timing:

- Blocking, Non-blocking, Asynchronous

I/O Controllers: Hardware that controls actual device

Processor Accesses through I/O instructions, load/store to special physical memory

Notification mechanisms

Interrupts

Polling: Report results through status register that processor looks at periodically

Device drivers interface to I/O devices

Provide clean Read/Write interface to OS above

Manipulate devices through PIO, DMA & interrupt handling

Three types: block, character, and network

Hard Disks

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 “interesting”

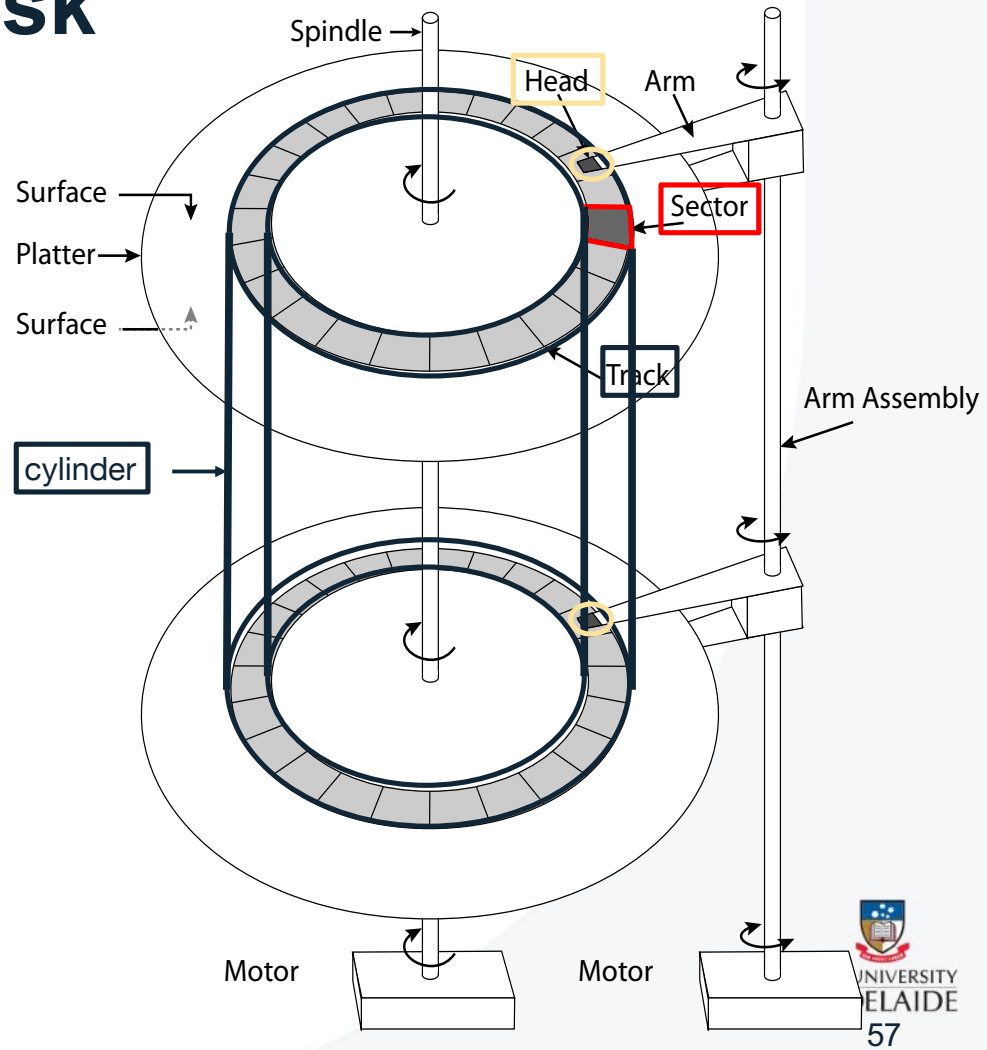
The Amazing Magnetic Disk

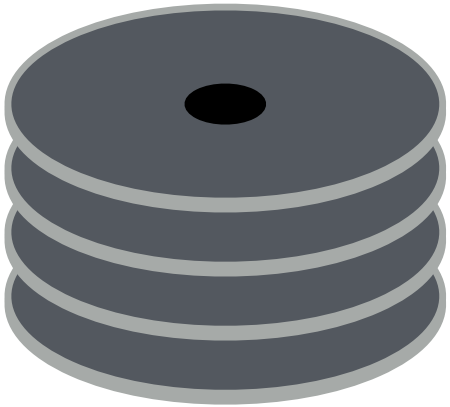
Unit of Transfer: Sector (512B or 4096B)

Ring of sectors form a **track**

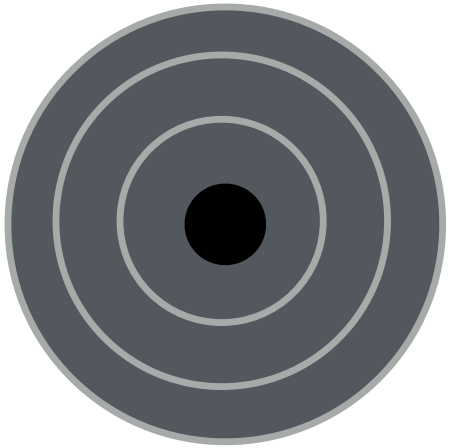
Stack of tracks form a **cylinder**

Heads position on **cylinders**

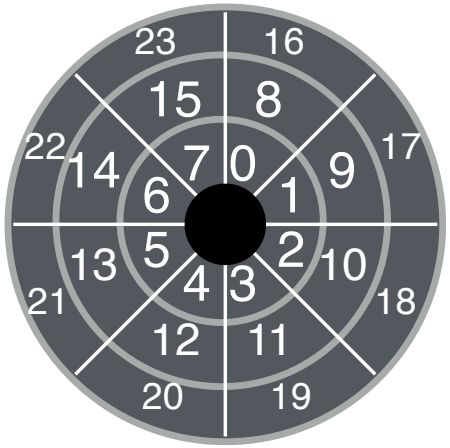




Many platters may be bound to the spindle.



