Previously, on CS4410...

The Working Set Model

- Choose Δ page references as WS sliding window
  - track WS for the last Δ page references
- WSS_i = # of distinct pages referenced by p_i in latest Δ references
  - Δ too small does not cover locality
  - Δ too large covers many localities
- Thrashing if $\sum_i WSS_i$ > # frames
  - if so, swap out one of the processes; free its frames
- If enough free frames, increase degree of multiprogramming

WS Page Replacement

$\Delta = 4$

Time	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Trace	e	d	a	c	c									
Page a														
Page b														
Page c														
Page d														
Page e														
Faults	✗	✗	✗	✗	✓									

Pages in Memory

WS Page Replacement

$\Delta = 4$

Time	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Trace	e	d	a	c	c	d								
Pages in Memory: Page a														
Page b														
Page c														
Page d														
Page e														
Faults	✗	✗	✗	✗	✓	✓								

unmapping e, since not referenced in the last 4 references

WS Page Replacement

$\Delta = 4$

Time	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Trace	e	d	a	c	c	d	b							
Pages in Memory: Page a														
Page b														
Page c														
Page d														
Page e														
Faults	✗	✗	✗	✗	✓	✓	✗							

WS Page Replacement

$$\Delta = 4$$

Time	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Trace	e	d	a	c	c	d	b	c						
Pages in Memory: Page a														
Page b														
Page c														
Page d														
Page e														
Faults	✗	✗	✗	✗	✓	✓	✗	✓						

WS Page Replacement

$\Delta = 4$

Time	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Trace	e	d	a	c	c	d	b	c	e					
Pages in Memory: Page a														
Page b														
Page c														
Page d														
Page e														
Faults	✗	✗	✗	✗	✓	✓	✗	✓	✗					

WS Page Replacement

$\Delta = 4$

Time	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Trace	e	d	a	c	c	d	b	c	e	c				
Pages in Memory: Page a														
Page b														
Page c														
Page d														
Page e														
Faults	✗	✗	✗	✗	✓	✓	✗	✓	✗	✓				

WS Page Replacement

$\Delta = 4$

Time	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Trace	e	d	a	c	c	d	b	c	e	c	e			
Pages in Memory: Page a														
Page b														
Page c														
Page d														
Page e														
Faults	✗	✗	✗	✗	✓	✓	✗	✓	✗	✓	✓			

WS Page Replacement

$\Delta = 4$

Time	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Trace	e	d	a	c	c	d	b	c	e	c	e	a		
Pages in Memory: Page a														
Page b														
Page c														
Page d														
Page e														
Faults	✗	✗	✗	✗	✓	✓	✗	✓	✗	✓	✓	✗		

WS Page Replacement

$\Delta = 4$

Time	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Trace	e	d	a	c	c	d	b	c	e	c	e	a	d	
Pages in Memory: Page a														
Page b														
Page c														
Page d														
Page e														
Faults	✗	✗	✗	✗	✓	✓	✗	✓	✗	✓	✓	✗	✗	

WS Page Replacement

$\Delta = 4$

Time	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Trace	e	d	a	c	c	d	b	c	e	c	e	a	d	
Pages in Memory: Page a														
Page b														
Page c														
Page d														
Page e														
Faults	✗	✗	✗	✗	✓	✓	✗	✓	✗	✓	✓	✗	✗	

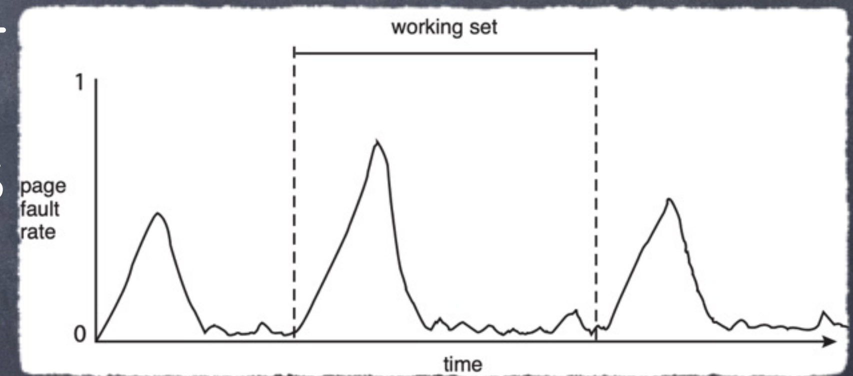
Approximating the Working Set

- Keep a k-bit tag in each page table entry (say, 2 bits)
- Set a timer interrupt to fire every Δ/k page references
  - if Δ = 10,000, then every 5000 references
- On timer interrupt
  - Shift tag right one bit
  - Copy REF bit in tag's leftmost bit and clear REF
  - Add to a free list any page whose tag is zero
- When a frame is needed, use the free list (check also REF bit!)
  - if free list is empty, pick any frame

Note: Must scan all frames!

Working Sets and Page Fault Rates

- As the working set changes, the page fault rate increases
  - a steep increase in the page fault rate indicates a shift in locality, which may require a different WS



- Idea: Change the number of frames allocated to a process in response to changes to its Page Fault rate
  - as long as the working sets of all processes currently in memory does not exceed the size of physical memory, no thrashing

PFF (Page Fault Frequency) Algorithm

Keep time t_{last} of last page fault

On page fault:

if $t_{current} - t_{last} \leq \tau$ (threshold) then add faulting page to the working set

else unmap all pages not referenced in $[t_{last}, t_{current}]$

PFF Page Replacement

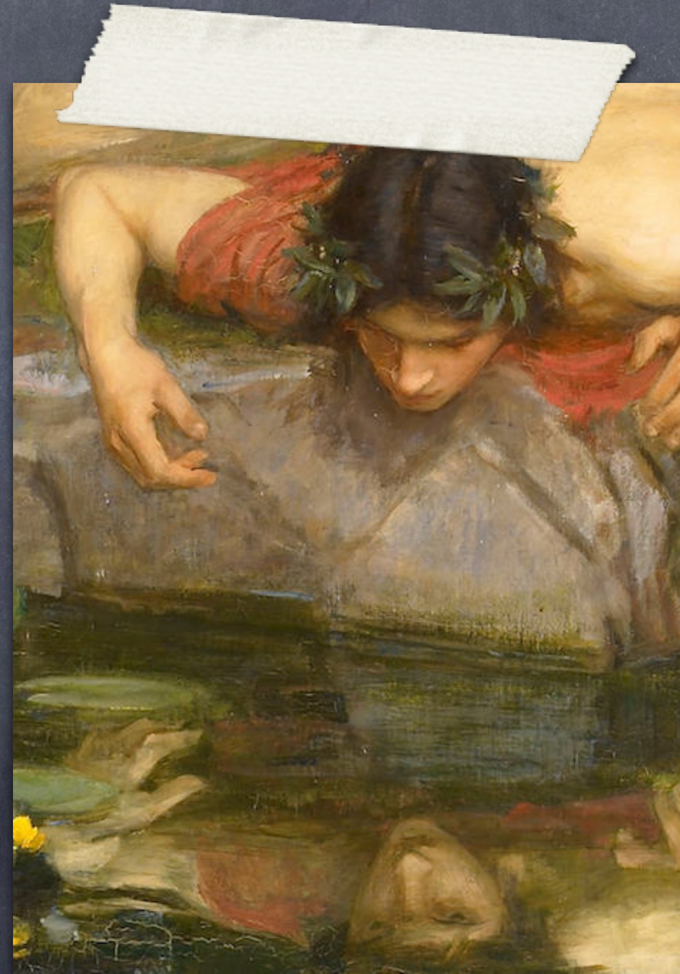
$$\tau = 2$$

Time		0	1	2	3	4	5	6	7	8	9	10	11	12	13
Trace		d	a	e	c	c	d	b	c	e	c	e	a	d	
Pages in Memory	Page a														
	Page b														
	Page c														
	Page d														
	Page e														
Faults		X	X	X	X			X		X			X	X	
$t_{\text{current}} - t_{\text{last}}$		0	1	1	1			3		2			3	1	

