



slides kindly provided by:

Tam Vu, Ph.D
Professor of Computer Science
Director, Mobile & Networked Systems Lab
Department of Computer Science
University of Colorado Boulder

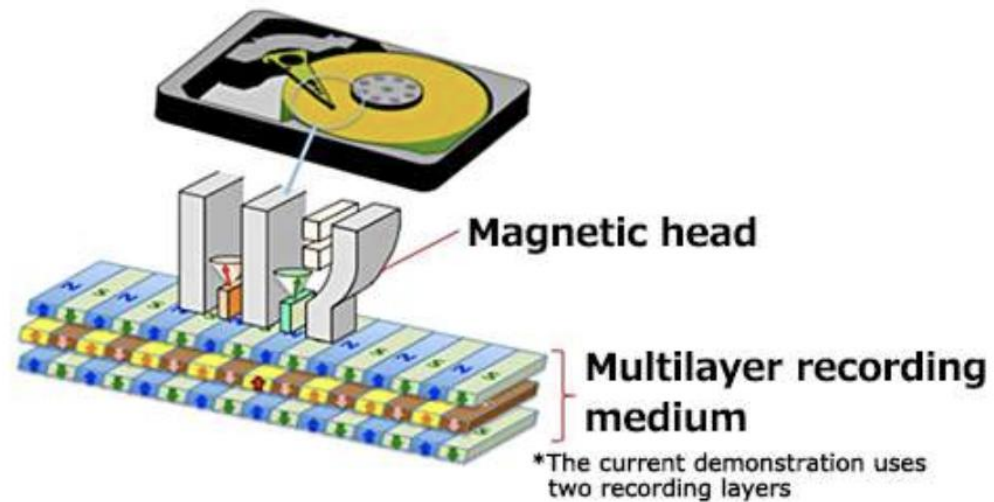
CSCI 3753 Operating Systems Summer 2020

Christopher Godley
PhD Student
Department of Computer Science
University of Colorado Boulder



Storage Management

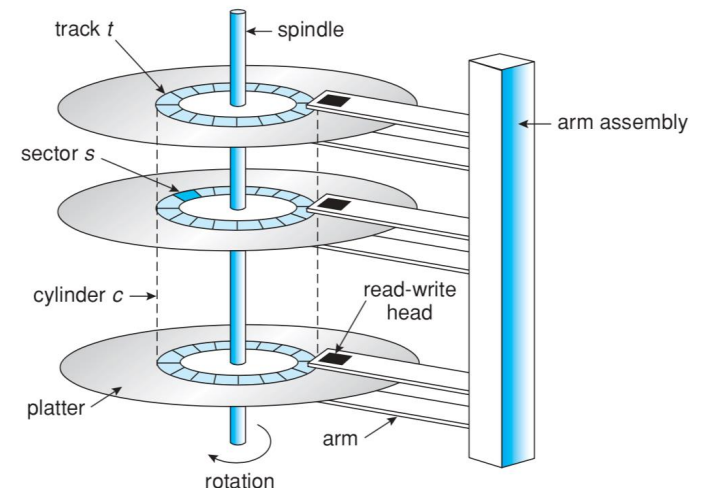
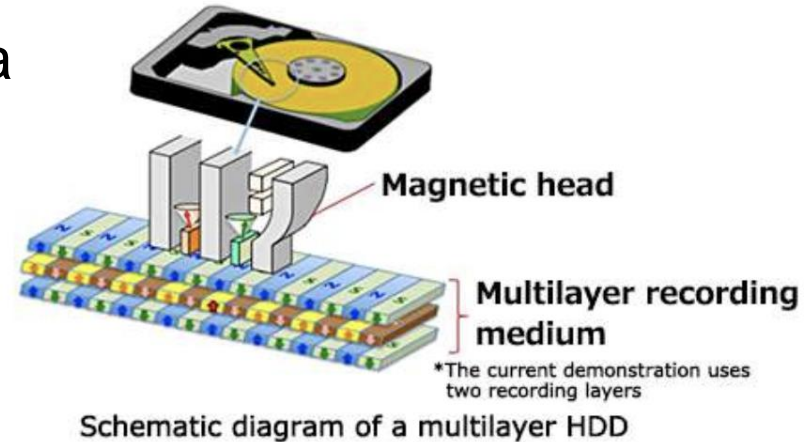
Magnetic Media



Schematic diagram of a multilayer HDD

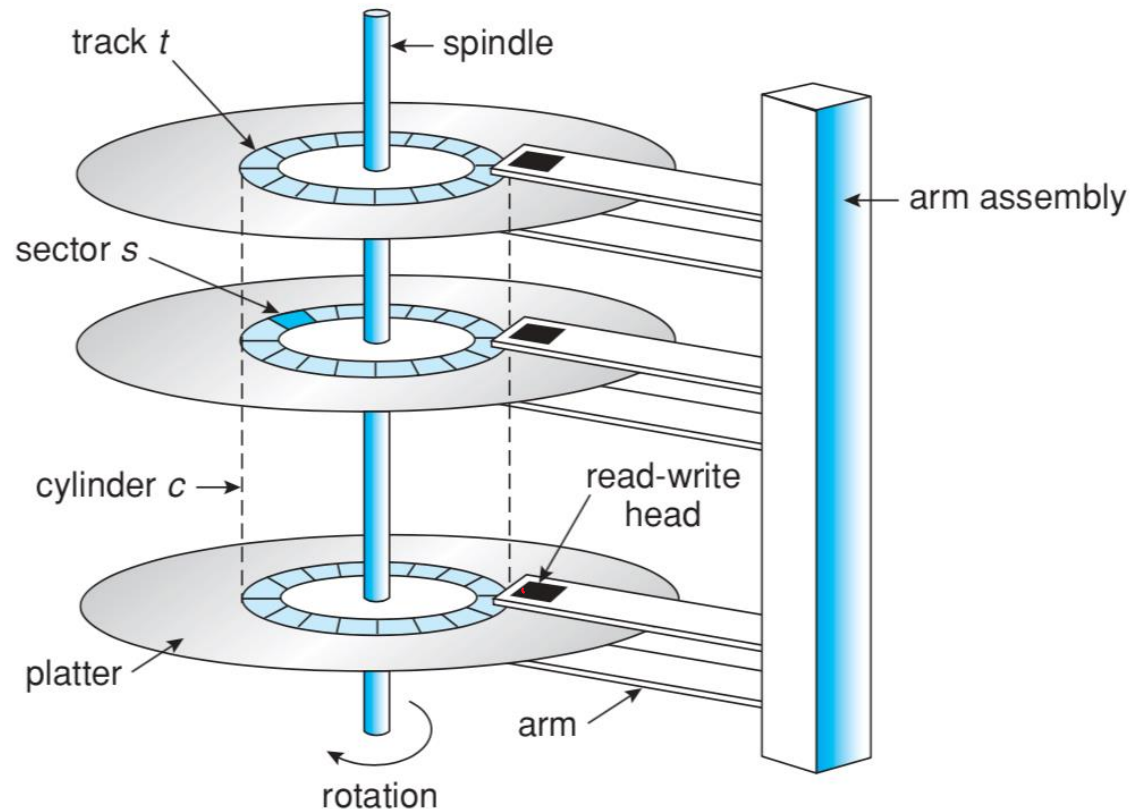
Magnetic Media

- Can set a bit to 1 or 0 by applying a strong enough magnetic field to a small region of a magnetic film
 - region retains its magnetic sense even when power is removed
- To read a sequence of bits,
 - Either move the magnetic head over the magnetic media,
 - Moving (or rotating) the media under the (static) magnetic head – this is the easier approach
- Supports both random and sequential access
- Array of magnetic disk platters increases bit storage at marginal cost in space and new disk heads



Magnetic Disk

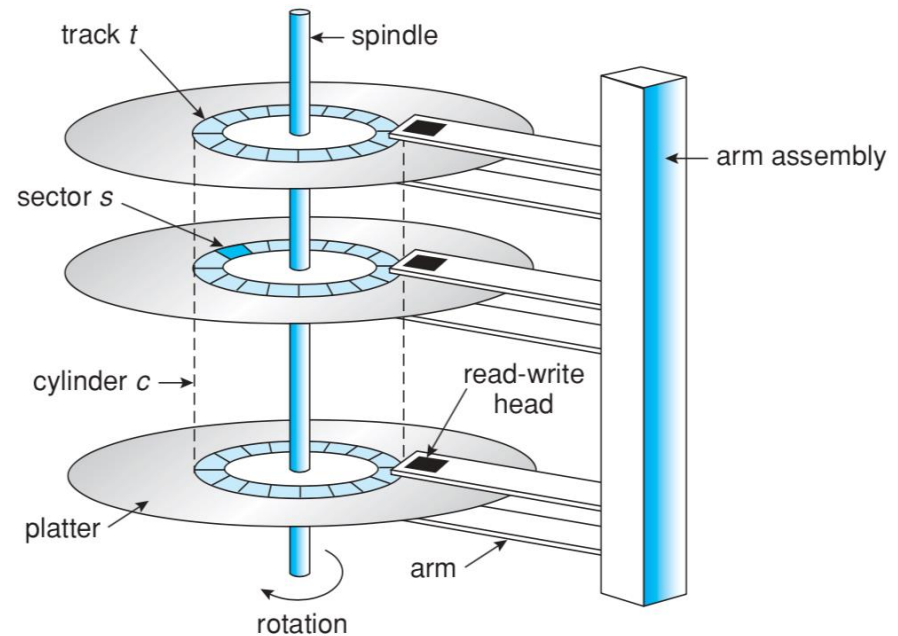
- Platter
- R/W head
- Arm
- Track
- Sector
- Cylinder



Magnetic Disk

Time to access data

- Move head into position to read correct track (Seek time)
- Wait for sector to rotate under head (rotational latency)
- Read data from sector on platter (transfer into memory)