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.

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:

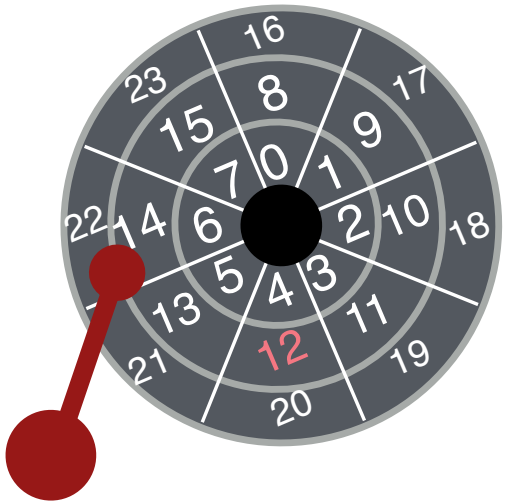
Record placement information every few (3-5) sectors

When head cross servo burst, figure out location and adjust as needed

Let’s Read 12!

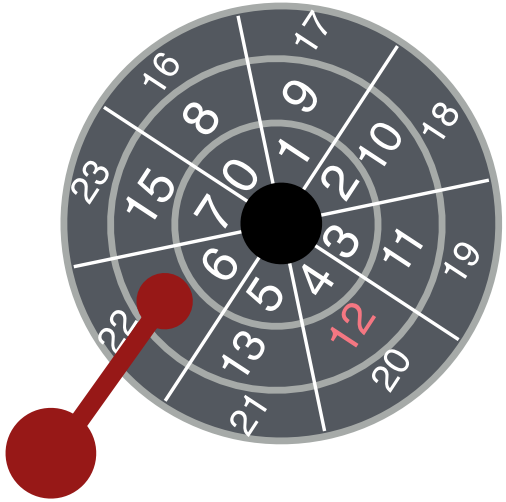
Seek to right track.

Seek to right track.

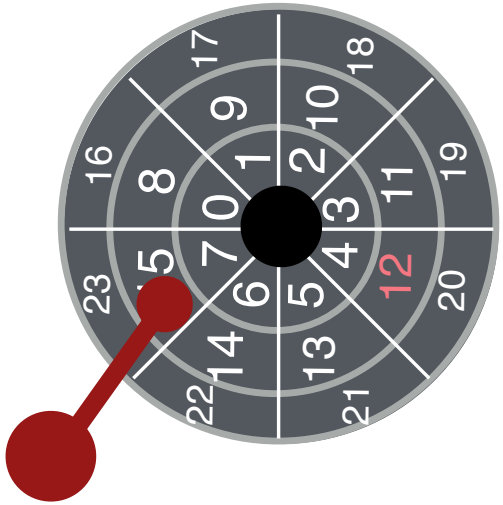


Seek to right track.

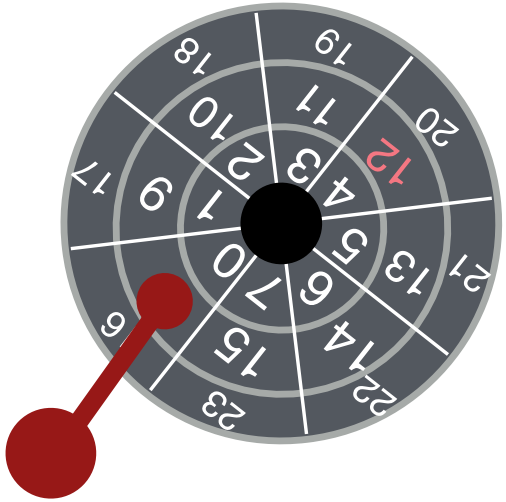
Wait for rotation.



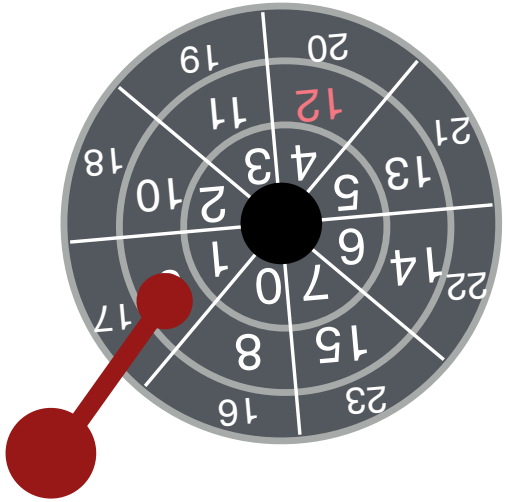
Wait for rotation.



