CS 140: Operating Systems Lecture 19: FFS and Logging File Systems

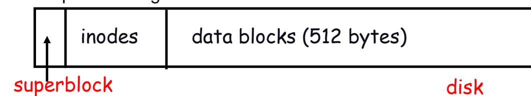
Mendel Rosenblum

Last time, this time

- Last Time – Crash Recovery techniques
  - **Write data twice in two different places**
    - **Used state duplication and idempotent actions to create arbitrary sized atomic disk updates.**
  - **Careful updates**
    - **Order writes to aid in recovery.**
- This time - What modern FS do:
  - **FFS performance optimization.**
  - **Logging - write twice but two different forms.**

Original Unix File System

- Simple and elegant:



- Nouns:
  - **Data blocks.**
  - **inodes (directories represented as files)**
  - **Hard links.**
  - **Superblock. (specifies number of blks in FS, counts of max # of files, pointer to head of free list).**
- Problem: slow
  - **Only gets 20Kb/sec (2% of disk maximum) even for sequential disk transfers!**

A plethora of performance costs

- Blocks too small (512 bytes)
  - **file index too large**
  - **too many layers of mapping indirection**
  - **transfer rate low (get one block at time)**
- Sucky clustering of related objects:
  - **Consecutive file blocks not close together**
  - **Inodes far from data blocks**
  - **Inodes for directory not close together**
  - **poor enumeration performance: e.g., "ls", "grep foo *.c"**
- Next: how FFS fixes these problems (to a degree)

Problem 1: Too small block size

- Why not just make bigger?

Block size	space wasted	file bandwidth
512	6.9%	2.6%
1024	11.8%	3.3%
2048	22.4%	6.4%
4096	45.6%	12.0%
1MB	99.0%	97.2%

- Bigger block increases bandwidth, but how to deal with wastage ("internal fragmentation")?
  - **Use idea from malloc: split unused portion.**

Handling internal fragmentation

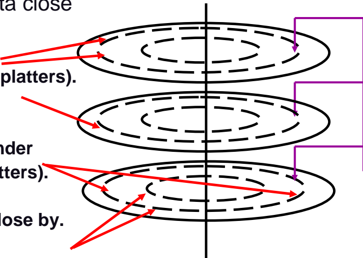
- BSD FFS:
  - **Has large block size (4096 or 8192).**
  - **Allow large blocks to be chopped into small ones ("fragments").**
  - **Used for little files and pieces at the ends of files.**

file a
File b

- Best way to eliminate internal fragmentation?
  - **Variable sized splits of course.**
  - **Why does FFS use fixed-sized fragments (1024, 2048)?**

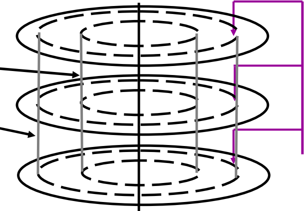
Prob' 2: Where to allocate data?

- Our central fact:
  - **Moving disk head expensive.**
- So? Put related data close
  - **Fastest: adjacent sectors (can span platters).**
  - **Next: in same cylinder (can also span platters).**
  - **Next: in cylinder close by.**



Clustering related objects in FFS

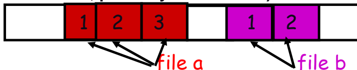
- 1 or more consecutive cylinders into a "cylinder group"



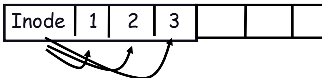
  - **Key: can access any block in a cylinder without performing a seek. Next fastest place is adjacent cylinder.**
  - **Tries to put everything related in same cylinder group**
  - **Tries to put everything not related in different group (?!)**

Clustering in FFS

- Tries to put sequential blocks in adjacent sectors
  - **(access one block, probably access next)**



- Tries to keep inode in same cylinder as file data:
  - **(if you look at inode, most likely will look at data too)**



- Tries to keep all inodes in a dir in same cylinder group
  - **(access one name, frequently access many)**
  - **"ls -l"**

What's a cylinder group look like?