Disk Access Latency

total delay to
read/write
from/to disk

=

seek time

+

rotational
latency

+

transfer time

typically 5-10 ms

typically 1-5 ms,
typical RPM is ~10000 revolutions
per minute, so if on average it takes
half a revolution to rotate to the right
sector, then $0.5 / (10000 / 60) \approx$ 3 ms

typically in 10-100 μ s,
typical 1 Gb/s data
transfer rate, so $\rightarrow (80 \times 10^3) \text{ Kb}$
retrieving 10 KB of
file data = $80000 / (1 \text{ Gb})$
= .08 ms

- Total disk access delay is often dominated by seek times and rotational latency



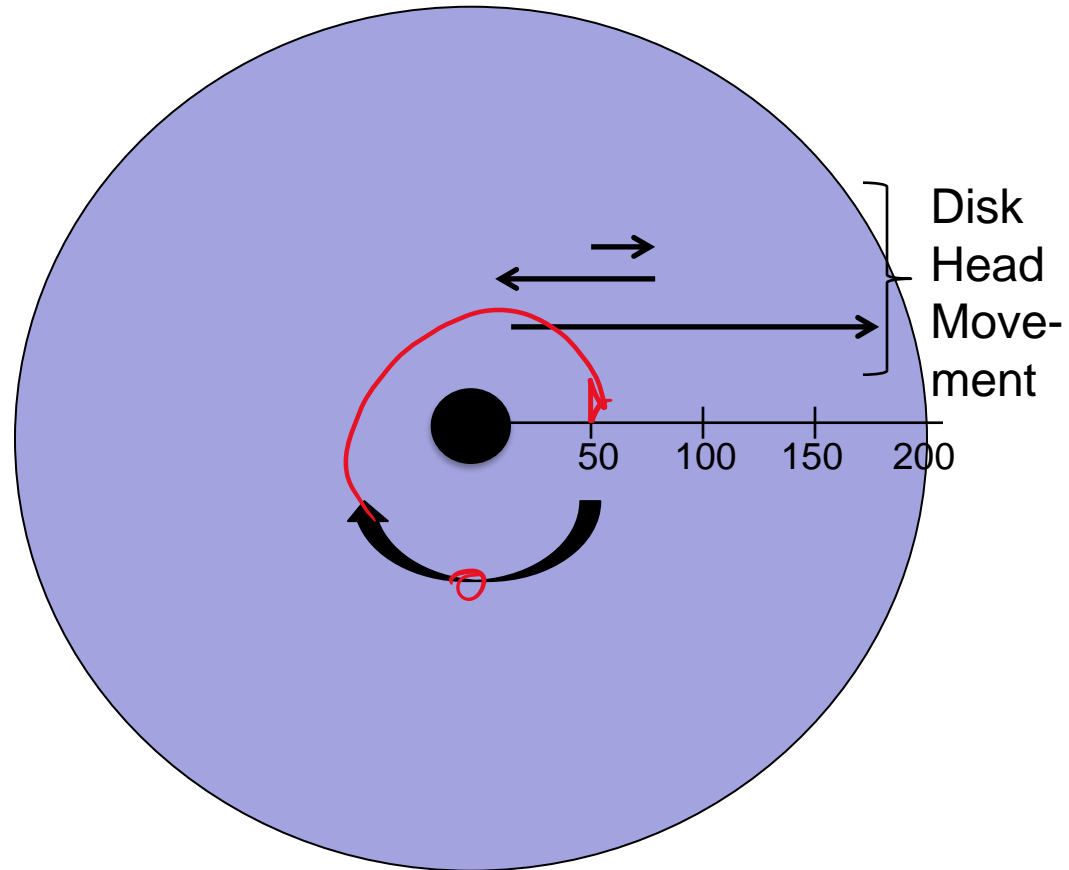
Disk Scheduling

Disk Scheduling

- Any technique that can **reduce seek times and rotational latency** is a big win in terms of improving disk access times
- Disk scheduling is designed to reduce seek times and rotational latency
- Disk operations take time to be processed
 - At any given time there will be multiple requests to access data from different places on the disk
 - These requests are stored in the queue
- The OS can choose to intelligently serve or schedule these requests so as to minimize latency
- **Our goal: find an algorithm that minimizes seek time ...**

Disk Scheduling

Rotating Disk



Disk Scheduling

- **Our goal: find an algorithm that minimizes seek time ...**
 - suppose we are given a series of disk I/O requests for different disk blocks that reside on the following different cylinder/tracks in the following order:
 - 98, 183, 37, 122, 14, 124, 65, 67
(Initial track location is 53)
 - The **total number of tracks traversed** is used as an evaluation metric of the algorithms.

Disk Scheduling

- FCFS Disk Scheduling


- Requests would be scheduled in the same order as the order of arrival
- Total number of cylinders/tracks traversed by the disk head under FCFS algorithm would be

$$|53-98| + |98-183| + |183-37| + |37-122| + \dots = 640 \text{ cylinders}$$


- Observation:

- disk R/W head would move less if 37 and 14 could be serviced together,
- similarly for 122 and 124
- easy to implement – but slow

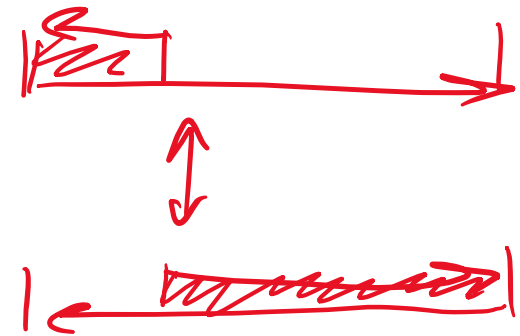
98, 183, 37, 122, 14, 124, 65, 67
(Initial track location is 53)



Disk Scheduling

- SSTF Disk Scheduling
 - Shortest-Seek-Time-First (SSTF) scheduling selects the next request with the minimum seek time from the current R/W head position
 - SSTF services requests in this order
 - 65, 67, 37 (closer than 98), 14, 98, 122, 124, 183 =>> ???
 - Locally optimal, but why not globally optimal?
 - Better to move disk head from 53 to 37 then 14 then 65 (this would result in ??? cylinders total)
 - Another drawback: starvation
 - While the disk head is positioned at 37, a series of new nearby requests may come into queue for 39, 35, 38, ...
98, 183, 37, 122, 14, 124, 65, 67
(Initial track location is 53)

Disk Scheduling



- OPT Disk Scheduling

- OPT looks to minimize the back and forth traversing
- Move the disk head from the current position to the closest edge track, and **sweep through** once (either innermost to outermost, or outermost to innermost)
- This should minimize the overlap of back and forth and give the minimum # of tracks covered
- OPT services requests as follows:
 - 53 → 37 → 14 → 65 → 67 → 98 → 122 → 124 → 183
- OPT ignores the initial direction and changes direction
 - total # cylinders traversed = 39 down + 169 up = 208 cylinders
- **The problem with OPT is that:**
 - you have to recalculate every time there's a new request for a disk track
- Best approximation to OPT is probably to just scan the disk in one direction, and then change the direction of scanning

98, 183, 37, 122, 14, 124, 65, 67
(Initial track location is 53)

Disk Scheduling

Scheduling Algorithm	# Cylinders
OPT	208
FCFS	640
SSTF	236
SCAN	333

- SCAN Disk Scheduling

- Move the read/write head in one direction from innermost to outermost track

- Then reverse direction

- Sweeping across the disk in alternate directions

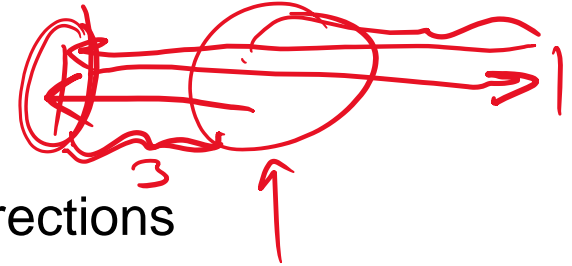
- SCAN services requests as follows:

- 53→65→67→98→122→124→183→200→37→14

- Advantages:

- Approximates OPT
 - Starvation free solution
 - handles dynamic arrival of new requests
 - simple to implement

98, 183, 37, 122, 14, 124, 65, 67
(Initial track location is 53)



Disk Scheduling

- SCAN Disk Scheduling

- Disadvantages:

- Problem 1: SCAN only approximates OPT
 - There is significantly more overlap compared to the OPT service sequence
 - Problem 2: unnecessarily goes to extreme edges of disk even if no requests there
 - Problem 3: unfair for tracks on the edges of the disk (both inside and outside)
 - Writes to the edge tracks take the longest since after SCAN has reversed directions at one edge, the writes on the other edge always wait the longest to be served
 - After two full scans back and forth, middle tracks get serviced twice, edge tracks only once

98, 183, 37, 122, 14, 124, 65, 67
(Initial track location is 53)

Disk Scheduling

Scheduling Algorithm	# Cylinders
OPT	208
FCFS	640
SSTF	236
SCAN	333
C-SCAN	384

- C-SCAN Disk Scheduling
 - Treat disk as a queue of tracks
 - When reaching one edge, move the R/W head all the way back to the other edge
 - Continue scanning in the same direction rather than reversing direction
 - Circular SCAN, or C-SCAN
 - After reaching an edge, requests at the opposite edge get served first
 - This is more fair in service times for all tracks
 - C-SCAN services requests as follows:
 - 53→65→67→98→122→124→183→200→1→14→37
 - For a total distance of 384 tracks
 - Note how no scanning is done as the disk head is repositioned from 200 all the way down to 1 to ensure circularity
 - 98, 183, 37, 122, 14, 124, 65, 67
(Initial track location is 53)