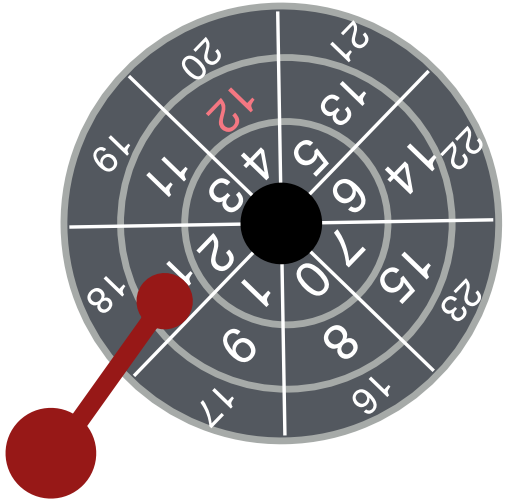
Wait for rotation.



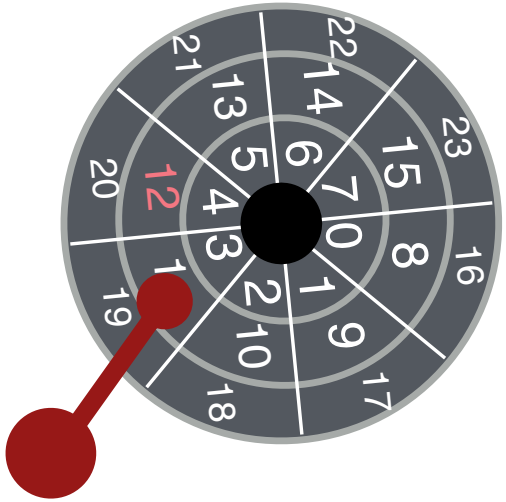
Wait for rotation.



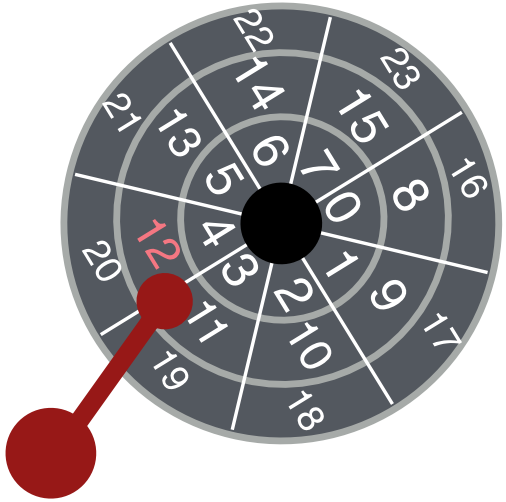
Wait for rotation.



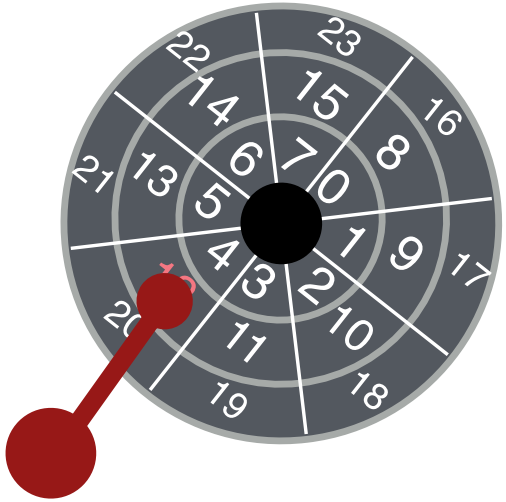
Wait for rotation.



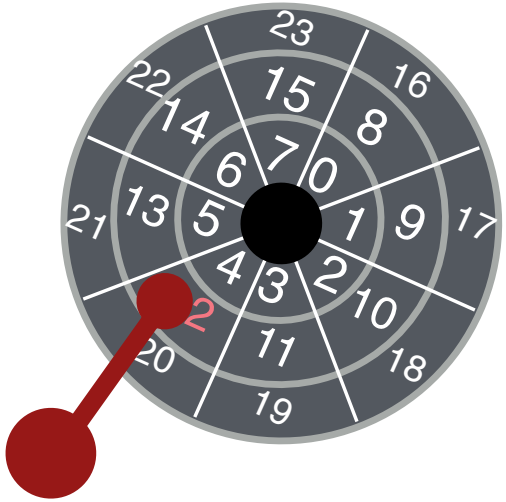
Transfer data.



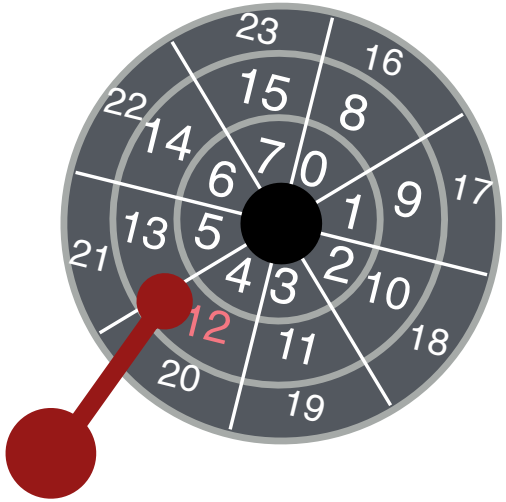
Transfer data.



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 ms

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?

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

	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 + 125μs = 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 + 125μs = 6.1ms

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

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

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?

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

	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?

