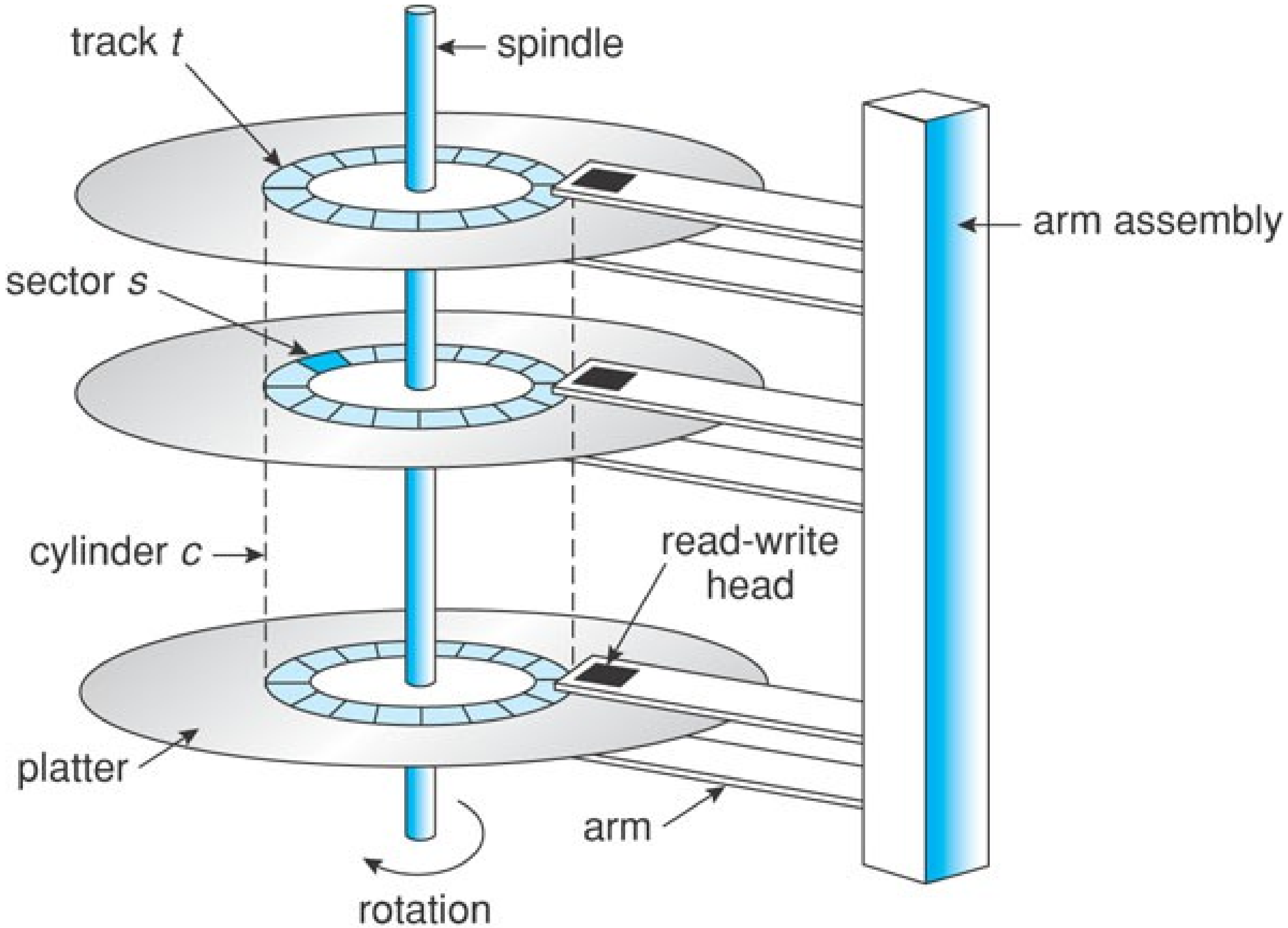
Mass-Storage

ICS332
Operating Systems

Magnetic Disks

- Magnetic disks are (still) the most common secondary storage devices today
- They are “messy”
  - Errors, bad blocks, missed seeks, moving parts
- And yet, the data they hold is critical
- The OS used to hide all the “messiness” from higher-level software
  - Programs shouldn’t have to know anything about the way the disk is built
- This has been done increasingly with help from the hardware
  - i.e., the disk controller
- What do disks look like?

Disk Structure



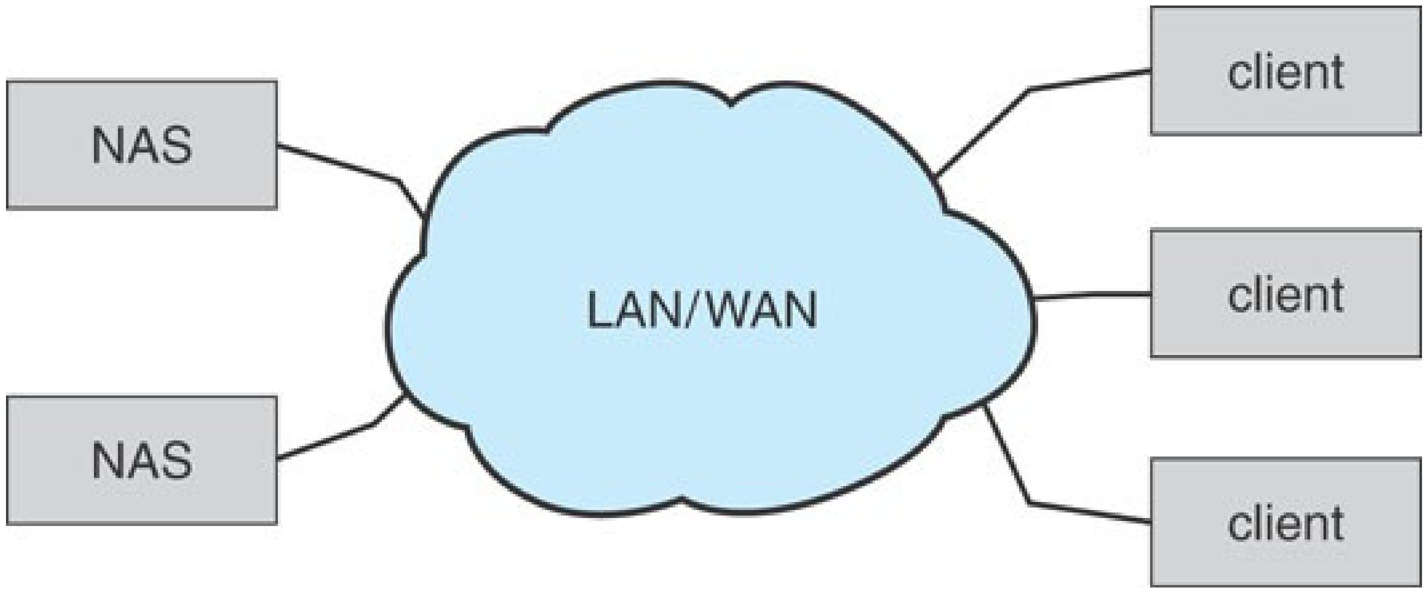
Disk Access

- A disk requires a lot of information for an access
  - Head #, sector #, track #, etc.
- Disks today are more complicated than the simple picture
  - e.g., sectors of different sizes to deal with varying densities and radial speeds with respect to the distance to the spindle
- Nowadays, disks comply with standard interfaces
  - EIDE, ATA, SATA, USB, Fiber Channel, SCSI
- The disk, in these interfaces, is seen as an array of logical blocks (512 bytes)
- The device, in hardware, does the translation between the block # and the platter #, sector #, track #, etc.
- This is good:
  - The kernel code to access the disk is straightforward
  - The controller can do a lot of work, e.g., transparently hiding bad blocks
- The cost is that some cool optimizations that the kernel could perhaps do are not possible, since all its hidden from it

