

# CSC 369

Week 10:

Disk I/O and File System Optimization



University of Toronto, Department of Computer Science



# Overview

---

- Last time:
  - File systems – Interface, structure, basic implementation
- This week:
  - Review FS basics in more detail
  - UNIX inode structure
  - File system optimizations
  - Magnetic disks as secondary storage
  - Disk scheduling



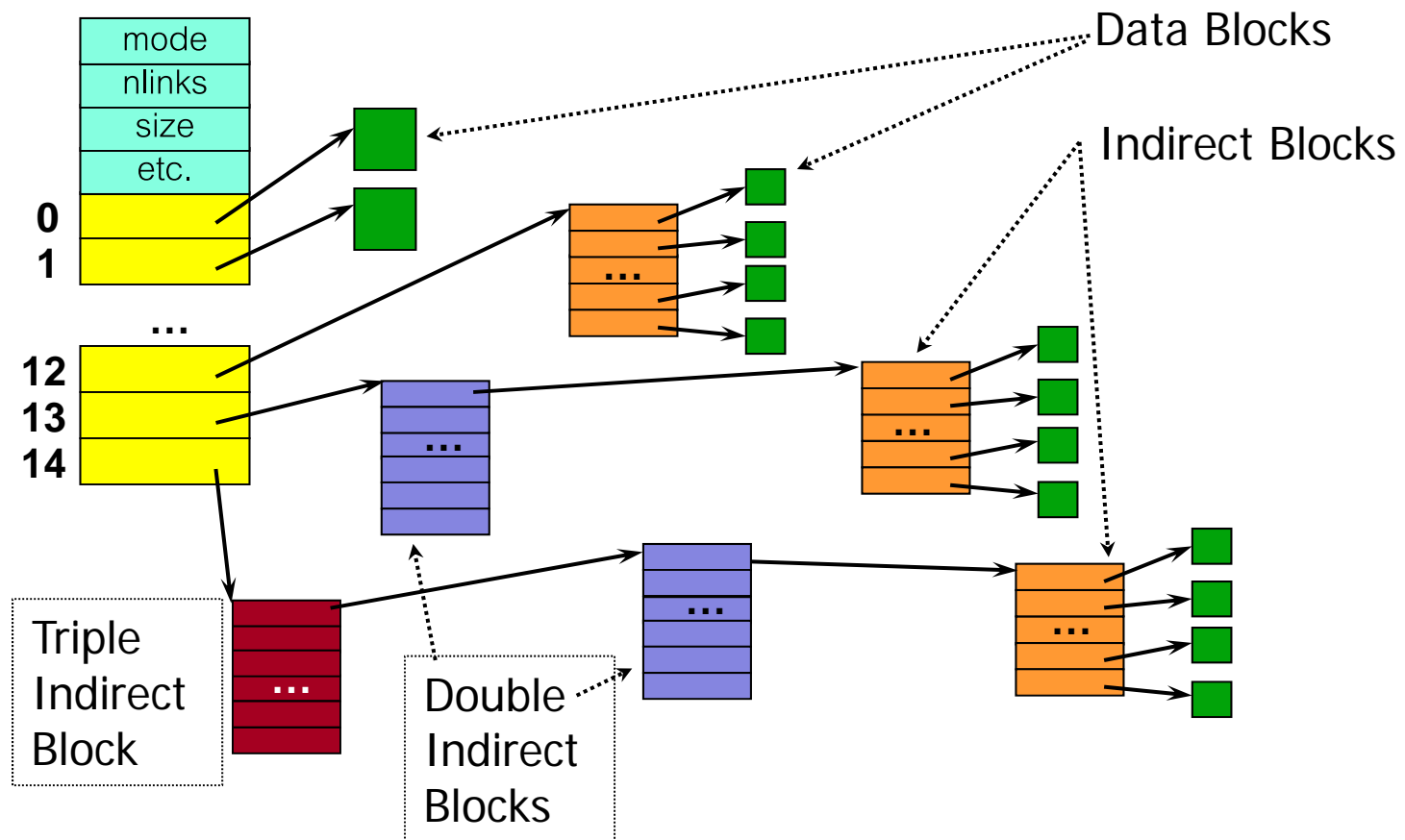
# Review of File Systems

---

- Many file system implementations, literally from AFS to ZFS
  - Check out [https://en.wikipedia.org/wiki/List\\_of\\_file\\_systems](https://en.wikipedia.org/wiki/List_of_file_systems)
- Most well-known:
  - Windows: FAT32, NTFS
  - MAC OS X: HFS+
  - BSD, Solaris: UFS, ZFS
  - Linux: ext2 (see A4 too!), ext3, ext4, ReiserFS, XFS, JFS, btrfs, etc.
- We'll have a look at a very simple file system (VSFS)
  - See readings as well!



# Remember: inode-based FS organization





# The main idea

---

- We need to create a file system for an unformatted disk
- We need to create some structure in it, so that things (data) will be easy to find and organize
- Key questions
  - Where do we store file data and metadata structures (inodes)?
  - How do we keep track of data allocations?
  - How do we locate file data and metadata?
  - What are the limitations (max file size, etc.)?



*Bad programmers worry about the code.  
Good programmers worry about data structures  
and their relationships.  
-- Linus Torvalds*

# Implementation of a Very Simple File System (VSFS)



University of Toronto, Department of Computer Science



# An unformatted raw disk

---

Total size = 256KB







# Overall Organization

The whole disk is divided into fixed-sized **blocks**.

Block size: 4KB

Number of blocks: 64

Total size: 256KB

0																15
16																31
32																47
48																63



# Data Region

Most of the disk should be used to actually store **user data**, while leaving a little space for storing other things like **metadata**. In VSFS, we reserve the **last 56 blocks** as data region.

0								D	D	D	D	D	D	D	D	15
16	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	31
32	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	47
48	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	63



# Metadata: inode table

FS needs to track information about each file.

In VSFS, we keep the info of each file in a struct called inode. And we use 5 blocks for storing all the inodes.

Size	Name	What is this inode field for?
2	mode	can this file be read/written/executed?
2	uid	who owns this file?
4	size	how many bytes are in this file?
4	time	what time was this file last accessed?
4	ctime	what time was this file created?
4	mtime	what time was this file last modified?
4	dtime	what time was this inode deleted?
2	gid	which group does this file belong to?
2	links_count	how many hard links are there to this file?
4	blocks	how many blocks have been allocated to this file?
4	flags	how should ext2 use this inode?
4	osd1	an OS-dependent field
60	block	a set of disk pointers (15 total)
4	generation	file version (used by NFS)
4	file_acl	a new permissions model beyond mode bits
4	dir_acl	called access control lists
4	faddr	an unsupported field
12	i_osd2	another OS-dependent field

ext2 inode with size of 128B

0				I	I	I	I	I	D	D	D	D	D	D	D	15
16	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	31
32	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	47
48	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	63

**How many files can this VSFS hold at most?**

Maximum number of inodes it can hold:  $5 * 4\text{KB} / 128\text{B} = 160 \Rightarrow$  can store at most 160 files.



# Allocation Structures

- Keep track of which blocks are being used and which ones are free
- We use a data structure (a **bitmap**) for this purpose
  - Each bit indicates if one block is free (0) or in-use (1)
- A bitmap for the **data region** and a bitmap for the **inode region**
  - Reserve one block for each bitmap.

0		IB	DB	I	I	I	I	I	D	D	D	D	D	D	D	D	15
16	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	31
32	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	47
48	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	63

**How many blocks can a 4KB data bitmap keep track of?**

4KB = 32K bits, can keep track of 32K blocks, so yeah it's overkill in this VSFS.



# Superblock

- Superblock contains information about this particular file system:
  - what type of file system it is (“VSFS” indicated by a magic number)
  - how many inodes and data blocks are there (160 and 56)
  - where the inode table begins (block 3), etc.

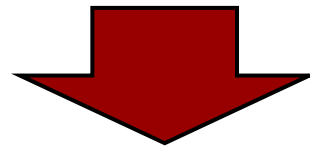
0	S	IB	DB	I	I	I	I	I	D	D	D	D	D	D	D	D	15
16	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	31
32	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	47
48	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	63

When mounting a file system, the OS first reads the superblock, identifies its type and other parameters, then attaches the volume to the file system tree with proper settings.



