CSE221 Lecture 12

Aronya Baksy

November 2024

1 Disks and Storage Technology

- SSD (flash storage, faster/reliable) used in mobile devices, HDDs (magnetic storage, less cost per byte) used in datacenters
- Disk organization: concentric rings called **tracks**, each track is split into 512-byte **sectors**
- Single track on all platters is one **cylinder**, group of tracks on all platters make a **cylinder group**
- One **head** per platter, all heads held in place by the **arm**.
- Three components of disk access time:
  - **Seek time**: move head to correct track (ms)
  - **rotation time**: move platter to have the correct sector under the head (ms)
  - **transfer time**: move the entire sector over the head to access its data. (μs)
- Disk interface to OS: at the time of FFS, specify platter/track/sector, modern disks use **logical blocks** whose size is multiple of sector size
- CDs and DVDs optimized for sequential access, organize data in spirals

2 UNIX File Systems

- Logical block size: smallest unit of Disk I/O (4KB typically)
- Inodes store file metadata, direct/indirect pointers to data blocks (design is optimized for small files that don't need too many levels of indirect blocks)
- Directories are a special kind of file consisting of file name and inode pairs

3 A Fast File System for UNIX

3.1 Goals and Claims

- Claim: more flexible allocation policies use locality of reference and increase throughput
- Two different block sizes to adjust to small/large files, avoid wastage of memory and disk
- Enhancements to UNIX file API: place advisory locks on files, long file names etc.

3.2 Existing UNIX File System

- Each disk drive split into *logical partitions*, each partition has a file system, files never span multiple partitions
- FS superblock (located at the beginning of the partition) contains the basic parameters of the file system, i.e. number of data blocks in the file system, maximum number of files, and a pointer to the free list (linked list of free blocks)
- Every file has a descriptor called an *inode* associated with it, contains info about last modified time, access rights, and list of indices to file blocks

- **Drawbacks**:
  - accessing a file normally incurs a long seek from the file's inode to its data
  - suboptimal block allocation since only 512B is allocated at a time
  - small block size, limited read-ahead capability, lots of seeks
- Increasing file block size from 512B to 1024B improved throughput but was still using only 4% of disk b/w because of bad allocation policies causing the free list to become entirely random and making lots of disk seeks happen per file access

3.3 New File System Organization

- Superblock is replicated across the partition, in case of crashes, done when FS is created
- Minimum FS block size is now 4096B (4KiB), and each FS can choose its own block size (any power of 2 that is ≥ 4096).
- **Cylinder group**: group of consecutive cyls in a disk, subdivision of a partition, has bookkeeping info which is placed at a fixed offset (1 + previous cyl group offset, reduces chances of system failure if HDD crashes)
- Bookkeeping info includes superblock copy, inodes, free block bitmap
- Large block size **wastes space** since (back then) users had lots of small files - solution is dividing each FS block into **fragments**
- Each entry of the free block bitmap contains N bits per block (N = number of frag)
- wasted space with fragments is equivalent to FS with block size equal to fragment size while providing flexibility for small files
- The FS API is modified to enable devs to supply the optimal write size (block size or buffer size) so as to avoid overheads of writing one fragment at a time
- If free blocks number in the FS reaches below the **free space reserve** then only sysadmins can create new files (full FS has half throughput compared to 90% occupied)
- FS tries to allocate new blocks on the same cylinder as the previous block in the same file
- **Parameterized FS**: FS is **aware of underlying CPU and disk h/w params** to allocate *rotationally efficient* blocks
- Cyl group info contains a count of the available blocks in a cylinder group at different rotational positions (superblock contains this info in a vector of lists called *rotational layout table*) (we don't do this anymore, in lieu of read-ahead buffers on disk)