Network-Attached Storage

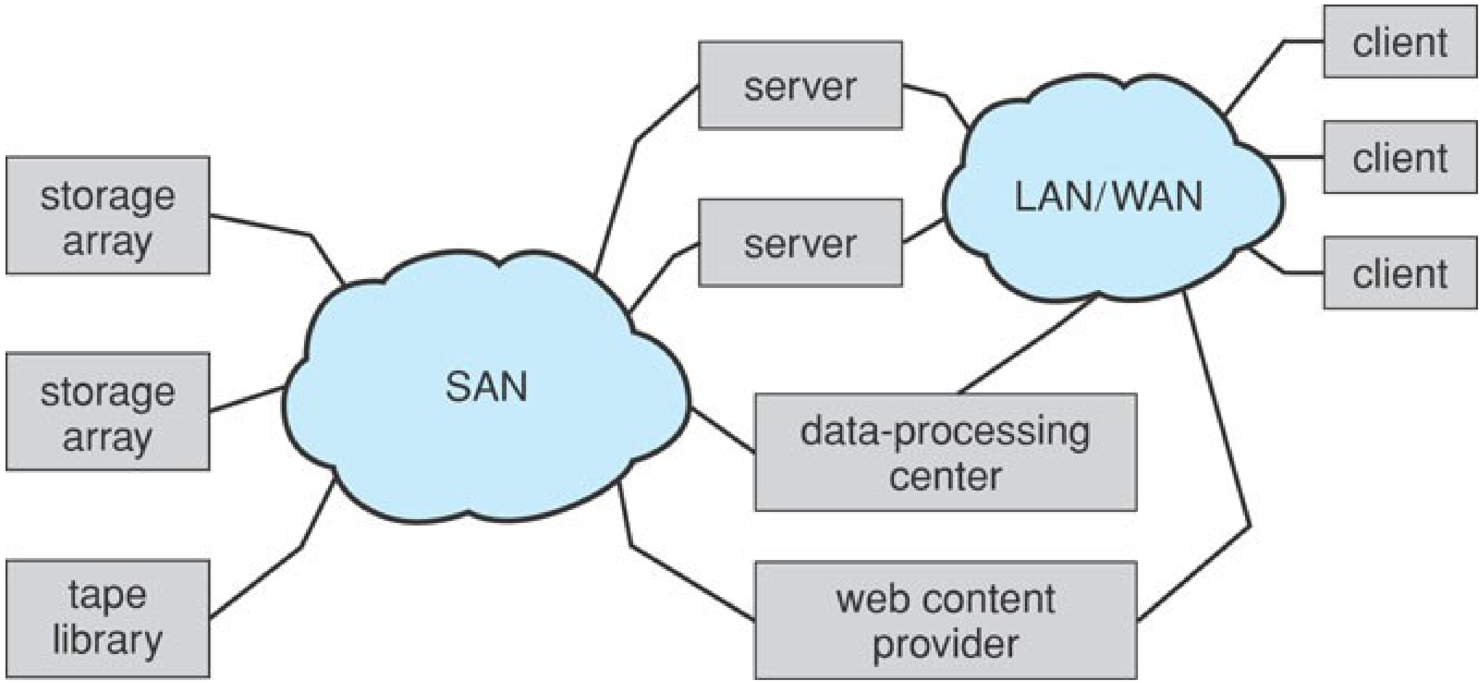
- Typically, one thinks of a disk as attached to a host (i.e., a computer)
  - called "host-attached storage"
- However, it is often convenient to think of compute resources and storage resources as separate
  - e.g., Web servers that answer http requests vs. the database that holds web application records
- One doesn't have to think of a disk as within a host, but as an "appliance" that can be put on a network
  - These appliances are accessed over the network, using some standard protocol (e.g., NFS + RPC)
  - No more, say, SCSI interfaces and SCSI ports, but instead network protocols and network cards
    - Although there is a SCSI interface (SCSI over IP), making the host unaware that it's accessing storage over the network
- This is called Network-Attached Storage (NAS)
  - Many appliances sold by many vendors, and pretty cheaply
  - e.g., 2TB NAS on Amazon around $80 in 2016

Network-Attached Storage



Storage-area Networks

- One drawback of NAS is that the network can be overloaded with I/O requests
  - Not a big deal if the applications/users don't use the network much
- A Storage-area Network (SAN) is a private network for network-attached storage devices



Disk Performance

- We've said many times that disks are slow
- Disk request performance depends on three steps
  - Seek - moving the disk arm to the correct cylinder
    - Depends on how fast disk arm can move (increasing very slowly over the years)
  - Rotation - waiting for the sector to rotate under the head
    - Depends on rotation rate of disk (increasing slowly over the years)
  - Transfer - transferring data from surface into disk controller electronics, sending it back to the host
    - Depends on density (increasing rapidly over the years)
- When accessing the disk, the OS and controller try to minimize the cost of all these steps

Disk Scheduling

- Just like for the CPU, one must schedule disk activities
- The OS receives I/O requests from processes, some for the disk
- These requests consist of
  - Input or output
  - A disk address
  - A memory address
  - The number of bytes (in fact sectors) to be transferred
- Given how slow the disk is and how fast processes are, it is common for the disk to be busy when a new request arrives
- The OS maintains a queue of pending disk requests
  - Processes are in the blocked state and placed in the device's queue maintained by the kernel
- After a request completes, a new request is chosen from the queue
- Question: which request should be chosen?

Seek Time

- Nowadays, the average seek time is in orders of milliseconds
  - Swinging the arm back and forth takes time
- This is an eternity from the CPU's perspective
  - 2 GHz CPU
  - 5ms seek time
  - 10 million cycles!

Credit: Alpha six

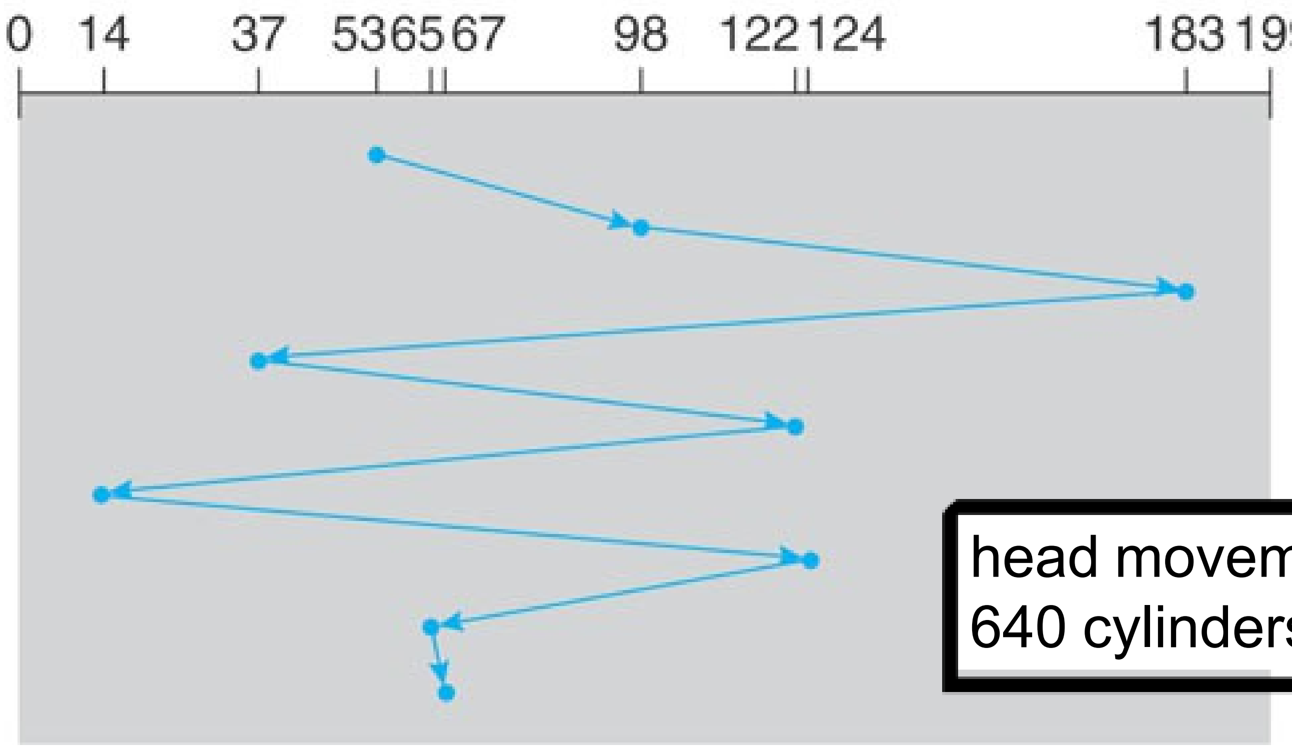
- A good goal is to minimize seek time
  - i.e., minimize arm motion
  - i.e., minimize the number of cylinders the head travels over

First Come First Serve (FCFS)

- FCFS: as usual, the simplest

queue = 98, 183, 37, 122, 14, 124, 65, 67 (cylinder #)