# Formatting disk into VSFS, done!

Raw Disk

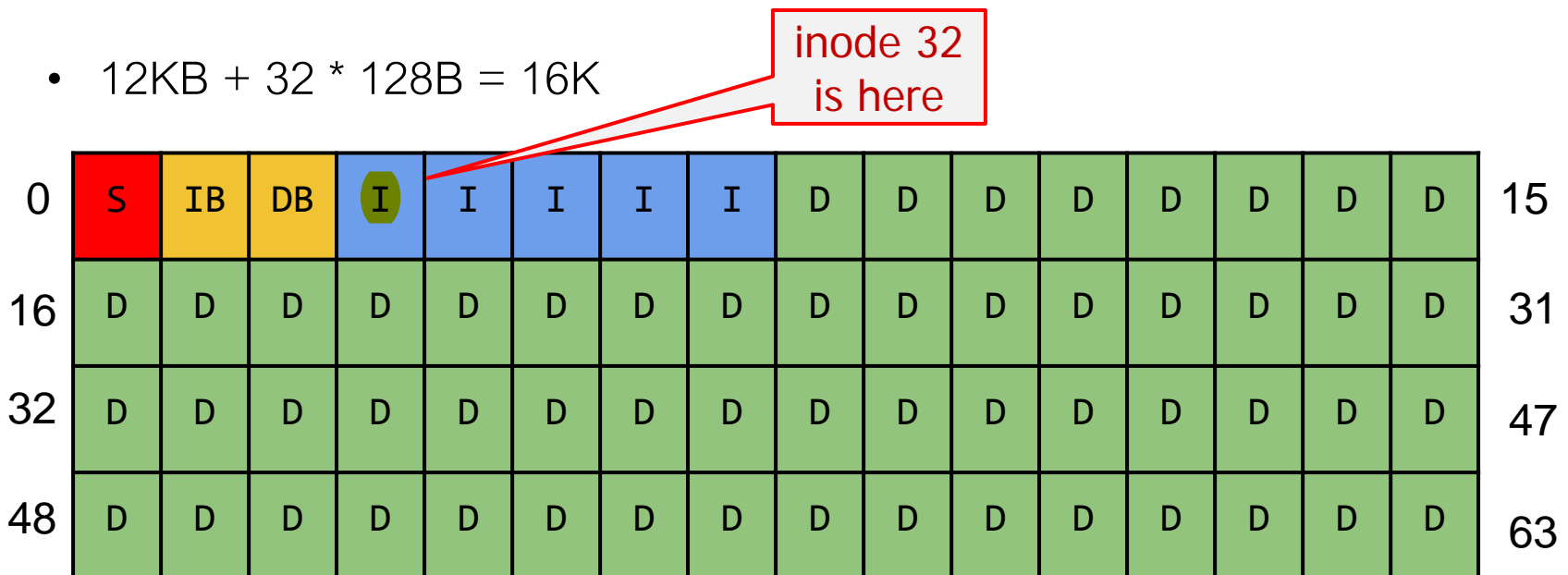


0	S	IB	DB	I	I	I	I	I	D	D	D	D	D	D	D	D	15
16	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	31
32	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	47
48	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	63



## Example: Read a file with inode number 32

- From the superblock, we know
  - inode table begins at Block 3, i.e., 12KB
  - inode size is 128B
- Calculate the address of inode 32
  - $12\text{KB} + 32 * 128\text{B} = 16\text{K}$



So we have the inode, but which blocks have the **data**?



# From inode to data

- Say the inode contains an array of 15 **direct pointers** that point to 15 data blocks that belong to the file.
- Maximum file size supported:
  - $15 * 4\text{KB} = 60\text{KB}$
- If we need a file larger than 60KB, we need to do something more sophisticated.

Size	Name	What is this inode field for?
2	mode	can this file be read/written/executed?
2	uid	who owns this file?
4	size	how many bytes are in this file?
4	time	what time was this file last accessed?
4	ctime	what time was this file created?
4	mtime	what time was this file last modified?
4	dtime	what time was this inode deleted?
2	gid	which group does this file belong to?
2	links_count	how many hard links are there to this file?
4	blocks	how many blocks have been allocated to this file?
4	flags	how should ext2 use this inode?
4	osd1	an OS-dependent field
60	block	a set of disk pointers (15 total)
4	generation	file version (used by NFS)
4	file_acl	a new permissions model beyond mode bits
4	dir_acl	called access control lists
4	faddr	an unsupported field
12	i_osd2	another OS-dependent field

ext2 inode with size of 128B

0	S	IB	DB	I	I	I	I	I	D	D	D	D	D	D	D	15
16	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	31
32	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	47
48	D	D	D	D	D	D	D	D	D	D	D	D	D	D	D	63





# Multi-Level Index with Indirect Pointers

- Direct pointers to disk blocks do not support large files
- Idea: **indirect pointer**
  - Instead of pointing to a block of **user data**, it points to a block that contains **more pointers**
  - From the 15 pointers we have in an inode, use the first 14 as direct pointers and 15th as an indirect pointer
- **How big a file can we support now?**
  - 14 direct pointers in total: 14 data blocks
  - Indirect pointer points to a block (4KB) which can hold 1K pointers => 1K data blocks in addition
  - Total size supported:  $4K * (14 + 1K) = 4152KB$

**What if I want even BIGGER?**



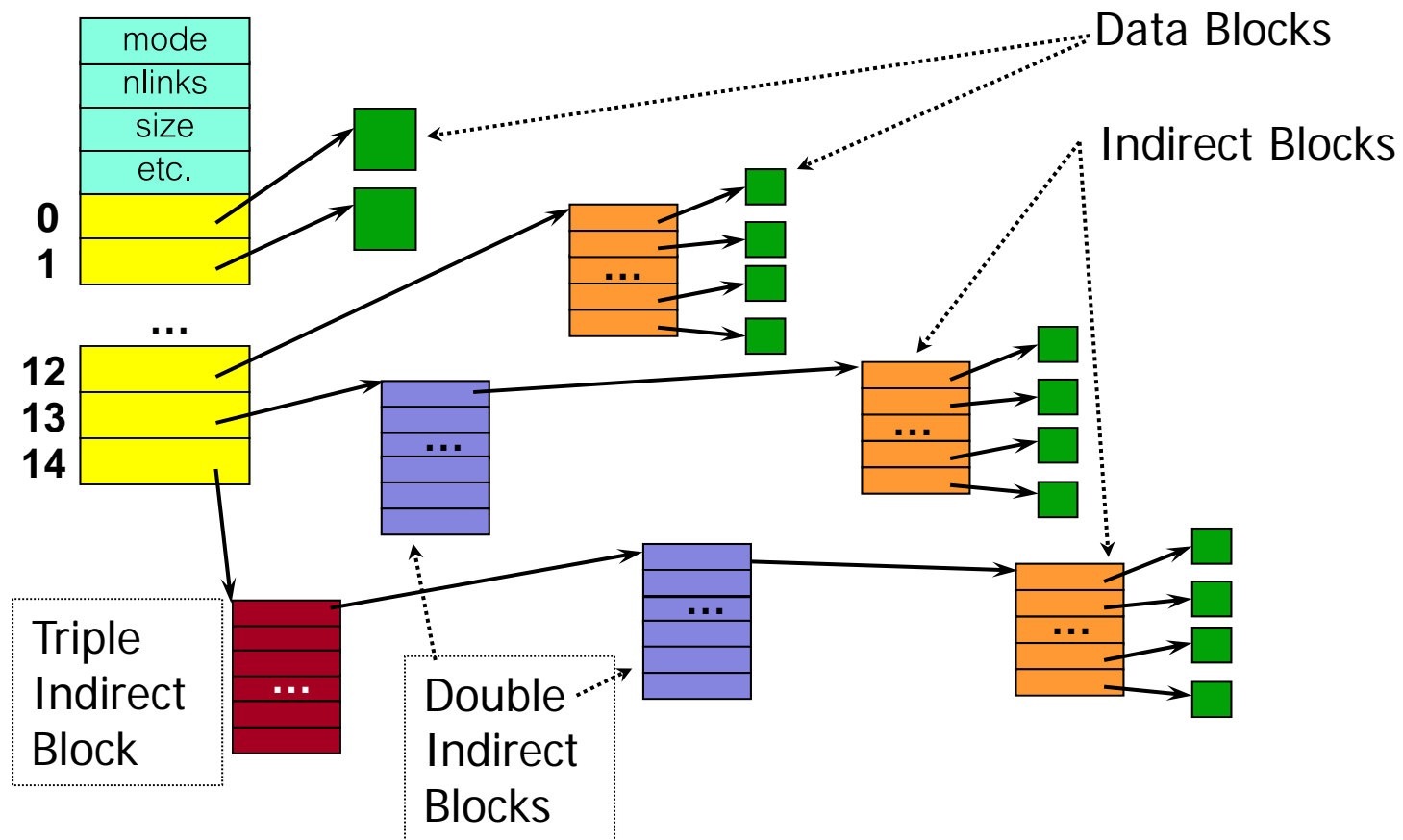
# Double Indirect Pointer!