Disk Scheduling

Scheduling Algorithm	# Cylinders
OPT	208
FCFS	640
SSTF	236
SCAN	333
C-SCAN	384
LOOK	299

- LOOK Disk Scheduling
 - don't travel to the extreme edge if no requests there
 - look at the furthest request in the current direction
 - only move the disk head that far
 - then reverse direction
 - No starvation in this algorithm either
 - LOOK would service requests in the following order:
 - 53→65→67→98→122→124→183→37→14
 - total # cylinders traversed = 130 up + 169 down = 299 cylinders

98, 183, 37, 122, 14, 124, 65, 67
(Initial track location is 53)



Disk Scheduling

Scheduling Algorithm	# Cylinders
OPT	208
FCFS	640
<u>SSTF</u>	236
SCAN	333
C-SCAN	384
<u>LOOK</u>	299
C-LOOK	322

- C-LOOK Disk Scheduling
 - Improve LOOK by making it more fair
 - Circular LOOK
 - After moving disk head to furthest request in current direction
 - Move disk head directly all the way back to the further request in the other direction
 - Continue scanning in the same direction
 - C-LOOK would service requests in the following order:
 - 53→65→67→98→122→124→183→14→37
 - total # cylinders traversed = 130 up + 169 down + 23 up = 322 cylinders

98, 183, 37, 122, 14, 124, 65, 67
(Initial track location is 53)

Disk Scheduling

- Disk Scheduling in Practice
 - In the previous examples, the # of cylinders traversed by SCAN, LOOK, and C-LOOK are much lower if the initial direction was down
 - We can't keep changing the direction
 - Can cause starvation/unfairness.
 - Preserving directionality of the scan is important for fairness, but we sacrifice some optimality/performance
 - Either SSTF or LOOK are reasonable choices as default disk scheduling algorithms 
 - These algorithms for reducing seek times are typically embedded in the disk controller today
 - There are other algorithms for reducing rotational latency, also embedded in the disk controller 

Disk Scheduling vs. File Layout

- File Layout and Disk Scheduling
 - Block allocation schemes can effect access times
 - When a file's data is spread across the disk, it increasing seek times.
 - Contiguous allocation minimizes seek times
 - Select free blocks near adjacent file data
 - Opening a file for the first time requires searching the disk for the directory, file description header, and file data
 - These are all spread out across the disk
 - Seek times are large
 - Could store directories in the middle tracks, so at most half the disk is traversed to find the file header and data
 - Could store the file header near the file data, or near the directory



Flash Memory



What is Flash Memory?

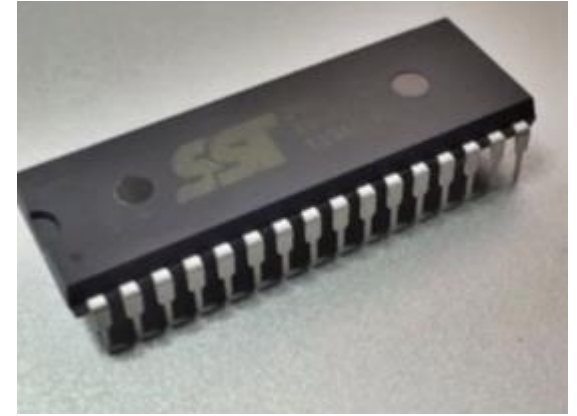
- Flash memory is a type of solid state storage
 - bits are stored in an electronic chip
 - not on a magnetic disk
 - Programmed and erased electrically
 - no mechanical parts
 - Non-volatile storage
 - i.e. “permanent” with caveats
- Primarily used in solid state drives (SSDs), memory cards, USB flash drives, mobile smartphones, digital cameras, etc.



What is Flash Memory?

- Flash memory is a form of
→ EEPROM

- Electrically Erasable Programmable Read Only Memory
- bits can be rewritten again and again using ordinary electrical signals
- non-volatile



- Compared to:

RAM (Random Access Memory)
– main memory, volatile

ROM (Read Only Memory) – code is burned in at the factory and can only be read thereafter, no writes

PROM (Programmable ROM) – code can be written once using a special machine

EPROM (Erasable PROM) – code can be written multiple times with a special machine

What is Flash Memory?

- Advantages vs. disk:
 - much faster access (lower latency),
 - more resistant to kinetic shock (& intense pressure, water immersion etc.),
 - more compact,
 - lighter,
 - lower power (typically 1/3-1/2 of disks), ...

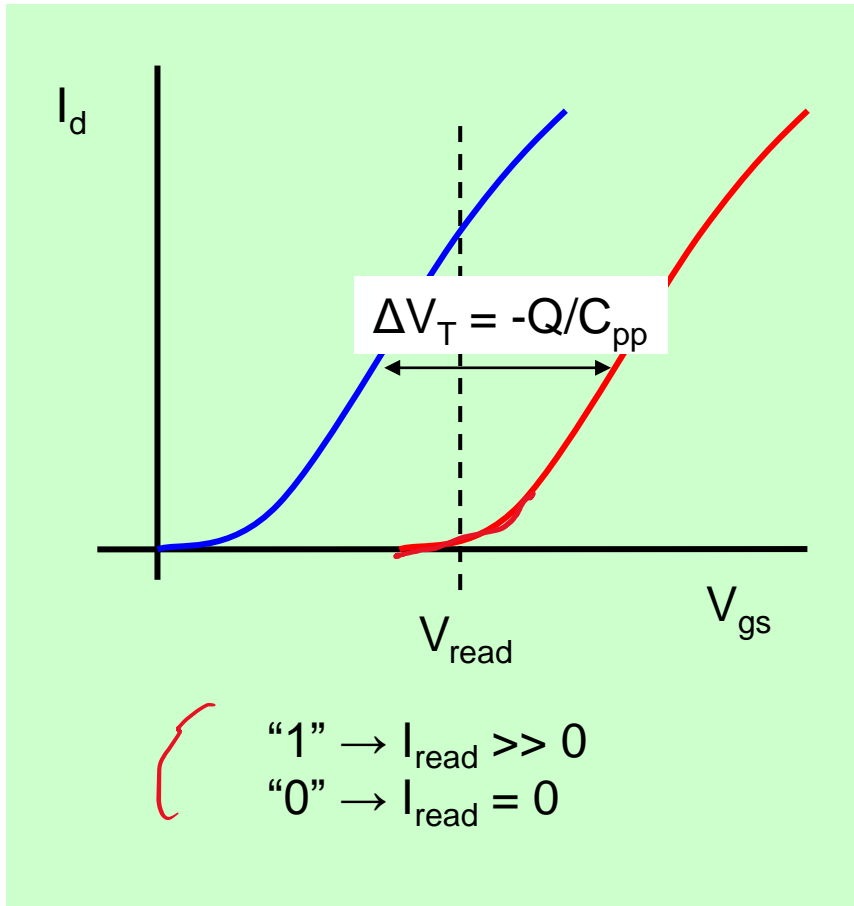


What is Flash Memory?

- Disadvantages vs. disk:
 - more costly per byte,
 - limited lifetime,
 - erases are costly, ...
- Flash's name:
 - can inexpensively erase a large amount of solid state memory quickly, like a “flash” of light



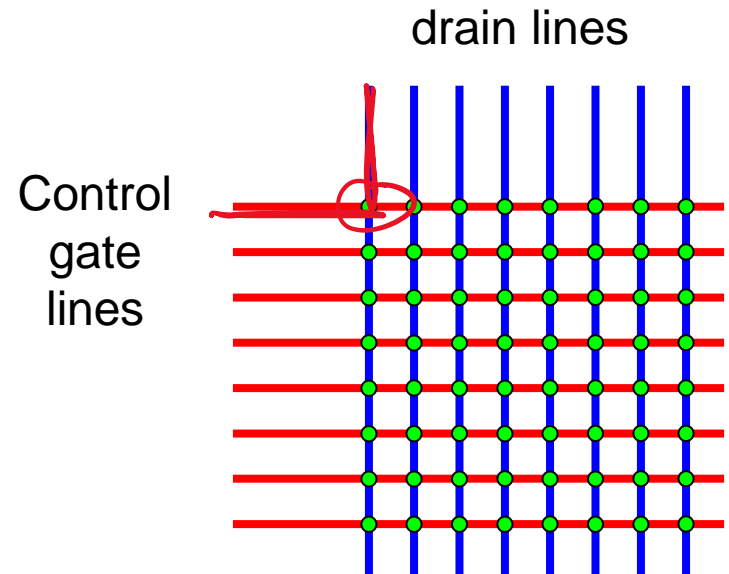
Logic “0” and “1”



Reading a bit means:

1. Apply V_{read} on the control gate
2. Measure drain current I_d of the floating-gate transistor

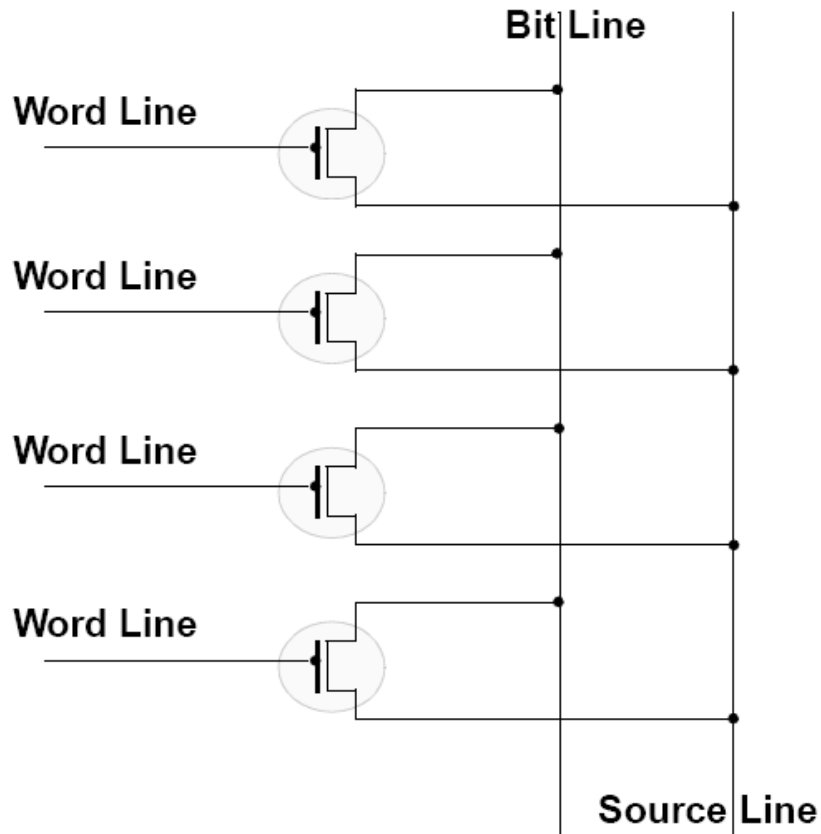
When cells are placed in a matrix:



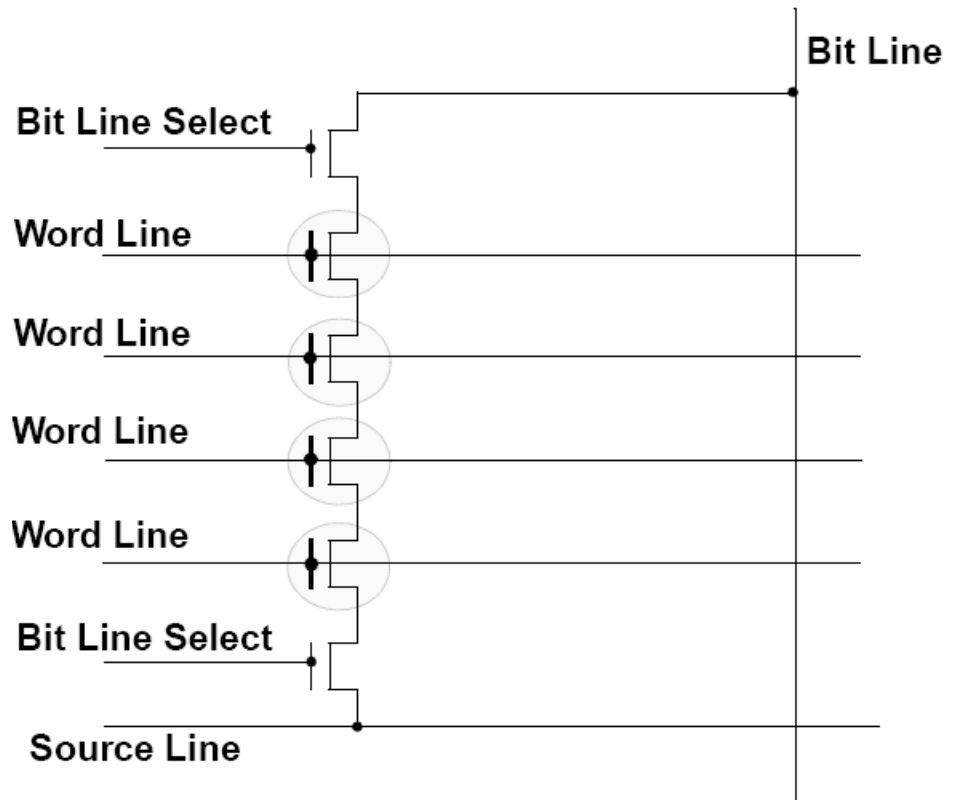
NOR or NAND addressing

‘Word’ = control gate; ‘bit’ = drain

•NOR



•NAND



less contacts → more compact

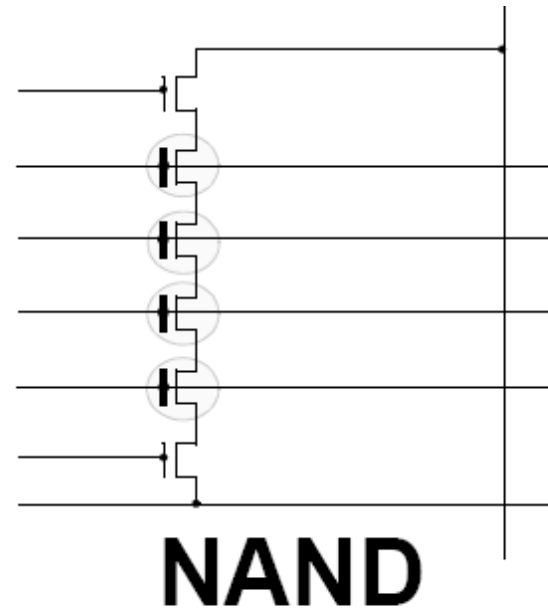
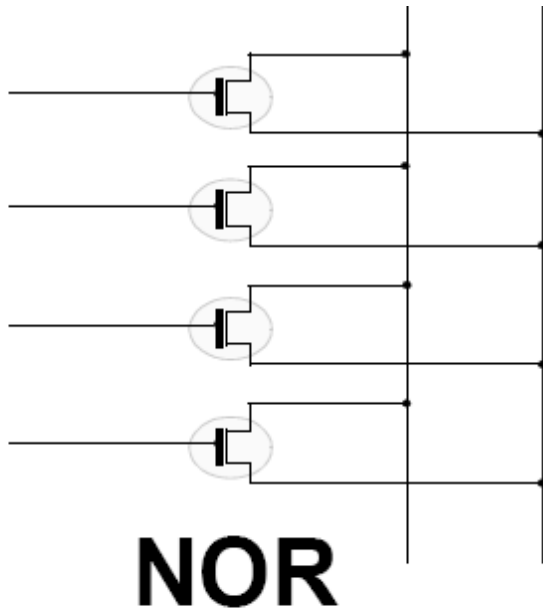
NAND Flash

- Smaller memory cell area (less expensive)
- Word accessible
 - large granularity, 100-1000 bits/word,
 - good for secondary storage where files don't mind being read out in large chunks/words)
- Slower random byte access (have to read a word)
- Short erasing (~2 ms) and programming times (~5 MB/sec sustained write speed)
- Longer write lifetime

NOR Flash

- Larger memory cell area
- Byte accessible (good for ROM-like program storage, where instructions need to be read out on a byte-size granularity)
- Faster random byte access
- Longer erasing (~750 ms) and programming times (~.2 MB/sec sustained write speed)
- Shorter write lifetime
- iPhone uses both multi-GB NAND flash for multimedia file storage and multi-MB NOR flash for code/app storage

NAND versus NOR



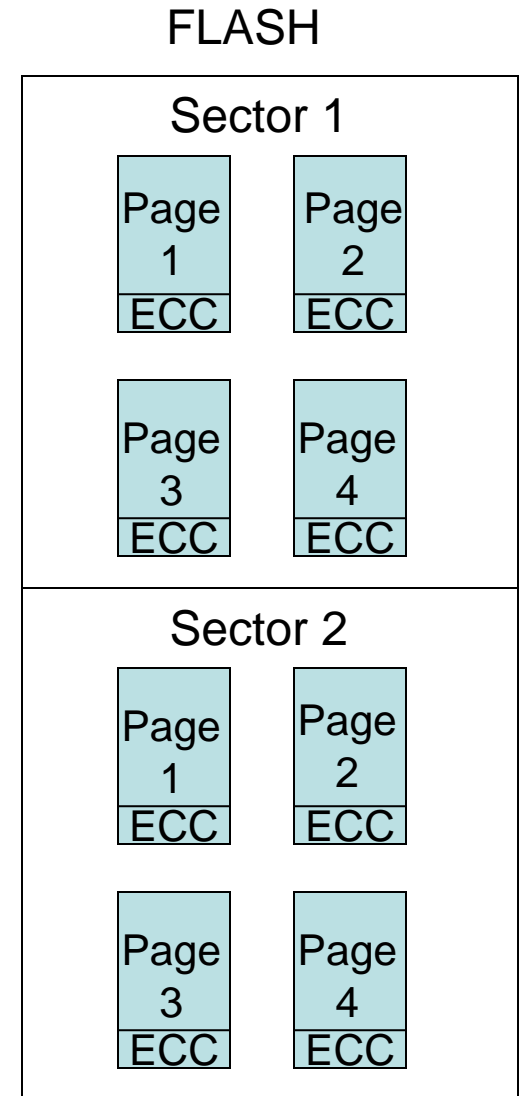
10x better endurance
Fast read (~ 100 ns)
Slow write (~ 10 μ s)
Used for Code