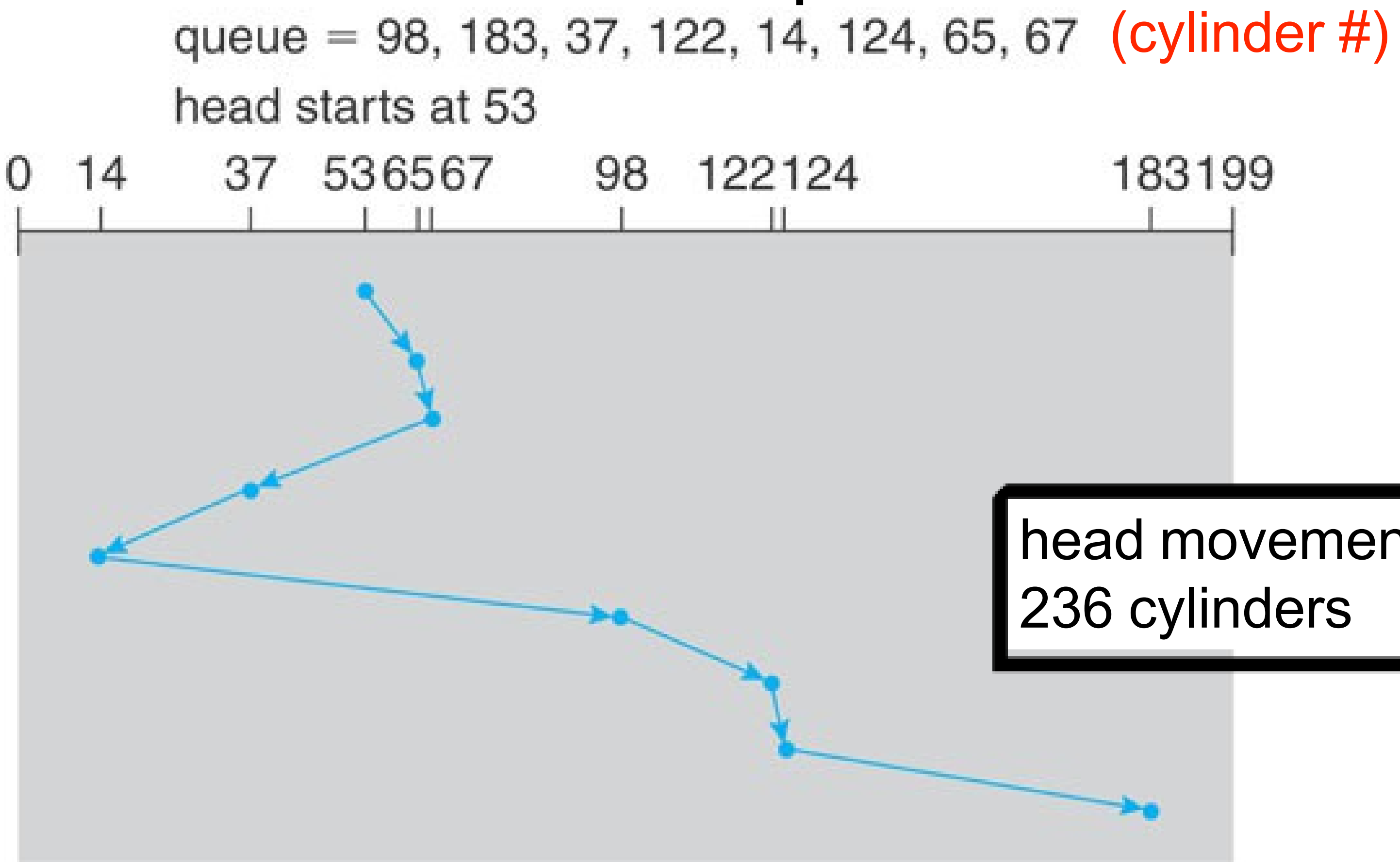
head starts at 53



head movement:
640 cylinders

Shortest Seek Time First (SSTF)

- SSTF: Select the request that's the closest to the current head position



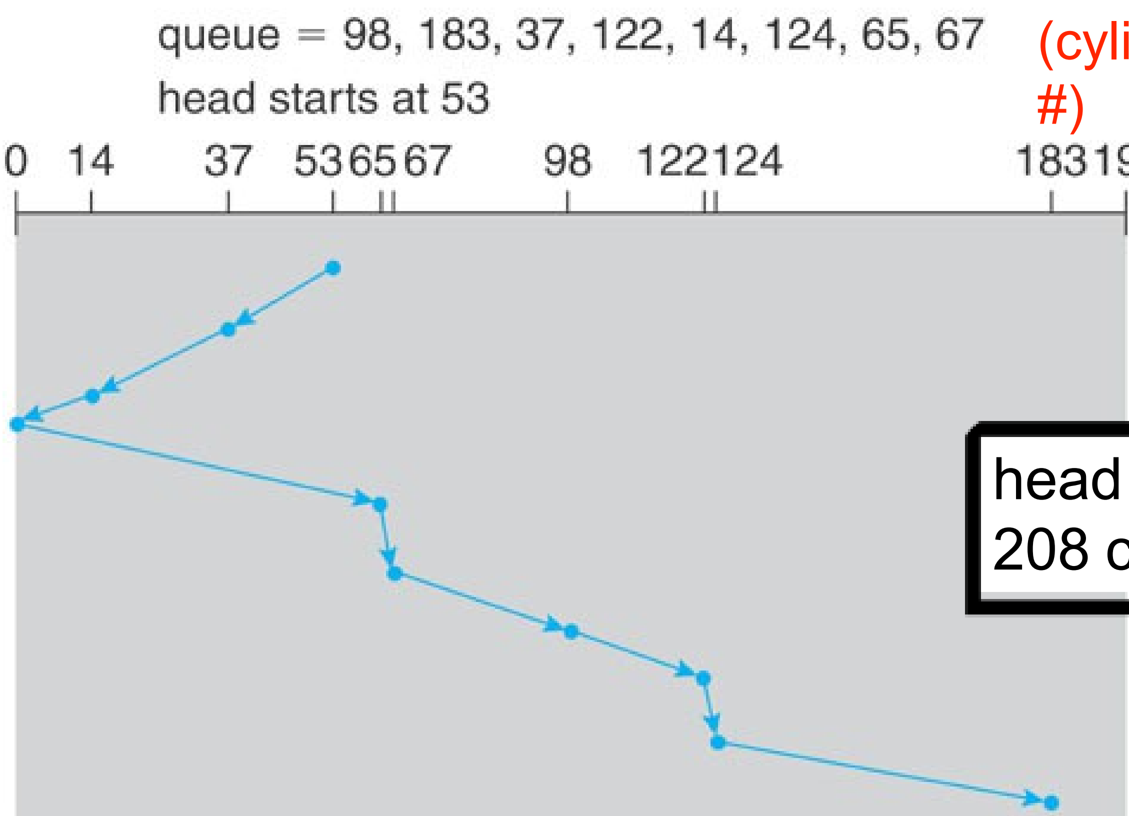
head movement: 236 cylinders

SSTF

- SSTF is basically SJF (Shortest job First), but for the disk
- Like SJF, it may cause starvation
  - If the head is at 80, and if there is a constant stream of requests for cylinders in [50,100], then a request for cylinder 200 will never be served
- Also, it is not optimal in terms of number of cylinders
  - On our example, it is possible to achieve as low as 208 head movements

SCAN Algorithm

- The head goes all the way up and down, just like an elevator
  - It serves requests as it reaches each cylinder



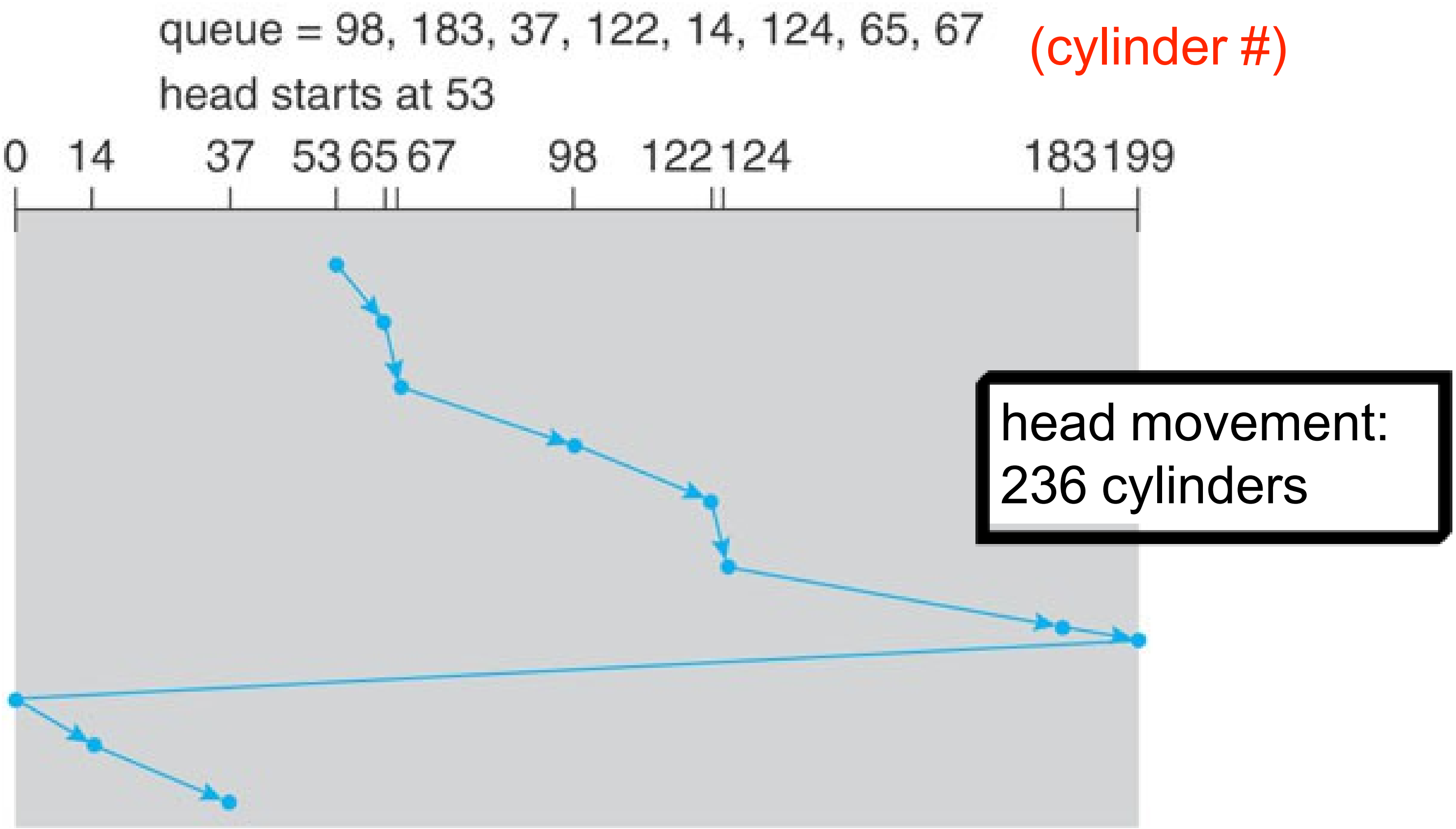
(cylinder #)