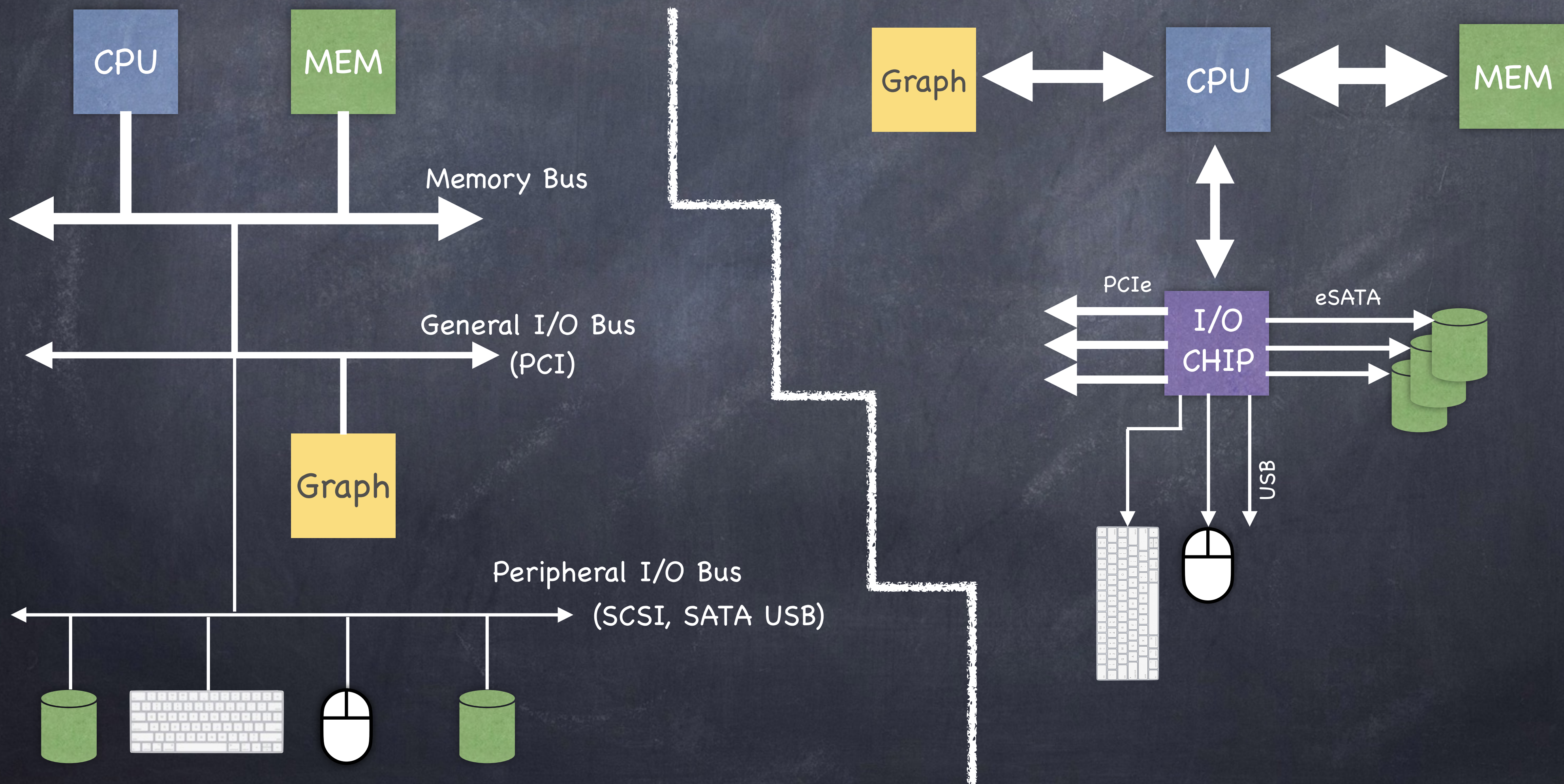
I/O Devices

You Need to Get Out More!

- How does a computer connect with the outside world?



I/O Architecture



Interacting with a Device

Abstraction

(what the user sees)

Interacting with a Device

Interface

(what the OS sees)

Internals

(what is needed to implement the abstraction)

Interacting with a Device

Registers | Status | Command | Data

Microcontroller
Memory
Other device specific chips

Internals
(what is needed to implement the abstraction)

Interacting with a Device

- OS controls device by reading/writing registers

Registers	Status	Command	Data

Internals

Microcontroller

Memory

Other device specific chips

(what is needed to implement the abstraction)

```
while (STATUS == BUSY)
    ; // wait until device is not busy
write data to DATA register
write command to COMMAND register
    // starts device and executes command
while (STATUS == BUSY)
    ; // wait until device is done with request
```

Tuning It Up

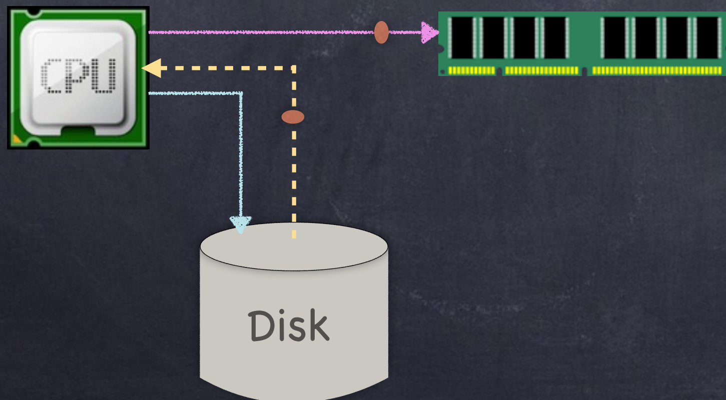
- CPU is polling
  - use interrupts
  - run another process while device is busy
  - what if device returns very quickly?
- CPU is copying all the data to and from DATA
  - use Direct Memory Access (DMA)

```
while (STATUS == BUSY)
      ; // wait until device is not busy
write data to DATA register
write command to COMMAND register
     // starts device and executes command
while (STATUS == BUSY)
     ; // wait until device is done with request
```

From interrupt-driven I/O to DMA

- Interrupt driven I/O
  - Device ⟷ CPU ⟷ RAM

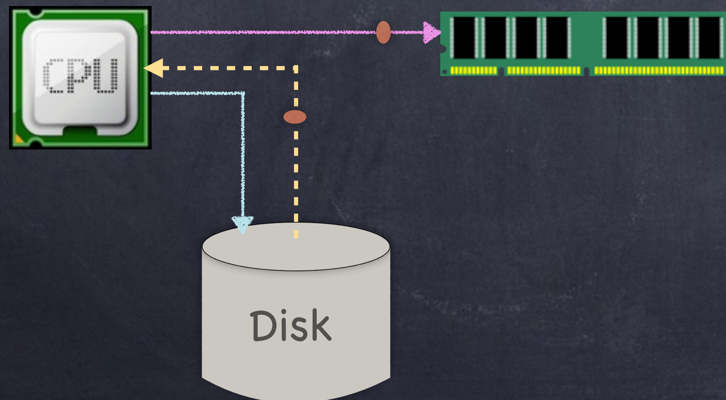
    for $(i = 1 \ldots n)$
    - CPU issues read request
    - device interrupts CPU with data
    - CPU writes data to memory



