CSE 153
Design of Operating Systems

Fall 2018

Lecture 14: File system – optimizations and advanced topics

There’s more to filesystems ☺

- Standard Performance improvement techniques
- Alternative important File systems
  - FFS: Unix Fast File system
  - JFS: making File systems reliable
  - LFS: Optimizing write performance
- Improve the performance/reliability of disk drives?
  - RAID
- Generalizations
  - Network file systems
  - Distributed File systems
  - Internet scale file systems

Improving Performance

- Disk reads and writes take order of milliseconds
  - Very slow compared to CPU and memory speeds
- How to speed things up?
  - File buffer cache
  - Cache writes
  - Read ahead

File Buffer Cache

- Applications exhibit significant locality for reading and writing files
- 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
  - Even a 4 MB cache can be very effective
- Issues
  - The file buffer cache competes with VM (tradeoff here)
  - Like VM, it has limited size
  - Need replacement algorithms again (LRU usually used)

Caching Writes

- On a write, some applications assume that data makes it through the buffer cache and onto the disk
    - As a result, writes are often slow even with caching
- Several ways to compensate for this
    - "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 next block after last block written
        - » Complicated

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 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)

FFS, JFS, LFS, RAID

- Now we're going to look at some example file and storage systems
  - BSD Unix Fast File System (FFS)
  - Journaling File Systems (JFS)
  - Log-structured File System (LFS)
  - Redundant Array of Inexpensive Disks (RAID)

Fast File System

- The original Unix file system had a simple, straightforward implementation
  - Easy to implement and understand
  - But very poor utilization of disk bandwidth (lots of seeking)
- BSD Unix folks did a redesign (mid 80s) that they called the Fast File System (FFS)
  - Improved disk utilization, decreased response time
  - McKusick, Joy, Leffler, and Fabry
- Now the FS to which all other Unix FS's are compared
- Good example of being device-aware for performance

Data and Inode Placement

Original Unix FS had two placement problems:

1. Data blocks allocated randomly in aging file systems
   - Blocks for the same file allocated sequentially when FS is new
   - As FS “ages” and fills, need to allocate into blocks freed up when other files are deleted
   - Problem: Deleted files essentially randomly placed
   - So, blocks for new files become scattered across the disk
2. Inodes allocated far from blocks
   - All inodes at beginning of disk, far from data
   - Traversing file name paths, manipulating files, directories requires going back and forth from inodes to data blocks

Both of these problems generate many long seeks

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
  - Files in same directory allocated in same cylinder
  - Inodes for files allocated in same cylinder as file data blocks
- 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

