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

How many disks blocks are accessed?

Let's say you want to open the file "/one/two/three"

1. Master block for the location of inode for "/"
2. Directory block for "/" - search for "one"
3. Inode block containing inode for "one"
4. Directory block for "one" - search for "two"
5. Inode block containing inode for "two"
6. Directory block for "two" - search for "three"
7. Inode block for "three"

- Now we can read the first block of file "three"
- This is why we need **`open`**

Disk Operations Excercise

- Let's look at a text-based representation of a file system:

```
inode bitmap 11110000

inodes       [d a:0 r:3] [f a:1 r:1] [f a:-1 r:1] [d a:2 r:2] [] …

data bitmap  11100000

data         [(.,0) (..,0) (y,1) (z,2) (f,3)] [u] [(.,3) (..,0)]
             [] ...
```

This FS has 4 inodes in use, and 3 data blocks in use

Disk Operations Excercise

- Let's look at a text-based representation of a file system:

```
inode bitmap 11110000

inodes       [d a:0 r:3] [f a:1 r:1] [f a:-1 r:1] [d a:2 r:2] [] …

data bitmap  11100000

data         [(.,0) (..,0) (y,1) (z,2) (f,3)] [u] [(.,3) (..,0)]
             [] ...
```

The first inode [d a:0 r:3] is the inode for the root directory.

a:0 - the address is data block 0

r: 3 - there are 3 references to this directory

Disk Operations Excercise

- Let's look at a text-based representation of a file system:

```
inode bitmap 11110000

inodes       [d a:0 r:3] [f a:1 r:1] [f a:-1 r:1] [d a:2 r:2] [] …

data bitmap  11100000

data         [(.,0) (..,0) (y,1) (z,2) (f,3)] [u] [(.,3) (..,0)]
             [] ...
```

Disk Operations Excercise

- Let's look at a text-based representation of a file system:

```
inode bitmap 11110000

inodes       [d a:0 r:3] [f a:1 r:1] [f a:-1 r:1] [d a:2 r:2] [] …

data bitmap  11100000

data         [(.,0) (..,0) (y,1) (z,2) (f,3)] [u] [(.,3) (..,0)]
             [] ...
```

Disk Operations Excercise

- Let's look at a text-based representation of a file system:

```
inode bitmap 11110000

inodes       [d a:0 r:3] [f a:1 r:1] [f a:-1 r:1] [d a:2 r:2] [] …

data bitmap  11100000

data         [(.,0) (..,0) (y,1) (z,2) (f,3)] [u] [(.,3) (..,0)]
             [] ...
```

File z is empty. The inode does not point to any data blocks

Disk Operations Excercise

- Let's look at a text-based representation of a file system:

```
inode bitmap 11110000

inodes       [d a:0 r:3] [f a:1 r:1] [f a:-1 r:1] [d a:2 r:2] [] …

data bitmap  11100000

data         [(.,0) (..,0) (y,1) (z,2) (f,3)] [u] [(.,3) (..,0)]
             [] ...
```

File y is using one data block (at index 1). For this exercise we don't care what the content is.

Quercus exercise

- (or on paper…)

