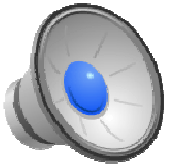


Logging File Systems

- Learning Objectives
 - Explain the difference between journaling file systems and log-structured file systems.
 - Give examples of workloads for which each type of system will excel/fail miserably.
 - Compare and contrast the recovery and durability properties of systems such as soft-updates, journaling, and log-structured file systems.
 - Explain the technological motivation for log-structured systems.
- Topics
 - Log-structured file systems



Journaling Pros/Cons

- Advantages
 - Write log records sequentially
 - Data (random) writes can be buffered much longer.
Implications:
 - You may get multiple writes into the buffer cache before you have to write to disk.
 - You can accumulate a large number of blocks to write to disk and then schedule them cleverly, so they are faster than the same number of random writes..
 - Facilitate fast recovery: just replay the log (no costly FSCK).
- Disadvantages
 - You write everything twice.
 - If you log only metadata, then you can end up with intact file system structures, but incorrect or missing data.

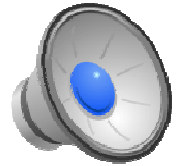


Looking Forward

- Annoying tidbits
 - You end up writing all your meta-data twice: once to the log and once to the file system.
 - If you want the data recoverable, then you have to log that too and that means you're writing the data twice.

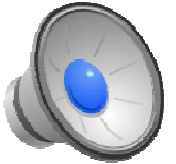
Isn't there a better way???

- Log-structured file systems
 - Is there a way to make the log records themselves be the actual data, so you don't have to write everything twice (once to the log and once to the file system data).