Smaller cell size
Slow read (~ 1 μ s)
Faster write (~ 1 μ s)
Used for Data



Organizing Flash Memory

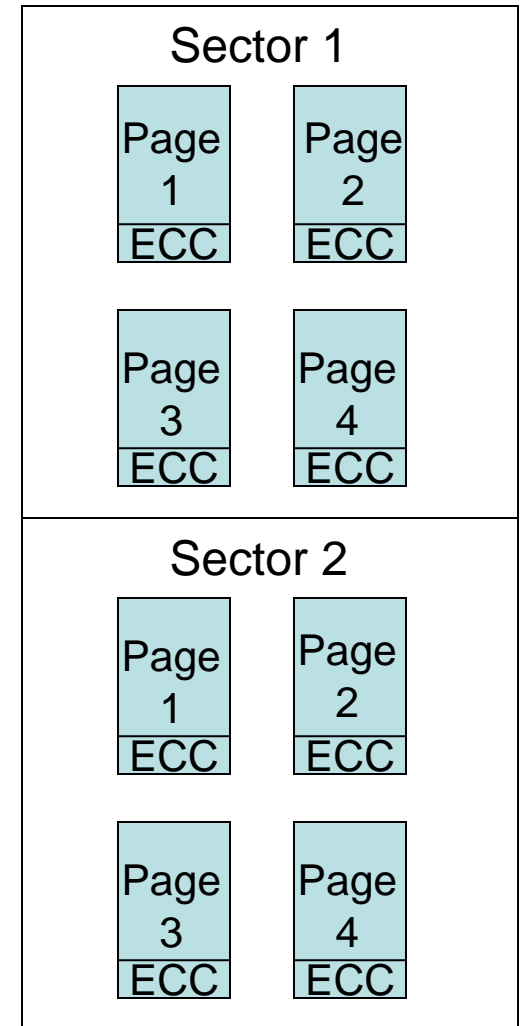
- Flash memory is typically divided into blocks or *sectors*
 - There can be many *pages* per block/sector
 - e.g. 64 pages per block, 16 blocks per flash, and 4 KB per block => 4 MB of flash
 - The last 12-16 bytes of each page is reserved for error detection/correction (ECC) and bookkeeping
 - the flash file system can put information in this space



Operations on Flash Memory

- Reads and writes occur on a sub-page level granularity
 - Can read a byte (NOR) or word (NAND) by giving **page + offset**
- However, writes are tricky
 - Writes of a byte (NOR) or word (NAND) only proceed immediately if that memory has **been cleanly reset/erased and not yet written to**
 - “Rewrites” to memory that has already been written to ***require an erase first before the write***

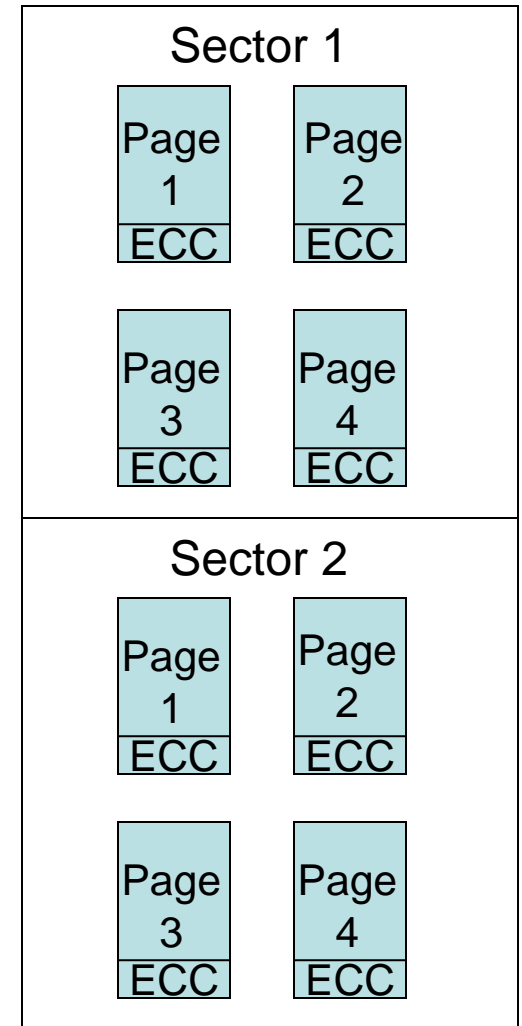
FLASH



Rewrites Are Costly

- Unfortunately, erasing in flash can **only occur on a sector granularity!**
 - This is the price we pay for cheap “flash” technology
- Hence, to write just one byte into a memory location that has already been written to
 - First you have to erase the entire sector containing that byte address
 - Then write the byte!

FLASH



Rewrites Are Costly

- Rewrites happen all the time.
 - Example: deleting a file
 - De-allocate the flash pages containing the file data
 - If we want to allocate these free flash pages to other files, we have to first erase the entire sectors containing these pages
- In comparison, the cost of rewrites is the same as writes for magnetic disk:
 - Can simply have magnetic head reset of the orientation of the bits on magnetic media
 - No need to erase before writing

Rewrites Are Costly

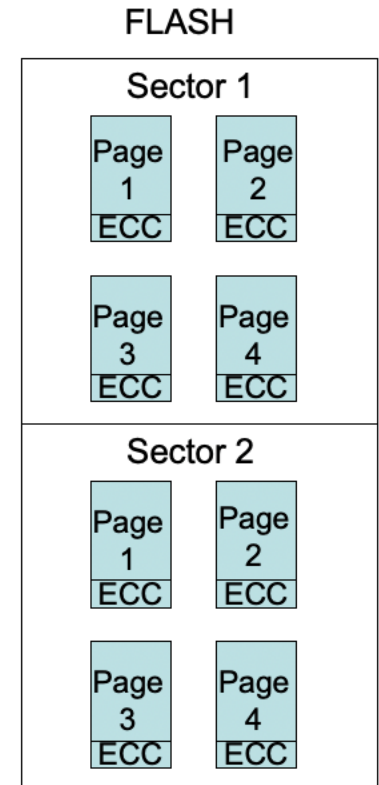
- To rewrite over a page of flash memory *while preserving* the other pages in the same sector:
 - Either:
 1. Copy the entire sector into RAM
 2. In RAM, write the changes to the page that you want to modify
 3. Erase the sector on flash
 4. Write all the sector's pages currently in RAM to the newly erased sector, including the modified page
 - Or:
 - Copy the entire sector into a new clean sector, except for the page to be modified
 - Write the one page to the correct location in the new sector (page is clean)

Limited Lifetime of Flash Memory

- Memory wear - Flash memory has a limited write lifetime
 - NOR flash memory can withstand 100,000 write-erase cycles before becoming unreliable
 - NAND flash memory can withstand 1,000,000 write-erase cycles
 - Reason: The strong electric field needed to set and reset bits eventually breaks down the silicon transistor
- Eventually, your mobile devices can no longer save data!

Wear Leveling Solution

- To extend the life of flash memory chips, write and erase operations are **spread out over the entire flash memory**
 - Keep a count in 12-16 byte trailer of each page of how many writes have been made to that page, and choose the free page with the lowest write count
 - Randomly scatter data over the entire flash memory span, since it doesn't take any longer to extract data from nearby pages as distant pages
 - Many of today's flash chips implement wear leveling in the hardware's device controller



Problems Applying Standard Disk File Systems to Flash Memory

1. A typical disk file system stores data/metadata in fixed locations
 - the directory structure, file headers, allocation structures (e.g. FAT) and free space structures (e.g. bitmap) and even file data are in relatively fixed locations
 - Repeated modifications to these structures in static locations on flash would result in rapidly exhausting the write lifetimes of those locations, hence all of flash

Problems Applying Standard Disk File Systems to Flash Memory

2. Disk file systems are not optimized for rewrites
 - Disk file system rewrites are viewed essentially the same as writes, and repeated rapid rewrites can be quickly done because there is little disk seek time
 - Flash rewrites incur a much greater penalty, are much longer than writes to clean flash pages, and repeated rapid rewrites in flash would cause disastrously long latency, not to mention reduced lifetime

Flash File Systems

- Must be designed to deal with 2 major problems of flash memory:
 - Rewrites requires erases – hold on to writes
 - Limited lifetime
- **Log-structured file systems are well-suited to flash memory!**
 - Writes to a file are added/appended to the end of the log, as are rewrites. The log is the file system.
 - The log grows sequentially and doesn't rewrite over the same location
 - Hence less erasing & also longer lifetime

Flash File Systems

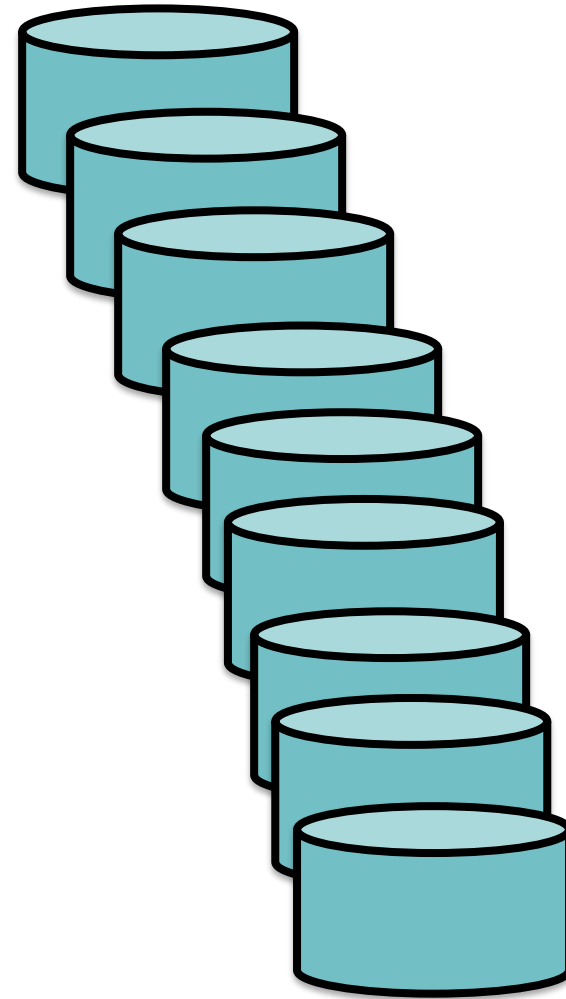
- Journaling Flash File System (JFFS) and JFFS2
 - Is a Log-structured file system similar to log-based recovery seen earlier for file system reliability
 - The file system is the log, and is written sequentially over flash pages,
 - The entries in the log contain all the data we need to reconstruct the file (recall the X & Y old and new values)
 - The log may be circular, eventually wrapping around once all free flash pages have been consumed.
 - Free space = distance between head and tail of log
 - As free space decreases, garbage collection is performed to free up some more space.



