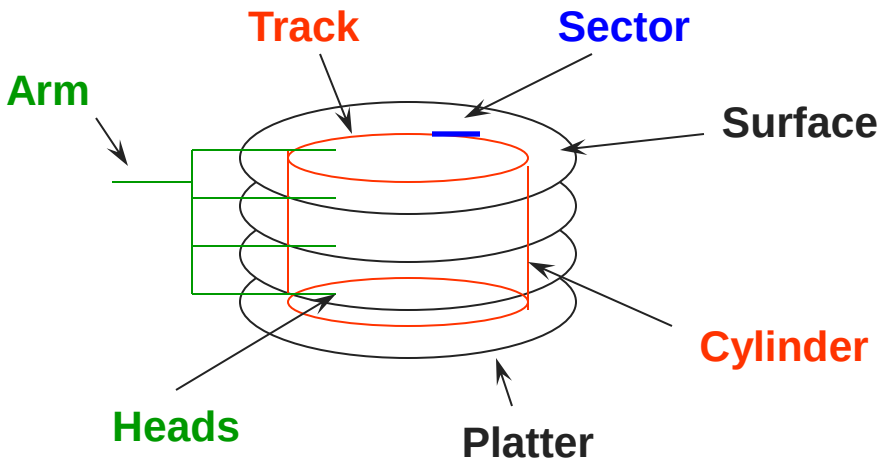
# CS 153
# Design of Operating Systems

## Winter 2016

Lecture 21: File system optimizations

# Physical Disk Structure

- Disk components
  - Platters
  - Surfaces
  - Tracks
  - Sectors
  - Cylinders
  - Arm
  - Heads



# Cylinder Groups

- BSD FFS addressed these problems using the notion of a cylinder group
  - 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
  - Same for inodes
- 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
    - » Only used by root – this is why “df” may report >100%

# Other Problems

- Small blocks (1K) caused two problems:
  - Low bandwidth utilization
  - Small max file size (function of block size)
- Fix: Use a larger block (4K)
  - Very large files, only need two levels of indirection for 2^32
  - 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

# The Results

Table IIa. Reading Rates of the Old and New UNIX File Systems

| Type of file system | Processor and bus measured | Speed (Kbytes/s) | Read bandwidth % | % CPU |
|---|---|---|---|---|
| Old 1024 | 750/UNIBUS | 29 | 29/983 3 | 11 |
| New 4096/1024 | 750/UNIBUS | 221 | 221/983 22 | 43 |
| New 8192/1024 | 750/UNIBUS | 233 | 233/983 24 | 29 |
| New 4096/1024 | 750/MASSBUS | 466 | 466/983 47 | 73 |
| New 8192/1024 | 750/MASSBUS | 466 | 466/983 47 | 54 |

Table IIb. Writing Rates of the Old and New UNIX File Systems

| Type of file system | Processor and bus measured | Speed (Kbytes/s) | Write bandwidth % | % CPU |
|---|---|---|---|---|
| Old 1024 | 750/UNIBUS | 48 | 48/983 5 | 29 |
| New 4096/1024 | 750/UNIBUS | 142 | 142/983 14 | 43 |
| New 8192/1024 | 750/UNIBUS | 215 | 215/983 22 | 46 |
| New 4096/1024 | 750/MASSBUS | 323 | 323/983 33 | 94 |
| New 8192/1024 | 750/MASSBUS | 466 | 466/983 47 | 95 |

# Log-structured File System

- The Log-structured File System (LFS) was designed in response to two trends in workload and technology:
  1. Disk bandwidth scaling significantly (40% a year)
     - » While seek latency is not
  2. Large main memories in machines
     - » Large buffer caches
     - » Absorb large fraction of read requests
     - » Can use for writes as well
     - » Coalesce small writes into large writes
- LFS takes advantage of both of these to increase FS performance
  - Rosenblum and Ousterhout (Berkeley, 1991)

# LFS Approach

- Treat the disk as a single log for appending
  - Collect writes in disk cache, write out entire collection in one large disk request
    - » Leverages disk bandwidth
    - » No seeks (assuming head is at end of log)
  - All info written to disk is appended to log
    - » Data blocks, attributes, inodes, directories, etc.