Disk I/O
File System Optimization

CSC369

Karen Reid

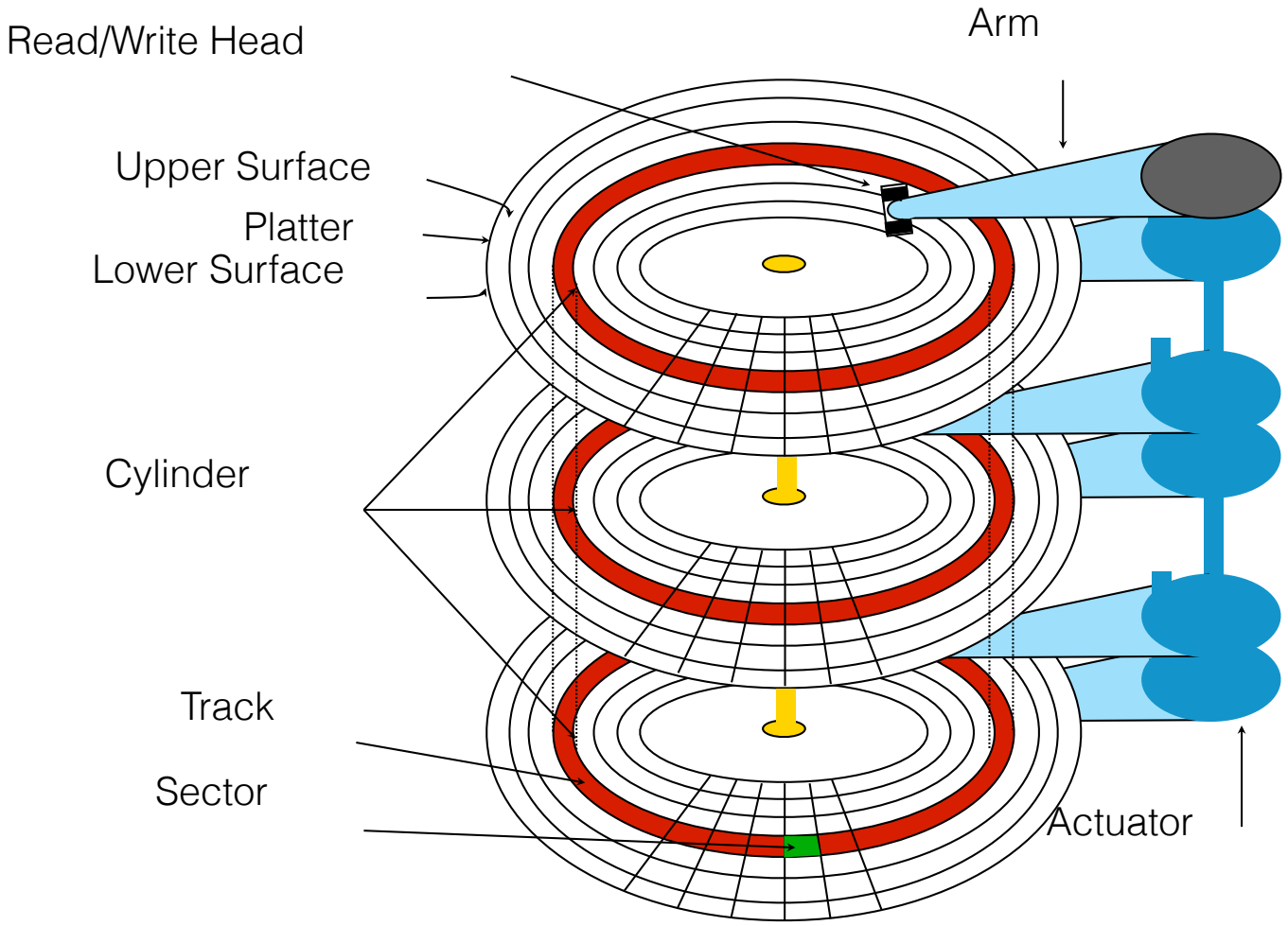
Overview

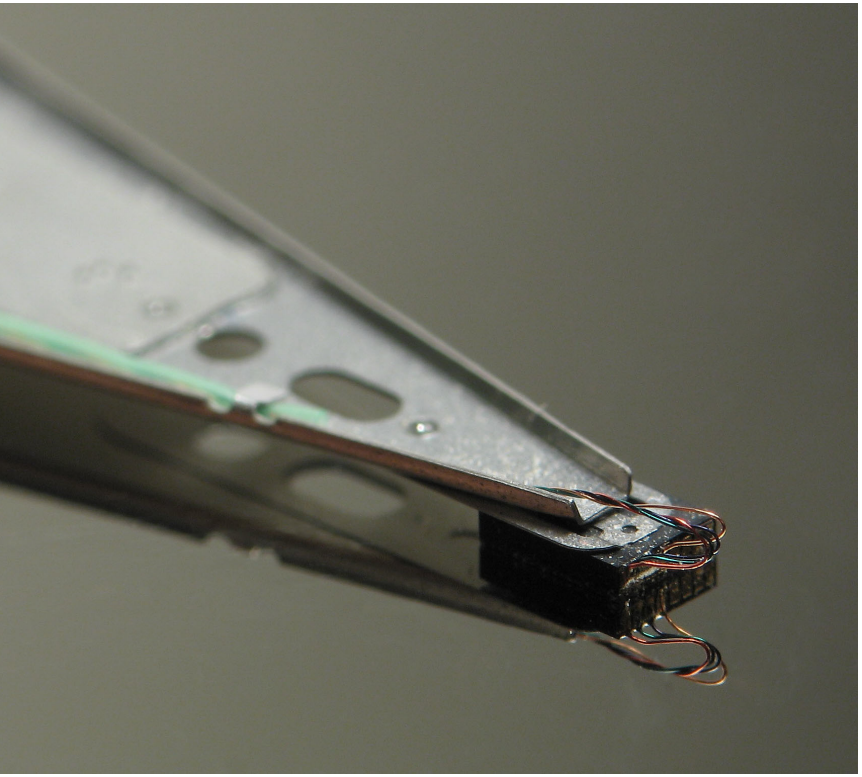
- Last time:
  - File systems – Interface, structure, basic implementation
- Today:
  - Magnetic disks as secondary storage
  - Disk management
  - File system optimizations