head movement:
208 cylinders

SCAN Algorithm

- There can be no starvation with SCAN
- Moving the head from one cylinder to the next takes little time and is better than swinging back and forth
- One small problem: After reaching one end, assuming requests are uniformly distributed, when the head reverses direction it will find very few requests initially
  - Because it just served them on the way up
  - Not quite like an elevator in this respect
- This leads to non-uniform wait times
  - Requests that just missed the head close to one end have to wait a long time
- Solution: C-SCAN
  - When the head reaches one end, it “jumps” to the other end instead of reversing direction
  - Just as if the cylinder were organized in a circular list

C-SCAN



queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53
(cylinder #)
0
14
37
53
65
67
98
122
124
183
199
head movement:
236 cylinders

Disk Scheduling Recap

- As usual, there is no "best" algorithm
  - Highly depends on the workload
- Do we care?
  - For home PCs, there aren't that many I/O requests, so probably not
  - For servers, disk scheduling is crucial
    - And SCAN-like algorithms are "it"
- Modern disks implement the disk scheduling themselves
  - SCAN, C-SCAN
  - Also because the OS can't do anything about rotation latency, while the disk controller can
    - It's not all about minimizing seek time
- However, the OS must still be involved
  - e.g., not all requests are created equal

Disk Reliability

- Disks are not reliable
  - MTTF (Mean Time To Failure) is not infinite
  - And failures can be catastrophic
- Yearly “Hard drive reliability” studies
- Google looked at over 100,000 disks in 2007 and looked at failure statistics

- Let’s look at one of their graphs

Disk Reliability

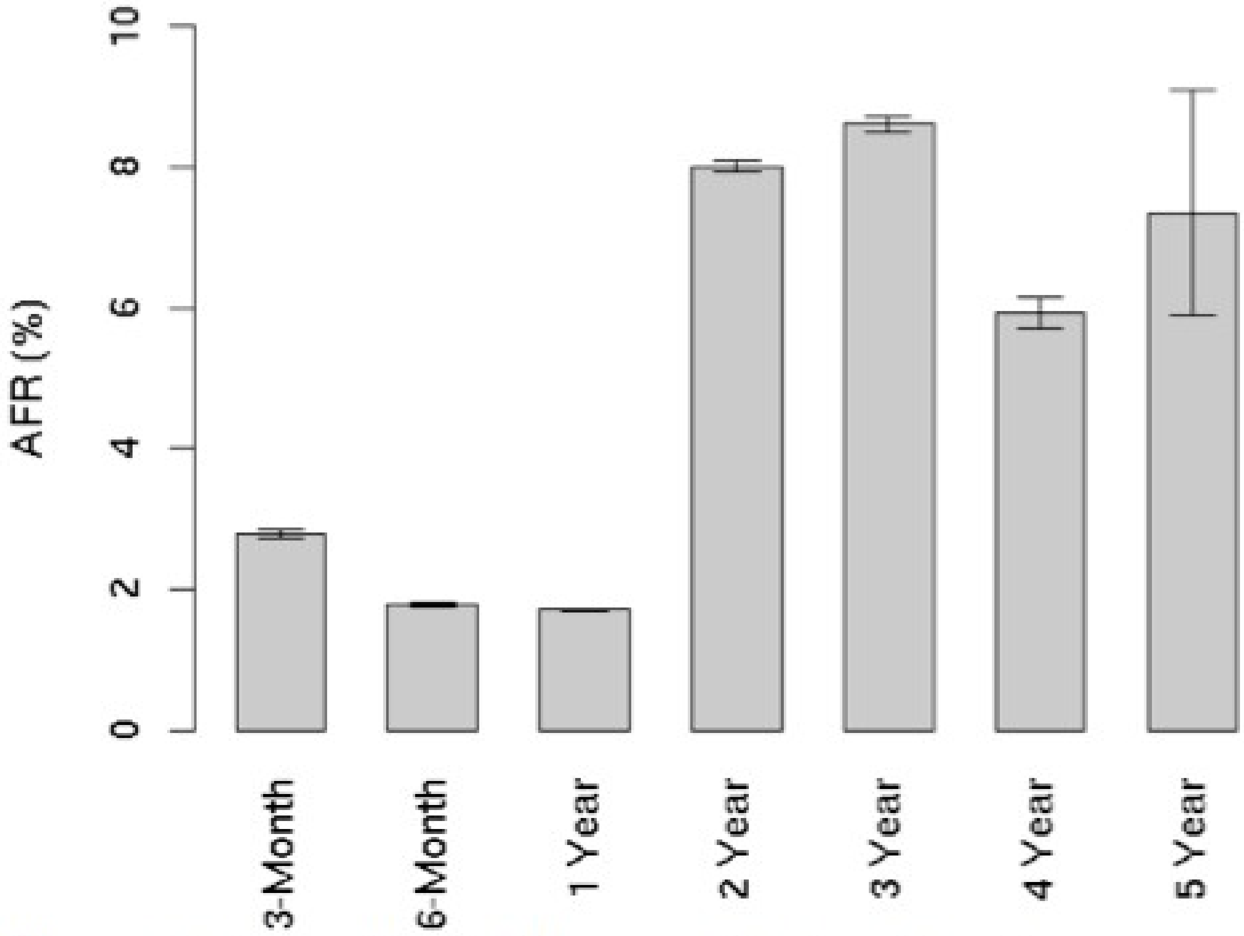
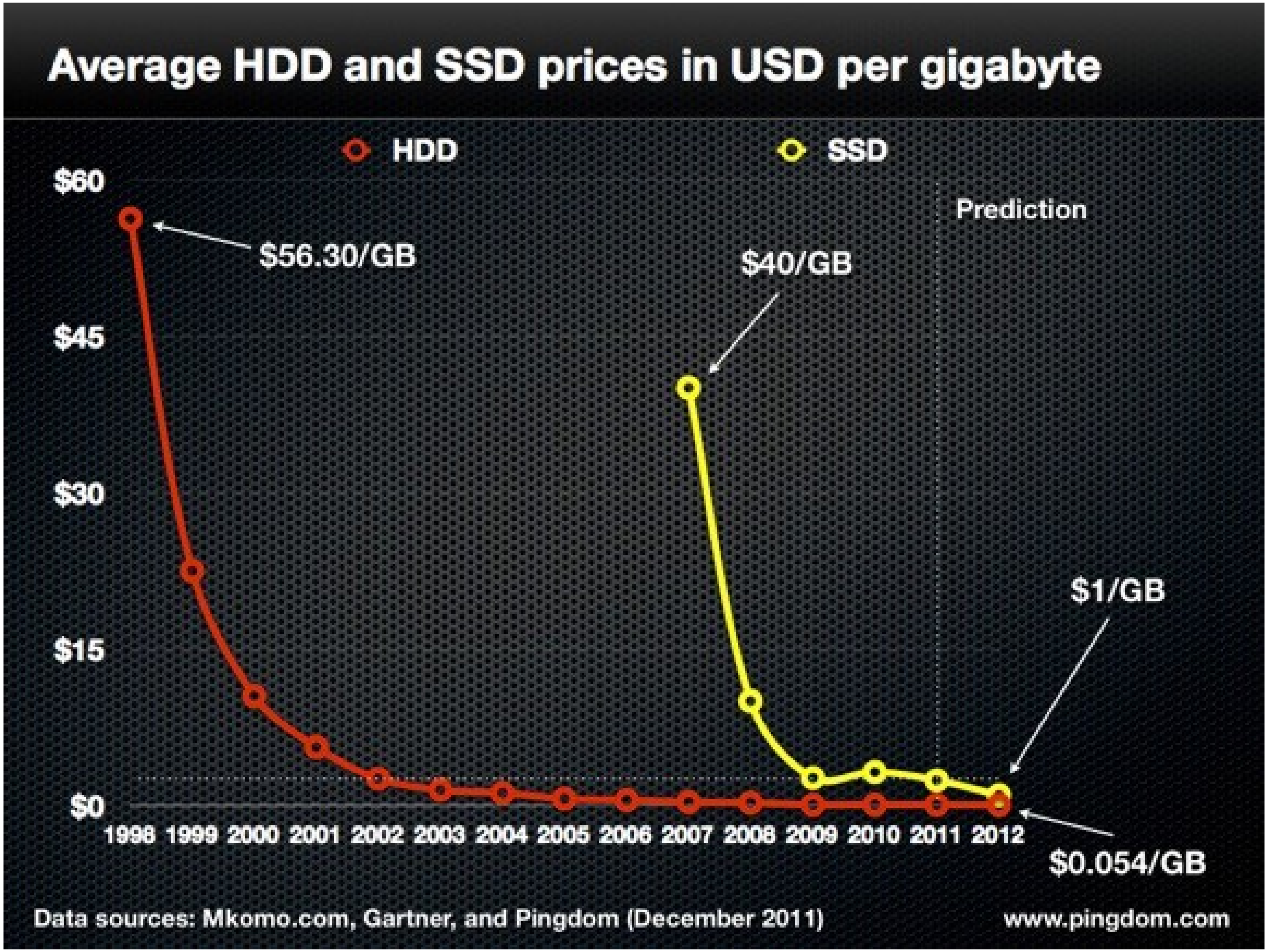