- Looks simple, but only in abstract

# LFS Challenges

- LFS has two challenges it must address for it to be practical
  1. Locating data written to the log
     - » FFS places files in a location, LFS writes data “at the end”
  2. Managing free space on the disk
     - » Disk is finite, so log is finite, cannot always append
     - » Need to recover deleted blocks in old parts of log

# LFS: Locating Data

- FFS uses inodes to locate data blocks
  - Inodes pre-allocated in each cylinder group
  - Directories contain locations of inodes
- LFS appends inodes to end of the log just like data
  - Makes them hard to find
- Approach
  - Use another level of indirection: inode maps
  - inode maps maintain the location of each inode
  - inode map is itself divided into blocks that are written to the log
  - Fixed checkpoint region on disk stores locations of all inode maps
  - Cache inode maps in memory for performance

# LFS Layout

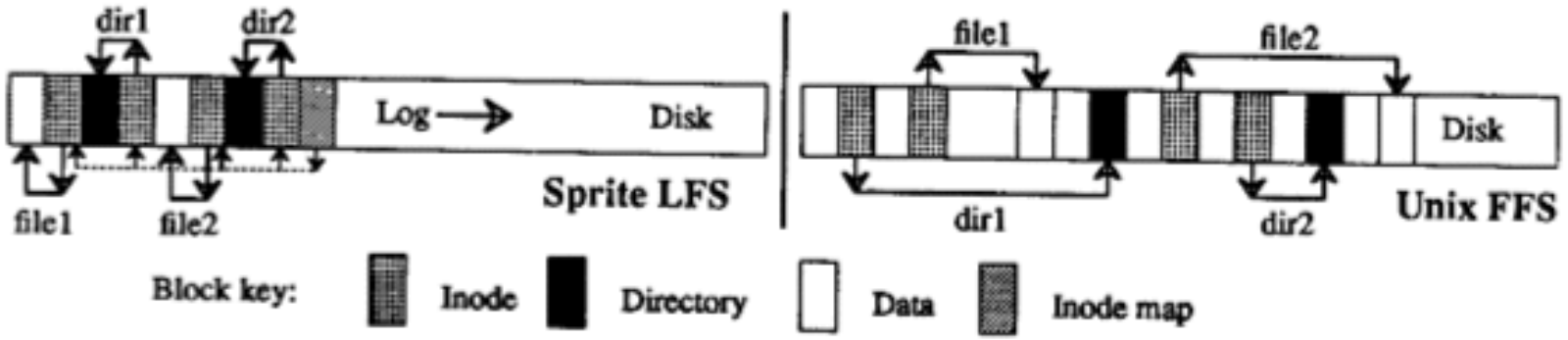
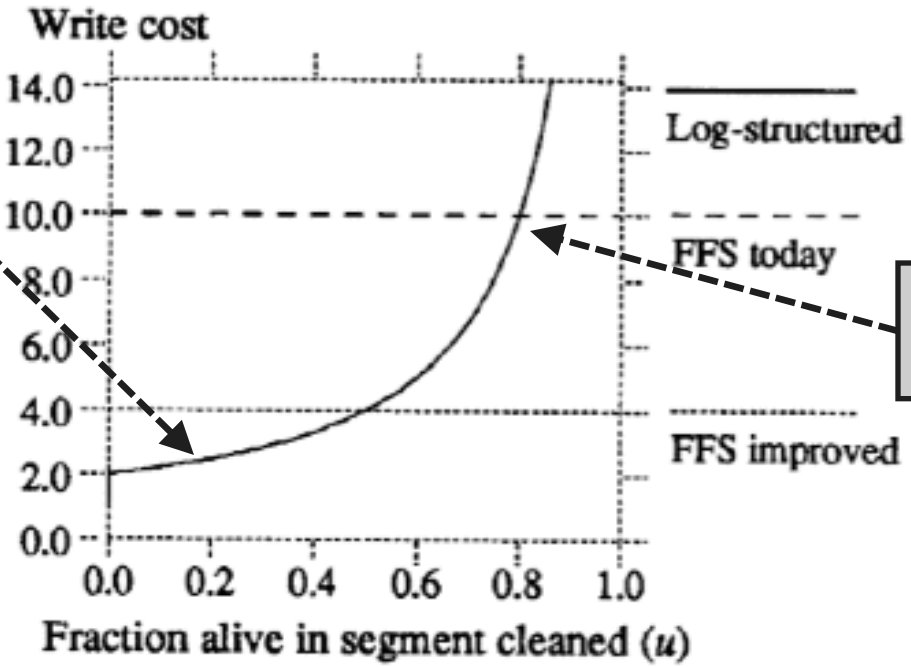