Barracuda time = 9ms + 4.1ms + 149μs = 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 + 149μs = 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

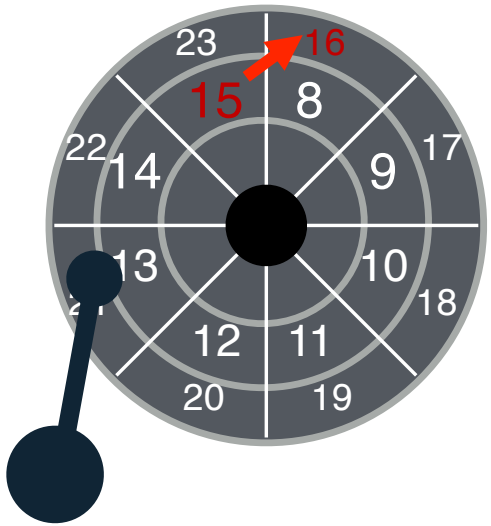
Track Skew

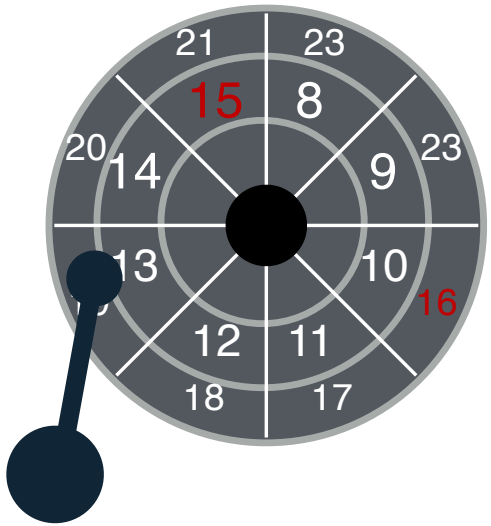
Zones

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

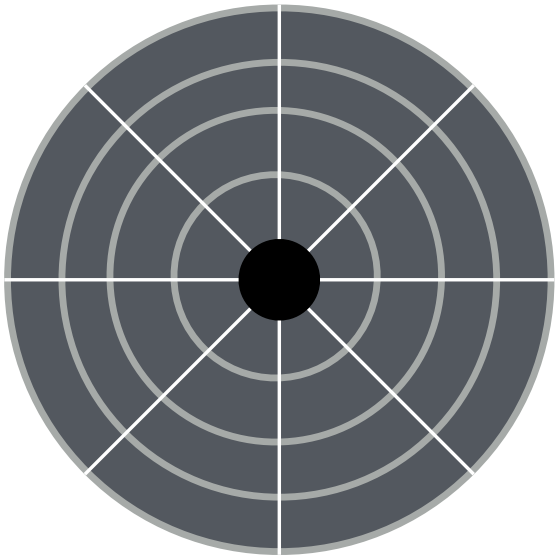
Other Improvements

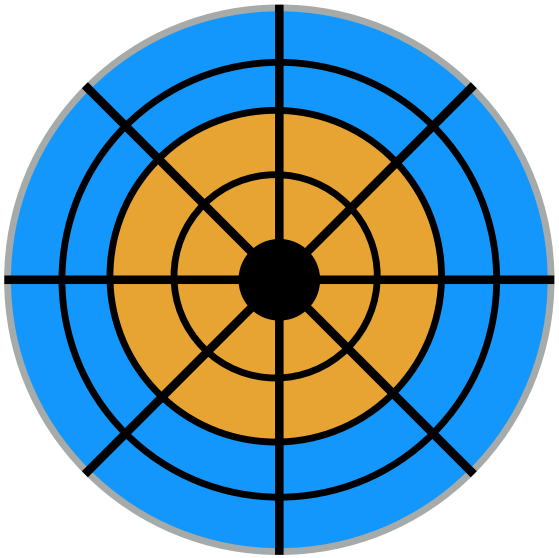
Track Skew

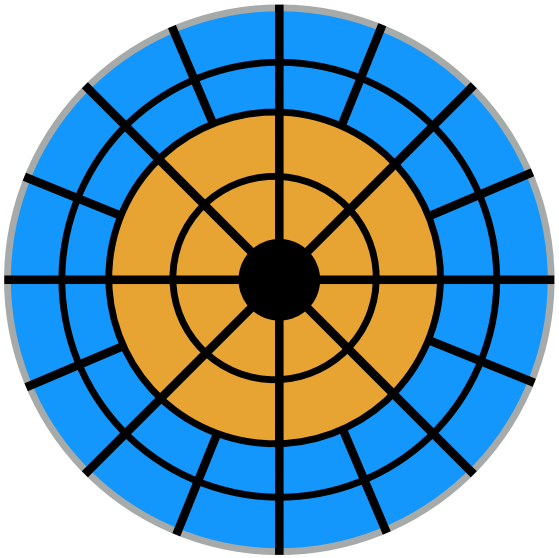
Zones

Cache

THE UNIVERSITY of ADELAIDE







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

Other Improvements

Track Skew

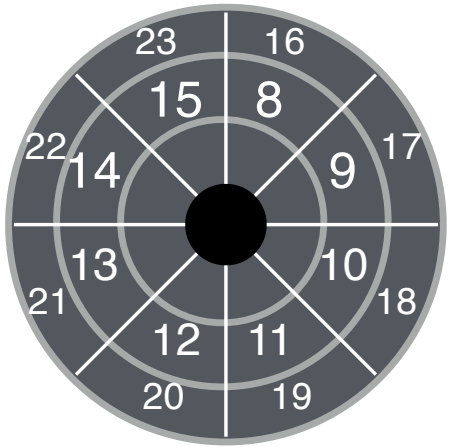
Zones

Cache

Drive Cache

Drives may cache both reads and writes.