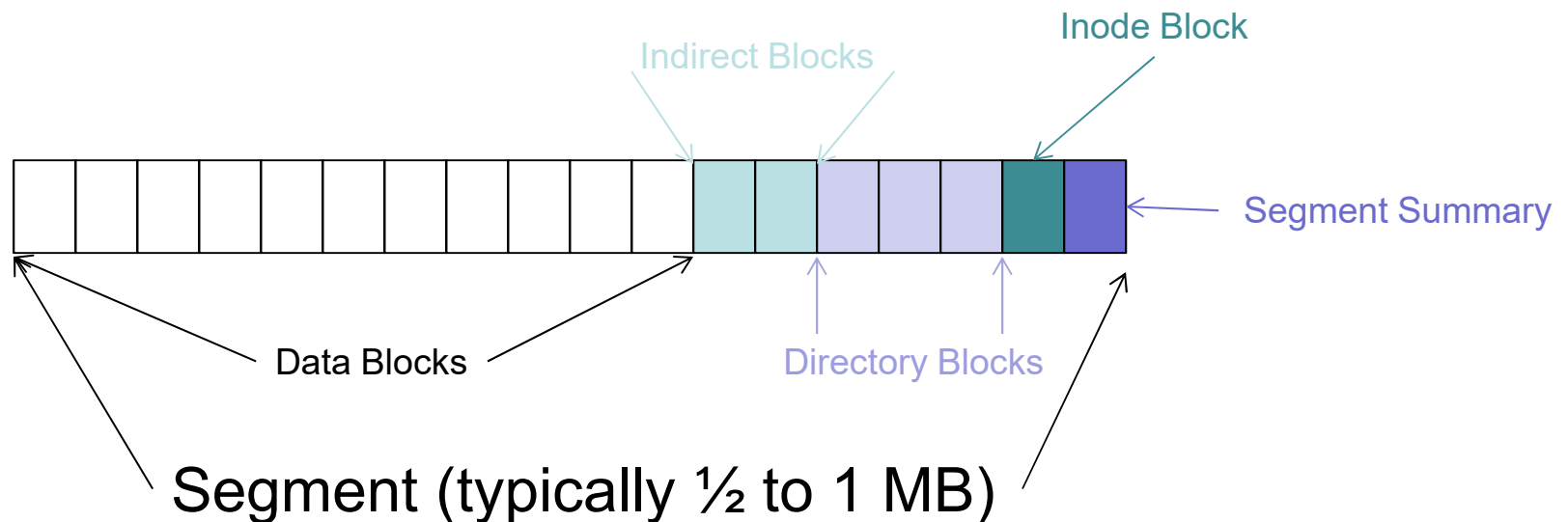
Log-Structured File Systems

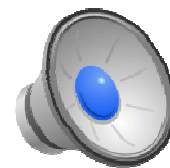
- Let's take a look at current technology trends and workload patterns and design a file system that addresses those challenges.
- 1984: A Trace-Based Analysis of the 4.2 BSD File System (Ousterhout et al).
 - Most files are small.
 - Most bytes belongs to large files.
 - Most files are read and written sequentially.
- 1988: The Case for Redundant Arrays of Inexpensive Disks (Patterson et al.)
 - Increase I/O bandwidth by putting multiple disks together.
 - Add extra disks for reliability (parity or ECC).
 - “Good” arrangements make big, sequential I/O fast, but small, random I/O slow.
- Other trends:
 - I/O gap widening
 - Machines have large caches that reduce read traffic.
 - Writes however, must go to disk.
 - Writing sequentially is significantly better than writing randomly.
 - No significant improvements in disk access (seek or rotation) times.
 - We need to get rid of small, random access and synchronous I/O.



Overview

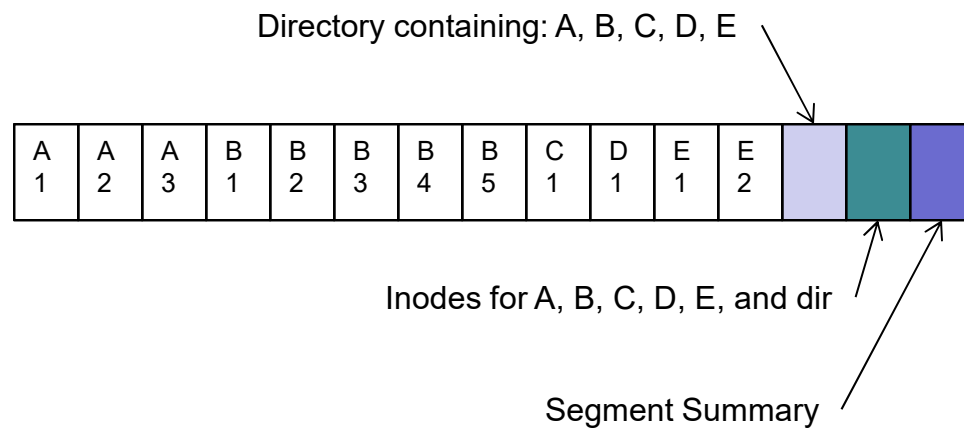
- Cache data in memory, even dirty data.
- Coalesce lots of dirty data together (inodes, directories, data blocks, indirect blocks, etc) into a large chunk of data.
- Write that data to disk sequentially as a log, but *make the log the only persistent representation of the file system.*

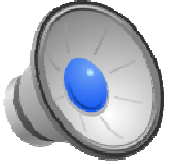




LFS Example (1)

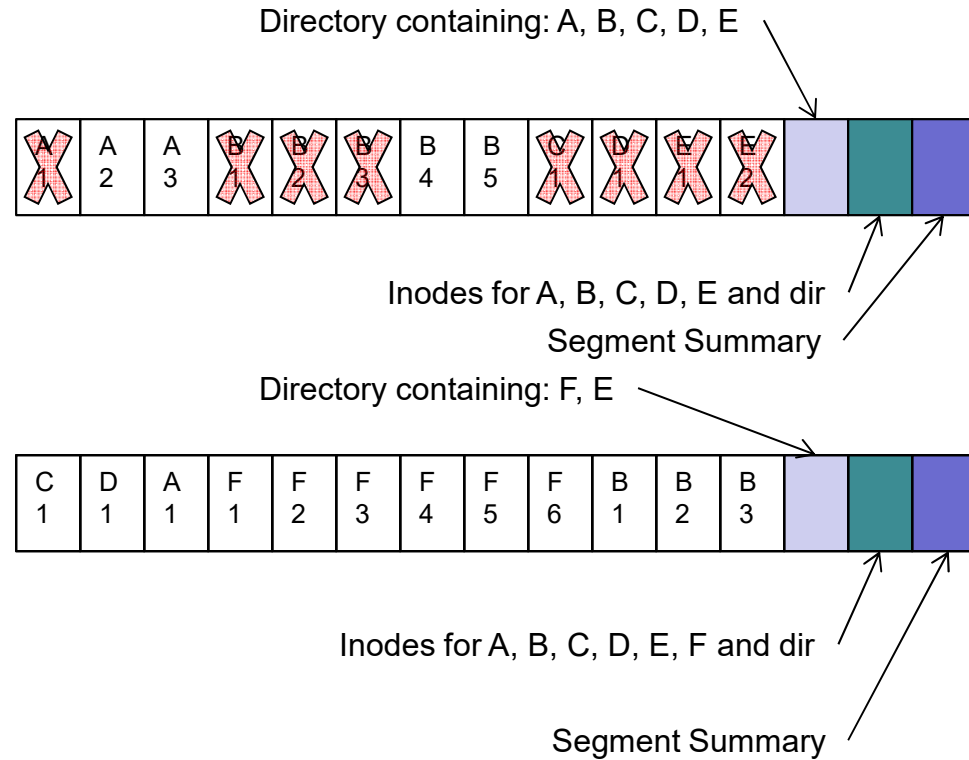
Create A; write blocks 1-3
Create B; write blocks 1-5
Create C; write block 1
Create D; write block 1
Create E; write blocks 1-2