- A **double indirect pointer** points to a **block full of indirect pointers** which point to **blocks of direct pointers**.
  - E.g., for the 15 pointers, we use the first 13 as direct pointers, the 14th as an indirect pointer, and the 15th as a double indirect pointer.
- **How big a file do we support now?**
  - From direct pointers: 13 data blocks
  - From indirect pointers: 1024 data blocks
  - From double indirect pointers:  $1024 * 1024 = 1\text{M}$  data blocks
  - Total size =  $4\text{K} * (13 + 1024 + 1024 * 1024) \approx 4.004\text{GB}$

**Still not big enough? Use a Triple Indirect Pointer**



# A tree-view of multi-level indirect pointers





# A tree-view of multi-level indirect pointers

---

- You might wonder: this tree is super **imbalanced**.
- Frequently access big files => the blocks with the double and/or triple indirect pointers are going to be accessed like **crazy** (because most of the data blocks are in their subtrees).
- True, but it's fine: in practice, **most files are small**.
- For more practice-inspired decisions, read "*A Five-Year Study of File System Metadata*" (see course web page)



## Another approach: **extent-based**

- An **extent** == a disk pointer plus a length (in # of blocks), i.e., it allocates a few blocks in a row.
- Instead of requiring a pointer to every block of a file, we **just need a pointer to every several blocks** (every extent).
- *Disadvantage:* Less flexible than the pointer-based approach
- *Advantages:* Uses smaller amount of metadata per file, and file allocation is more compact.
- Adopted by ext4, HFS+, NTFS, XFS.



# Yet another approach: **Linked-Based**

- Instead of pointers to all blocks, the inode just has one pointer to the first data block of the file, then the first block points to the second block, etc.
- Works poorly if we want to access the last block of a big file.
- Use an in-memory **File Allocation Table**, indexed by address of data block
  - Faster in finding a block.
- FAT file system, used by Windows before NTFS.
- Focus on inode-based FSs in next lectures...



# Summary

---

- Inodes
  - Data structure representing a FS object (file, dir, etc.)
  - Attributes, disk block locations
  - No file name, just metadata!
- Directory
  - List of (name, inode) mappings
  - Each directory entry: a file, other directory, link, itself (.), parent dir (..), etc.



# Inodes and directories: Examples

- Inodes

- Data structure representing a FS object (file, dir, etc.)
- Attributes, disk block locations
- No file name, just metadata!

- Directory

- List of (name, inode) mappings
- Each directory entry: a file, other directory, link, itself (.), parent dir (..), etc.

- Use `stat` or `ls -lf` to check out information!
- On your machine, info about your FS:
  - `sudo dumpe2fs -h /dev/sda1`
  - `df -Th`





# Links: Examples

- Hard links
  - Multiple file names (and directory entries) mapped to the same inode
  - Reference count – only remove file when it reaches 0
- Soft (Symbolic) links
  - “Pointer” to a given file
  - Contains the path



# The content of a data block

---

- If it belongs to a regular file
  - Data of the file
- If it belongs to a directory
  - List of directory entries: (name, inode number) pairs, which are the entries under the directory
- If it belongs to a symbolic link
  - The path of the file that it links to



# Unix Inodes and Path Search

- Unix Inodes are **not** directories
- They describe where on the disk the blocks for a file are placed
  - Directories are files, so inodes also describe where the blocks for directories are placed on the disk
- Directory entries map file names to inodes
  - To open “/somefile”, use Master Block to read the inode for “/”
  - inode allows us to find data block for directory “/”
  - Read data block for “/”, look for entry for “somefile”
  - This entry identifies the inode for “somefile”
  - Read the inode for “somefile” into memory
  - The inode says where first data block is on disk
  - Read that block into memory to access the data in the file



# Next up

---

 Performance optimizations ...



# Caching

---

- File operations (`open()` / `read()` / `write()`) incur a good amount of disk I/Os
- Then how can the file system perform reasonably well?
  - Caching!



# File Buffer Cache

- Key observation: Applications exhibit significant **locality** for reading and writing files. How? (recall VM)
- Idea: Cache file blocks in memory to capture locality
  - This is called the **file buffer cache**
  - Cache is system wide, used and shared by all processes
  - Reading from the cache makes a disk perform like memory
  - Significant reuse: **spatial and temporal locality**
  - Even a 4 MB cache can be very effective
- **What do we want to cache?**
  - Inodes, directory entries, disk blocks for “hot” files, even whole files if small.



# Caching and Buffering

---

- So, we use memory to cache important blocks
- **Static partitioning:** at boot time, allocate a fixed-size cache in memory (typically 10% of total memory) -- early file systems
- **Dynamic partitioning:** integrate virtual memory pages and file system pages into a unified page cache, so pages of memory can be flexibly allocated for either virtual memory or file system, used by modern systems
- *Replacement policy:* typically use LRU
- Tradeoff between static vs dynamic partitioning
  - Applicable to any resource allocation kind of problem!



# Caching Writes

---

- Caching works really well for reads!
- For writes, not as much!
  - Writes still have to go to disk anyway to become persistent.
  - Once a block is modified in memory, the write back to disk may not be immediate (**synchronous**).





# Caching Writes

- Use a **buffer** to buffer writes. Buffering a batch of disk writes is helpful because:
  - Combine **multiple writes into one write**
    - e.g., updating multiple bits of the inode bitmap
  - Can improve performance by **scheduling the buffered writes (lazy updates)**
    - e.g., can schedule buffered writes in such a way that they happen sequentially on disk.
  - Can **avoid some writes**
    - e.g., one write changes a bit from 0 to 1, then another write changes it from 1 to 0, if buffered than no need to write anything to disk



# Tradeoff: speed vs durability

---

- Caching and buffering improves the speed of file system reads and writes
- However, it sacrifices the durability of data.
  - Crash occurs => buffered writes not written to disk yet, are lost
  - Better durability => sync to disk more frequently => worse speed
- Should I favour speed or durability?
  - It depends
  - On what? It depends on the application, e.g.,
    - web browser cache
    - bank database



# Approaches?

---

- Several ways to address these concerns
  - Delayed writes only for a specific amount of time
    - How long do we hold dirty data in memory?
  - Asynchronous writes (“write-behind”)
    - Maintain a queue of uncommitted blocks
    - Periodically flush the queue to disk
    - Unreliable
  - Battery backed-up RAM (NVRAM)
    - As with write-behind, but maintain queue in NVRAM
    - Expensive
  - Log-structured file system
    - Always write contiguously at end of previous write



# Read Ahead

- Many file systems implement “read ahead”
  - FS predicts that the process will request next block
  - FS goes ahead and requests it from the disk
  - This can happen while the process is computing on previous block
    - Overlap I/O with execution
  - When the process requests block, it will be in cache
  - Compliments the on-disk cache, which also is doing read ahead
- For sequentially accessed files, can be a big win
  - Unless blocks for the file are scattered across the disk
  - File systems try to prevent that, though (during allocation)



# Next up

---

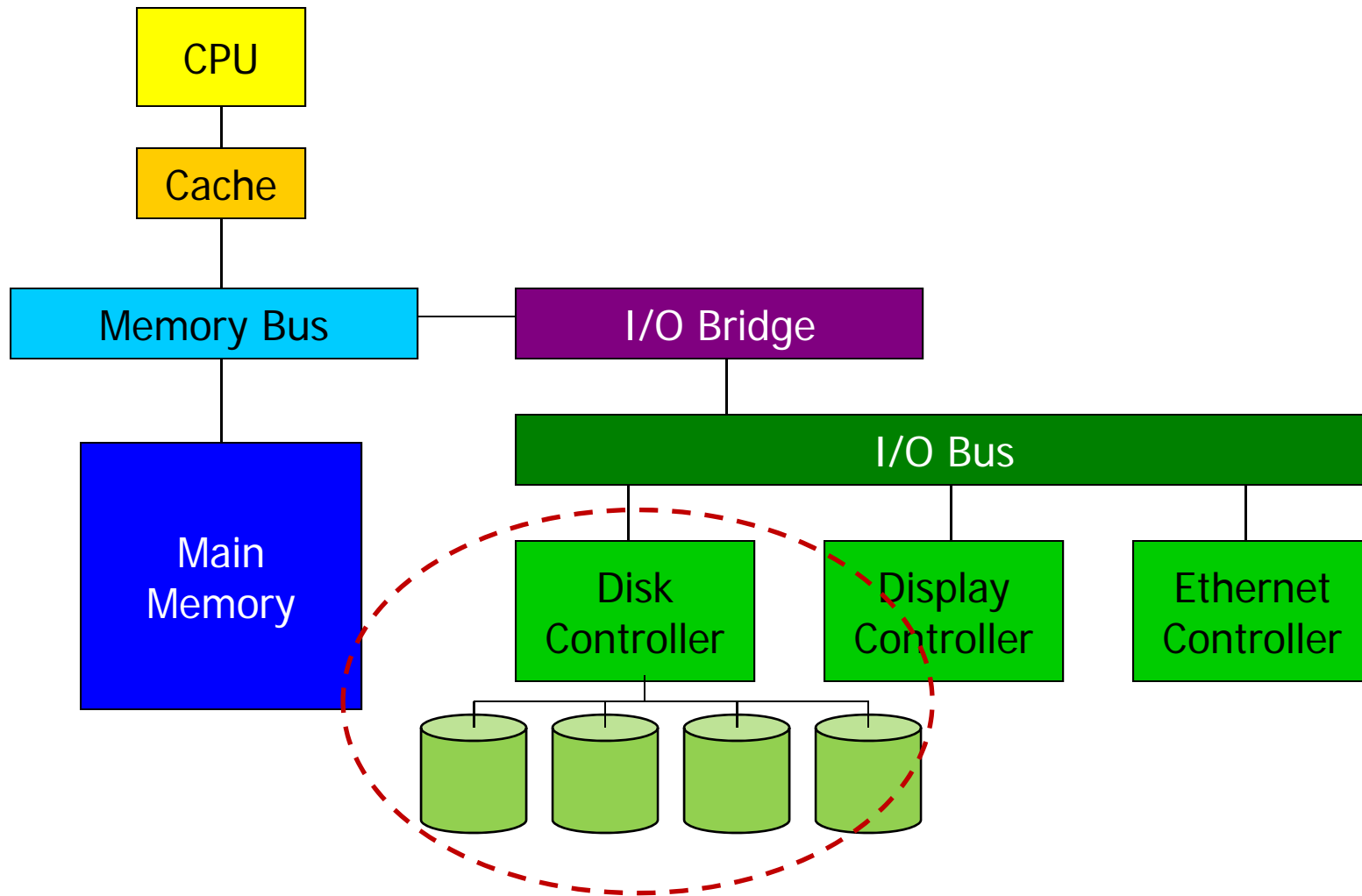
- We looked at performance with caching, buffering, read-ahead
- What about the actual disk's physical characteristics?



