

Lecture 14: File System Performance

CSE 120: Principles of Operating Systems
Alex C. Snoeren



Lab 2 Due THURSDAY



UCSDCSE



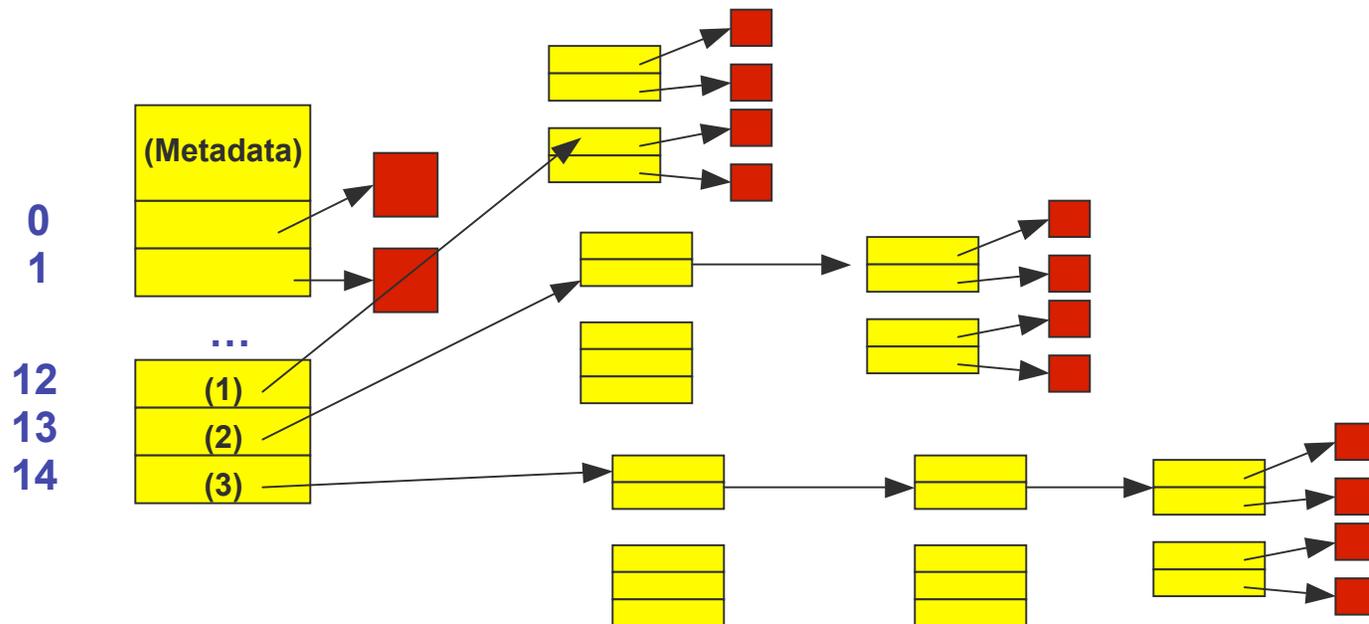
Overview

- Last time we discussed how file systems work
 - ◆ Files, directories, inodes, data blocks, etc.
 - ◆ Didn't focus much on where the data was actually placed
- Now we'll focus on how to make them *perform*
 - ◆ By being smart about how we lay out the data on disk
 - ◆ Or, even better, avoiding the disk altogether
- Three (somewhat dated, but illustrative) case studies:
 - ◆ BSD Unix Fast File System (FFS)
 - ◆ Log-structured File System (LFS)
 - ◆ Redundant Array of Inexpensive Disks

Unix Inodes



- Unix **inodes** implement an indexed structure for files
 - ◆ Also store metadata info (protection, timestamps, length, ref count...)
- Each inode contains 15 block pointers
 - ◆ First 12 are direct blocks (e.g., 4 KB blocks)
 - ◆ Then single, double, and triple indirect



Unix Inodes and Path Search



- Inodes describe where on the disk the blocks for a file are placed
 - ◆ Unix Inodes are *not* directories
 - ◆ 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 “/one”, use Master Block to find inode for “/” on disk
 - ◆ Open “/”, look for entry for “one”
 - ◆ This entry gives the disk block number for the inode for “one”
 - ◆ Read the inode for “one” into memory
 - ◆ The inode says where first data block is on disk
 - ◆ Read that block into memory to access the data in the file



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

- Applications assume writes make it to 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)



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 from which all other Unix FS's have been 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 – why it is possible for “df” to report >100%



Other Problems

- Small blocks (1K) caused two 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}
 - ◆ Problem: internal fragmentation
 - ◆ Fix: Introduce “fragments” (1K pieces of a block)
- Problem: Media failures
 - ◆ Replicate master block (superblock)
- Problem: Reduced seeks, but even one is expensive
 - ◆ What if we can avoid going to disk at all?



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)
 - » 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, '91)



FFS Problems

- LFS also addresses some problems with FFS
 - ◆ Placement is improved, but still have many small seeks
 - » Possibly related files are physically separated
 - » Inodes separated from files (small seeks)
 - » Directory entries separate from inodes
 - ◆ Metadata requires synchronous writes
 - » With small files, most writes are to metadata (synchronous)
 - » Synchronous writes very slow



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.
- Simple, eh?
 - ◆ Alas, 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: 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

RAID



- Redundant Array of Inexpensive Disks (RAID)
 - ◆ A storage system, not a file system
 - ◆ Patterson, Katz, and Gibson (Berkeley, '88)
- 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 Challenges

- Small files (small writes less than a full stripe)
 - ◆ Need to read entire stripe, update with small write, then write entire segment 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 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 UltraDMA PC RAID disk cards

Next time...



- Network file systems
- Read Chapter 11.9
- Finish Project 2!