Redundant Arrays of Inexpensive Disks (RAID)

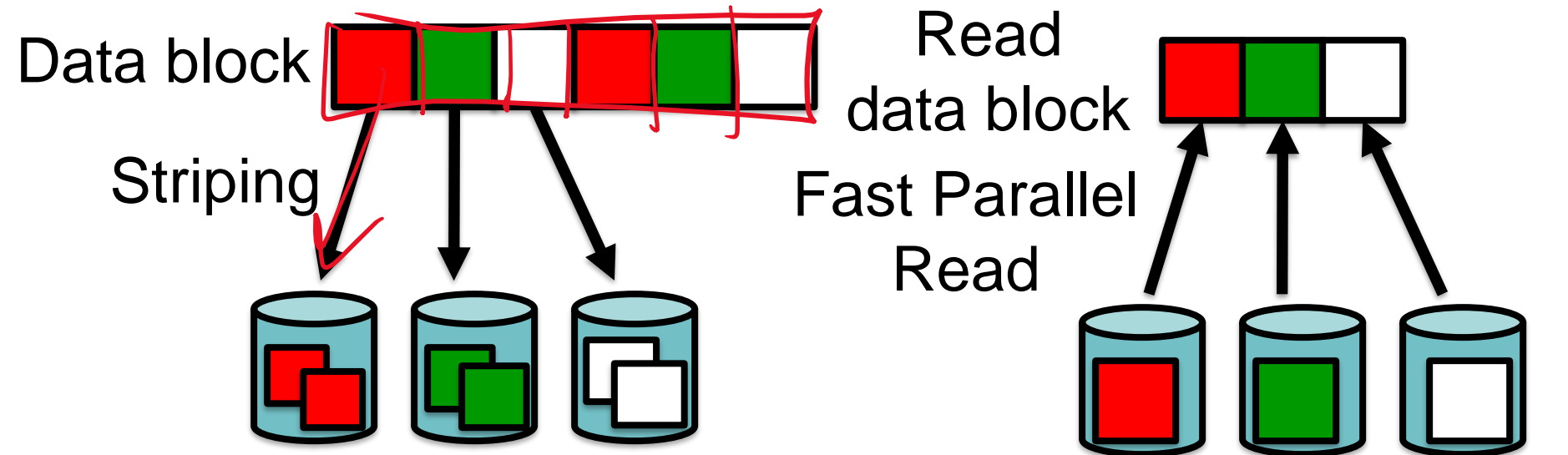
Redundant Arrays of Inexpensive Disks (RAID)

- Magnetic disks are cheap these days.
- Attaching an array of magnetic disks to a computer brings several advantages compared to a single disk:
 - Faster read/write access to data by having multiple reads/writes in parallel.
 - Data is striped across different disks, e.g. each byte of an 8-byte word is striped onto a different disk
 - Better fault tolerance/reliability
 - if one disk fails, a copy of the data could be stored on another disk – redundancy to the rescue!



RAID0

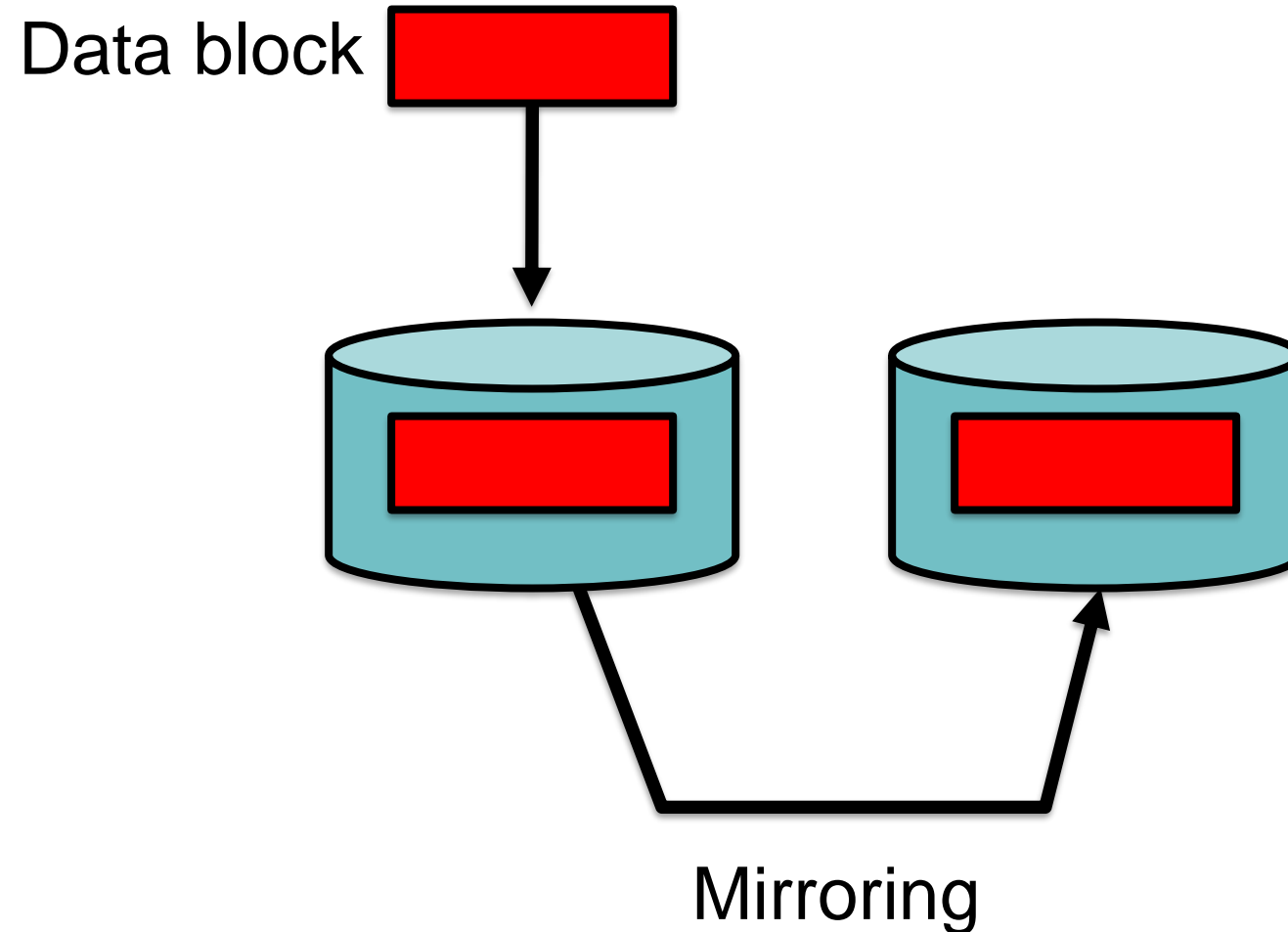
- RAID has different levels with increasing redundancy
 - RAID0 = data striping with no redundancy



Also fast parallel writes

RAID1

– RAID1 = mirror each disk



Get redundancy,
but not parallel
performance
speedup of
RAID0.

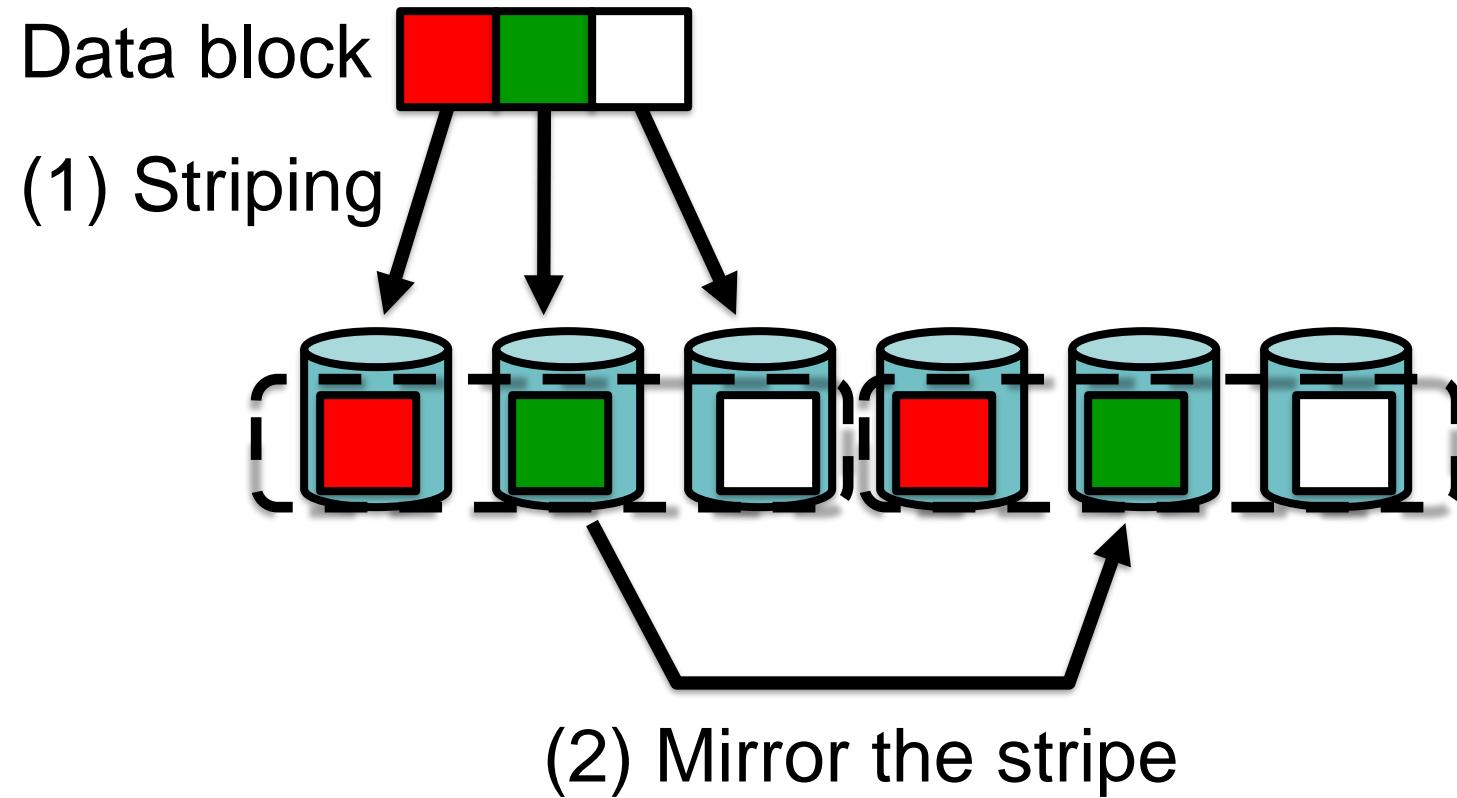
Combine RAID0
with RAID1

Can get limited
read speedup by
submitting the
read to all mirrors,
and taking the 1st
data result