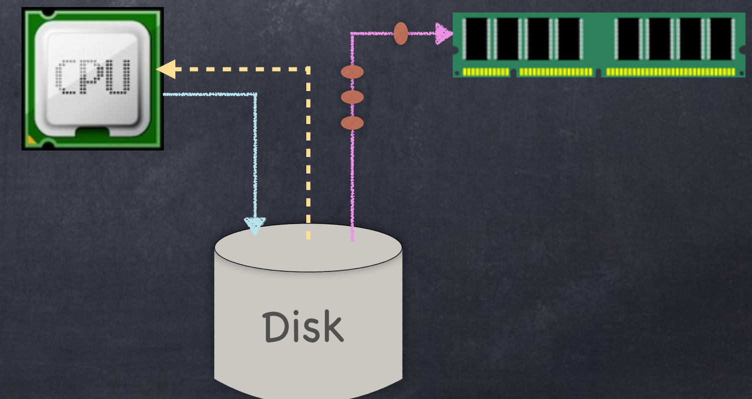
From interrupt-driven I/O to DMA

- Interrupt driven I/O
  - Device ⟷ CPU ⟷ RAM

    for $(i = 1 \ldots n)$
    - CPU issues read request
    - device interrupts CPU with data
    - CPU writes data to memory



- + Direct Memory Access
  - Device ⟷ RAM
    - CPU sets up DMA request
    - Device puts data on bus & RAM accepts it
    - Device interrupts CPU when done

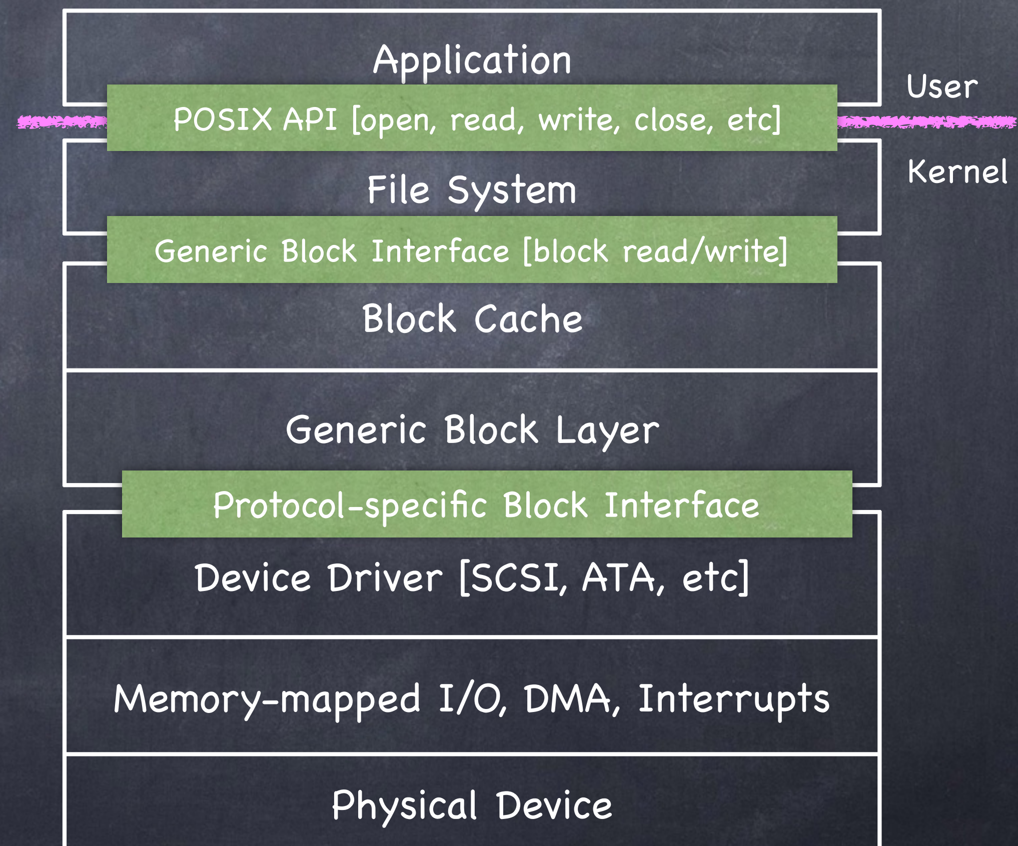


Communicating with devices

- Explicit I/O instructions (privileged)
  - in and out instructions in x86
- Memory-mapped I/O
  - map device registers to memory location
  - use memory load and store instructions to read/write to registers

How can the OS handle a multitude of devices?

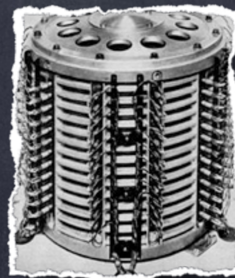
- Abstraction!
  - Encapsulate device specific interactions in a device driver
  - Implement device neutral interfaces above device drivers
- Humans are about 70% water…
  - …OSs are about 70% device drivers!