OS caches data too



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

Solid State Disks (SSDs)

1995 – Replace rotating magnetic media with non-volatile memory (battery backed DRAM)

2009 – Use NAND Multi-Level Cell (2 or 3-bit/cell) flash memory

Sector (4 KB page) addressable, but stores 4-64 “pages” per memory block

Trapped electrons distinguish between 1 and 0

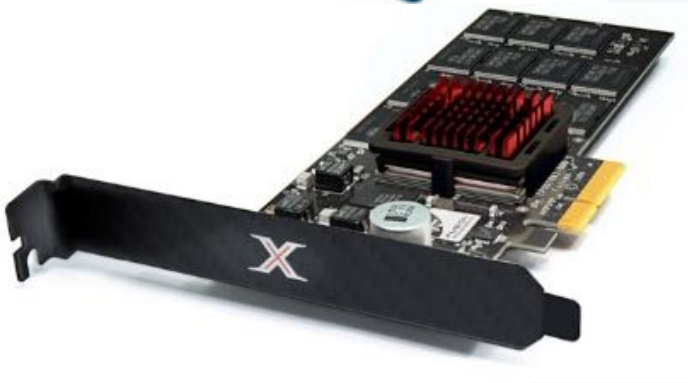
No moving parts (no rotate/seek motors)

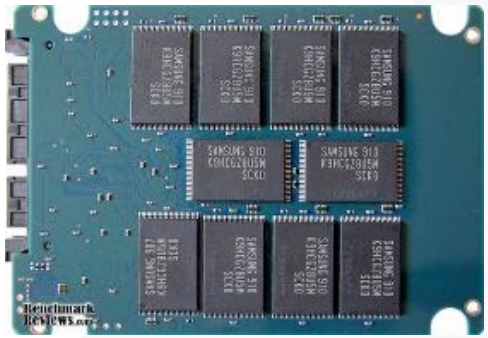
Eliminates seek and rotational delay (0.1-0.2ms access time)

Very low power and lightweight

Limited “write cycles”

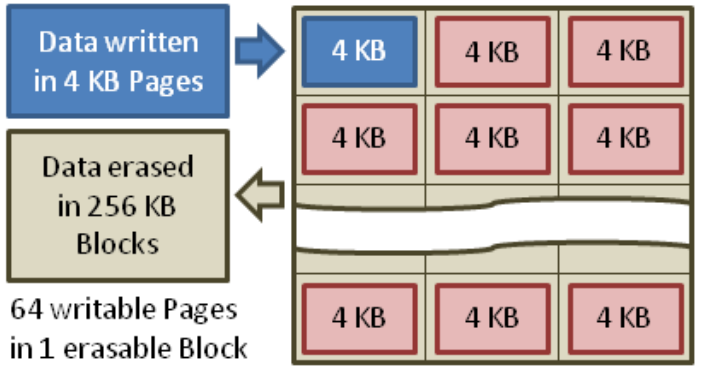
Rapid advances in capacity and cost ever since!



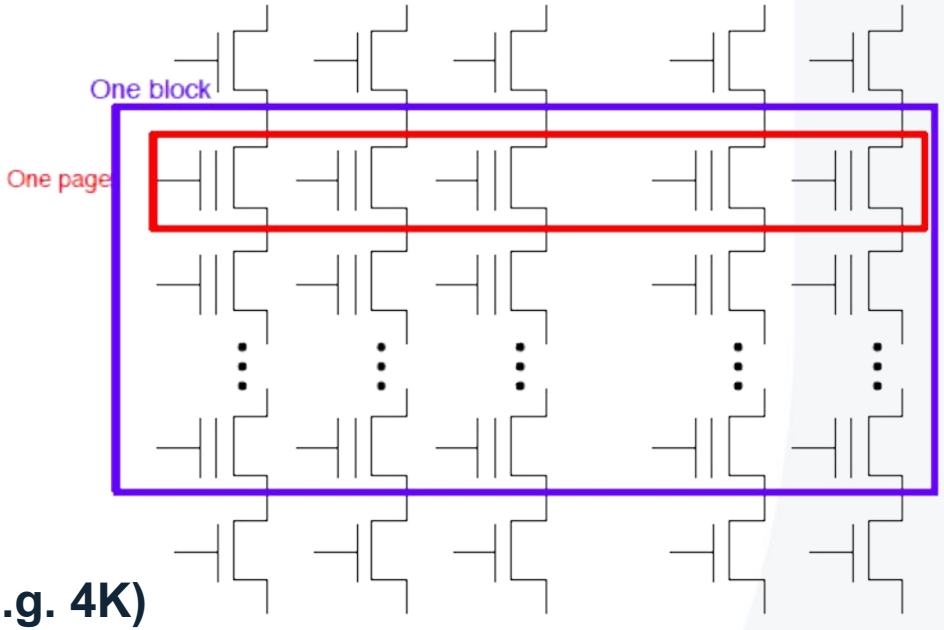


THE UNIVERSITY of ADELAIDE

Flash Memory



https://en.wikipedia.org/wiki/Solid-state_drive



Data read and written in page-sized chunks (e.g. 4K)

Cannot be addressed at byte level

Random access at block level for reads (no locality advantage)