Secondary Storage Devices

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

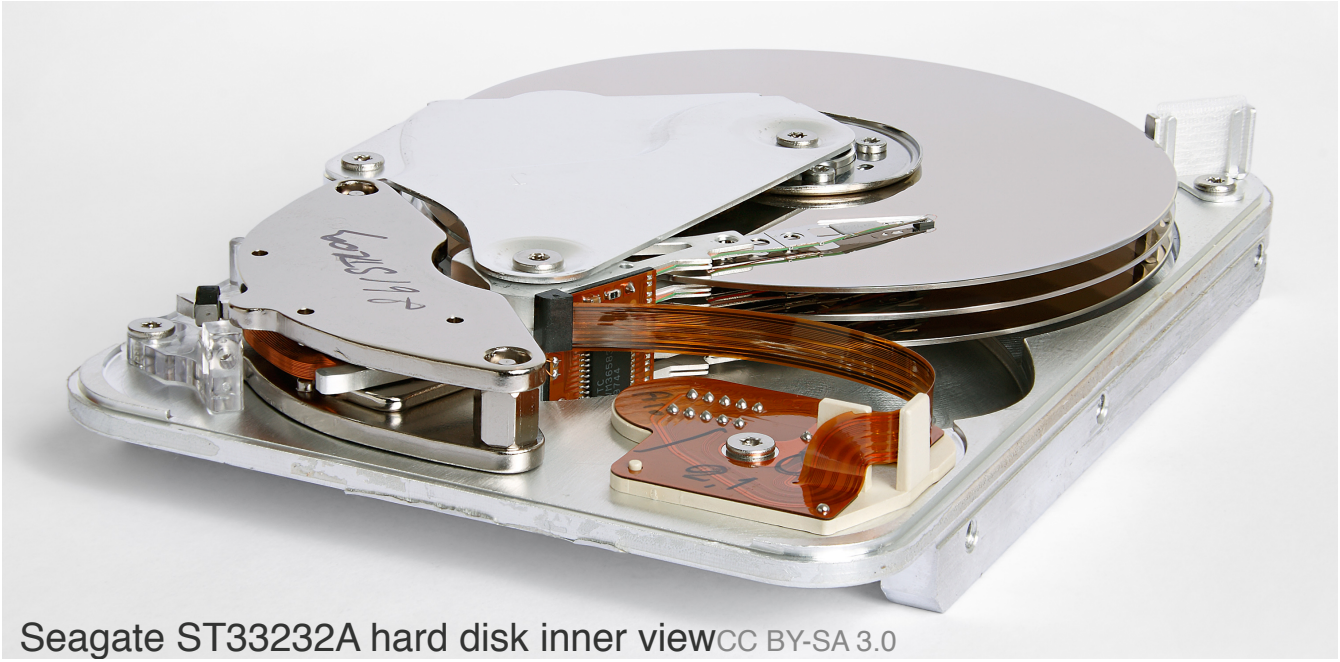
Disk Components





Read/Write Head © Wikipedia Commons

Seagate ST33232A hard disk head and platters detail

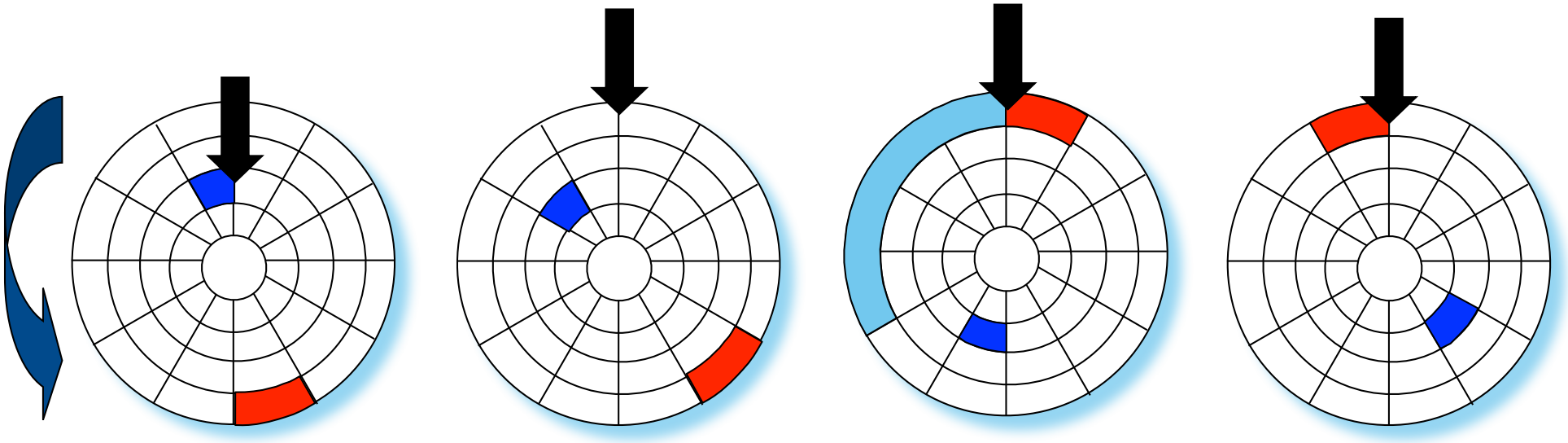


Seagate ST33232A hard disk inner view

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 ½ rotation (~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)
    - ~100 MB/s, average sector transfer time of ~5us
    - improving rapidly (~40% per year)

Traditional service time components



After BLUE read

Seek for RED

Rotational latency

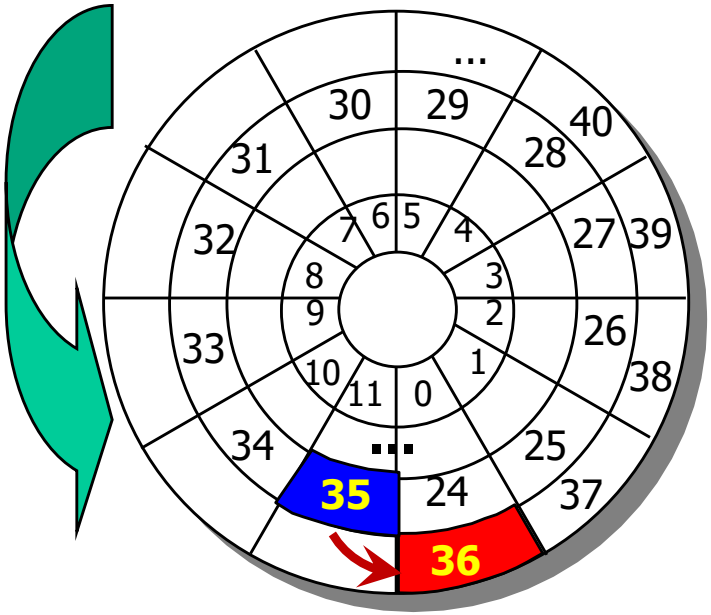
After RED read

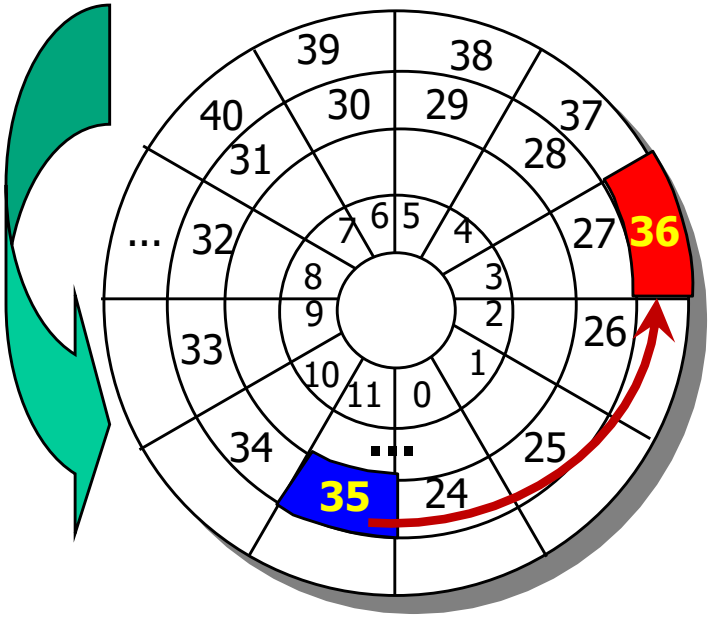
- When the OS uses the disk, it tries to minimize the cost of all of these steps
  - Particularly seeks and rotation