A closer look at secondary storage, where files live...



# I/O Diagram





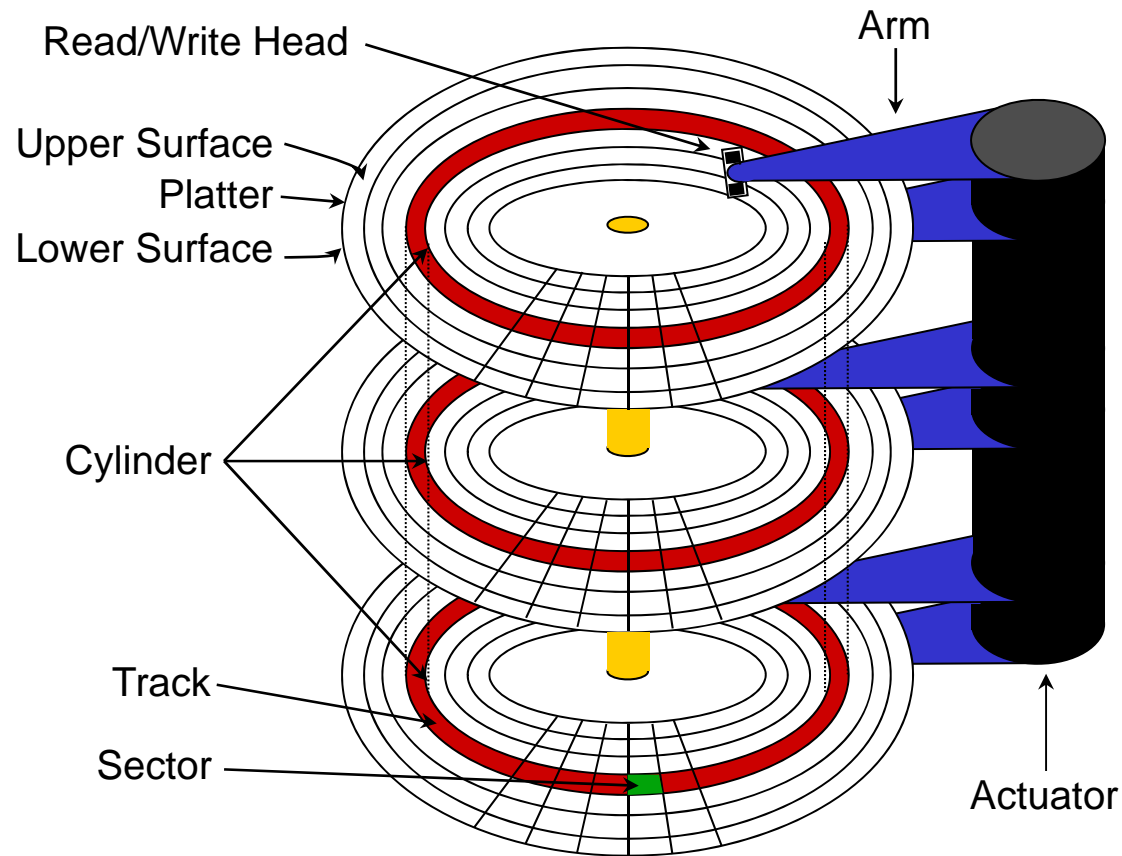
# Secondary Storage Devices

---

- Drums
  - Ancient history
- Magnetic disks
  - Fixed (hard) disks
  - Removable (floppy) disks
- Optical disks
  - Write-once, read-many (CD-R, DVD-R)
  - Write-many, read-many (CD-RW)
- We're going to focus on the use of fixed (hard) magnetic disks for implementing secondary storage



# Disk Components





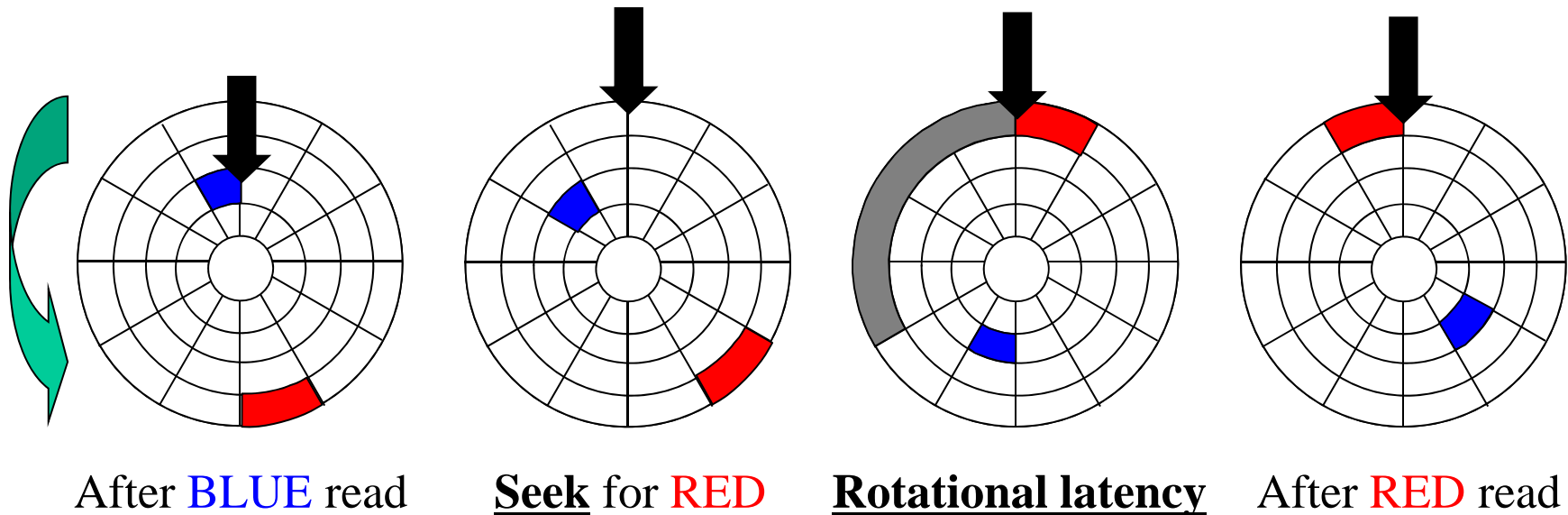


# Disk Performance

- Disk request performance depends on a number of steps
  - **Seek** – moving the disk arm to the correct cylinder
    - Depends on how fast disk arm can move
    - Typical times: 1-15ms, depending on distance (avg 5-6 ms)
    - Improving very slowly (7-10% per year)
  - **Rotation** – waiting for the sector to rotate under the head
    - Depends on rotation rate of disk (7200 RPM SATA, 15K RPM SCSI)
    - Average latency of  $\frac{1}{2}$  rotation ( $\sim 4$  ms for 7200 RPM disk)
    - Has not changed in recent years
  - **Transfer** – transferring data from surface into disk controller electronics, sending it back to the host
    - Depends on density (increasing quickly)
    - $\sim 100$  MB/s, average sector transfer time of  $\sim 5\mu\text{s}$
    - improving rapidly ( $\sim 40\%$  per year)