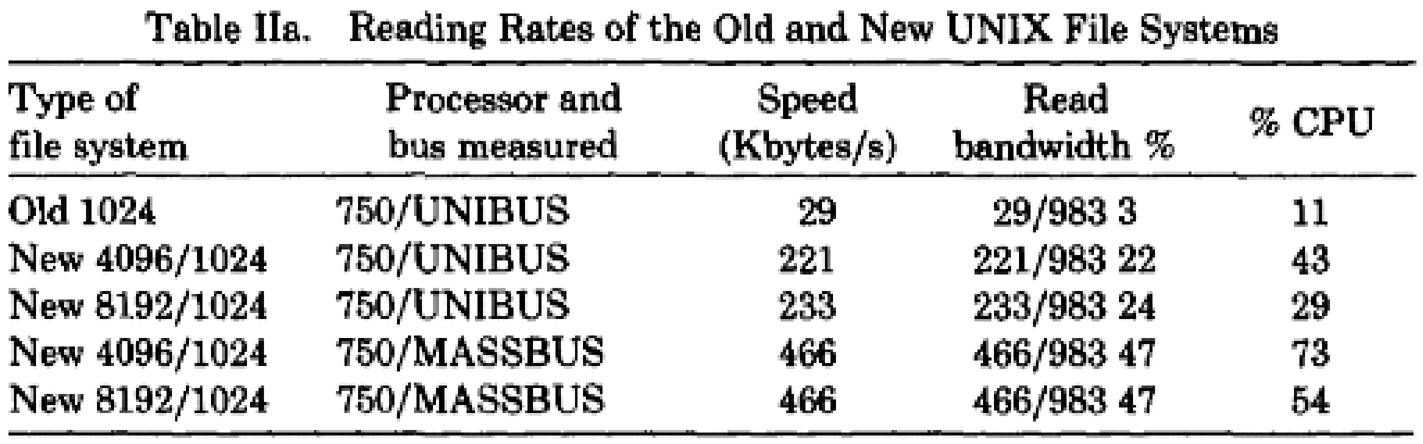


Table IIa. Reading Rates of the Old and New UNIX File Systems

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

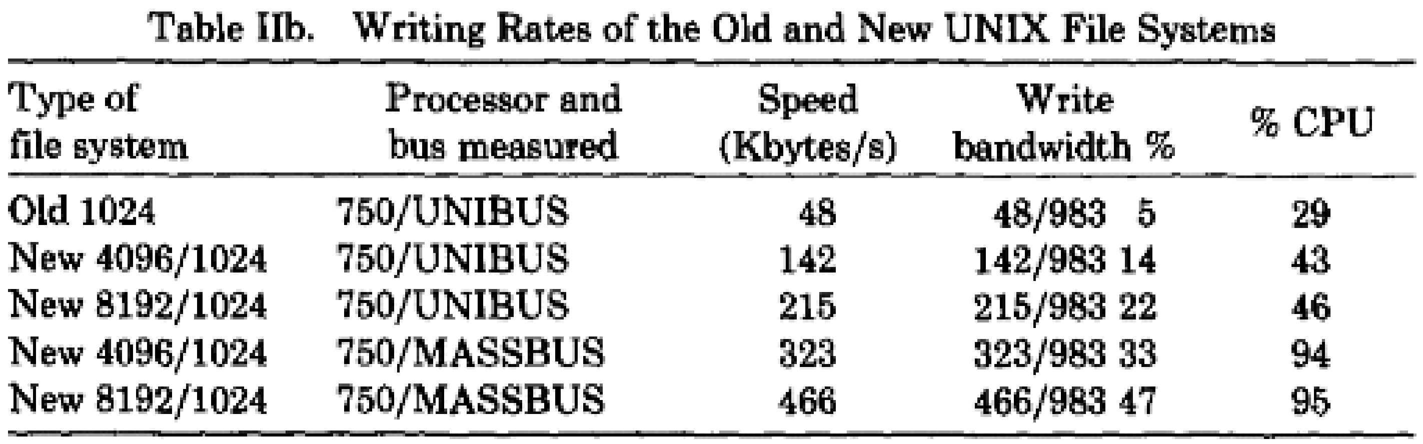


Table IIb. Writing Rates of the Old and New UNIX File Systems

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

Problem: crash consistency

- Updates to data and meta data are not atomic

- Consider, what happens when you delete a file
  1. Remove directory entry
  2. Remove the inode(s)
  3. Mark the free map (for all the i-node and data blocks you freed)
  - What happens if you crash somewhere in the middle?

Journaling File Systems

- Journaling File systems make updates to a log
  - Log plans for updates to a journal first
  - When a crash happens you can replay the journal to restore consistency
- What if we crash when writing journal?
  - Problem. Possible solution, bracket the changes
    - » Introduce checksum periodically
    - » Replay only parts where there is checksum that matches
- Journal choices (regular file? Special partition?)
- Log meta-data and data?