Some Hardware Optimizations

- Track Skew
- Zones
- Cache

Track Skew

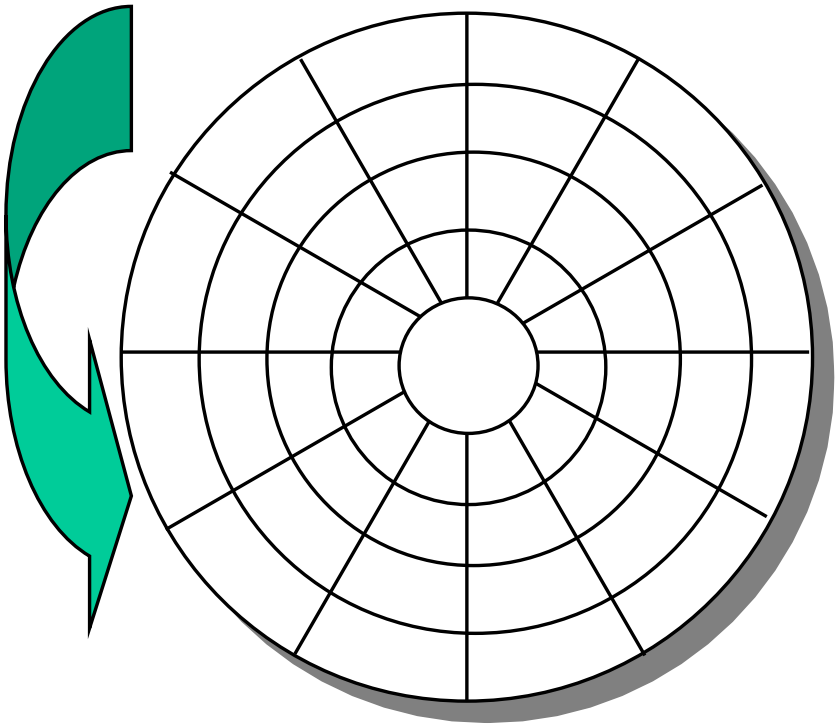




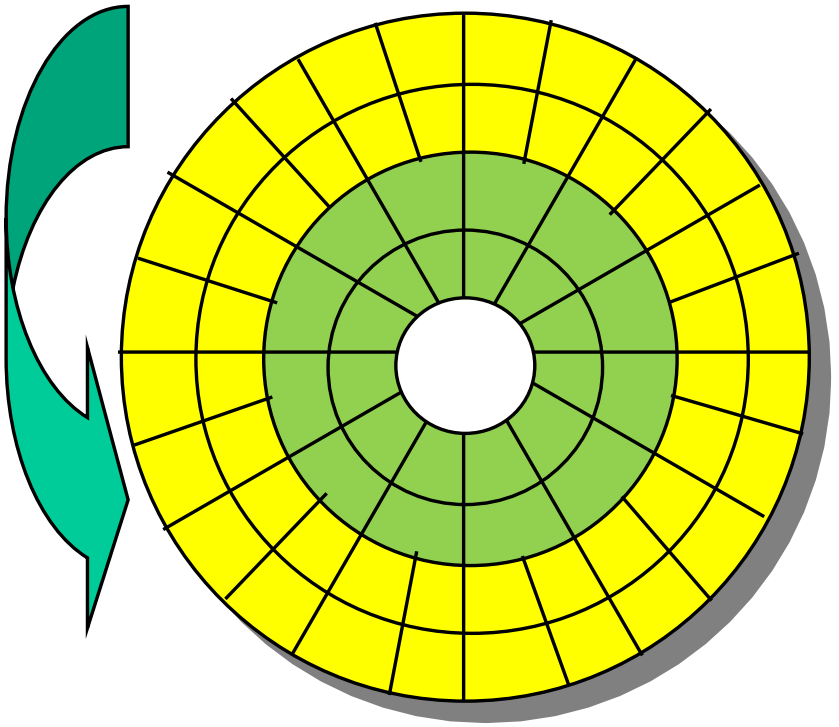
If the arm moves to outer track too slowly, may miss sector 36 and have to wait for a whole rotation.

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 OS cache
  - 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

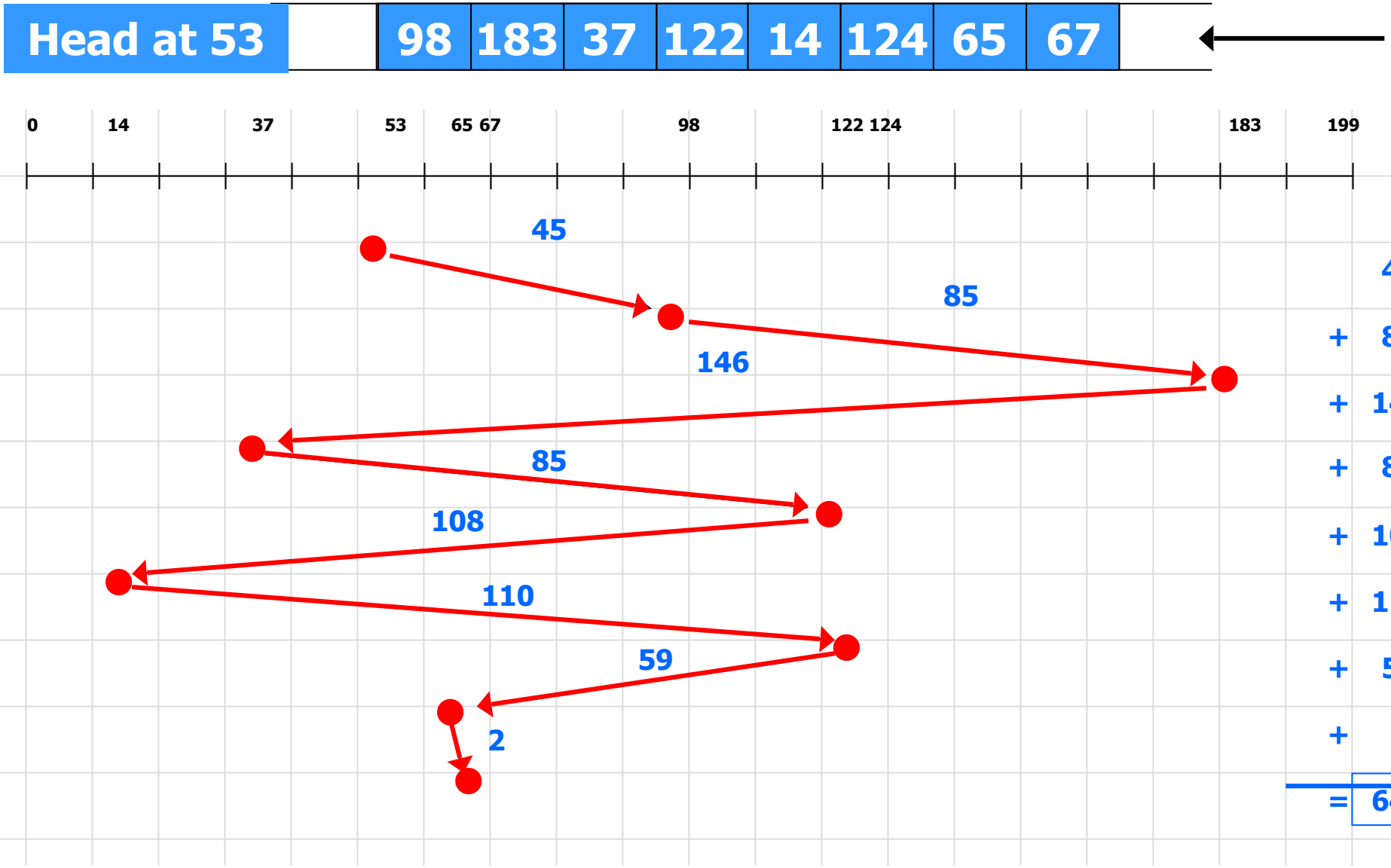
Disks and the OS

- 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 #)