# Traditional service time components



- When the OS uses the disk, it tries to minimize the cost of all of these steps
  - Particularly seeks and rotation



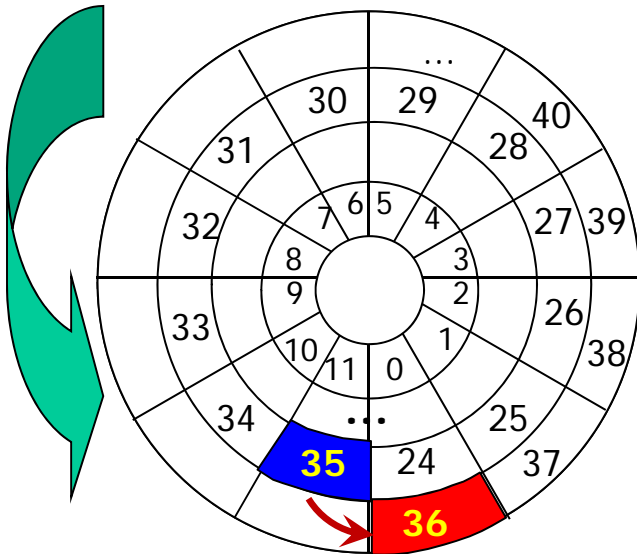
# Some hardware optimizations

---

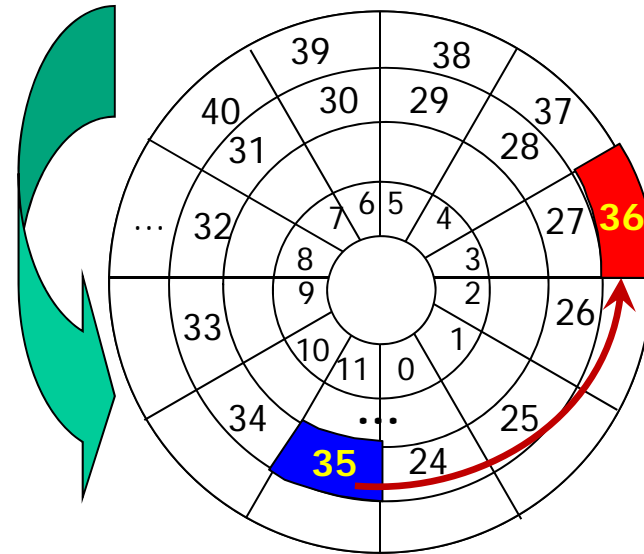
- Track skew
- Zones
- Cache



# Track skew



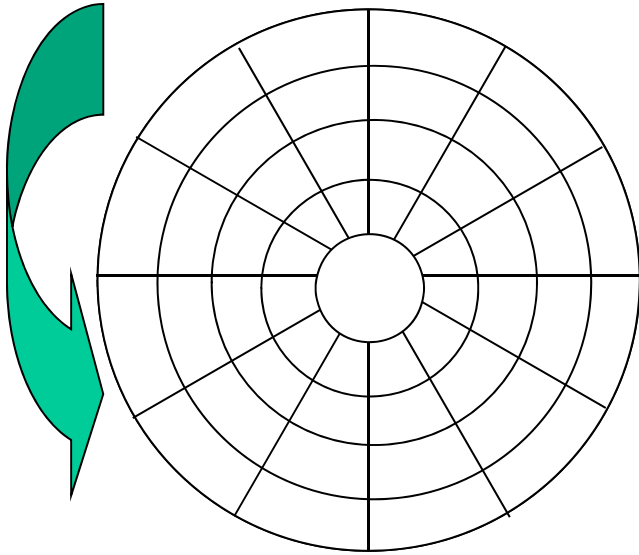
If the arm moves to outer track too slowly, may miss sector 36 and have to wait for a whole rotation.



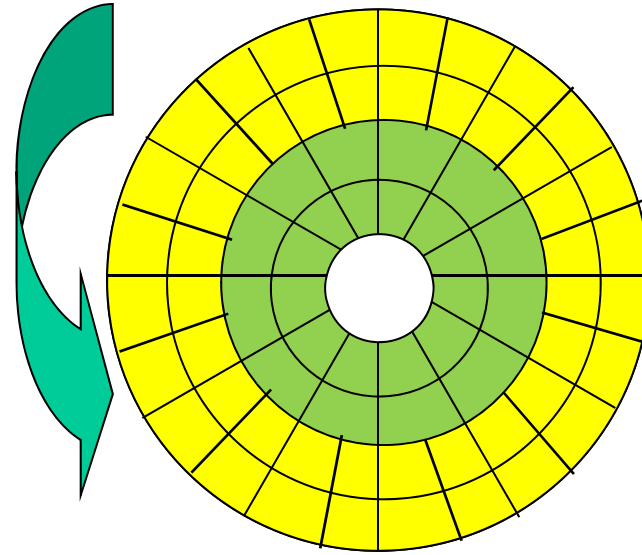
Instead, skew the track locations, so that we have enough time to position.



# Zones



Each sector is 512 bytes  
Notice anything though?



Outer tracks are larger by geometry, so they should hold more sectors.



# Cache, aka Track Buffer

---

- A small memory chip, part of the hard drive
  - Usually 8-16MB
- Different from cache that OS has
  - Unlike the OS cache, it is aware of the disk geometry
  - When reading a sector, may cache the whole track to speed up future reads on the same track



# Disks and the OS

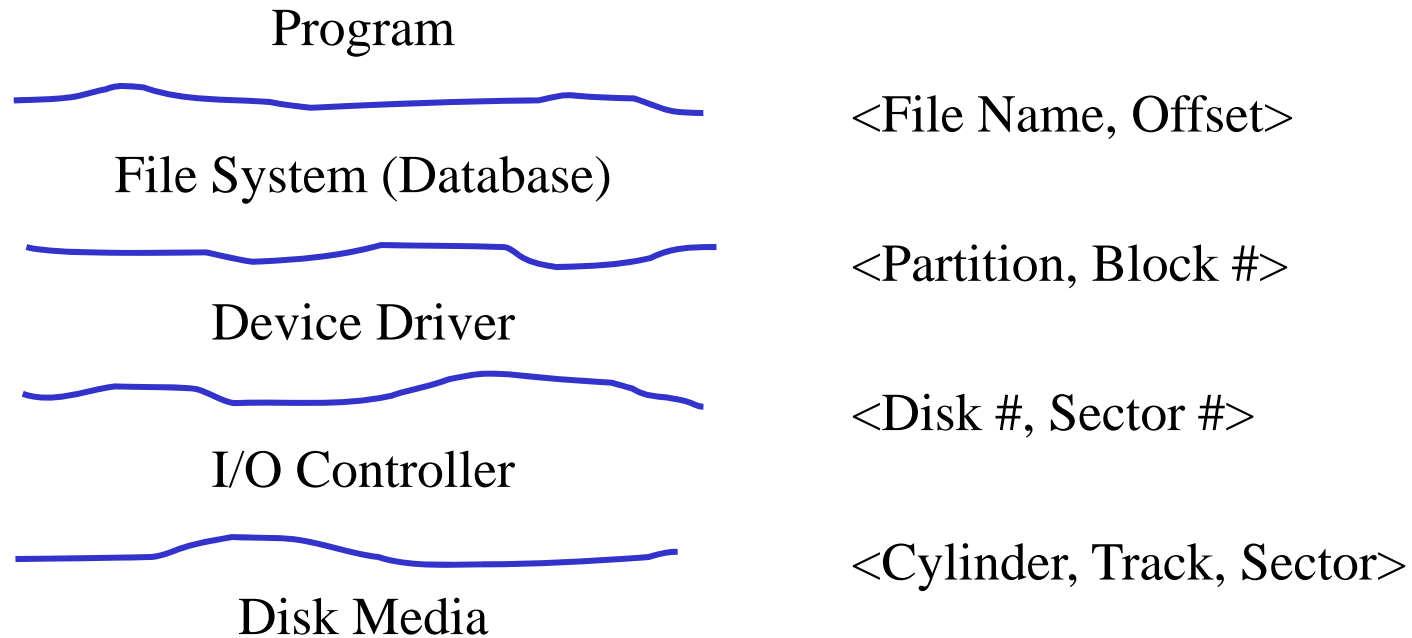
---

- Disks are messy physical devices:
  - Errors, bad blocks, missed seeks, etc.
- The job of the OS is to hide this mess from higher level software
  - Low-level device control (initiate a disk read, etc.)
  - Higher-level abstractions (files, databases, etc.)
- The OS may provide different levels of disk access to different clients
  - Physical disk (surface, cylinder, sector)
  - Logical disk (disk block #)
  - Logical file (file block, record, or byte #)