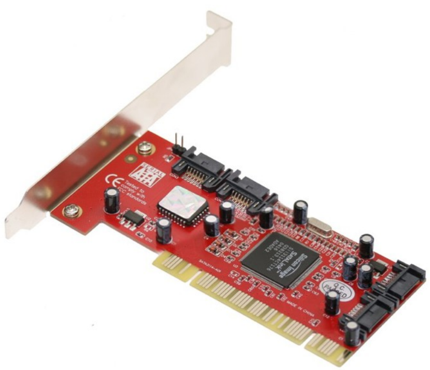
Figure 2: Annualized failure rates broken down by age groups

Disks are Cheap



RAID

- Disks are unreliable, slow, but cheap
- Simple idea: let's use redundancy
  - Increases reliability
    - If one fails, you have another one (increased perceived MTTF)
  - Increases speed
    - Aggregate disk bandwidth if data is split across disks
- Redundant Array of Independent Disks
  - The OS can implement it with multiple bus-attached disks
  - A RAID controller in hardware
  - A "RAID array" as a stand-alone box



RAID Techniques

- Data Mirroring
  - Keep the same data on multiple disks
    - Every write is to each mirror, which takes time
- Data Striping
  - Keep data split across multiple disks to allow parallel reads
    - e.g., read bits of a byte from 8 disks
- Error-Code Correcting (ECC) - Parity Bits
  - Keep information from which to reconstruct lost bits due to a drive failing
- These techniques are combined at will

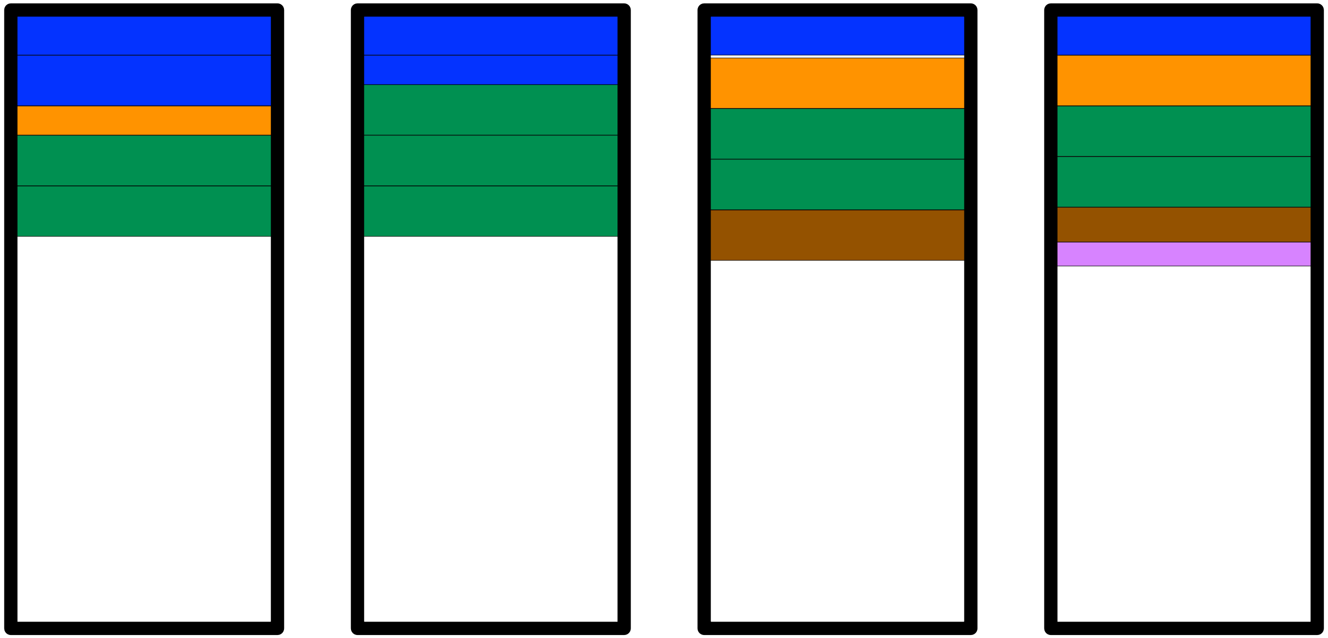
RAID Levels

- Combinations of the techniques are called “levels”
  - More of a marketing tool, really
- You should know about common RAID levels, i.e.: 0, 1, 1+0, 0+1, 5, 5+0, 6, 6+0
  - The book talks about all of them
    - but for level 2, which is not used

RAID 0

- Data is striped across multiple disks
  - Using a fixed strip size
- Gives the illusion of a larger disk with high bandwidth when reading/writing a file
  - Accessing a single strip is not any faster
- Improves performance, but not reliability
- Useful for high-performance applications

RAID 0 Example

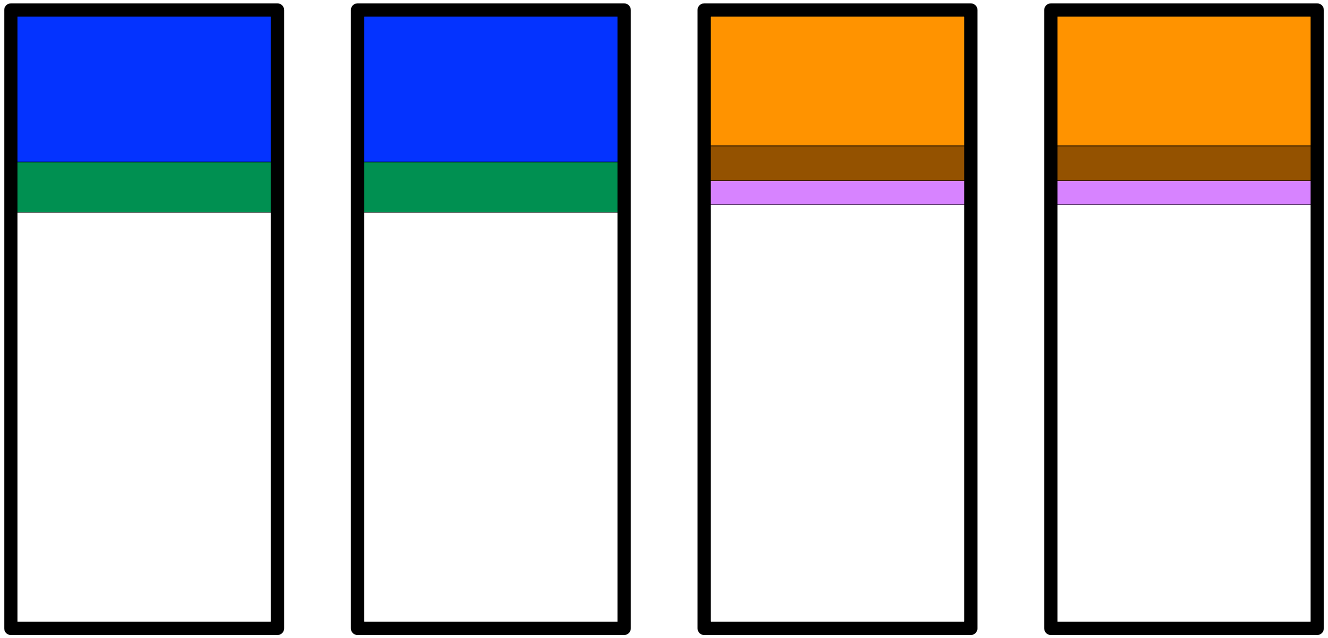


- Fixed strip size
- 5 files of various sizes
- 4 disks

RAID 1

- Mirroring (also called shadowing)
- Write every written byte to 2 disks
  - Uses twice as many disks as RAID 0
- Reliability is ensured unless you have (extremely unlikely) simultaneous failures
- Performance can be boosted by reading from the disk with the fastest seek time
  - The one with the arm the closest to the target cylinder