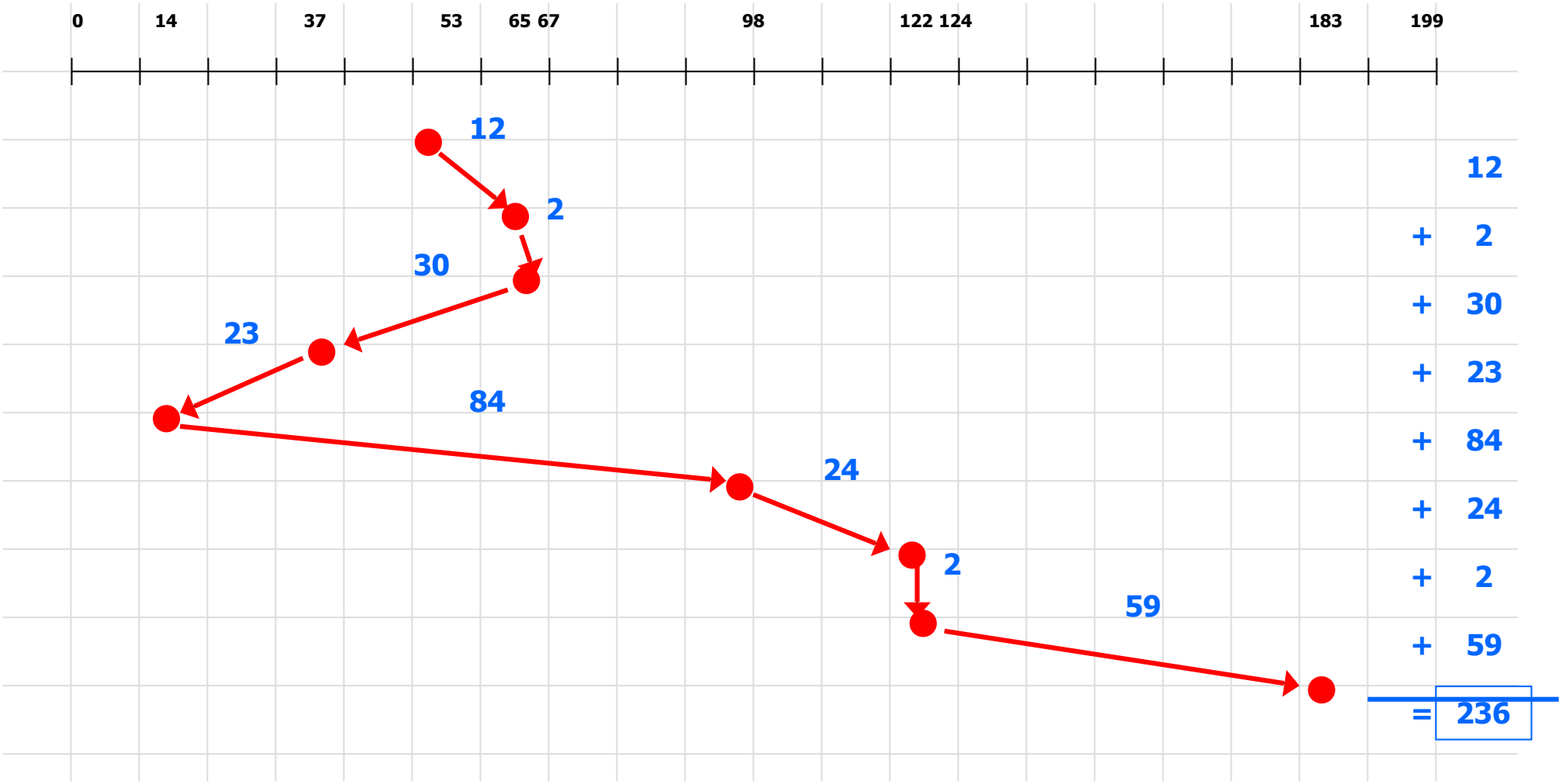
Disk Scheduling

- Because seeks are so expensive (milliseconds!), the OS tries to schedule disk requests that are queued waiting for the disk
  - 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)