# Software Interface Layers

---



- Each layer abstracts details below it for layers above it





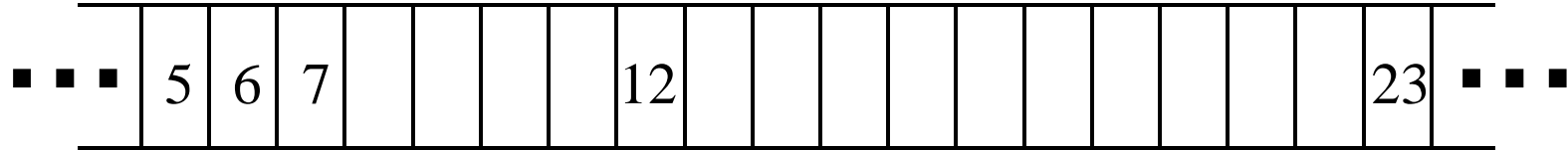
# Disk Interaction

- Specifying disk requests requires a lot of info:
  - Cylinder #, surface #, track #, sector #, transfer size...
- Older disks required the OS to specify all of this
  - The OS needed to know all disk parameters
- Modern disks are more complicated
  - Not all sectors are the same size, sectors are remapped, etc.
- Current disks provide a **higher-level interface (SCSI)**
  - The disk exports its data as a **logical array of blocks** [0...N]
    - Disk maps logical blocks to cylinder/surface/track/sector
  - Only need to **specify the logical block # to read/write**
  - But now the disk parameters are hidden from the OS



# The common storage device interface

---



OS's view of storage device

Storage exposed as **linear array of blocks**

Common block size: 512 bytes

Number of blocks: device capacity / block size



# Back to File Systems...

---

- Key idea: File systems need to be **aware of disk characteristics** for performance
  - **Allocation algorithms** to enhance performance
  - **Request scheduling** to reduce seek time



# Enhancing achieved disk performance

---

- High-level disk characteristics yield two goals:
  - **Closeness**
    - reduce seek times by putting related things close to each other
    - generally, benefits can be in the factor of 2 range
  - **Amortization**
    - amortize each positioning delay by grabbing lots of useful data
    - generally, benefits can reach into the factor of 10 range



# Allocation strategies

---

- Disks perform best if seeks are reduced and large transfers are used
  - Scheduling requests is one way to achieve this
  - Allocating related data “close together” on the disk is even more important



# FFS: a disk-aware file system



# Original Unix File System

- Recall FS sees storage as linear array of blocks
  - Each block has a *logical block number* (LBN)

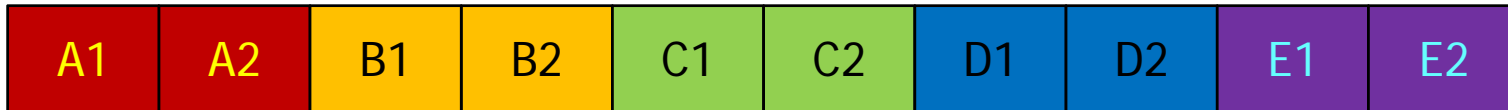


- Simple, straightforward implementation
  - Easy to implement and understand
  - But very poor utilization of disk bandwidth. *Why?*

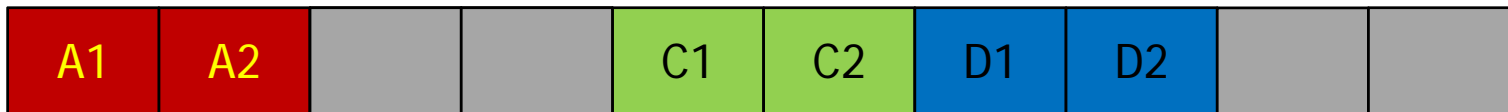


# Data and Inode Placement – problem #1

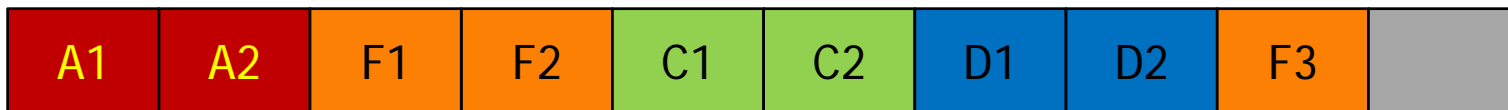
- On a new FS, blocks are allocated sequentially, close to each other.



- As the FS gets older, files are being deleted and create random gaps



- In aging file systems, data blocks end up allocated far from each other:

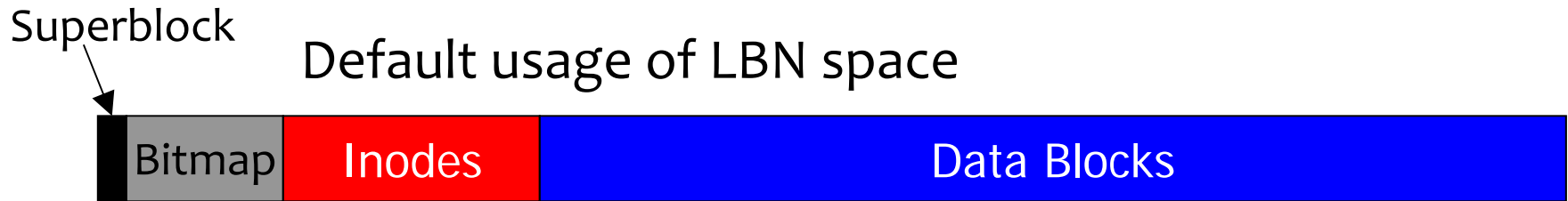


- Data blocks for new files end up scattered across the disk!
- Fragmentation of an aging file system causes more seeking!





# Data and Inode Placement – problem #2



- Inodes allocated far from blocks
  - All inodes at beginning of disk, far from data
- Recall that when we traverse a file path, at each level we inspect the inode first, then access the data block.
  - Traversing file name paths, manipulating files, directories requires going back and forth from inodes to data blocks
- => Again, lots of seeks!



# FFS

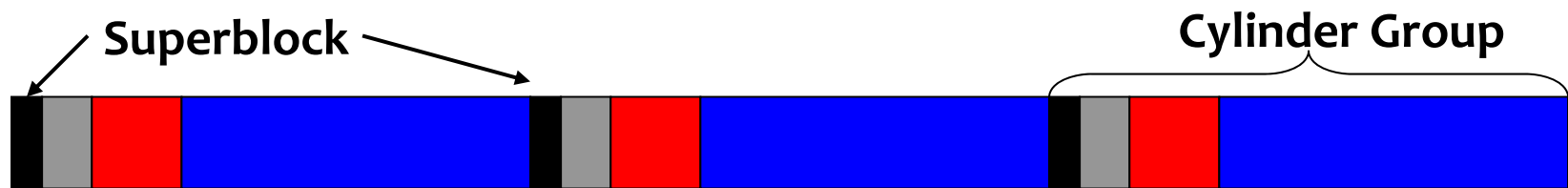
---

- BSD Unix folks did a redesign (early-mid 80s?) that they called the Fast File System (FFS)
  - Improved disk utilization, decreased response time
  - McKusick, Joy, Leffler, and Fabry, ACM TOCS, Aug. 1984
- A major breakthrough in the history of File Systems
- All modern FSs draw from the lessons learned from FFS
- **Good example of being device-aware for performance!**



# Cylinder Groups

- BSD FFS addressed placement problems using the notion of a **cylinder group** (aka *allocation groups* in lots of modern FS's)
  - Disk partitioned into groups of cylinders
  - Data blocks in same file allocated in same cylinder group
  - Files in same directory allocated in same cylinder group
  - Inodes for files are allocated in same cylinder group as file data blocks



Cylinder group organization



# Cylinder Groups (continued)

---

- Allocation in cylinder groups provides *closeness*
  - Reduces number of long seeks
- Free space requirement
  - To be able to allocate according to cylinder groups, the disk must have free space scattered across cylinders
  - 10% of the disk is reserved just for keeping the disk partially free all the time
  - When allocating large files, break it into large chunks and allocate from different cylinder groups, so it does not fill up one cylinder group
  - If preferred cylinder group is full, allocate from a “nearby” group



# More FFS solutions

---

- Small blocks (1K) in orig. Unix FS caused 2 problems:
  - Low bandwidth utilization
  - Small max file size (function of block size)
- Fix using a larger block (4K)
  - Very large files, only need two levels of indirection for  $2^{32}$
  - New Problem: internal fragmentation
  - Fix: Introduce “fragments” (1K pieces of a block)
- Problem: Media failures
  - Replicate master block (superblock)
- Problem: Device oblivious
  - Parameterize according to device characteristics