Fig. 1. A comparison between Sprite LFS and Unix FFS. This example shows the modified disk blocks written by Sprite LFS and Unix FFS when creating two single-block files named dir1/file1 and dir2/file2. Each system must write new data blocks and inodes for file1 and file2, plus new data blocks and inodes for the containing directories. Unix FFS requires ten nonsequential writes for the new information (the inodes for the new files are each written twice to ease recovery from crashes), while Sprite LFS performs the operations in a single large write. The same number of disk accesses will be required to read the files in the two systems. Sprite LFS also writes out new inode map blocks to record the new inode locations

# LFS: Free Space Management

- LFS append-only quickly runs out of disk space
  - Need to recover deleted blocks
- Approach:
  - Fragment log into segments
  - Thread segments on disk
    - » Segments can be anywhere
  - Reclaim space by cleaning segments
    - » Read segment
    - » Copy live data to end of log
    - » Now have free segment you can reuse
- Cleaning is a big problem
  - Costly overhead

# Write Cost Comparison




Write cost of 2 if 20% full

Write cost of 10 if 80% full

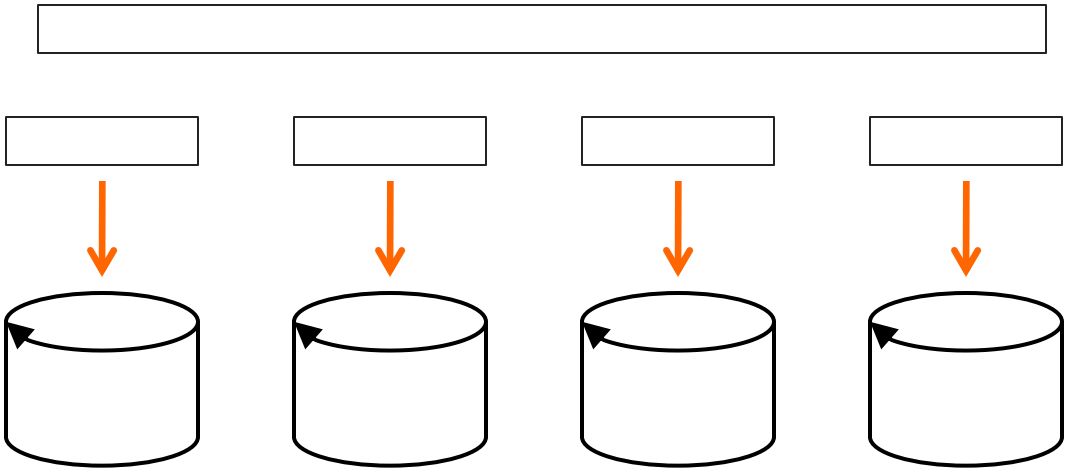
Fig. 3. Write cost as a function of $u$ for small files In a log-structured file system, the write cost depends strongly on the utilization of the segments that are cleaned. The more live data in segments cleaned, the more disk bandwidth that is needed for cleaning and not available for writing new data. The figure also shows two reference points: "FFS today," which represents Unix FFS today, and "FFS improved," which is our estimate of the best performance possible in an improved Unix FFS. Write cost for Unix FFS is not sensitive to the amount of disk space in use.

# RAID

- Redundant Array of Inexpensive Disks (RAID)
  - A storage system, not a file system
  - Patterson, Katz, and Gibson (Berkeley, 1988)

- Idea: Use many disks in parallel to increase storage bandwidth, improve reliability
  - Files are striped across disks
  - Each stripe portion is read/written in parallel
  - Bandwidth increases with more disks