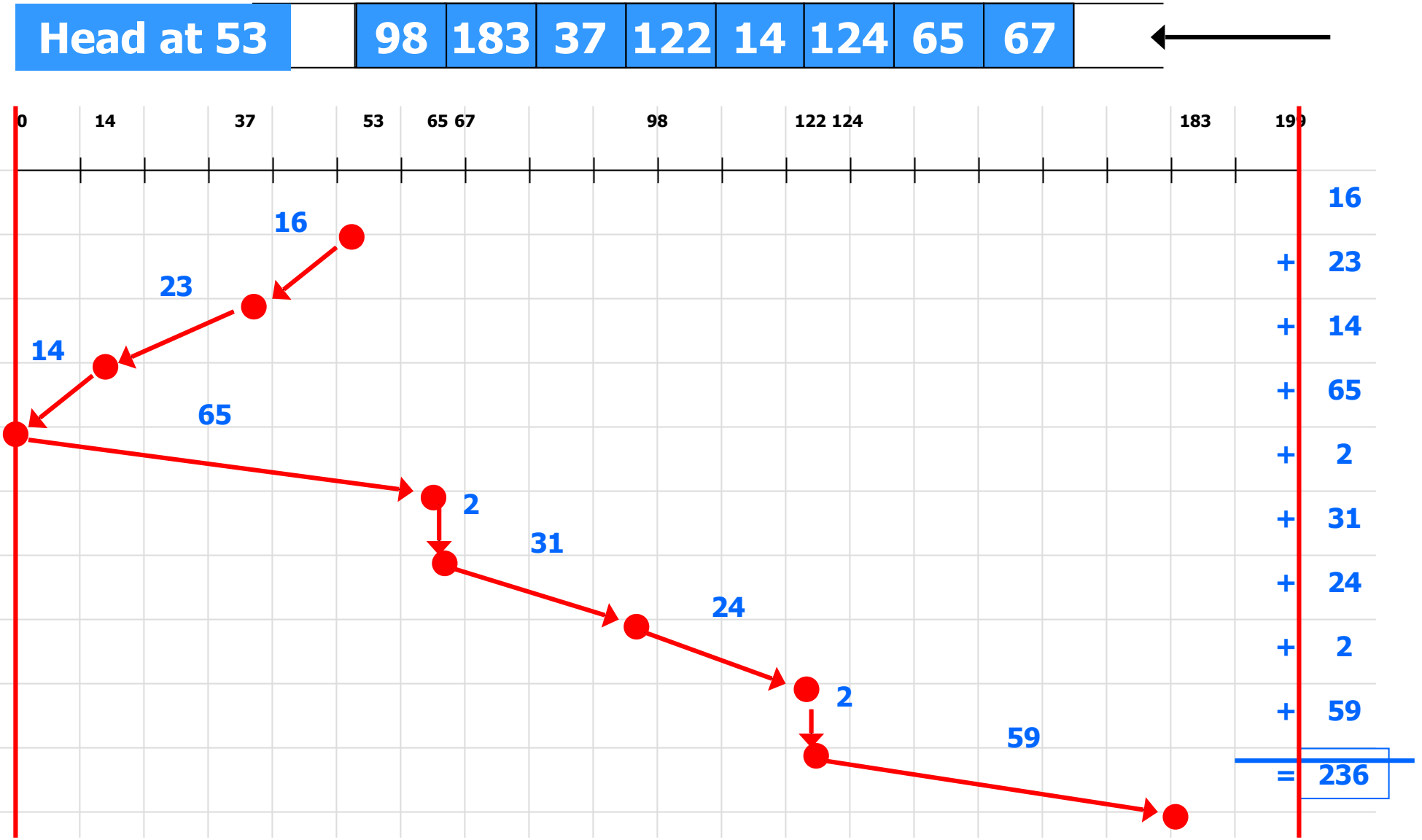
Example: FCFS



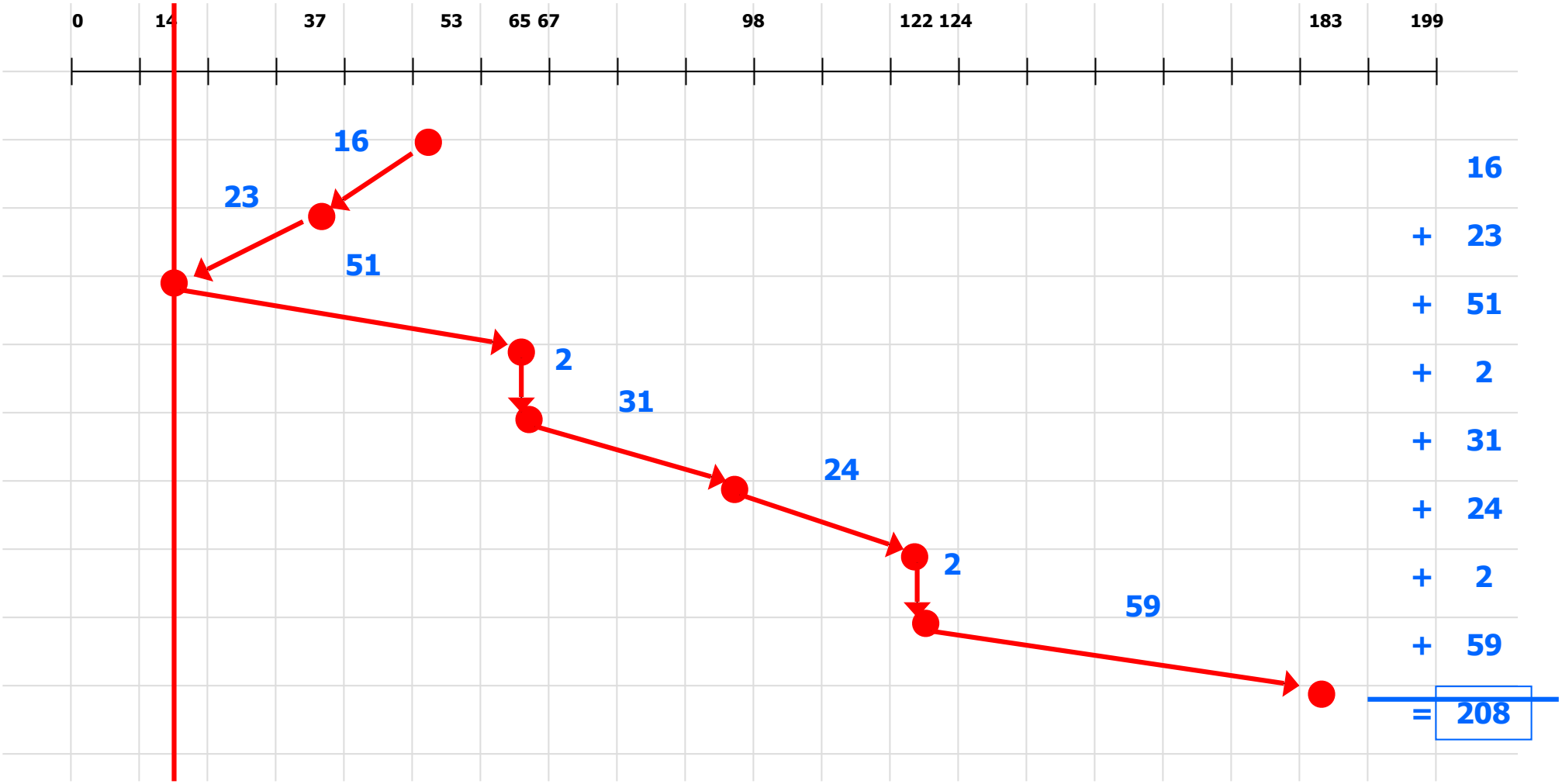
Example: SSTF



Example: Scan



Example: Look



Disk Scheduling (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)