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 map file #s to inode location
  - Location of inode map blocks kept in checkpoint region
  - Checkpoint region has a fixed location
  - Cache inode maps in memory for performance

LFS Layout

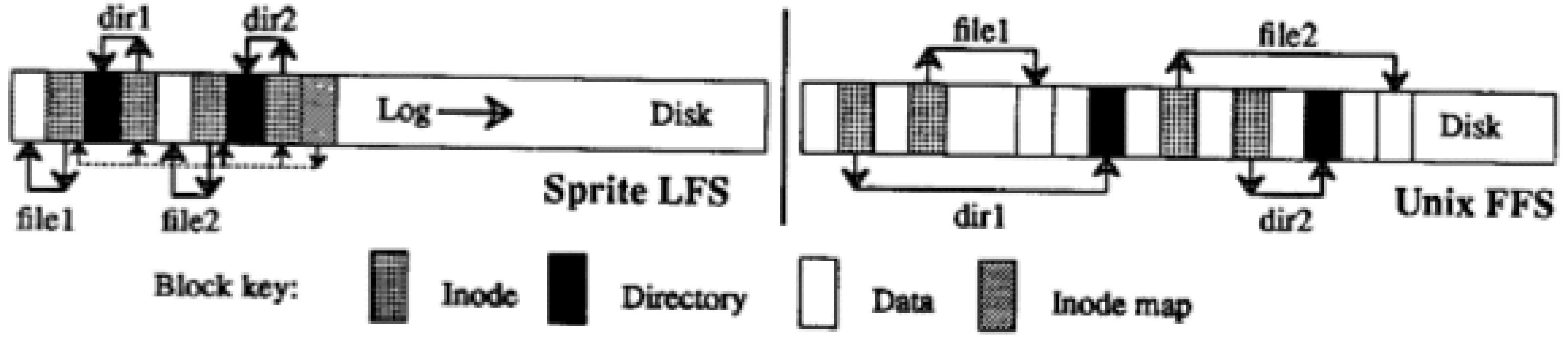


Fig. 1. A comparison between Sprite LFS and Unix FFS. This example shows the modified disk blocks written by Sprite LFS and Unix FFS when creating two single-block files named dir1/file1 and dir2/file2. Each system must write new data blocks and inodes for file1 and file2, plus new data blocks and inodes 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