Writes delay x 2
on synchronous
write , but OS can
mask this by
delaying writes

RAID0+1

– RAID0+1

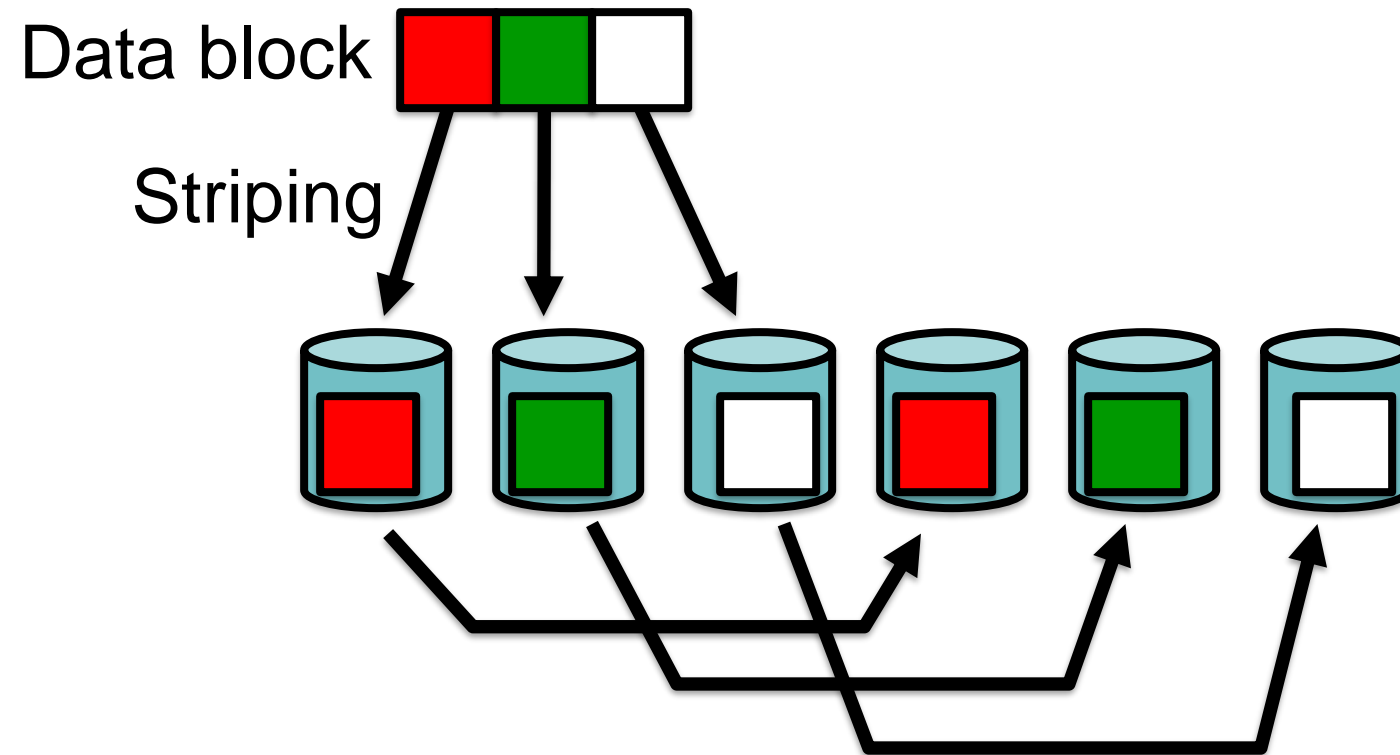


RAID 0+1 = RAID
0 data striping for
performance +
RAID 1 mirroring
of stripes for
redundancy

Note: if both
primary stripe and
second stripe fail
then the entire
data block is lost

RAID1+0

– RAID1+0, aka “RAID 10”



(1) Mirroring each disk

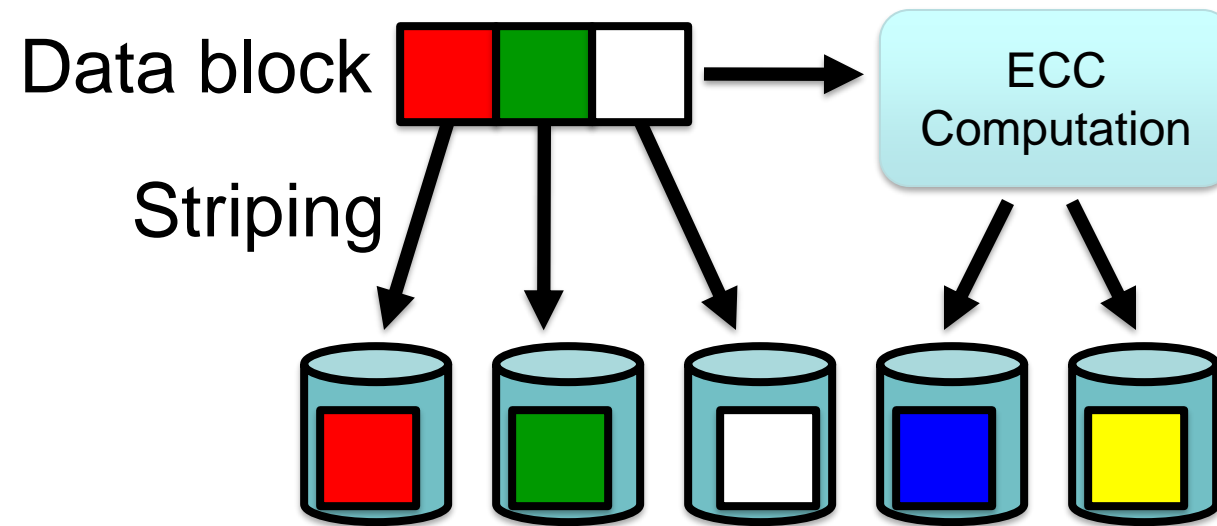
RAID 1+0 = mirror each disk then stripe.

Any two disks may fail, and the data can still be retrieved, unless the two disks mirror the same data is failed

Thus, RAID 1+0 is more robust than RAID 0+1.

RAID2

- RAID2 = put Error Correction Code (ECC) bits on other disks

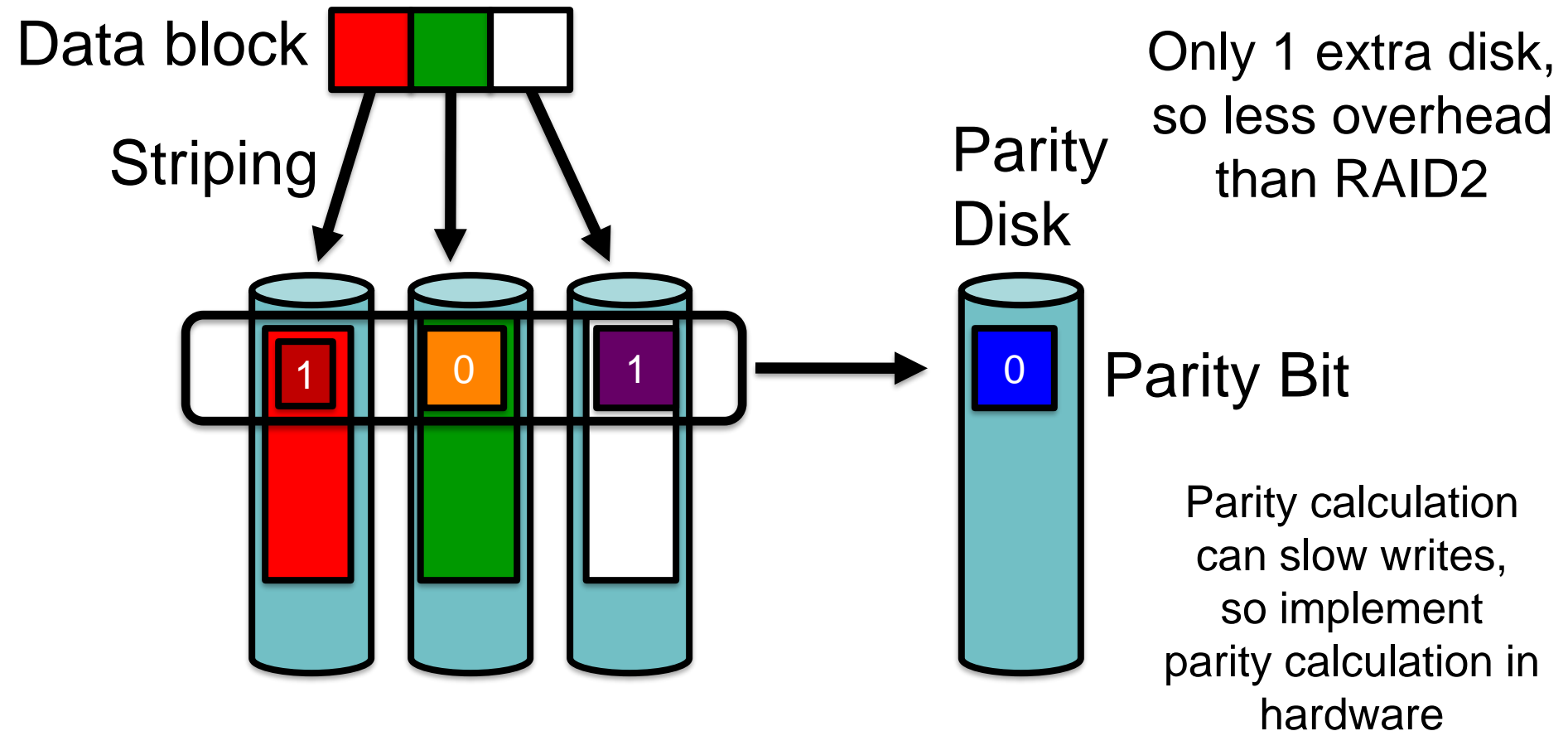


Error correction codes are more compact than just copying the data, and provide strong statistical guarantees against error

e.g. a crashed disk's lost data can be corrected with the redundant data stored in the ECC disks