Persistent Storage

Storage Devices

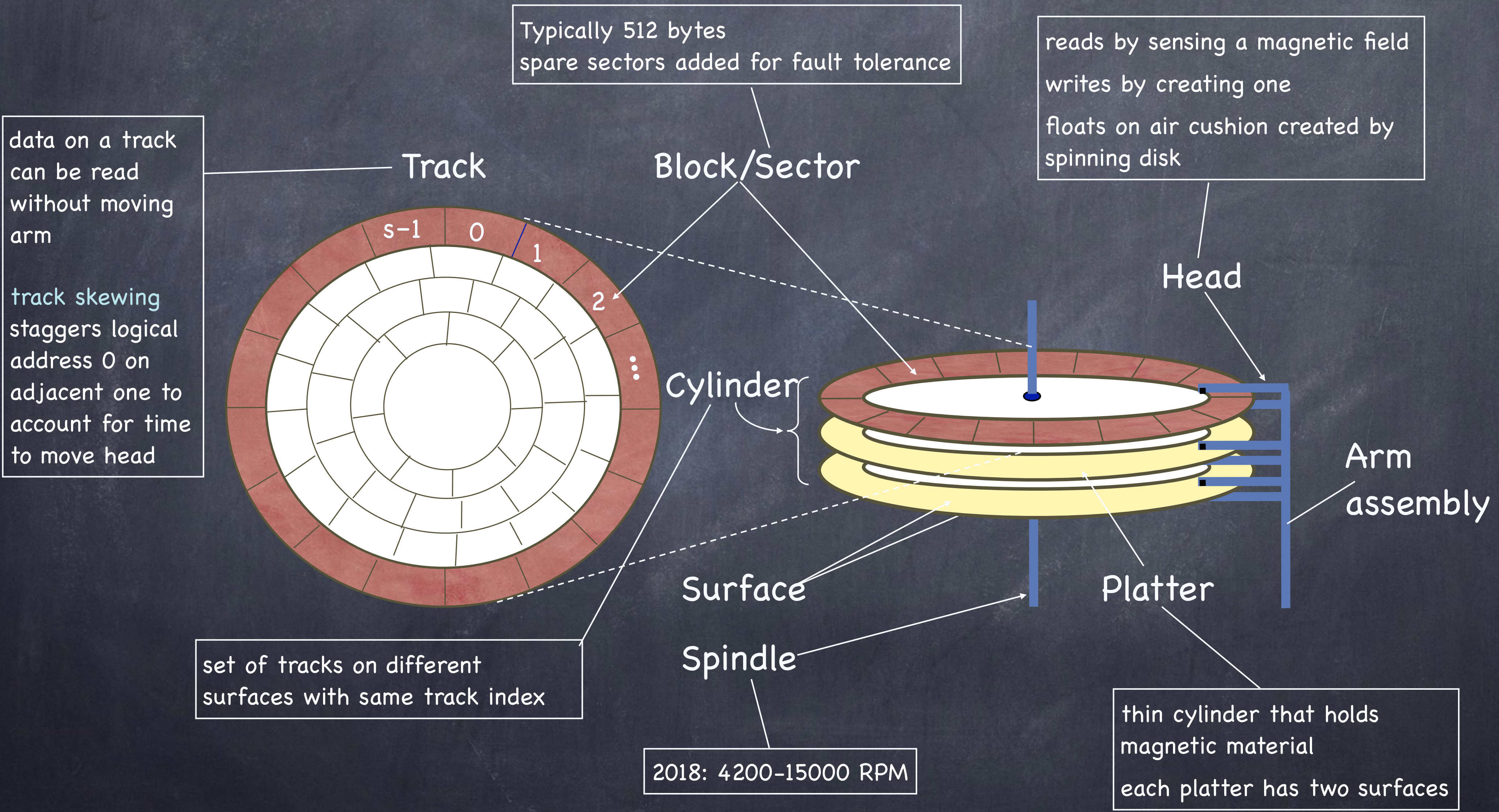
- We focus on two types of persistent storage
  - magnetic disks
    - servers, workstations, laptops
  - flash memory
    - smart phones, tablets, cameras, laptops
- Other exist(ed)
  - tapes
  - drums
  - clay tablets



Magnetic disk

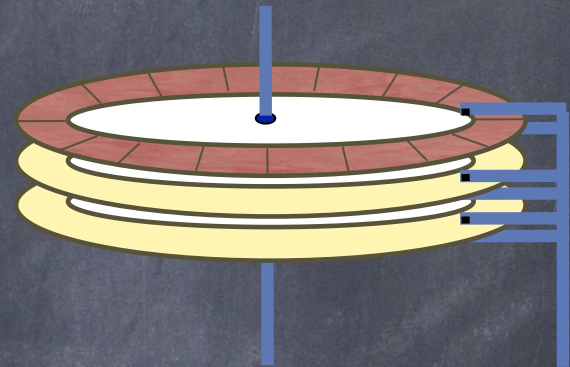
- Store data magnetically on thin metallic film bonded to rotating disk of glass, ceramic, or aluminum

Disk Drive Schematic



Disk Read/Write

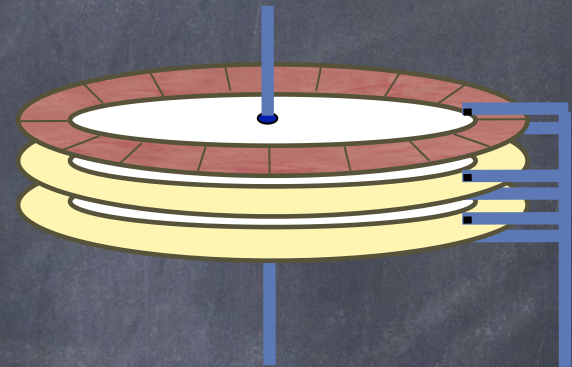
- Present disk with a sector address
  - Old: CHS = (cylinder, head, sector)
  - New abstraction: Logical Block Address (LBA)
    - linear addressing 0…N-1
- Heads move to appropriate track
  - seek
  - settle
- Appropriate head is enabled
- Wait for sector to appear under head
  - rotational latency
- Read/Write sector
  - transfer time



Disk access time:

Disk Read/Write

- Present disk with a sector address
  - Old: CHS = (cylinder, head, sector)
  - New abstraction: Logical Block Address (LBA)
    - linear addressing 0…N-1
- Heads move to appropriate track
  - seek (and though shalt approximately find)
  - settle (fine adjustments)
- Appropriate head is enabled
- Wait for sector to appear under head
  - rotational latency
- Read/Write sector
  - transfer time

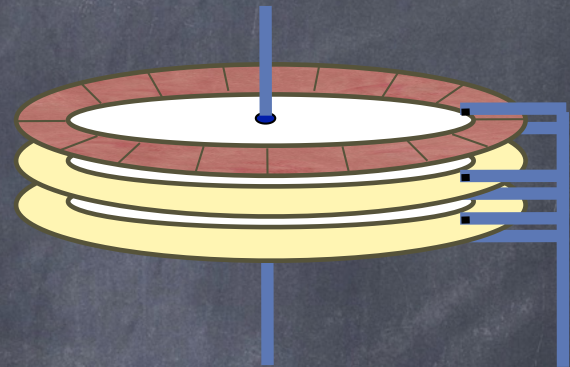


Disk access time:

seek time +

Disk Read/Write

- Present disk with a sector address
  - Old: CHS = (cylinder, head, sector)
  - New abstraction: Logical Block Address (LBA)
    - linear addressing 0...N-1
- Heads move to appropriate track
  - seek (and though shalt approximately find)
  - settle (fine adjustments)
- Appropriate head is enabled
- Wait for sector to appear under head
  - rotational latency
- Read/Write sector
  - transfer time



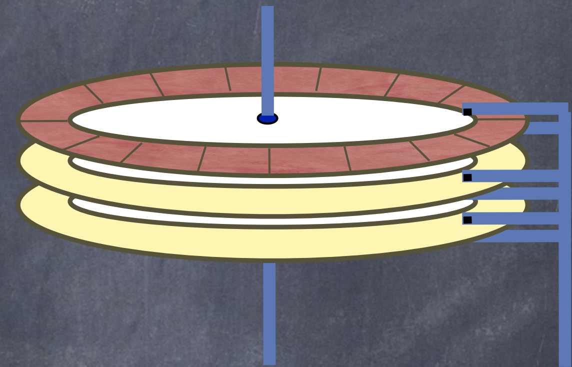
Disk access time:

seek time +

rotation time +

Disk Read/Write

- Present disk with a sector address
  - Old: CHS = (cylinder, head, sector)
  - New abstraction: Logical Block Address (LBA)
    - linear addressing 0…N-1
- Heads move to appropriate track
  - seek (and though shalt approximately find)
  - settle (fine adjustments)
- Appropriate head is enabled
- Wait for sector to appear under head
  - rotational latency
- Read/Write sector
  - transfer time



Disk access time:

seek time +

rotation time +

transfer time

Seek time: A closer look

- Minimum: time to go from one track to the next
  - 0.3-1.5 ms
- Maximum: time to go from innermost to outermost track
  - more than 10ms; up to over 20ms
- Average: average across seeks between each possible pair of tracks
  - approximately time to seek 1/3 of the way across disk

See notes for how that time is computed!