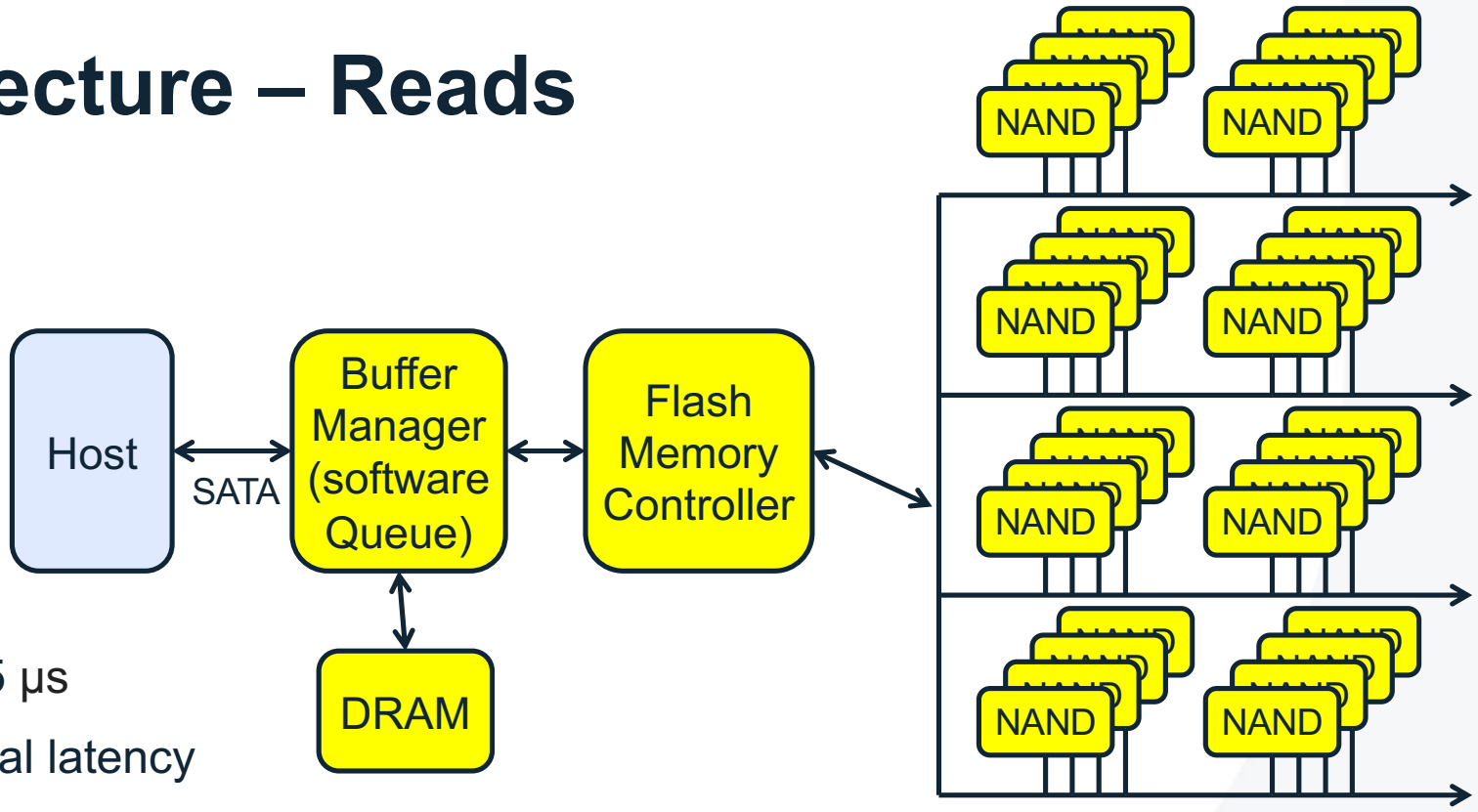
Writing of new blocks handled in order (kinda like a log)

Before writing, must be *erased* (256K block at a time)

Requires free-list management

CANNOT write over existing block (Copy-on-Write is normal case)

SSD Architecture – Reads



Read 4 KB Page: ~25 µs

No seek or rotational latency

Transfer time: transfer a 4KB page

- SATA: 300-600MB/s => ~4 x10^3 b / 400 x 10^6 bps => 10 µs

Latency = Queuing Time + Controller time + Xfer Time

Highest Bandwidth: Sequential OR Random reads

SSD Architecture – Writes

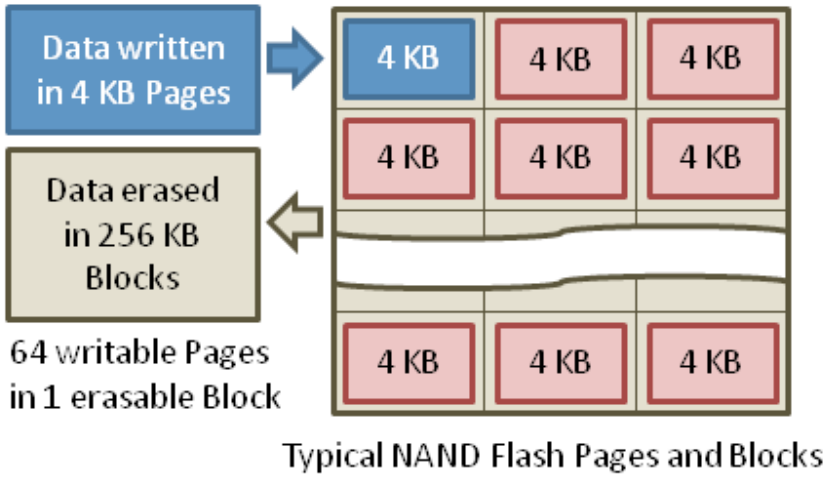
Writing data is complex! (~200µs – 1.7ms)

Can only write empty pages in a block

Erasing a block takes ~1.5ms

Controller maintains pool of empty blocks by coalescing used pages (read, erase, write), also reserves some % of capacity

Rule of thumb: writes 10x reads, erasure 10x writes



https://en.wikipedia.org/wiki/Solid-state_drive

SSD Architecture – Writes

SSDs provide same interface as HDDs to OS – read and write chunk (4KB) at a time

But can only overwrite data 256KB at a time!