- Basically a mini-Unix file system:

- How how to ensure there's space for related stuff?
  - **Place different directories in different cylinder groups.**
  - **Keep a "free space reserve" so can allocate near existing things.**
  - **When file grows to big (1MB) send its remainder to different cylinder group.**

Prob' 3: Finding space for related objects

- Old Unix (& dos): Linked list of free blocks
  - **Just take a block off of the head. Easy.**

  - **Bad: free list gets jumbled over time. Finding adjacent blocks hard and slow.**
- FFS: switch to bit-map of free blocks
  - **1010101111111000001111111000101100.**
  - **Easier to find contiguous blocks.**
  - **Small, so usually keep entire thing in memory.**
  - **Key: keep a reserve of free blocks. Makes finding a close block easier.**

Using a bitmap

- Usually keep entire bitmap in memory:
  - **4G disk / 4K byte blocks. How big is map?**
- Allocate block close to block x?
  - **Check for blocks near bmap[x/32].**
  - **If disk almost empty, will likely find one near.**
  - **As disk becomes full, search becomes more expensive and less effective.**
- Trade space for time (search time, file access time)
  - **Keep a reserve (e.g, 10%) of disk always free, ideally scattered across disk.**
  - **Don't tell users (df --> 110% full).**
  - **N platters = N adjacent blocks.**
  - **With 10% free, can almost always find one of them free.**

So what did we gain?

- Performance improvements:
  - **Able to get 20-40% of disk bandwidth for large files.**
  - **10-20x original Unix file system!**
  - **Better small file performance. (why?)**
- Is this the best we can do? No.
- Block based rather than extent based
  - **Name contiguous blocks with single pointer and length (Linux ext2fs)**
- Writes of meta data done synchronously
  - **Really hurts small file performance.**
  - **Make asynchronous with write-ordering ("soft updates") or logging (the episode file system, ~LFS).**
  - **Play with semantics (/tmp file systems).**

Logging

- What/need to know what was happening at the time of the crash.
  - **Observation: Only need to fix up things that are in the middle of changing. Normally a small fraction of total disk.**
- Idea:
  - **Lets keep track of what operations are in progress and use this for recovery. It's keep a "log" of all operations, upon a crash we can scan through the log and find problem areas that need fixing.**
- One small problem: Log needs to be in non-volatile memory!

Implementation

- Add log area to disk.

- Always write changes to log first – called *write-ahead logging* or *journaling*.
- Then write the changes to the file system.
- All reads go to the file system.
- Crash recovery – read log and correct any inconsistencies in the file system.

Issue - Performance

- Two disk writes (on different parts of the disk) for every change?
  - **Observation: Once written to the log, the change doesn't need to be immediately written to the file system part of disk. Why?**
  - **It's safe to use a write-back file cache.**
  - **Normal operation:**
    - **Changes made in memory and logged to disk.**
    - **Merge multiple changes to same block. Much less than two writes per change.**
- Synchronous writes are on every file system change?
  - **Observation: Log writes are sequential on disk so even synchronous writes can be fast.**
  - **Best performance if log on separate disk.**

Issue - Log management

- How big is the log? Same size as the file system?
  - **Can we reclaim space?**
- Observation: Log only need for crash recover.
- Checkpoint operation – make in-memory copy of file system (file cache) consistent with disk.
  - **After a checkpoint, can truncate log and start again.**
- Log only needs to be big enough to hold change being kept in memory.
- Most logging file systems only log metadata (file descriptors and directories) and not file data to keep log size down.

Log format

- What exactly do we log?
- Possible choices:
  - **Physical block image**
    - **Example: directory block and inode block.**
      - **+ easy to implement, -takes much space (slower)**
      - **Which block image?**
        - Before operation: Easy to go backward during recovery
        - After operation: Easy to go forward during recovery.
        - Both: Can go either way.
  - **Logical operation**
    - **Example: Add name "foo" to directory #41**
    - **+ more compact, -more work at recovery time, -tricky**

Current trend is towards logging FS

- Fast recovery: recovery time O(active operations) and not O(disk size)
- Better performance if changes need to be reliable
  - **If you need to do synchronous writes, sequential synchronous writes are much faster than non-sequential ones.**
  - **Note that no synchronous writes are faster than logging but can be dangerous.**
- Examples:
  - **Windows NTFS.**
  - **Veritas.**
  - **Many competing logging file system for Linux**
    - **ext3, jfs, xfs, riserfs**