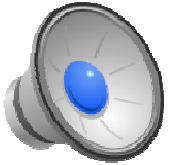




LFS Example (2)

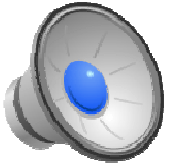
- Create A; write blocks 1-3
- Create B; write blocks 1-5
- Create C; write block 1
- Create D; write block 1
- Create E; write blocks 1-2
- Update file C; block 1
- Update file D; block 1
- Update file A; block 1
- Create file F; blocks 1-6
- Update file B; blocks 1-3
- Delete E





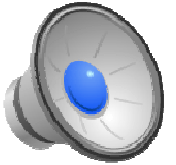
LFS File System Operations

- Most operations behave identically to typical FS (FFS).
 - Directories map names to inode numbers.
 - Inode numbers map to inodes.
 - Inodes map to data blocks.



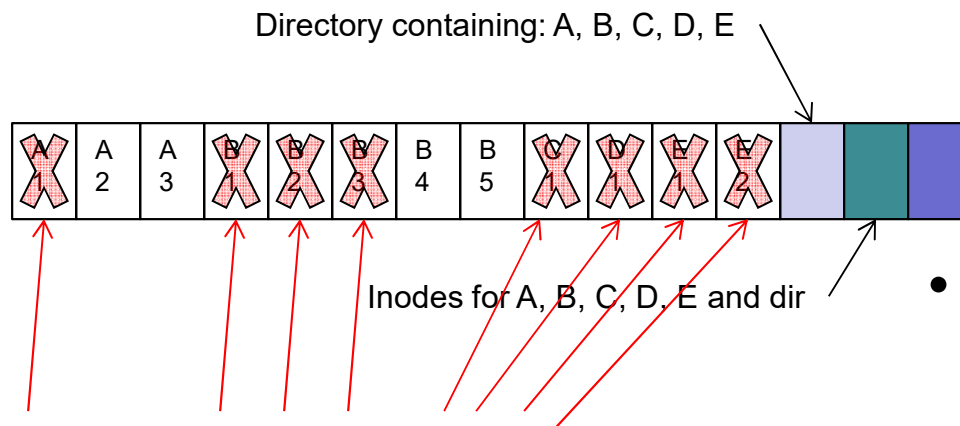
LFS: Finding Inodes

- Maintain an **inode map**
 - A large array with one entry for each inode.
 - The array contains the disk address of the inode.
 - Since you can place many inodes in a single block, make sure that you can figure out which inode is which in the inode block (store the inode number in the inode).
- Where do you place the inode map?
 - Option 1: Fixed location on disk
 - Option 2: In a 'special' file (the ifile)
 - Write the special file in segments just like we write regular files.
 - But then, how do we find the inode for the ifile?
 - Store the ifile inode address in a special place (i.e, superblock).



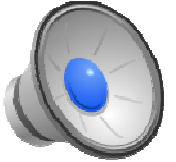
LFS: Free Space Management (1)

- Option 1: Threading
 - Leave live data in place.
 - Write new data to available places.
 - NetApp's WAFL uses this technique.