Seek time: A closer look

- Minimum: time to go from one track to the next
  - 0.3-1.5 ms
- Maximum: time to go from innermost to outermost track
  - more than 10ms; up to over 20ms
- Average: average across seeks between each possible pair of tracks
  - approximately time to seek 1/3 of the way across disk

How did we get that?

- To compute average seek time, add distance between every possible pair of tracks and divide by total number of pairs
  - assuming N tracks, N^2 pairs, and sum of distances is $\sum_{x=0}^{N}\sum_{y=0}^{N}|x-y|$ which we compute as $\int_{x=0}^{N}\int_{y=0}^{N}|x-y|dy\,dx$

How did we get that?

- To compute average seek time, add distance between every possible pair of tracks and divide by total number of pairs
  - assuming N tracks, N^2 pairs, and sum of distances is $\sum_{x=0}^{N}\sum_{y=0}^{N}|x-y|$ which we compute as $\int_{x=0}^{N}\int_{y=0}^{N}|x-y|dy\,dx$
  - The inner integral expands to $\int_{y=0}^{x}(x-y)dy + \int_{y=x}^{N}(y-x)dy$

    which evaluates to $x^2/2 + (N^2/2 - xN + x^2/2)$

How did we get that?

- To compute average seek time, add distance between every possible pair of tracks and divide by total number of pairs
  - assuming N tracks, N^2 pairs, and sum of distances is $\sum_{x=0}^{N}\sum_{y=0}^{N}|x-y|$ which we compute as $\int_{x=0}^{N}\int_{y=0}^{N}|x-y|dy\,dx$
  - The inner integral expands to $\int_{y=0}^{x}(x-y)dy + \int_{y=x}^{N}(y-x)dy$

    which evaluates to $x^2/2 + (N^2/2 - xn + x^2/2)$
  - The outer integral becomes $\int_{x=0}^{N}(x^2 + N^2/2 - xN)dx = N^3/3$

    which we divide by the number of pairs to obtain $N/3$

Seek time: A closer look

- Minimum: time to go from one track to the next
  - 0.3-1.5 ms
- Maximum: time to go from innermost to outermost track
  - more than 10ms; up to over 20ms
- Average: average across seeks between each possible pair of tracks
  - approximately time to seek 1/3 of the way across disk
- Head switch time: time to move from track i on one surface to the same track on a different surface
  - range similar to minimum seek time

Rotation time: A closer look

- Today most disk rotate at 4,200 to 15,000 RPM
  - ≈ 15ms to 4ms per rotation
  - good estimate for rotational latency is half that amount
- Head starts reading as soon as it settles on a track
  - track buffering to avoid "shoulda coulda" if any of the sectors flying under the head turn out to be needed

Transfer time: A closer look

- Surface transfer time
  - Time to transfer one or more sequential sectors to/from surface after head reads/writes first sector
  - Much smaller than seek time or rotational latency
    - 512 bytes at 100MB/s ≈ 5μs (0.005 ms)
  - Lower for outer tracks than inner ones
    - same RPM, but more sectors/track: higher bandwidth!
- Host transfer time
  - time to transfer data between host memory and disk buffer
    - 60MB/s (USB 2.0); 640 MB/s (USB 3.0); 25.GB/s (Fibre Channel 256GFC)

Disk Buffer

- Small cache [``Track buffer”, 8 to 16 MB] holds data
  - read from disk
  - about to be written to disk
- On write
  - write back (return from write as soon as data is cached)
  - write through (return once it is on disk)

Computing I/O time

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

- The rate of I/O is computed as

$$R_{I/O} = \frac{Size_{Transfer}}{T_{I/0}}$$

Example: Toshiba MK3254GSY (2008)

Size	
Platters/Heads	2/4
Capacity	320GB
Performance	
Spindle speed	7200 RPM
Avg. seek time R/W	10.5/12.0 ms
Max. seek time R/W	19 ms
Track-to-track	1 ms
Surface transfer time	54-128 MB/s
Host transfer time	375 MB/s
Buffer memory	16MB
Power	
Typical	16.35 W
Idle	11.68 W

500 Random Reads

Size	
Platters/Heads	2/4
Capacity	320GB
Performance	
Spindle speed	7200 RPM
Avg. seek time R/W	10.5/12.0 ms
Max. seek time R/W	19 ms
Track-to-track	1 ms
Surface transfer time	54-128 MB/s
Host transfer time	375 MB/s
Buffer memory	16MB
Power	
Typical	16.35 W
Idle	11.68 W

- Workload
  - 500 read requests, randomly chosen sector
  - served in FIFO order
- How long to service them?
  - 500 times (seek + rotation + transfer)
  - seek time: 10.5 ms (avg)
  - rotation time:
    - 7200 RPM = 120 RPS
    - rotation time 8.3 ms
    - on average, half of that: 4.15 ms
  - transfer time
    - at least 54 MB/s
    - 512 bytes transferred in (.5/54,000) seconds = 9.26μs
  - Total time:
    - 500 x (10.5 + 4.15 + 0.009) ms ≈ 7.33 sec

$$R_{I/O} = \frac{500 \times .5 \times 10^{-3} MB}{7.33\, s} = 0.034\, MB/s$$

500 Sequential Reads

Size	
Platters/Heads	2/4
Capacity	320GB
Performance	
Spindle speed	7200 RPM
Avg. seek time R/W	10.5/12.0 ms
Max. seek time R/W	19 ms
Track-to-track	1 ms
Surface transfer time	54-128 MB/s
Host transfer time	375 MB/s
Buffer memory	16MB
Power	
Typical	16.35 W
Idle	11.68 W

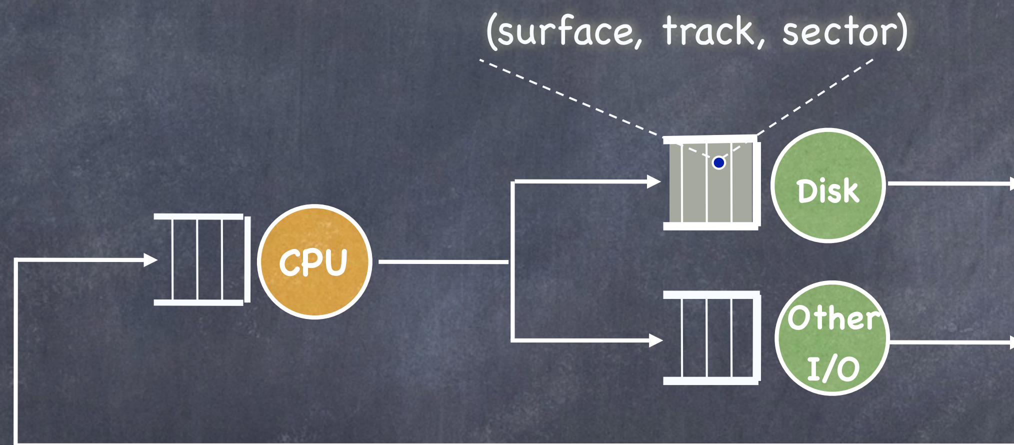
- Workload
  - 500 read requests for sequential sectors on the same track
  - served in FIFO order
- How long to service them?
  - seek + rotation + 500 times transfer
  - seek time: 10.5 ms (avg)
  - rotation time:
    - 4.15 ms, as before
  - transfer time
    - outer track: 500 x (.5/128000) ≈ 2ms
    - inner track: 500 x (.5/54000) seconds ≈ 4.6ms
  - Total time is between:
    - outer track: (2 + 4.15 + 10.5) ms ≈ 16.65 ms