3.4 Allocation Policies

- two levels: global decisions policies for new inode placement. Otherwise, local allocation routines to find locally optimal scheme to layout data.
- **Global allocator**: improve perf by increase locality to minimize seek latency, and improve layout to make large transfers possible
- inodes are clustered - all indoes in a single dir are placed within the same CG
- directories have spearate allocation policy to spread - new dir created in CG with more than avg. number of free blocks
- Directories and files should reside in same CG
- redirect data block allocation to a different cylinder when file ¿ 48Kib, avoids a single CG becoming full
- **Local Allocator** requests specific blocks within a CG
- if requested block is not avail. then : first check rotationally closest block to the one requested, then check in same cylinder, then in same CG, then find new CG using quadratic hash, else search all CGs

3.5 Evaluation

- Inode alloc policies work: `ls` command is 2-8 times faster for deeply nested and flat dirs respectively
- Much higher CPU utilization (43-73%) in benchmarks but much much faster throughput (10-20 times read b/w, 3-5 times write b/w)
- Larger block allocation overhead in new FS but less block allocations overall because of big blocks
- read and write speeds are identical on new FS (old FS had faster writes because writes are asynchronous)
- Future work: implement disk drivers to chain together kernel buffers. This would allow contiguous disk blocks to be read in a single disk transaction

3.6 Functional Enhancements

- **Advisory locks** on file system that are used only when programs request it
- **Symbolic links** i.e. softlinks
- **Rename** syscall instead of old approach of 3 syscalls (copy source to temp, move temp to dest, delete source)
- **Soft and hard quota limits** for disk usage per user

4 Design and Implementation of a Log-Structured File System

4.1 Goals

- Make more efficient use of disk bandwidth
- Use log structure for faster file writing and disaster recovery

4.2 Design

- Write all information to a sequential **log**, which is the *only* structure on the disk (not an auxiliary, tempoprary log)
- Improves write performance by removing seeks, makes DR easier as entire disk need not be scanned to re-build the memory
- **Technology trends**: CPU speed and main memory size were increasing exponentially, disk speed increasing slowly (large mem size means buffering becomes important and force writes to dominate disk workloads) and faster for sequential access
- **Problems with existing approaches**:
  - FFS puts files far apart, separates inodes from file contents within a CG(which makes some ops cheap but makes others like file creation, expensive)
  - Synchronous writes for directories and inodes even though file blocks themselves are written async in FFS

4.3 Implementation

- Main issues: how to find data using metadata, how to manage free space (avoid fragmentation)
- FFS uses inodes, indirect blocks, but FFS keeps inodes at specific locations on disk, while LFS maintains an *inode map* to maintain the current location of each inode in the log
- inode maps are small enough to remain in memory, not written to disk

4.4 Free Space Management

- How to handle fragmentation: either thread through fragments (makes reads as slow as traditional FS) or periodic copying (overheads)
- Solution: use both: disk is split into large **segments** which are written to sequentially, all live data to be copied out before segment is re-written
- Threading happens at the **segment level**, the segment size is chosen large enough that the transfer time to read or write a whole segment is much greater than the cost of a seek to the beginning of the segment.
- **Segment cleaning**:
  - read a number of segments into memory
  - identify the live data, discard unused blocks
  - write the live data back to a smaller number of clean segments.
- **segment summary block** in each segment contains file to block mapping, and the block offset within the file (solves problems in cleaning)
- No data structures like free block list needed, just match file UID stored in SSB with file UID stored in inode, if not matching then clean up the block
- **Heuristic cleaning policies**: start cleaning segments when the number of clean segments drops below a threshold value, ceans a few tens of segments at a time until the number of clean segments surpasses another threshold value

4.4.1 Cleaning Policies

- **Which segments** to clean?
- First approach: clean segments that fall below a certain utilization threshold, doesn't work in practice as it doesn't take into account how often data is written
- Final Approach: cold vs hot segments based on write frequency, clean cold segments aggressively, clean hot ones less aggressively i.e wait longer before cleaning - *cost-benefit policy*
- **How to group live blocks** that are written out?
- Group live blocks by last modified time (proxy for hot/cold)

4.5 LFS Today

- Flash Translation Layer in SSD choose how to map blocks to file devices to ensure that blocks aren't overwritten too many times