RAID 1 Example



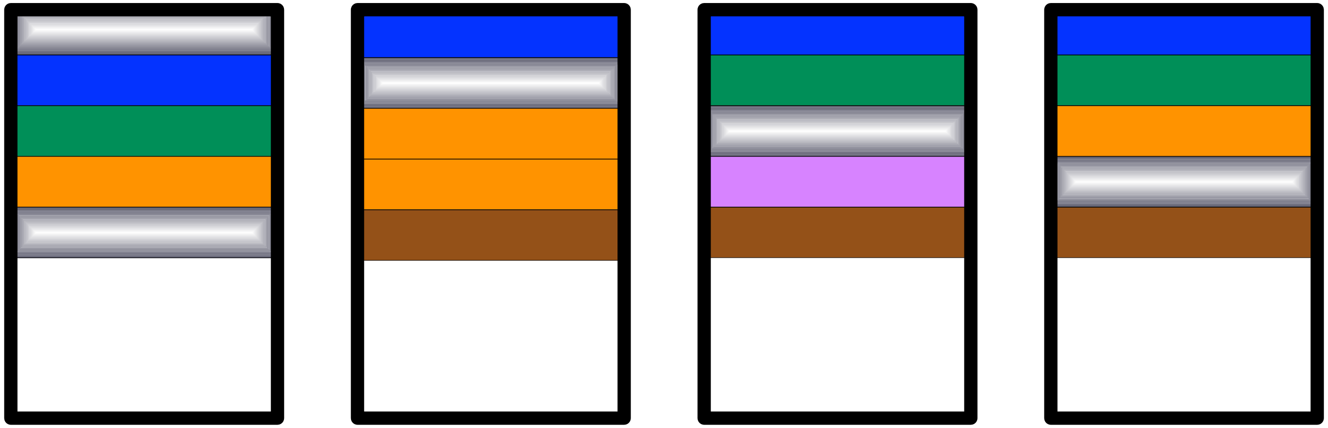
- 5 files of various sizes
- 4 disks

RAID 3

- Bit-interleaved parity
  - Each write goes to all disks, with each disk storing one bit
  - A parity bit is computed, stored, and used for data recovery
- Example with 4 disks an 1 parity disk
  - Say you store bits 0 1 1 0 on the 4 disks
  - The parity bit stores the XOR of those bits: (((0 xor 1) xor 1) xor 0) = **0**
  - Say you lose one bit: 0 ? 1 0
  - You can XOR the remaining bits with the parity bit to recover the lost bit: (((0 xor **0**) xor 1) xor 0) = 1
  - Say you lose a different bit: 0 1 1 ?
  - The XOR still works: (((0 xor 1) xor 1) xor **0**) = 0
- Bit-level striping increases performance
- XOR overhead for each write (done in hardware)
- Time to recovery is long (a bunch of XOR's)

RAID 4 and 5

- RAID 4: Basically like RAID 3, but interleaving it with strips
  - A (small) read involves only one disk
- RAID 5: Like RAID 4, but parity is spread all over the disks as opposed to having just one parity disk, as shown below



- RAID 6: like RAID 5, but allows simultaneous failures (rarely used)

OS Disk Management

- The OS is responsible for
  - Formatting the disk
  - Booting from disk
  - Bad-block recovery

Physical Disk Formatting

- Divides the disk into sectors
- Fills the disk with a special data structure for each sector
  - A header, a data area (512 bytes), and a trailer
- In the header and trailer is the sector number, and extra bits for error-correcting code (ECC)
  - The ECC data is updated by the disk controller on each write and checked on each read
  - If only a few bits of data have been corrupted, the controller can use the ECC to fix those bits
  - Otherwise the sector is now known as “bad”, which is reported to the OS
- Typically all done at the factory before shipping

Logical Formatting

- The OS first partitions the disk into one or more groups of cylinders: the partitions
- The OS then treats each partition as a separate disk
- Then, file system information is written to the partitions
  - See the File System lecture

Boot Blocks

- Remember the boot process from a previous lecture
  - There is a small ROM-stored bootstrap program
  - This program reads and loads a full bootstrap stored on disk
- The full bootstrap is stored in the boot blocks at a fixed location on a boot disk/partition
  - The so-called master boot record
- This program then loads the OS

Bad Blocks

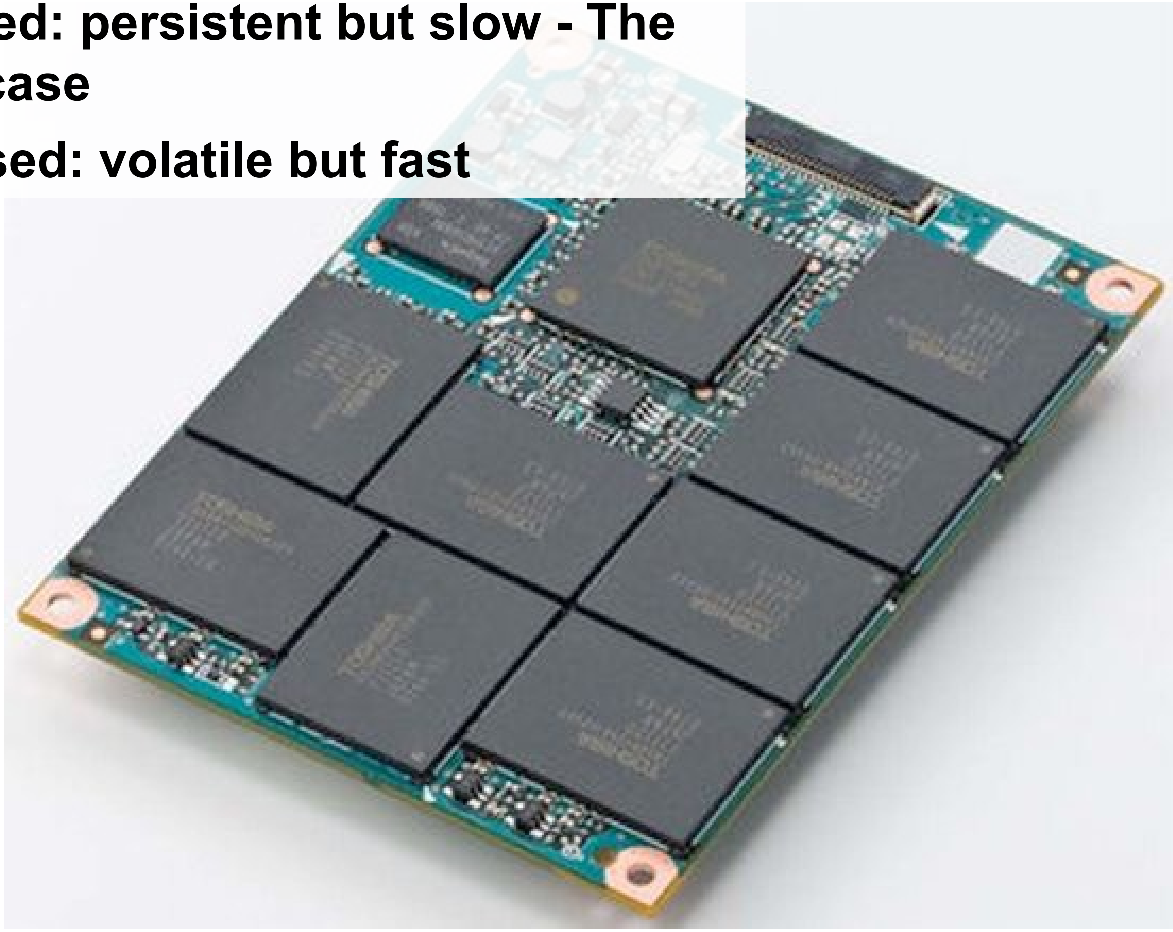
- Sometimes, data on the disk is corrupted and the ECC can't fix it
- Errors occur due to
  - Damage to the platter's surface
  - Defect in the magnetic medium due to wear
  - Temporary mechanical error (e.g., head touching the platter)
  - Temporary thermal fluctuation
- The OS gets a notification

Bad Blocks

- Upon reboot, the disk controller can be told to replace a bad block by a spare: sector sparing
  - Each time the OS asks for the bad block, it is given the spare instead
  - The controller maintains an entire block map
- Problem: the OS's view of disk locality may be very different from the physical locality
- Solution #1: Spares in each cylinders and a spare cylinder
  - Always try to find spares "close" to the bad block
- Solution #2: Shuffle sectors to bring the spare next to the bad block
  - Called sector splitting

Solid-State Drives (SSDs)