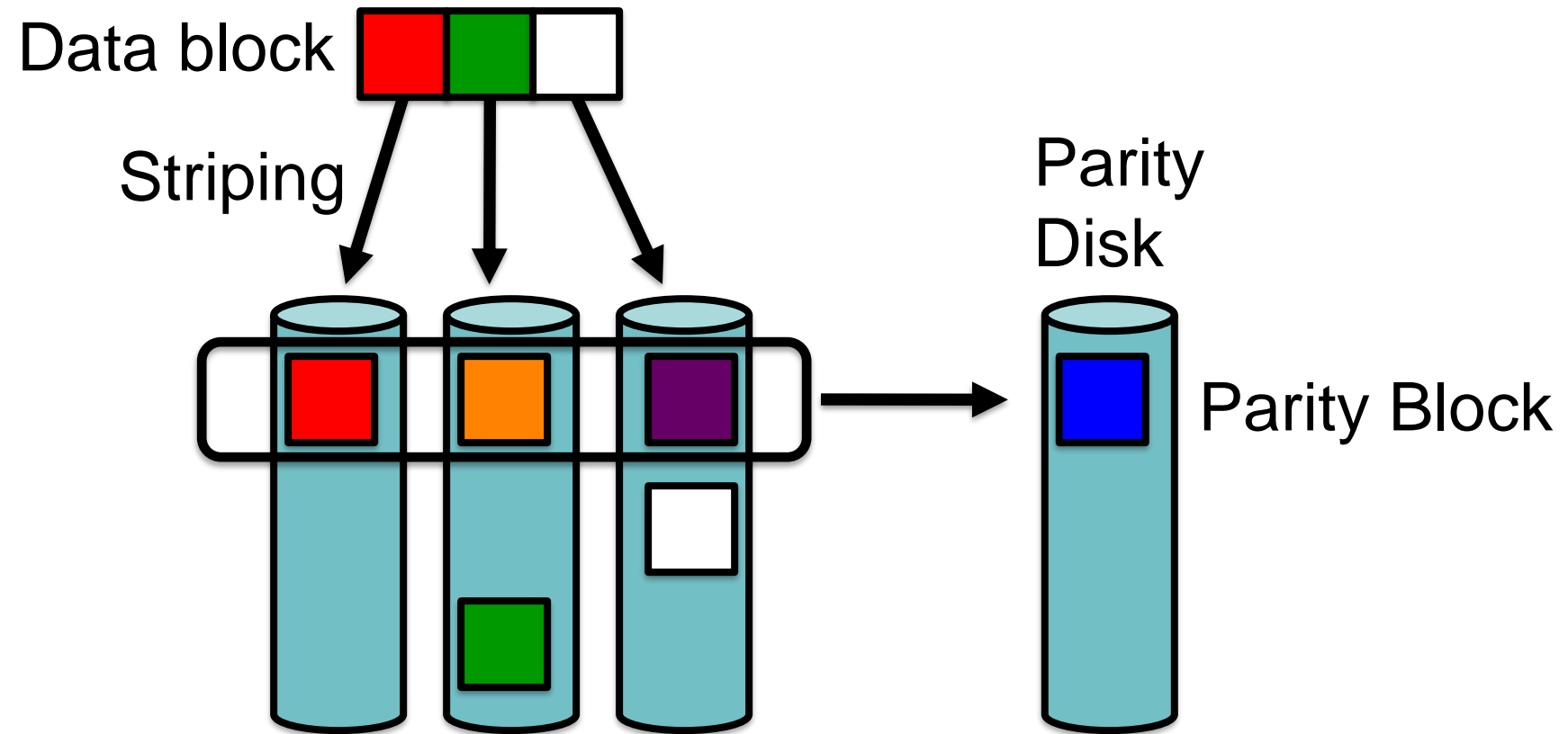
RAID3

- RAID3 = bit-interleaved parity: for each bit at the same location on different disks, compute a parity bit, that is stored on a parity disk



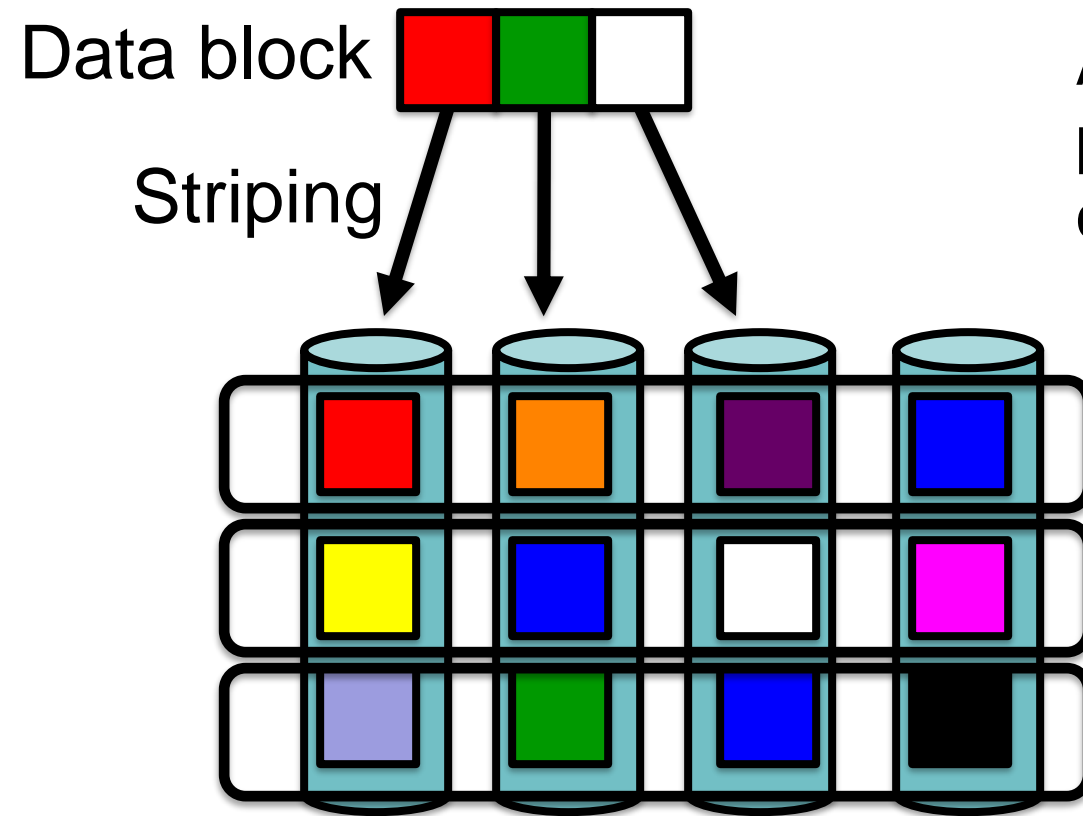
RAID4

- RAID4 = block-interleaved parity: for each block at the same location on different disks, compute a parity block, that is stored on a parity disk



RAID5

- RAID5 = block-interleaved *distributed* parity: spread the parity blocks to different disks, so every disk has both data and parity blocks



Avoids overuse of RAID4's parity disk, and is the most common parity RAID system

 = Parity Blocks

RAID6 replaces parity blocks with stronger Reed-Solomon ECC for multiple disk failures

RAID Implementation

- Three methods:
 - Software Based,
 - Firmware Based,
 - Hardware Based
- 1. Software Based: Plug in disks to your computer bus and implement RAID management in software in OS kernel
 - The RAID software layer sits above the device drivers
 - Cost effective and easy to implement, but not as efficient
 - In Linux, use the mdadm package to manage RAID arrays.
 - Cannot boot from RAID

RAID Implementation

2. Firmware/Driver Based: Plug in disks to your computer's I/O bus, and **an intelligent hardware RAID controller** recognizes them and integrates them into a RAID system automatically
 - Standard disk controller chips with special firmware and drivers
 - RAID instructions are stored in the firmware of the device
 - Can boot from RAID: During startup/boot, the RAID is essentially kick-started by the firmware
 - CPU still need to handle it => Costly

RAID Implementation

3. Hardware Based: Plug in disks to a RAID array, which is a stand-alone hardware unit with a RAID controller that looks like one physical device, e.g. SCSI, to your computer bus
 - File system doesn't have to know about RAID to use and benefit from this RAID disk array
 - Expensive but highly efficient

Summary

- Storage management
 - Magnetic
 - Disk Scheduling
 - Flash
 - Page/Sector management
 - RAID
 - Reliability vs. access time.