# Disk Scheduling Algorithms

- Because seeks are so expensive (milliseconds!), OS tries to schedule disk requests that are queued waiting for the disk
- Goal: Minimize seeks!
- Policies:
  - FCFS (do nothing)
    - Reasonable when load is low
    - Long waiting times for long request queues
  - SSTF (shortest seek time first)
    - Minimize arm movement (seek time), maximize request rate
    - Favors middle blocks
  - SCAN (elevator)
    - Service requests in one direction until done, then reverse
  - C-SCAN
    - Like SCAN, but only go in one direction (typewriter)



# Disk Scheduling Algorithms(2)

---

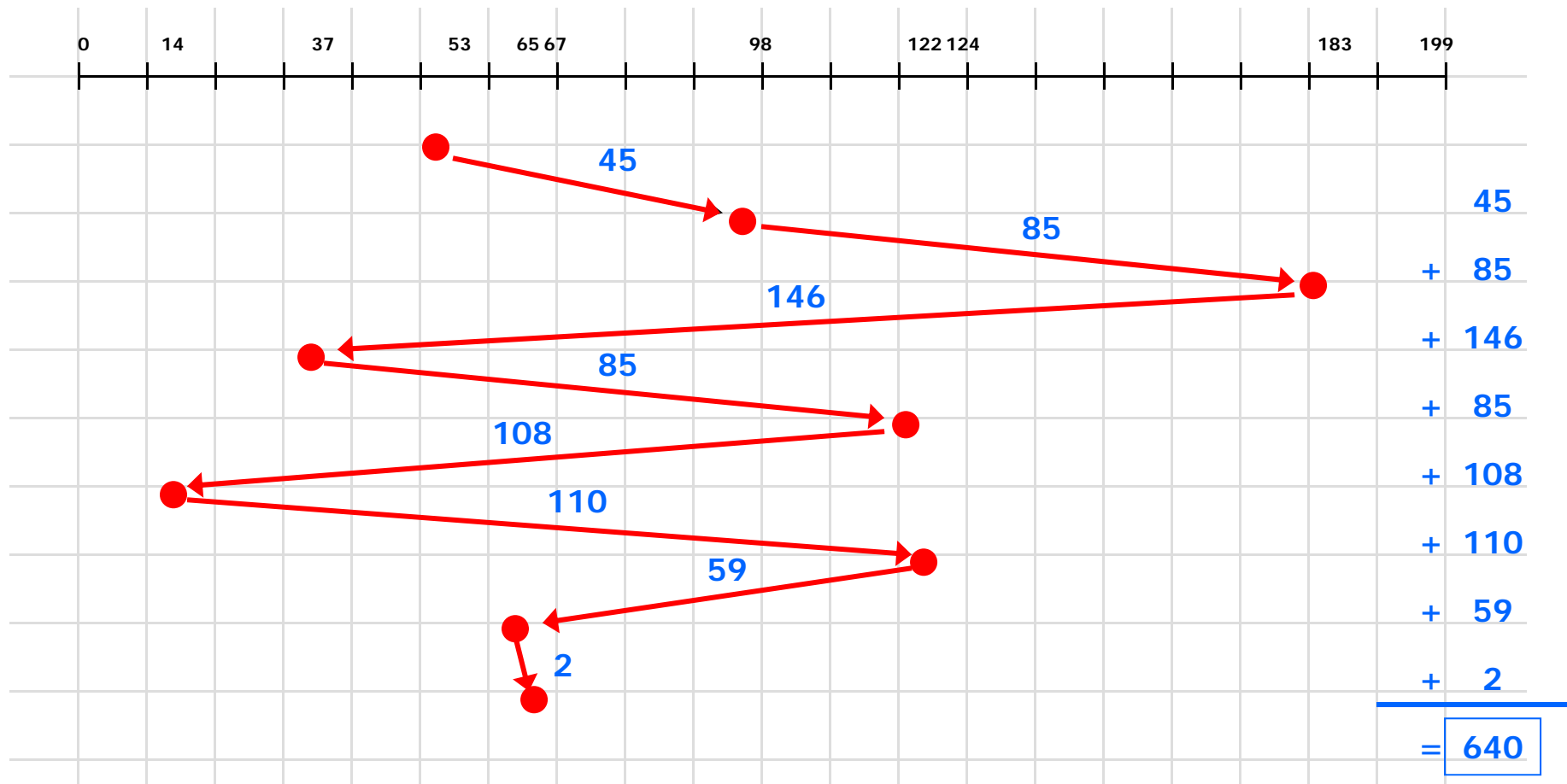
- LOOK / C-LOOK
  - Like SCAN/C-SCAN but only go as far as last request in each direction (not full width of the disk)
- In general, unless there are request queues, disk scheduling does not have much impact
  - Important for servers, less so for PCs
- Modern disks often do the disk scheduling themselves
  - Disks know their layout better than OS, can optimize better
  - If so, ignores/undoes any scheduling done by OS