      $R_{I/O} = \frac{500 \times .5 \times 10^{-3} MB}{16.65\, ms} = 15.02\, MB/s$
    - inner track: (4.6 + 4.15 + 10.5) ms ≈ 19.25 ms

      $R_{I/O} = \frac{500 \times .5 \times 10^{-3} MB}{19.25\, ms} = 12.99\, MB/s$

Disk Head Scheduling

- In a multiprogramming/time sharing environment, a queue of disk I/Os can form



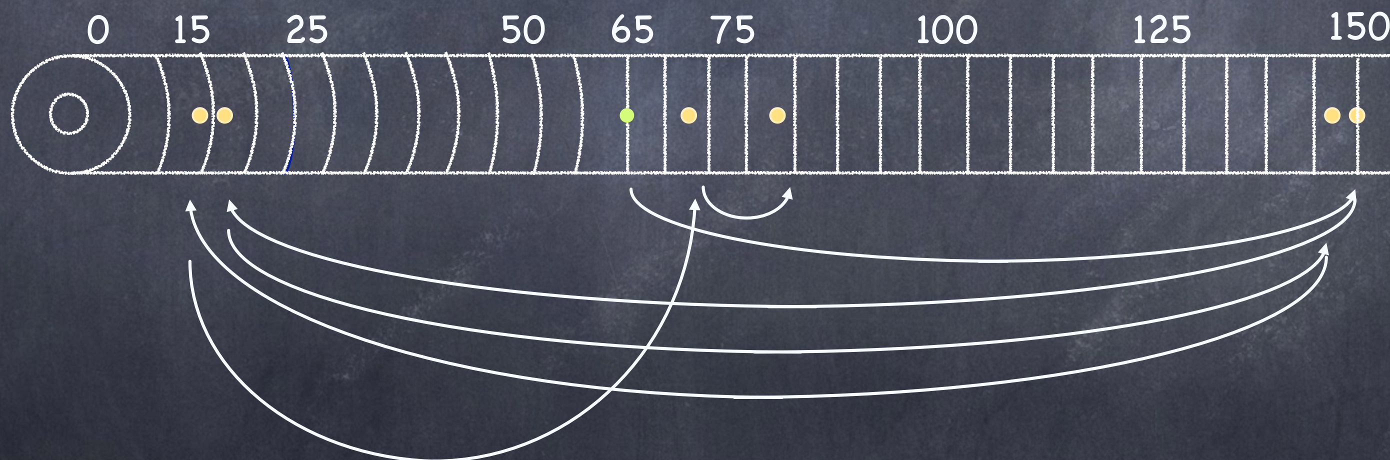
- OS maximizes disk I/O throughput by minimizing head movement through disk head scheduling
  - and this time we have a good sense of the length of the task!

FCFS

- Assume a queue of request exists to read/write tracks

....	83	72	14	147	16	150

and the head is on track 65



FCFS scheduling results in disk head moving 550 tracks

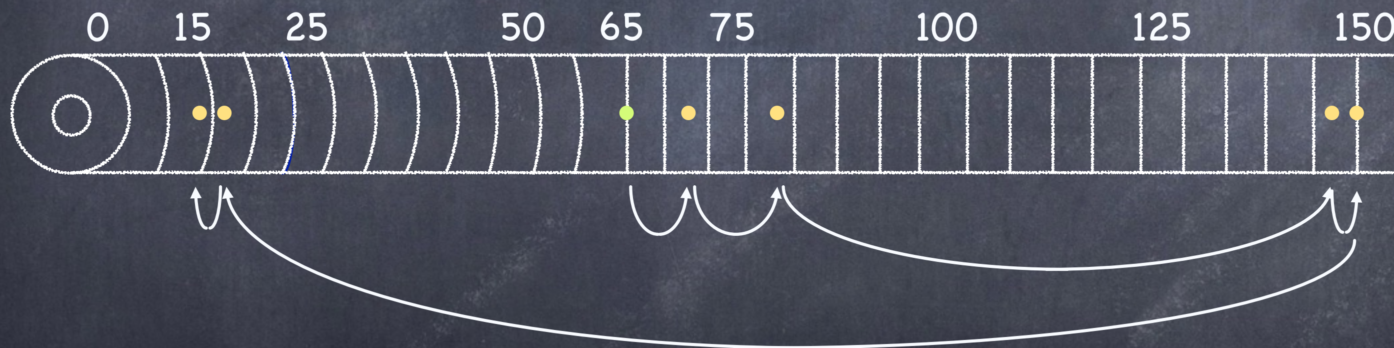
and makes no use of what we know about the length of the tasks!

SSTF: Shortest Seek Time First

- Greedy scheduling

Rearrange queue from: .... | 83 | 72 | 14 | 147 | 16 | 150 |

to: .... | 14 | 16 | 150 | 147 | 83 | 72 |



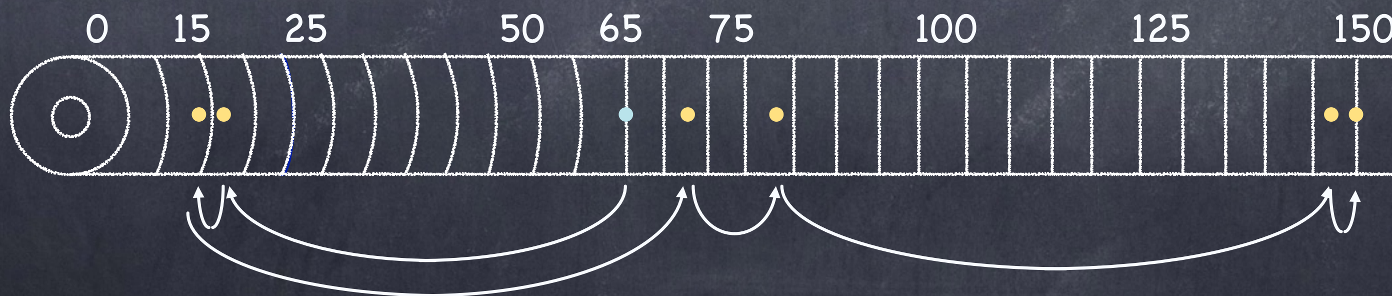
Head moves 221 tracks BUT

- OS knows blocks, not tracks (easily fixed)
- starvation

SCAN Scheduling
"Elevator"

- Move the head in one direction until all requests have been serviced, and then reverse
  - sweeps disk back and forth

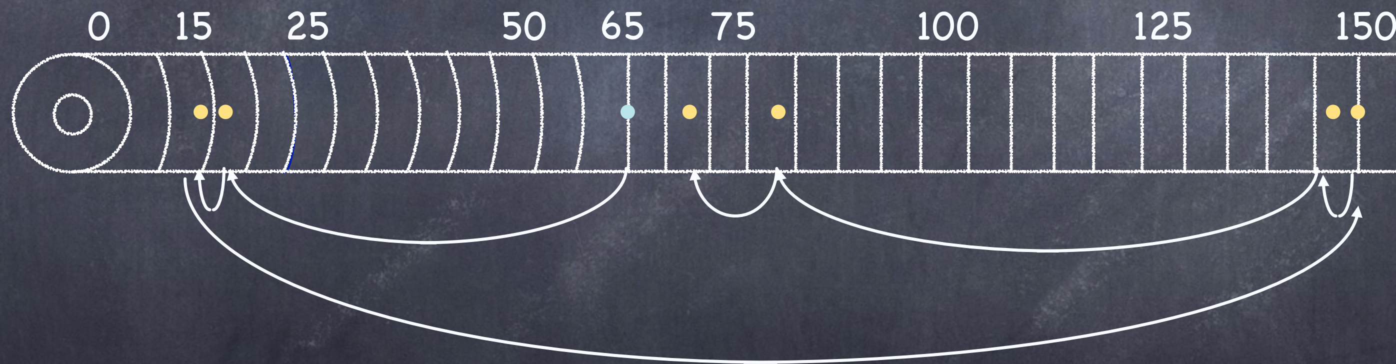
Rearrange queue from:	....	83	72	14	147	16	150
to:	....	150	147	83	72	14	16



Head moves 187 tracks.