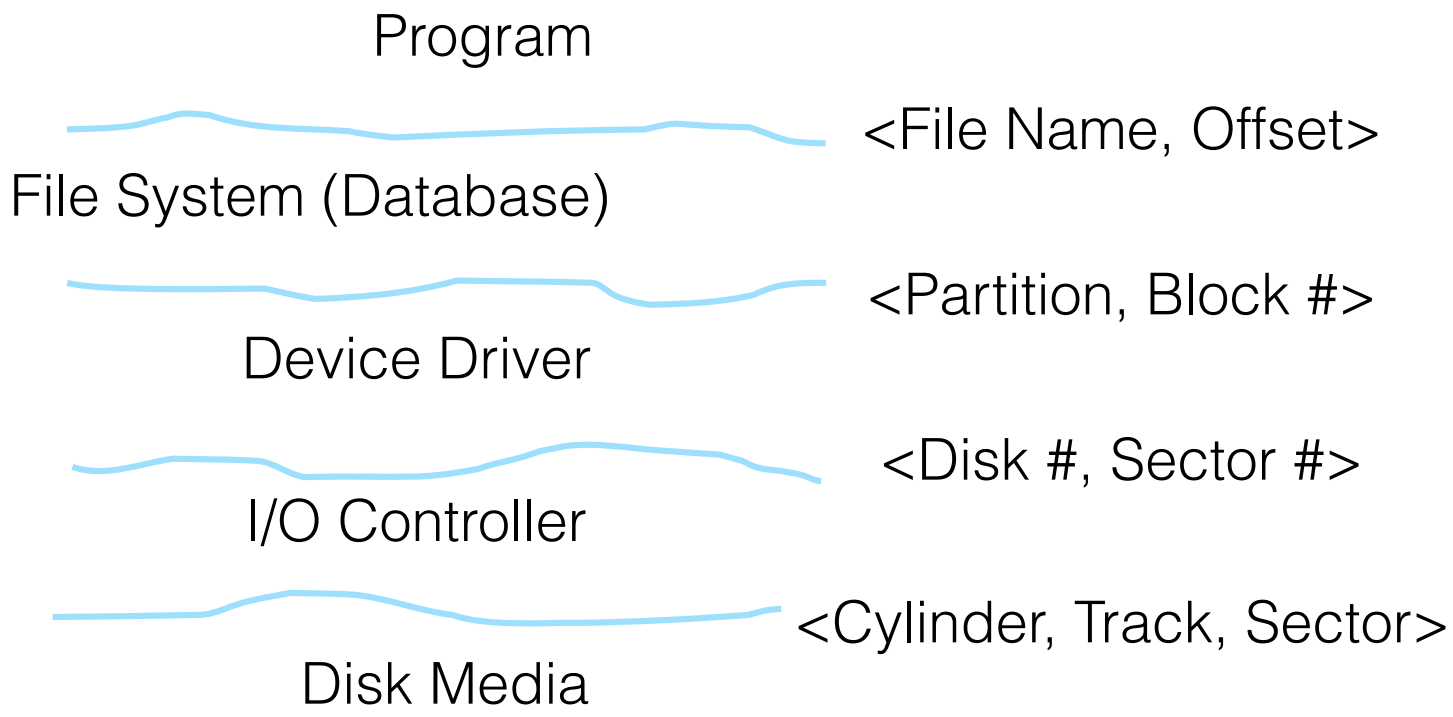
- In general, unless there are request queues, disk scheduling does not have much impact
  - Important for servers, less so for PCs

- Modern disks often do the disk scheduling themselves
  - Disks know their layout better than OS, can optimize better
  - Ignores, undoes any scheduling done by OS

Summary

- I/O overview
- Memory hierarchy
- Secondary storage
  - Large, persistent, but slow
- Disks
  - Physical structure
  - Interface
  - Performance
  - Scheduling

Software Interface Layers

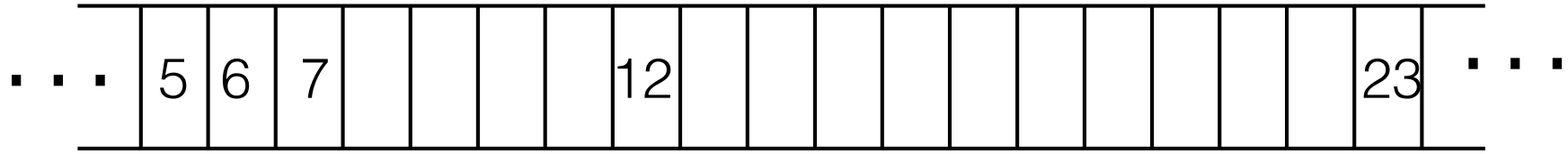


- Each layer abstracts details below it for layers above it
  - Naming and address mapping
  - Caching and request transformations

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 sizes: 512 bytes, 4096 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

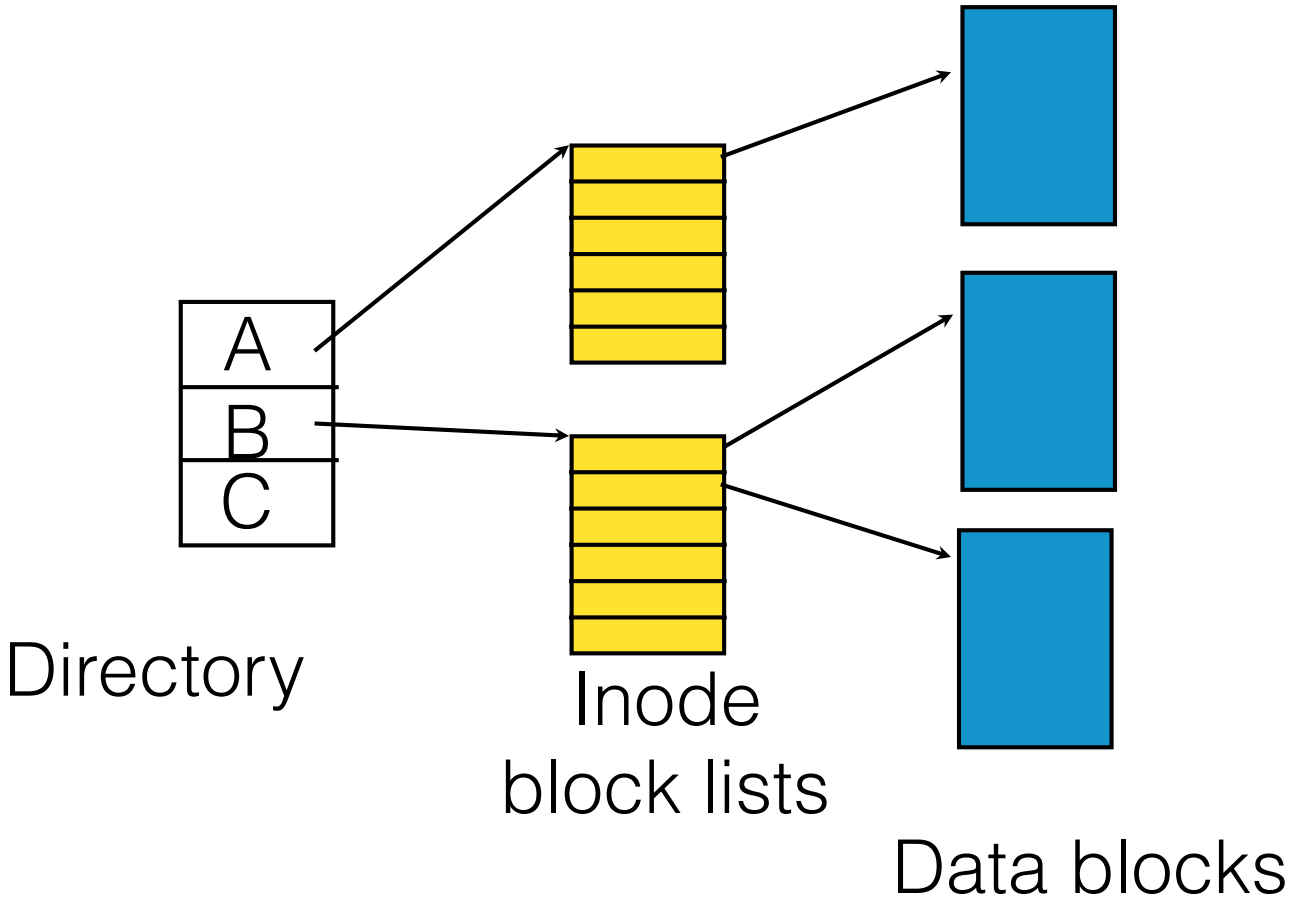
Enhancing disk performance

- High-level disk characteristics yield two goals:
  - Closeness
    - reduce seek times by putting related things close to one another
    - 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