# Example: FCFS

Head at 53

98 183 37 122 14 124 65 67







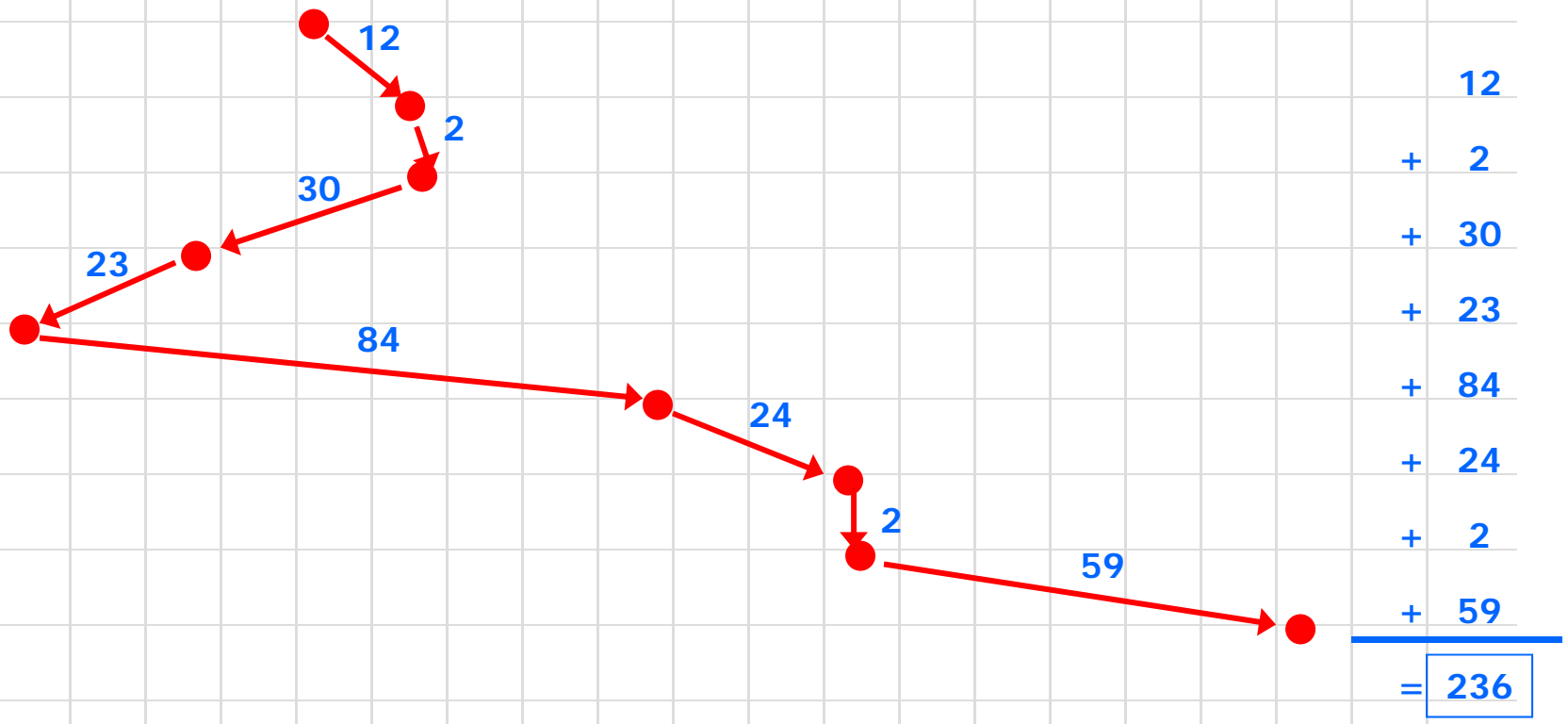
# Example: SSTF

Head at 53

98 183 37 122 14 124 65 67



0 14 37 53 65 67 98 122 124 183 199

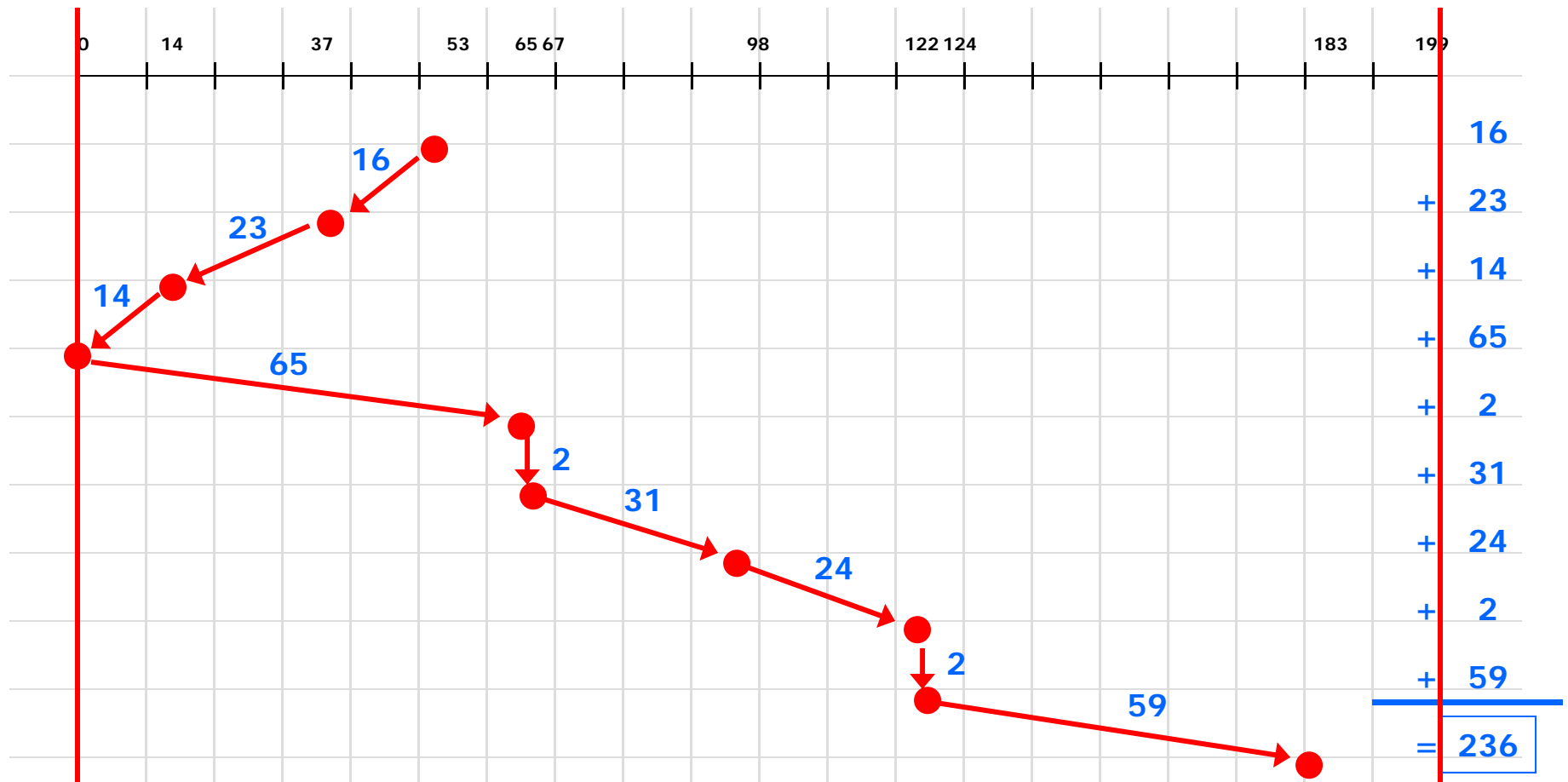




# Example: Scan

Head at 53

98 183 37 122 14 124 65 67

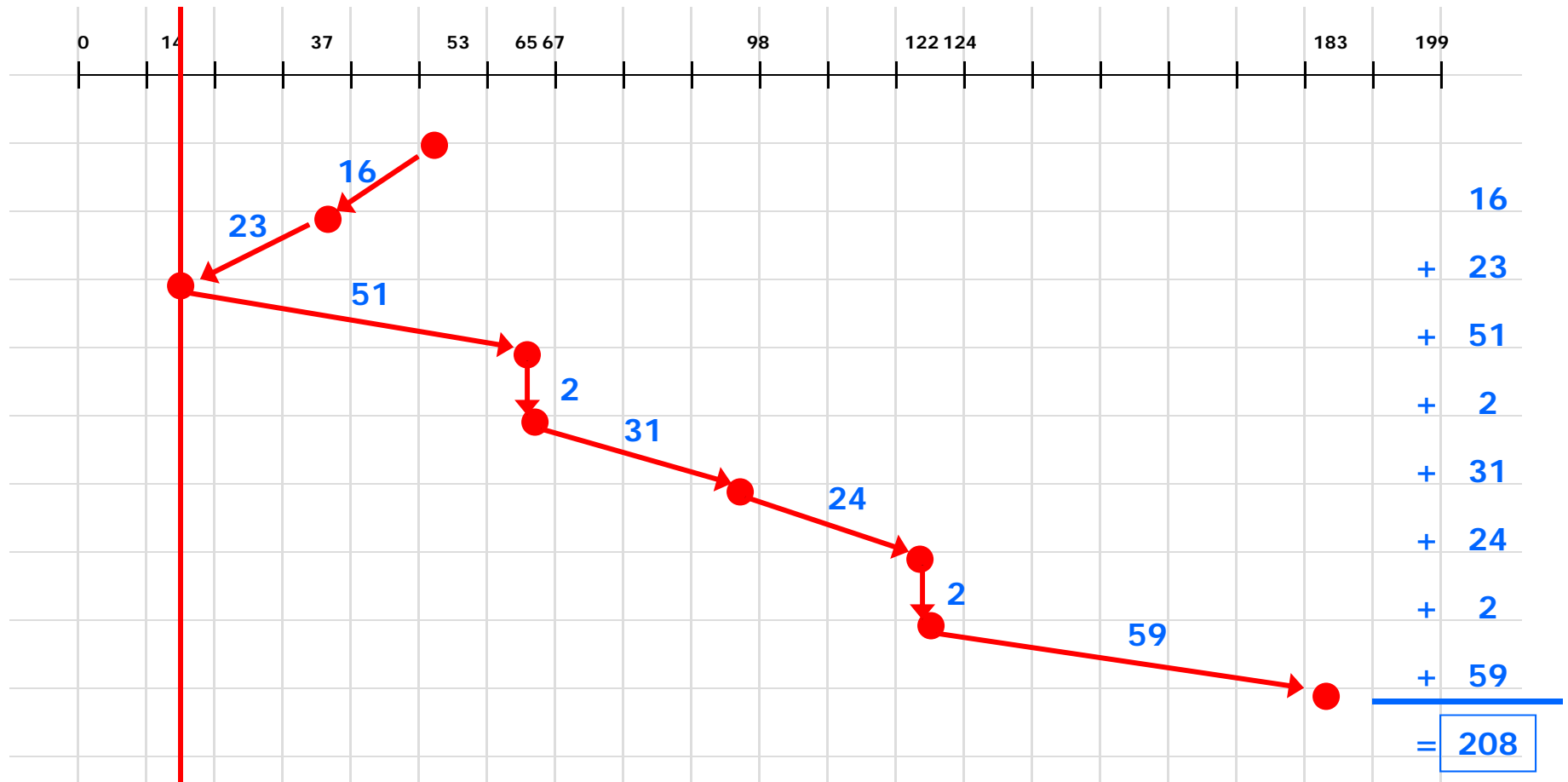




# Example: Look

Head at 53

98 183 37 122 14 124 65 67





# Summary

---

- File systems overview
  - Disks are large, persistent, but **slow**
  - Operations on files and directories, sharing
- File system organization
  - VSFS example, ext2, other strategies and design choices
- Performance enhancements: caching, read-ahead
- Disks
  - Physical structure
  - Placement problems and strategies, FFS
  - Disk scheduling algorithms



# Next time...

---

- More on File Systems
- Examples of modern file systems
  - Log-structured file system
  - Ext3
  - NTFS (time permitting..)



# Announcements

---

- Assignment 4!
- Three tutorial exercises to help you get started
  - Tutorial exercises 7 and 8 (due next week)
  - Tutorial exercise 9 (due the week after)
  - Work in pairs with your assignment partner (some code may be similar)
  - Start early, don't wait until the deadline for the exercises
    - If you do, you won't finish A4 on time!!