C-SCAN scheduling

- Circular SCAN
  - sweeps disk in one direction (from outer to inner track), then resets to outer track and repeats



- More uniform wait time than SCAN
  - moves head to serve requests that are likely to have waited longer

OS Outsources Scheduling Decisions

- Selecting which track to serve next should include rotation time (not just seek time!)
  - SPTF: Shortest Positioning Time First
- Hard for the OS to estimate rotation time accurately
  - Hierarchical decision process
    - OS sends disk controller a batch of "reasonable" requests
    - disk controller makes final scheduling decisions

Back to Storage…

What qualities we want from storage?

- Reliable: It returns the data you stored
- Fast: It returns the data you stored promptly
- Affordable: It does not break the bank
- Plenty: It holds everything you need

What we may instead get is a SLED!

- Single, Large, Expensive Disk

RAID

Redundant Array of Inexpensive* Disks

* In industry, “inexpensive” has been replaced by “independent” :-)

E Pluribus Unum

- Implement the abstraction of a faster, bigger and more reliable disk using a collection of slower, smaller, and more likely to fail disks
  - different configurations offer different tradeoffs
- Key feature: transparency
  - The Power of Abstraction™
  - to the OS looks like a single, large, highly performant and highly reliable single disk (a SLED, hopefully with lower-case "e"!)
    - a linear array of blocks
    - mapping needed to get to actual disk
    - cost: one logical I/O may translate into multiple physical I/Os
- In the box:
  - microcontroller, DRAM (to buffer blocks) [sometimes non-volatile memory, parity logic]

Failure Model

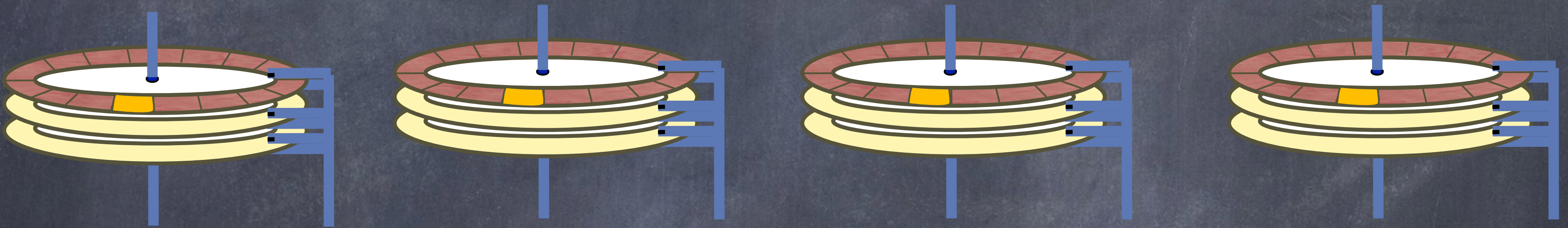
- RAID adopts the strong, somewhat unrealistic Fail-Stop failure model (electronic failure, wear out, head damage)
  - component works correctly until it crashes, permanently
    - disk is either working: all sectors can be read and written
    - or has failed: it is permanently lost
  - failure of the component is immediately detected
    - RAID controller can immediately observe a disk has failed and accesses return error codes
- In reality, disks can also suffer from isolated sector failures
  - Permanent: physical malfunction (magnetic coating, scratches, contaminants)
  - Transient: data is corrupted, but new data can be successfully read from/written to sector

How to Evaluate a RAID

- Capacity
  - what fraction of the sum of the storage of its constituent disks does the RAID make available?
- Reliability
  - How many disk faults can a specific RAID configuration tolerate?
- Performance
  - Workload dependent

RAID-0: Striping

Spread blocks across disks using round robin



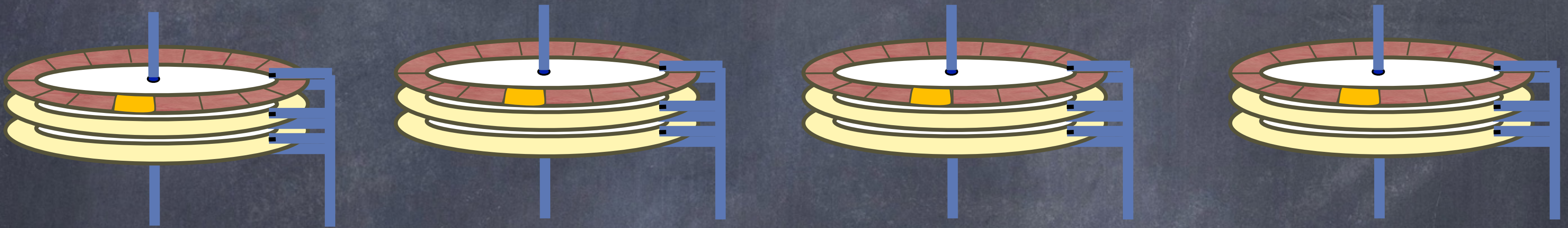
Stripe	0	1	2	3
Stripe	4	5	6	7
Stripe	8	9	10	11
Stripe	12	13	14	15

+ Excellent parallelism
  - can read/write from multiple disks

– Worst-case positioning time
  - wait for largest across all disks

RAID-0: Striping (Big Chunk Edition)

Spread blocks across disks using round robin



Stripe	0	2	4	6
	1	3	5	7
Stripe	8	10	12	14
	9	11	13	15

+ improve positioning time

— decrease parallelism

RAID-0: Evaluation

- Capacity
  - Excellent: N disks, each holding B blocks support the abstraction of a single disk with NxB blocks
- Reliability
  - Poor: Striping reduces reliability
    - Any disk failure causes data loss
- Performance
  - Workload dependent, of course
  - We'll consider two workloads
    - Sequential: single disk transfers S MB/s
    - Random: single disk transfer R MB/s
    - S >> R

RAID-0: Performance

- Single-block read/write throughput
  - about the same as accessing a single disk
- Latency
  - Read: T ms (latency of one I/O op to disk)
  - Write: T ms
- Steady-state read/write throughput
  - Sequential: N x S MB/s
  - Random: N x R MB/s

RAID-1: Mirroring

Each block is replicated twice

0	0	1	1
2	2	3	3
4	4	5	5
6	6	7	7

Read from any

Write to both

RAID-1: Evaluation

- Capacity
  - Poor: N disks of B blocks yield (N x B)/2 blocks
- Reliability
  - Good: Can tolerate the loss (not corruption!) of any one disk
- Performance
  - Fine for reads: can choose any disk
  - Poor for writes: every logical write requires writing to both disks
    - suffers worst seek+rotational delay of the two writes