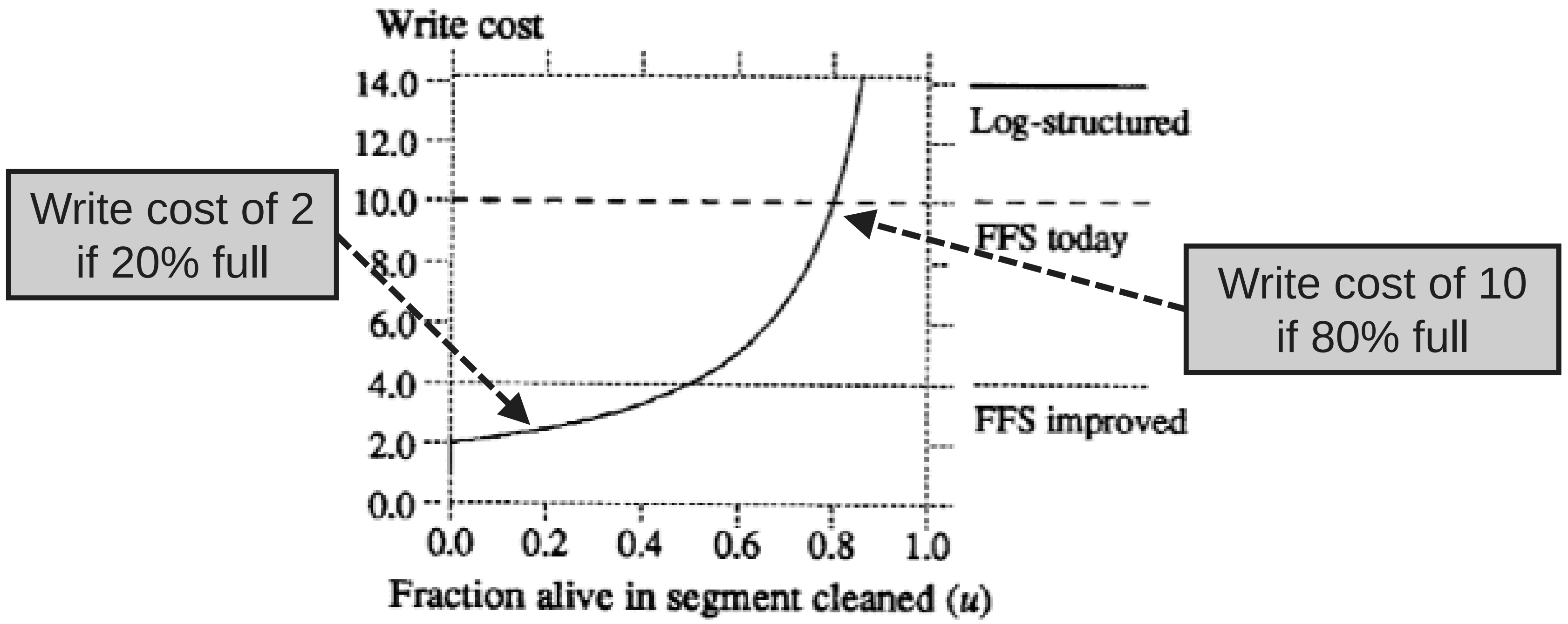


Fig. 3. Write cost as a function of *u* for small files In a log-structured file system, the write cost depends strongly on the utilization of the segments that are cleaned. The more live data in segments cleaned, the more disk bandwidth that is needed for cleaning and not available for writing new data. 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.

Write Cost: Simulation

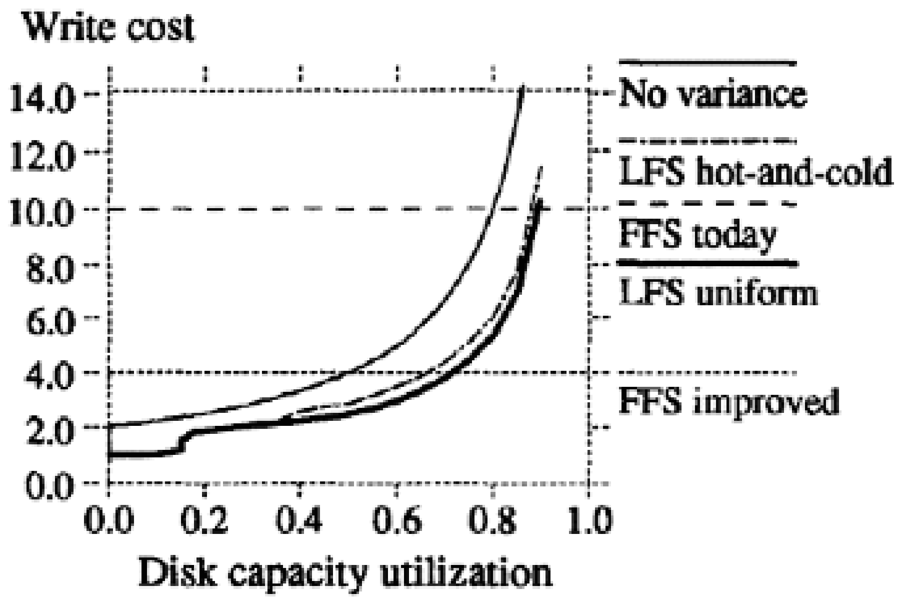
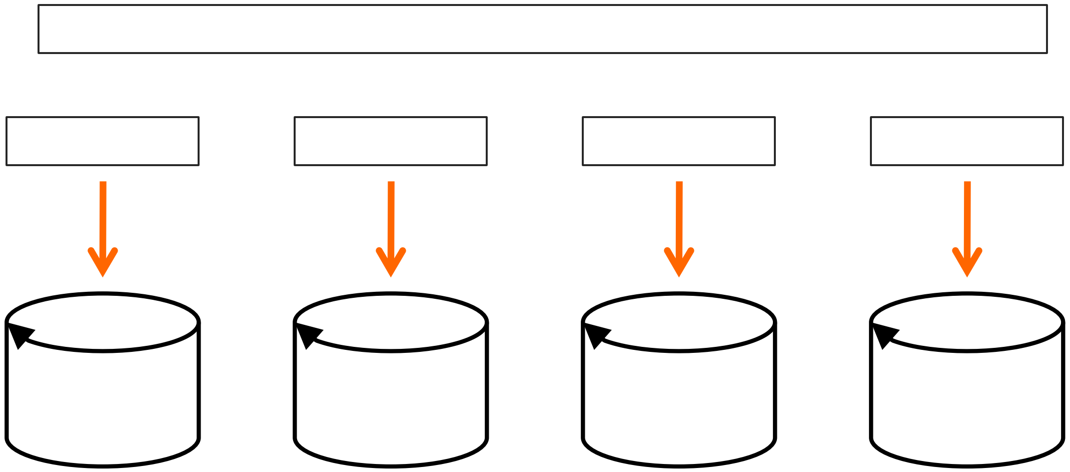