Inodes: Indirection & Independence

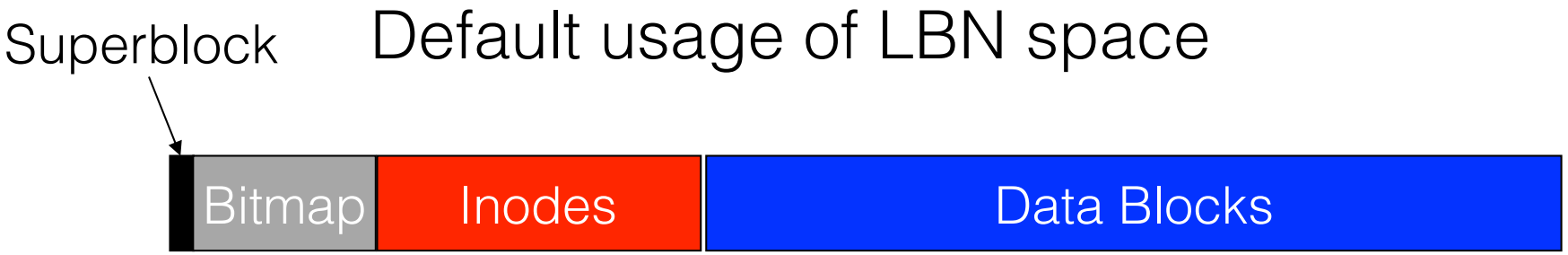


+ File size grows dynamically, allocations are independent
- Hard to achieve closeness and amortization

FFS: A disk-aware file system

Original Unix File System

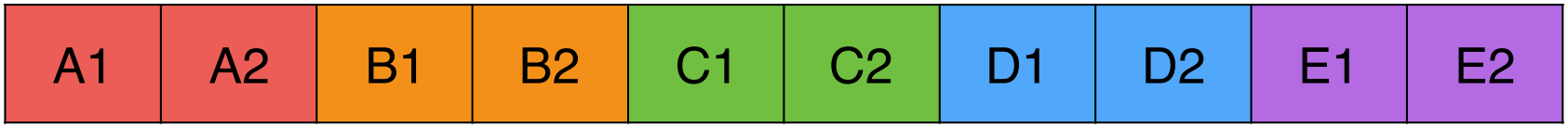
- Recall FS sees storage as linear array of blocks
- Each block has a logical block number (LBN)



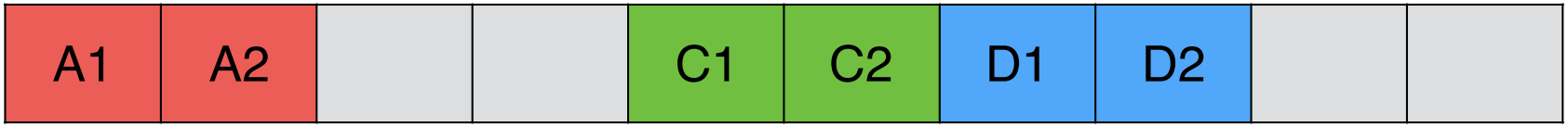
- Simple, straightforward implementation
  - Easy to implement and understand
  - But very poor utilization of disk bandwidth (lots of seeking)

Data and Inode Placement: Problem 1

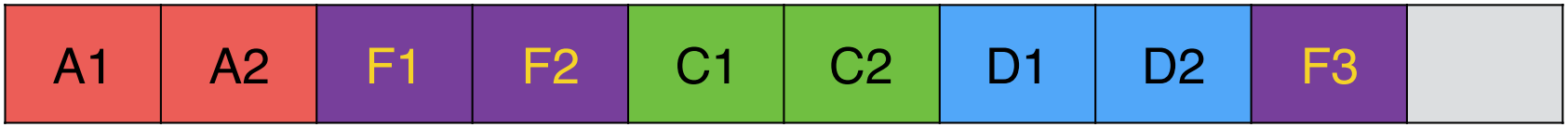
- On a new FS, blocks are allocated sequentially, close to each other.



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

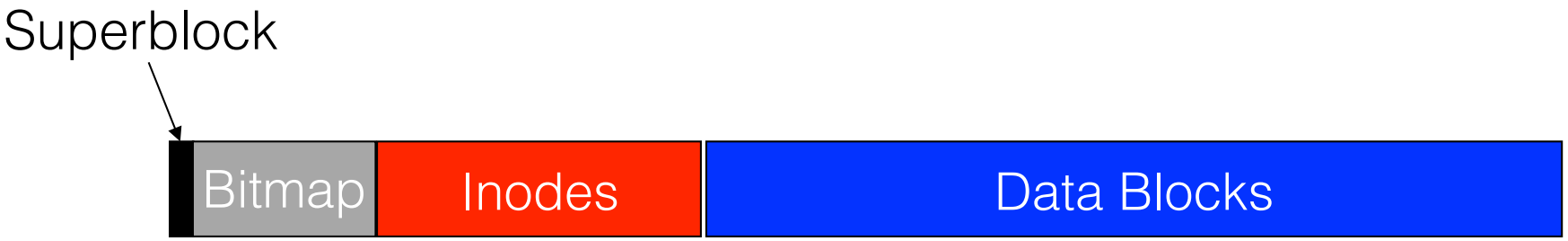


- In aging file systems, data blocks end up allocated far from each other:



- 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 (BSD 4.2) that they called the Fast File System (FFS)
  - Improved disk utilization, decreased response time
  - McKusick, Joy, Leffler, and Fabry, ACM TOCS, Aug. 1984
- Now the FS from which all other Unix FS's have been compared
- 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 allocated in same cylinder group as file data blocks

Cylinder group organization

Cylinder Groups (cont'd)

- 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 this purpose
  - When allocating a large file, 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