RAID-1: Performance

- Steady-state throughput
  - Sequential Writes: N/2 x S MB/s
    - Each logical Write involves two physical Writes
  - Sequential Reads: as low as N/2 x S MB/s

0	0	1	1
2	2	3	3
4	4	5	5
6	6	7	7

Suppose we want to read
0, 1, 2, 3, 4, 5, 6, 7

RAID-1: Performance

- Steady-state throughput
  - Sequential Writes: N/2 x S MB/s
    - Each logical Write involves two physical Writes
  - Sequential Reads: as low as N/2 x S MB/s

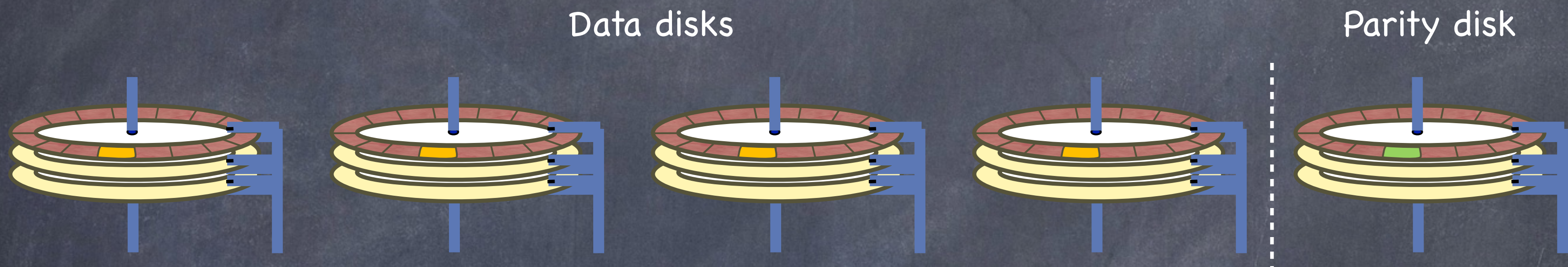
0	0	1	1
2	2	3	3
4	4	5	5
6	6	7	7

Suppose we want to read
0, 1, 2, 3, 4, 5, 6, 7

Each disk only delivers half of his bandwidth:
half of its blocks are skipped!

  - Random Writes: N/2 x R MB/s
    - Each logical Write involves two physical Writes
  - Random Reads: N x R MB/s
    - Reads can be distributed across all disks
- Latency for Reads and Writes: T ms

RAID-4: Block Striped, with Parity



Stripe 0	1	2	3	P0
Stripe 4	5	6	7	P1
Stripe 8	9	10	11	P2
Stripe 12	13	14	15	P3

1	1	0
0	1	0
0	0	1

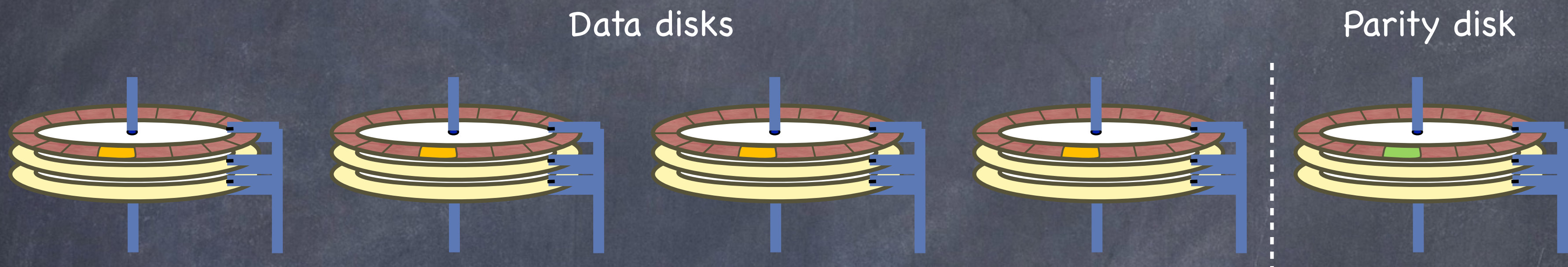
1	0	0
1	1	0
0	1	1

1	0	0
0	1	0
1	0	1

1	1	0
1	1	1
0	0	1

0	0	0
0	0	1
1	1	0

RAID-4: Block Striped, with Parity



Stripe 0	1	2	3	P0
Stripe 4	5	6	7	P1
Stripe 8	9	10	11	P2
Stripe 12	13	14	15	P3

1	1	0
0	1	0
0	0	1

1	0	0
1	1	0
0	1	1

1	0	0
0	1	0
1	0	1

1	1	0
1	1	1
0	0	1

0	0	0
0	0	1
1	1	0

Disk controller can identify faulty disk

- single parity disk can detect and correct errors

RAID-4: Evaluation

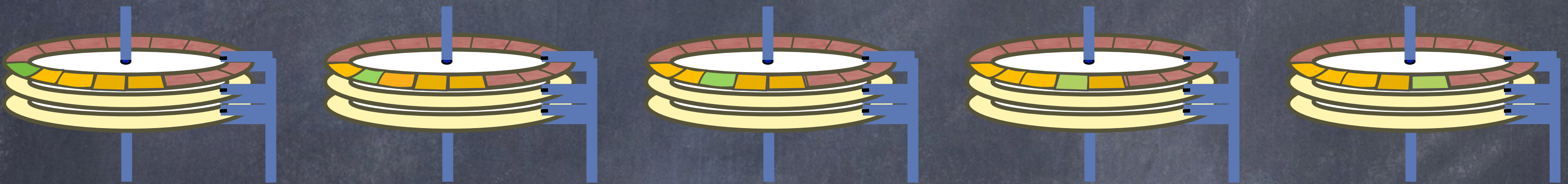
- Capacity
  - N disks of B blocks yield (N-1) x B blocks
- Reliability
  - Tolerates the failure of any one disk
- Performance
  - Fine for sequential read/write accesses and random reads
  - Random writes are a problem!

RAID-4: Performance

- Sequential Reads: (N-1) x S MB/s
- Sequential Writes: (N-1) x S MB/s
  - compute & write parity block once for the full stripe
- Random Read: (N-1) x R MB/s
- Random Writes: R/2 MB/s (N is gone! Yikes!)
  - need to read block B_{old} from disk and parity block P_{old}
  - Compute $P_{new} = (B_{old} \text{ XOR } B_{new}) \text{ XOR } P_{old}$
  - Write back B_{new} and P_{new}
  - Every write must go through parity disk, eliminating any chance of parallelism
  - Every logical I/O requires two physical I/Os at parity disk: can at most achieve 1/2 of its random transfer rate (i.e. R/2)
- Latency: Reads: T ms; Writes: 2T ms

RAID-5: Rotating Parity (avoids the bottleneck)

Parity and Data distributed across all disks



0	1	2	3	P0
5	6	7	P1	4
10	11	P2	8	9
15	P3	12	13	14
P4	16	17	18	19

RAID-5: Evaluation

- Capacity & Reliability
  - As in Raid-4
- Performance
  - Sequential read/write accesses as in RAID-4
    - (N-1) x S MB/s
  - Random Reads are slightly better
    - N x R MB/s (instead of (N-1) x R MB/s)
  - Random Writes much better than RAID-4: R/2 x N/2
    - as in RAID-4 writes involve two operations at every disk: each disk can achieve at most R/2
    - but, without a bottleneck parity disk, we can issue up to N/2 writes in parallel (each involving 2 disks)