Fig. 4. Initial simulation results. The curves labeled "FFS today" and "FFS improved" are reproduced from Figure 3 for comparison. The curve labeled "No variance" shows the write cost that would occur if all segments always had exactly the same utilization. The "LFS uniform" curve represents a log-structured file system with uniform access pattern and a greedy cleaning policy: the cleaner chooses the least-utilized segments. The "LFS hot-and-cold" curve represents a log-structured file system with locality of file access. It uses a greedy cleaning policy and the cleaner also sorts the live data by age before writing it out again. The x-axis is overall disk capacity utilization, which is not necessarily the same as the utilization of the segments being cleaned.

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

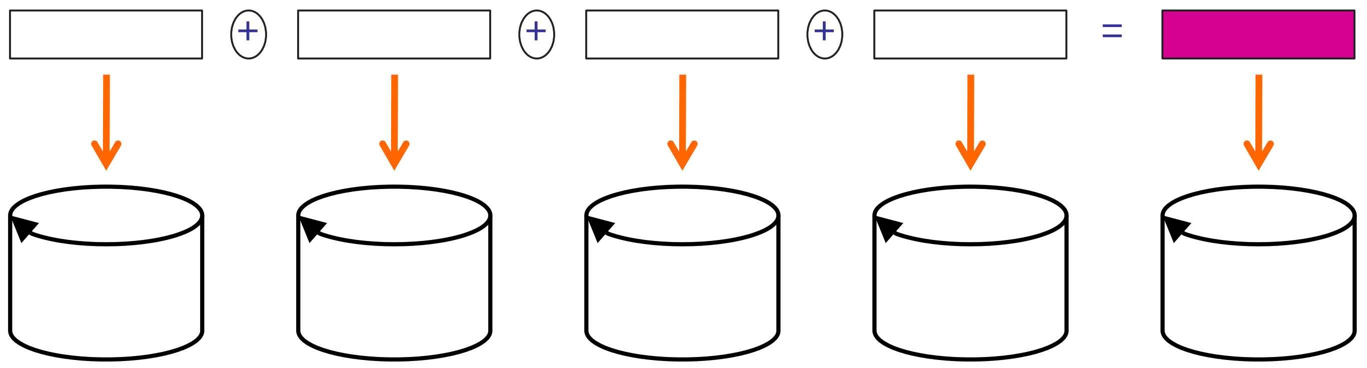
RAID



RAID Challenges

- Small files (small writes less than a full stripe)
  - Need to read entire stripe, update with small write, then write entire stripe out to disks
- 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 2,3 and 4: bit, byte and block level parity. Rarely used.
  - RAID 5, 6: 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

Other file system topics

- Network File systems (NFS)
  - Can a file system be shared across the network
  - The file system is on a single server, the clients access it remotely
- Distributed file systems: Can a file system be stored (and possibly replicated) across multiple machines
  - What if they are geographically spread?
  - Hadoop Distributed File System (HDFS), Google File System (GFS)
- File systems is an exciting research area
  - Take cs202 if interested!

Summary

- UNIX file system
  - Indexed access to files using inodes
- FFS
  - Improve performance by localizing files to cylinder groups
- JFS
  - Improve reliability by logging operations in a journal
- 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