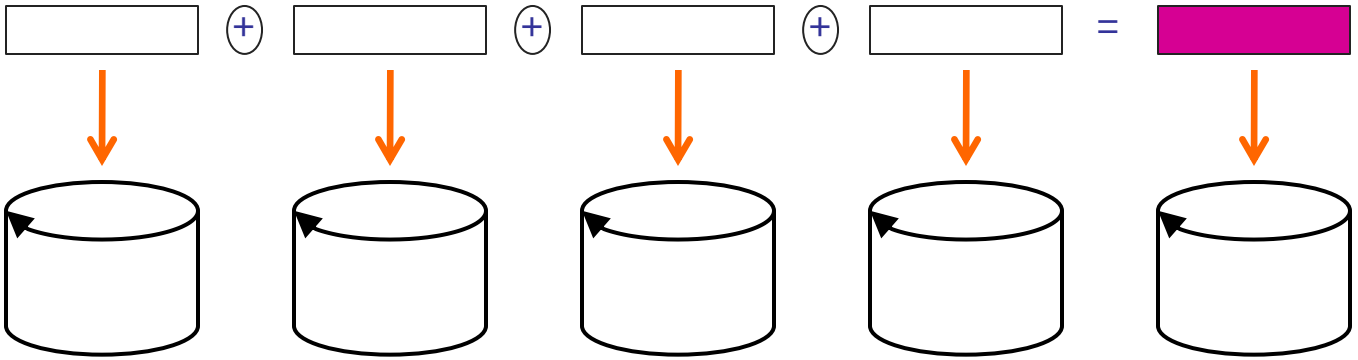
# RAID



# RAID Challenges

- Small files (small writes less than a full stripe)
  - Still write to one disk at a time
- Reliability
  - More disks increases the chance of media failure (MTBF)
- Turn reliability problem into a feature
  - Use one disk to store parity data
    - » XOR of all data blocks in stripe
  - Can recover any data block from all others + parity block
  - Hence “redundant” in name
  - Introduces overhead, but, hey, disks are “inexpensive”

# RAID with parity

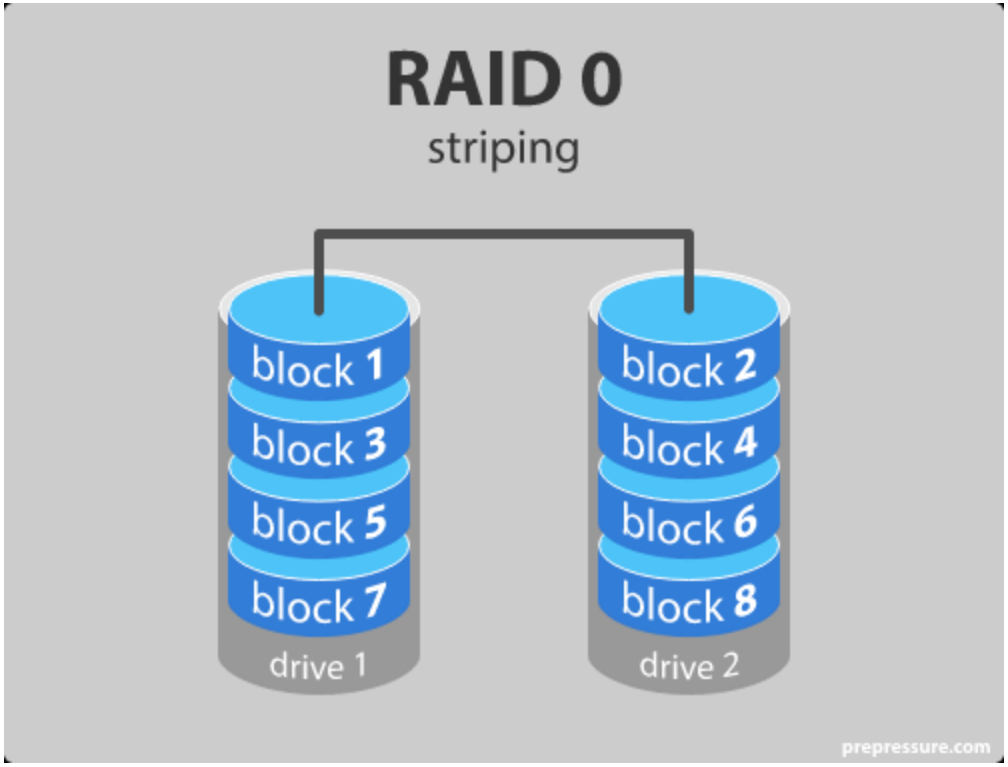


# RAID Levels

- In marketing literature, you will see RAID systems advertised as supporting different “RAID Levels”
- Here are some common levels:
  - RAID 0: Striping
    - » Good for random access (no reliability)
  - RAID 1: Mirroring
    - » Two disks, write data to both (expensive, 1X storage overhead)
  - RAID 5: Floating parity
    - » Parity blocks for different stripes written to different disks
    - » No single parity disk, hence no bottleneck at that disk
  - RAID “10”: Striping plus mirroring
    - » Higher bandwidth, but still have large overhead
    - » See this on PC RAID disk cards