Why not just erase and rewrite new version of entire 256KB block?

Erasure is very slow (milliseconds)

Each block has a finite lifetime, can only be erased and rewritten about 10K times

Heavily used blocks likely to wear out quickly

Solution – Two Systems Principles

1. Layer of Indirection

Maintain a *Flash Translation Layer (FTL)* in SSD

Map virtual block numbers (which OS uses) to physical page numbers (which flash memory controller uses)

Can now freely relocate data w/o OS knowing

2. Copy on Write

Do not overwrite a page when OS updates its data

Instead, write new version in a free page

Update FTL mapping to point to new location

Flash Translation Layer

No need to erase and rewrite entire 256KB block when making small modifications

SSD controller can assign mappings to spread workload across pages

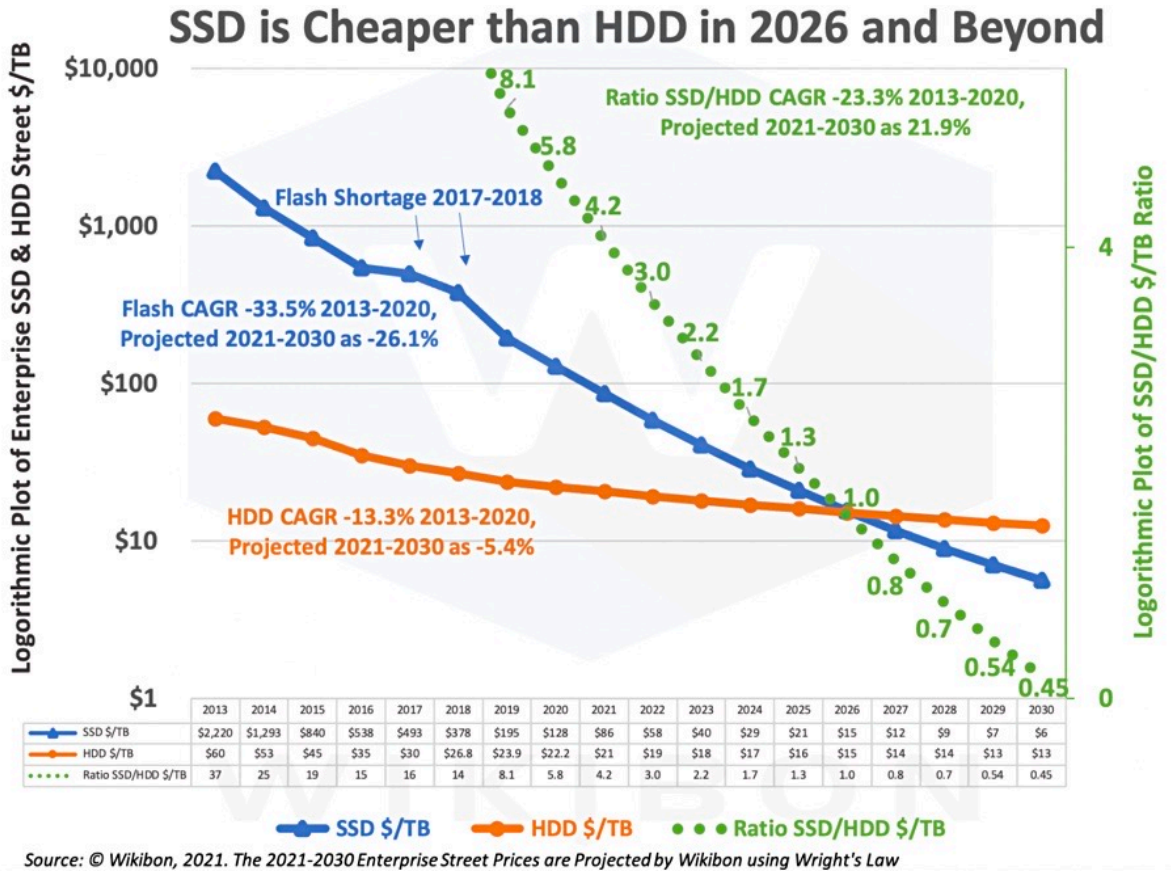
Wear Levelling

What to do with old versions of pages?