- Purely based on solid-state memory
  - Flash-based: persistent but slow - The common case
  - DRAM-based: volatile but fast

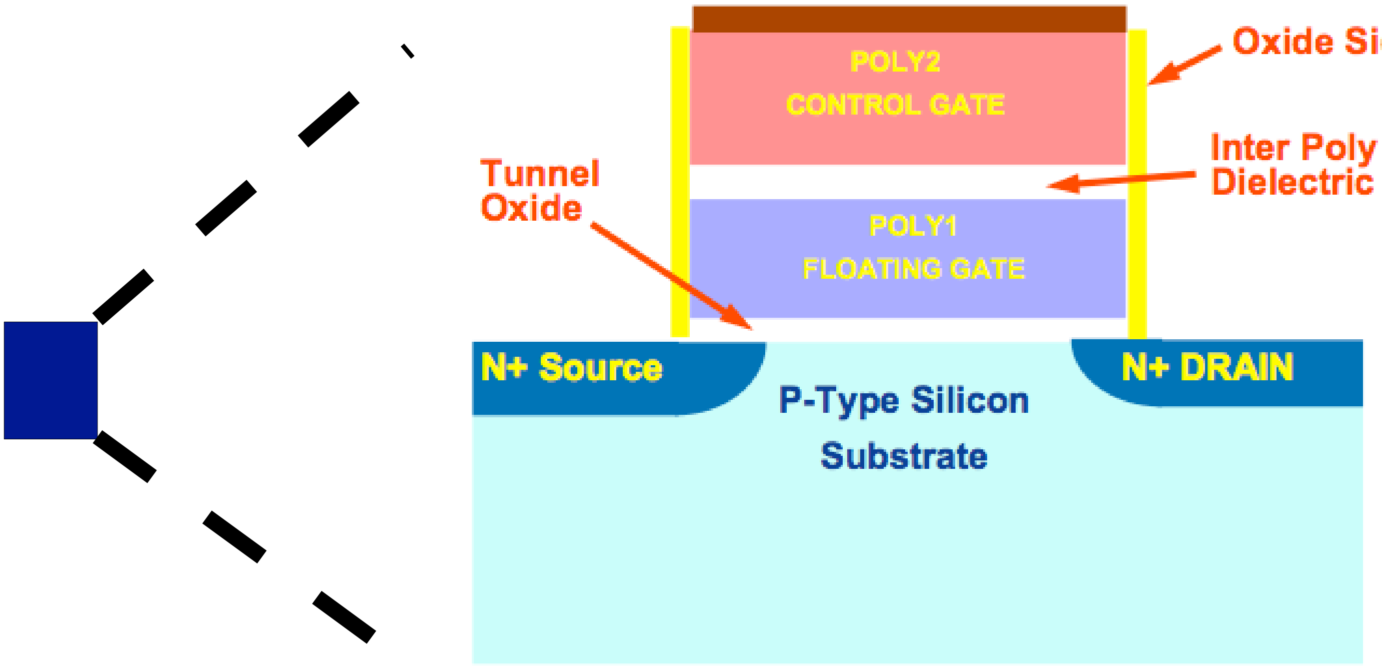


SSDs

- No moving parts!
- Flash SSDs competitive vs. hard drives
  - faster startups and reads
  - silent, low-heat, low-power
  - more reliable
  - less heavy
  - getting larger and cheaper, close to HDD
  - lower lifetime due to write wear off
    - Used to be a big deal, but now ok especially for personal computers
  - slower writes (????)
- SSDs are becoming more and more mainstream
- The death of HDD is not for tomorrow, but looks much closer than 5 years ago...