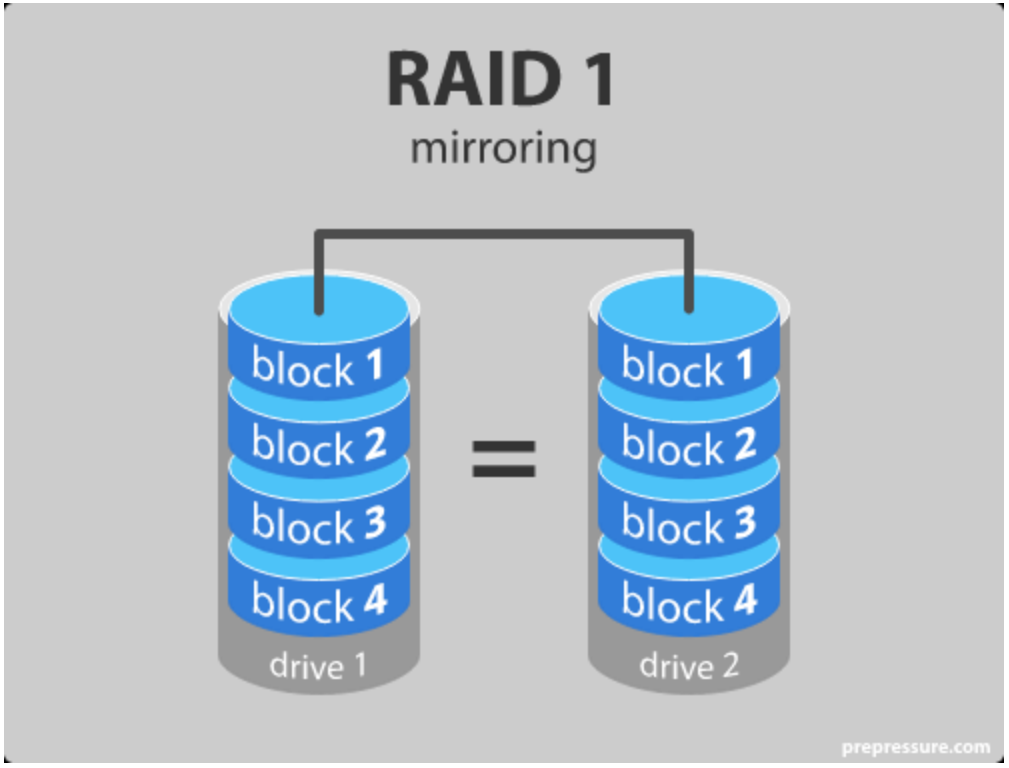
# RAID 0

- RAID 0: Striping
  - Good for random access (no reliability)
  - Better read/write speed