Garbage Collection in background

Erase blocks with old pages, add to free list

SSD prices drop faster than HDD



	2013	2014	2015	2016	2017	2018	2019	2020	2021	2022	2023	2024	2025	2026	2027	2028	2029	2030
SSD $/TB	$2,220	$1,293	$840	$538	$493	$378	$195	$128	$86	$58	$40	$29	$21	$15	$12	$9	$7	$6
HDD $/TB	$60	$53	$45	$35	$30	$26.8	$23.9	$22.2	$21	$19	$18	$17	$16	$15	$14	$14	$13	$13
Ratio SSD/HDD $/TB	37	25	19	15	16	14	8.1	5.8	4.2	3.0	2.2	1.7	1.3	1.0	0.8	0.7	0.54	0.45

Source: © Wikibon, 2021. The 2021-2030 Enterprise Street Prices are Projected by Wikibon using Wright's Law

SSD Summary

Pros (vs. hard disk drives):

- Low latency, high throughput (eliminate seek/rotational delay)
- No moving parts: Very lightweight, low power, silent, very shock insensitive

Read at memory speeds (limited by controller and I/O bus)

Cons

~~Small storage (0.1-0.5x disk)~~, ~~expensive (3-20x disk)~~

- Hybrid alternative: combine small SSD with large HDD

Asymmetric block write performance: read pg/erase/write pg

- Controller garbage collection (GC) algorithms have a major effect on performance

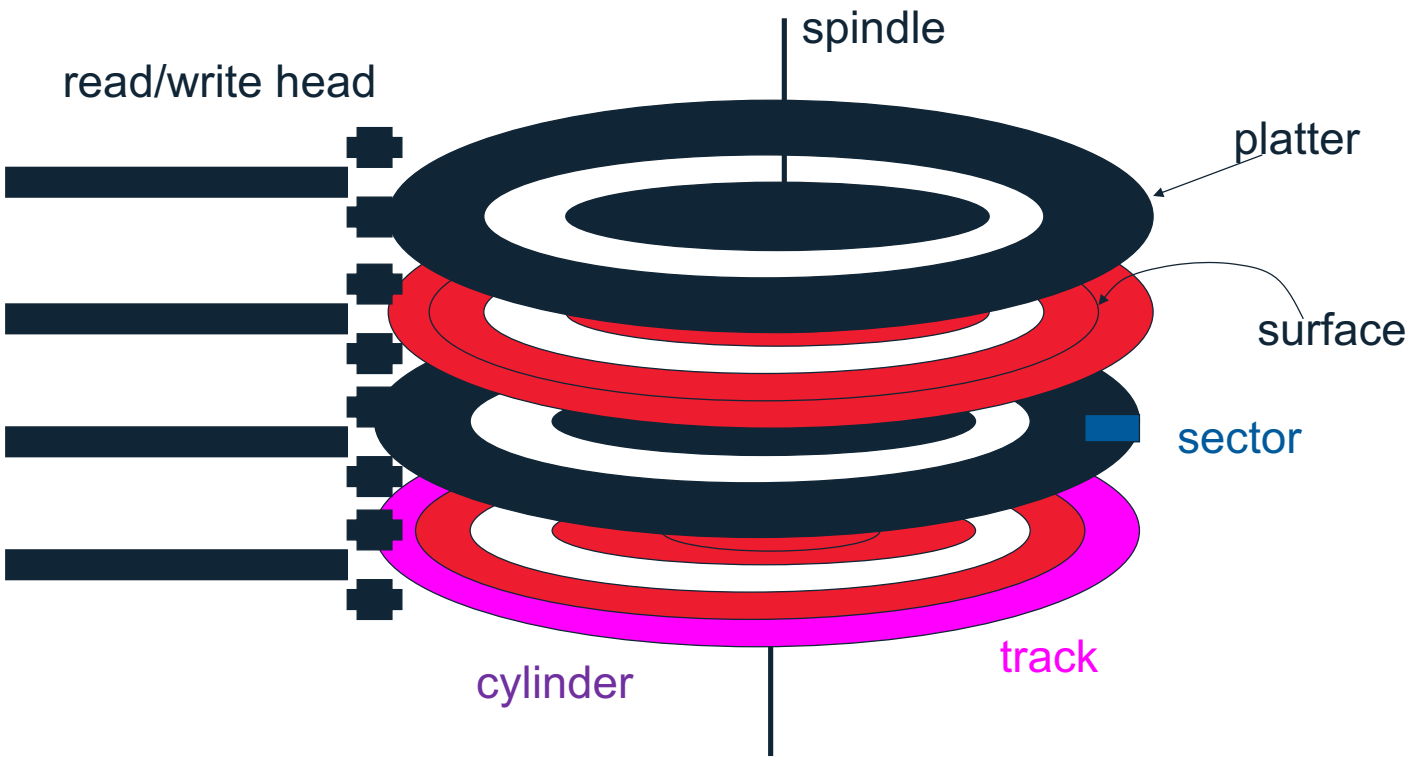
Limited drive lifetime

- 1-10K writes/page for MLC NAND
- Avg failure rate is 6 years, life expectancy is 9–11 years

These are changing rapidly!

No longer true!

Disk Summary



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

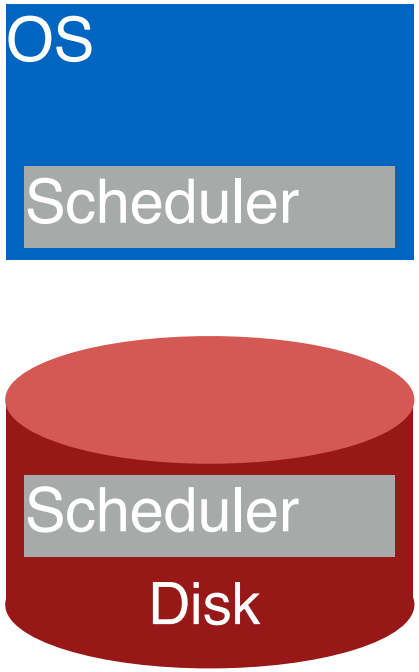
How long (roughly) does the below workload take?

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?

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, 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

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

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