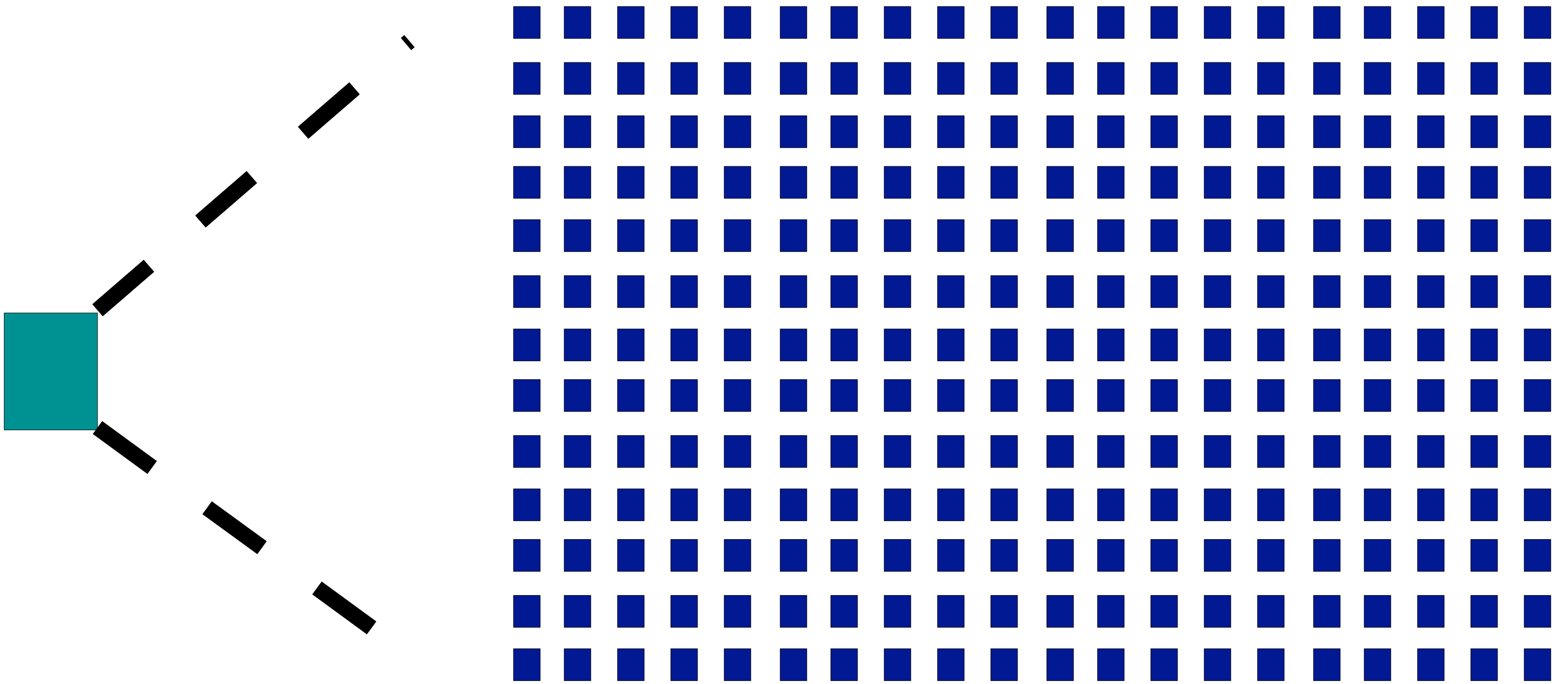
SSD Structure

- The flash cell



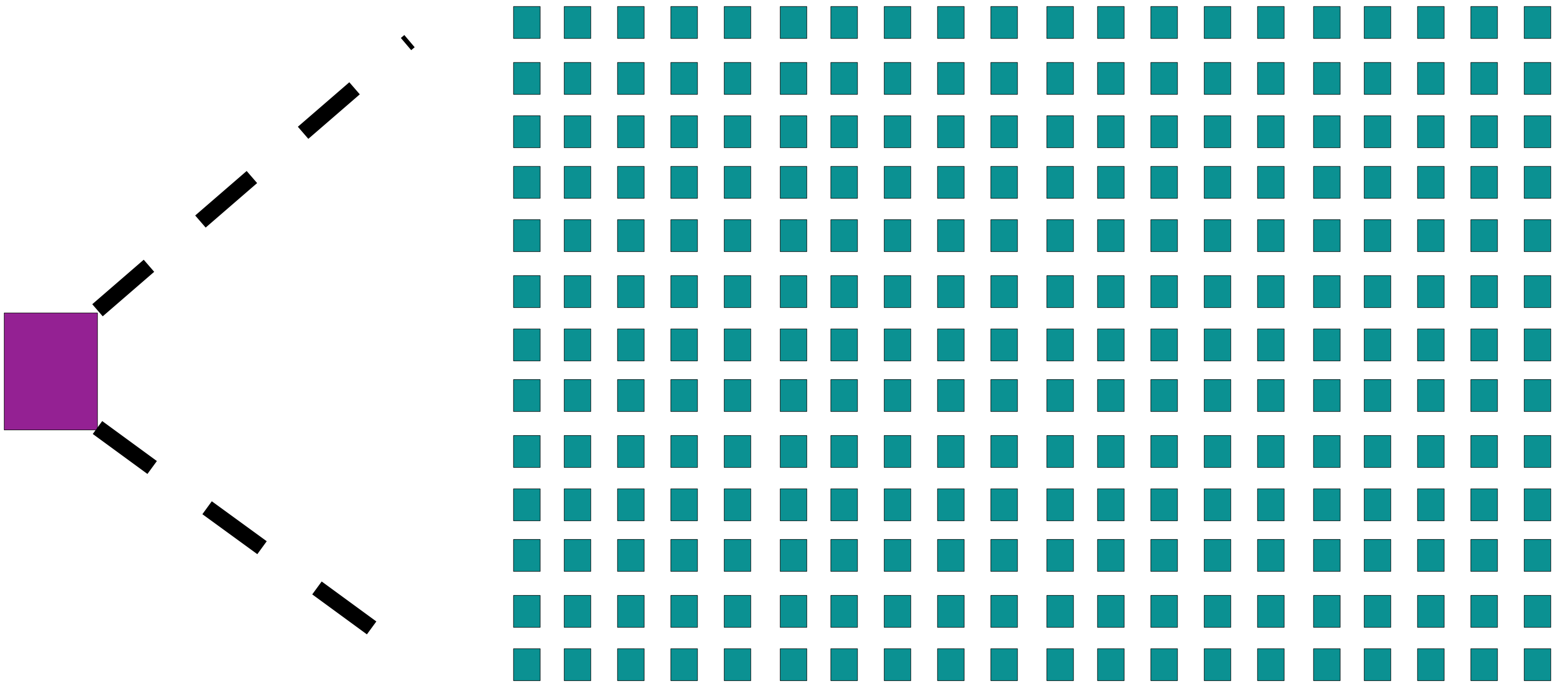
SSD Structure

- The page (4KB)



SSD Structure

- The block: 128 pages (512KB)



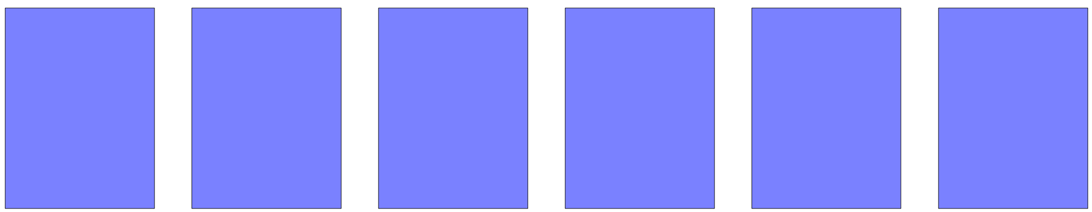
Why Slow Writes?

- Major concern: Before being written a page must be erased... but only blocks can be erased.

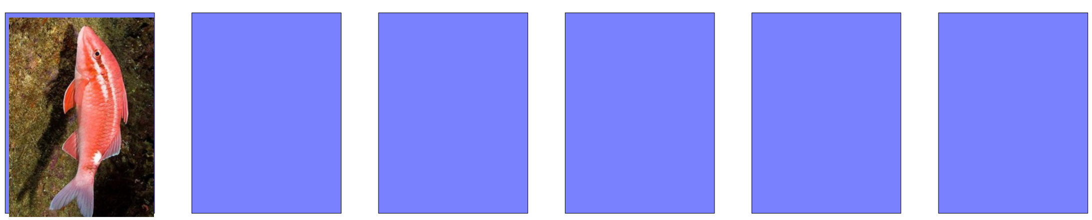
  Therefore valid pages must be read before being erased and rewritten...
- SSD writes are/were considered slow because of write amplification: as time goes on, a write x bytes of data in fact entails writing y>x bytes of data!!
- Reason:
  - The smallest unit that can be read: a 4KB page
  - The smallest unit that can be erased: a 512KB block
- Let's look at this on an example

Write Amplification

- Let’s say we have a 6-page block

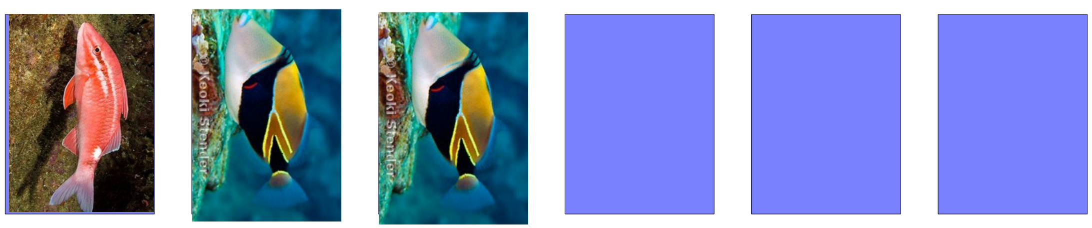


- Let’s write a 4KB file

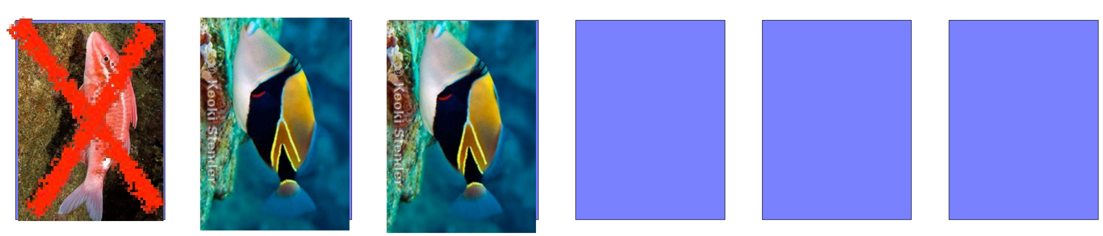


Write Amplification

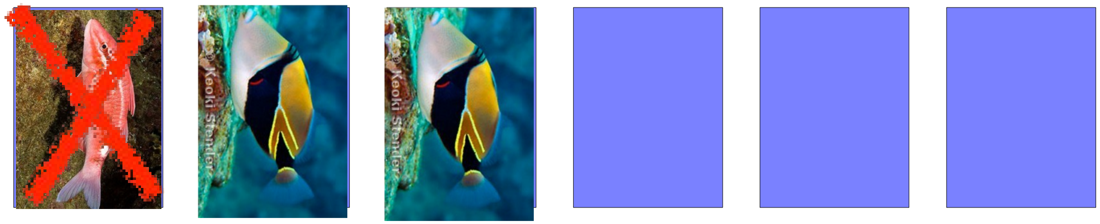
- Let’s write a 8KB file



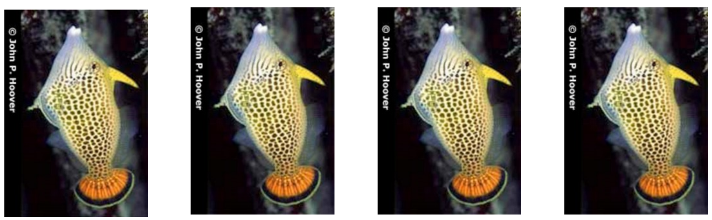
- Let’s “erase” the first file
  - We can’t erase the file without erasing the block, so we just mark it as invalid



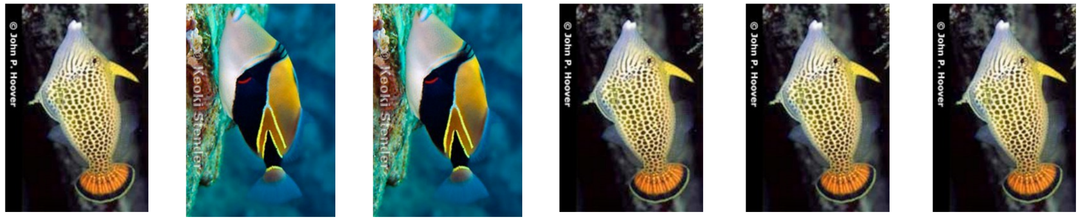
Write Amplification



- Let's write a 16KB file



- We have to
  - load the whole block into RAM (or controller cache)
  - Modify the in-memory block
  - Write back the whole block



Write Amplification

- To write 4KB + 8KB + 16KB = 28KB of application data, we had to write 4KB + 8KB + 24KB = 36KB of data to the SSD
- As the drive fills up and files get written / modified / deleted, writes end up amplified
- The controller keeps writing on the SSD until full, before it attempts any rewrite
- In the end, performance is still good relative to that of an HDD
- The OS can, in the background, clean up block with invalid pages so that they're easily writable when needed

SSDs vs. HDDs

- SSDs have many advantages of HDDs
  - Random read latency much smaller
  - SSDs are great at parallel read/write
  - SSDs are great at small writes
  - SSDs are great for random access in general
    - Which is typically the bane of HDDs

- Note that not all SSDs are made equal
  - Constant innovations/improvements

Conclusion

- HDDs are slow, large, unreliable, and cheap
- Disk scheduling by the OS/controller tries to help with performance
  - i.e., reduce seek time
- Redundancy is a way to cope with slow and unreliable HDDS
- SSDs provide a radically novel approach that may very well replace HDDs in the future
  - The two are likely to coexist for years to come
- The OS is involved in disk management functions, but with a lot of help from the drive controllers