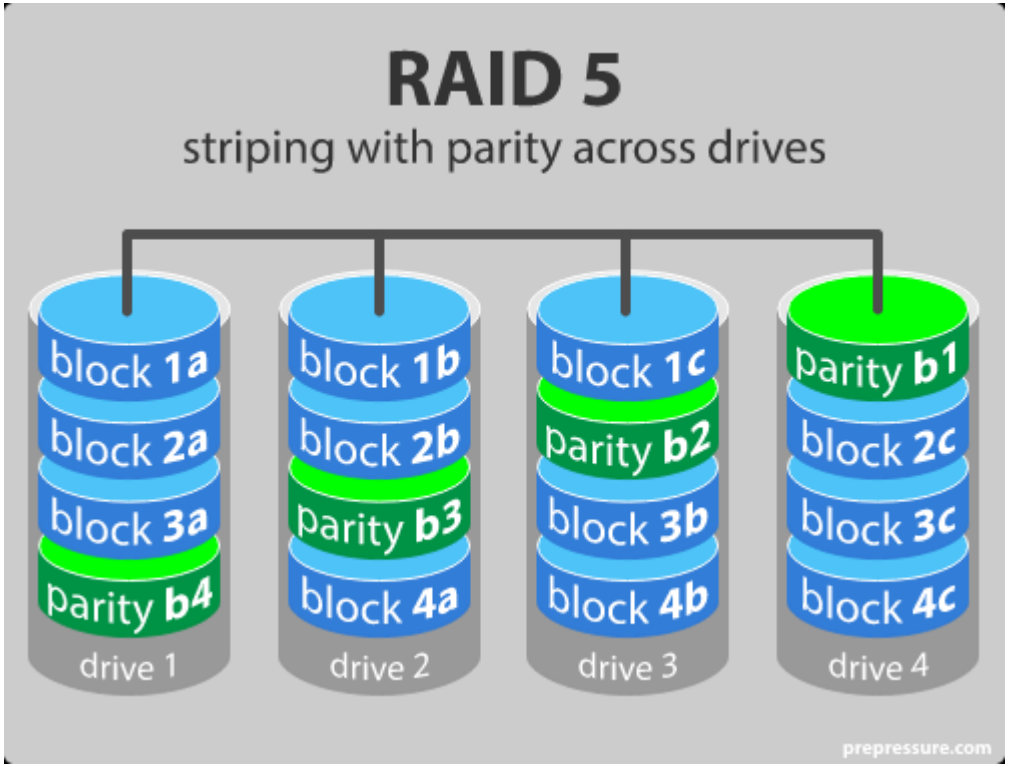
# RAID 1

- RAID 1: Mirroring
  - Two disks, write data to both (expensive, 1X storage overhead)



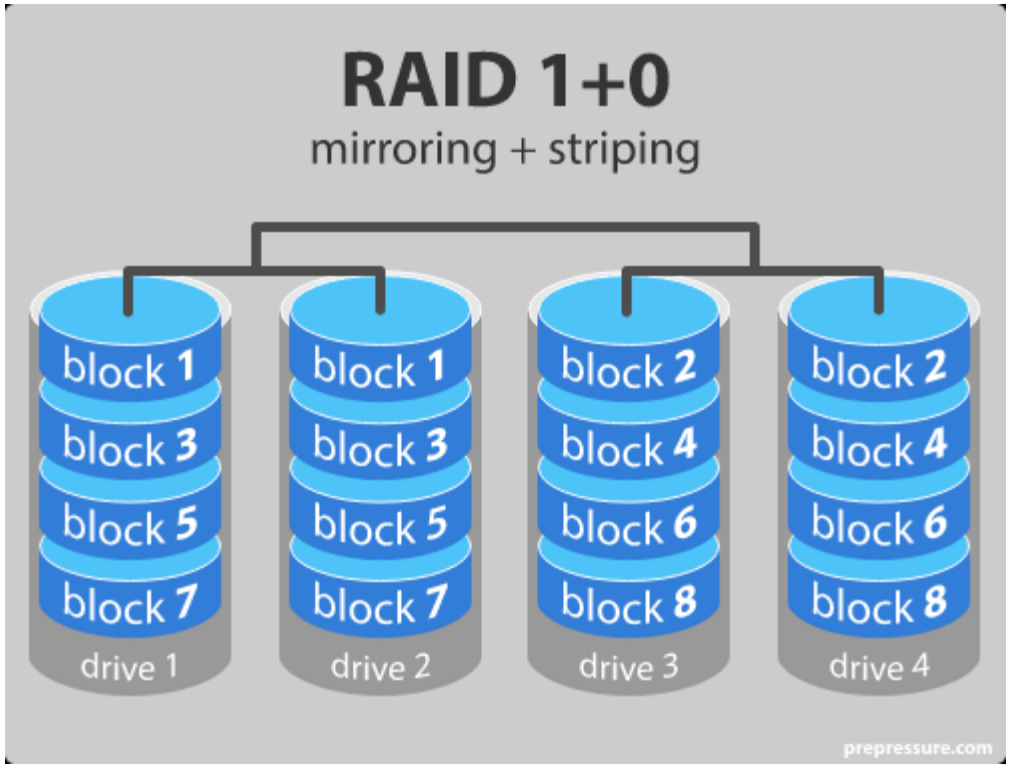
# RAID 5

- RAID 5: Floating parity
  - Parity blocks for different stripes written to different disks
  - No single parity disk, hence no bottleneck at that disk
  - Fast read while slower write (parity computation)



# RAID 1+0

- RAID “10”: Striping plus mirroring
  - Higher bandwidth, but still have large overhead
  - See this on PC RAID disk cards



# Summary

- LFS
  - Improve write performance by treating disk as a log
  - Need to clean log complicates things
- RAID
  - Spread data across disks and store parity on separate disk