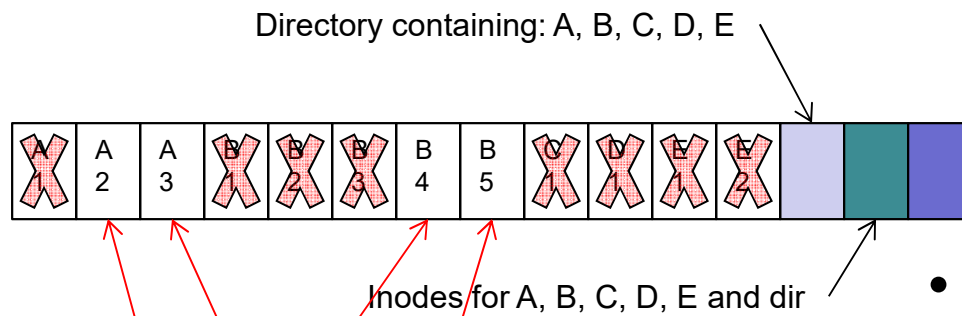
Use these free blocks

- **Problem**
 - Writes are no longer contiguous



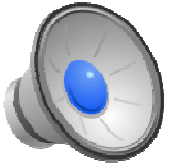
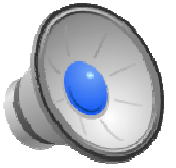
LFS: Free Space Management (2)

- Option 2: Cleaning
 - Copy and coalesce data into a new segment.
 - Old segment available for reclamation



Copy these blocks to end of log

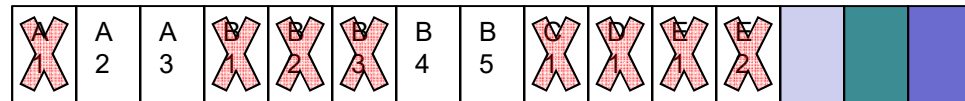
- **Problem**
 - Long-lived data gets copied a lot!



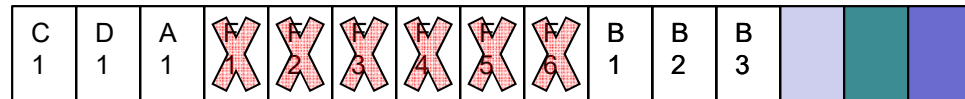
LFS: Free Space Management (3)

- Option 3: Hybrid
 - Use threaded segments.
 - Clean on a per segment basis.
 - Thread segments together.

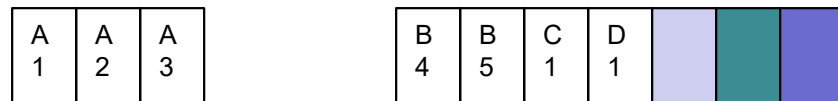
Delete F

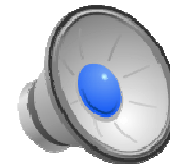


Reclaim space by cleaning
Replace two segments with one



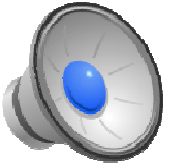
First two segments are now clean
And available for reallocation.





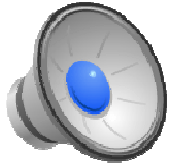
Cleaning Algorithm and Structures

- Three-step algorithm
 - Read N dirty segments.
 - Identify which blocks are live.
 - Write live data back to log.
- Identify each block
 - Must know which block of which file is being cleaned.
 - Segment summaries provide a description of all the blocks in the segment:
 - Identify data blocks and to which inode/blkno they belong.
 - Identify inodes stored in inode blocks
 - Must write a segment summary whenever you write “a batch” of blocks to disk.
 - These batches are called partial segments (unless they fill an entire segment).



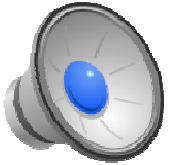
Cleaning Policies

- When should the cleaner run?
- How many segments should be cleaned at once?
- Which segments should be cleaned?
- How should cleaned blocks be grouped?
- The original paper addressed the last two; turns out that the first two are also very important.
 - Clean a few 10s of segments — requires a few tens of megabytes of kernel main memory.
 - Wait until clean segments are scarce; then begins cleaning (implication is that cleaning is always triggered when there is regular user activity; not a good idea).
 - Must maintain a segment usage table that tracks how full/empty each segment is.
 - Getting these calculations right is tricky.
 - Where do you put it? Right in the ifile with the inode map!



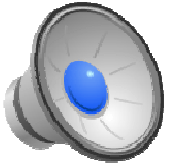
LFS Recovery (1)

- What do we do after a crash?
 - Good news: we know that writes always happen at the end of the log, so all we need to do is find that end.
 - Bad news: how do you find the end?
- Recovery structures:
 - Segment summaries let you parse each segment.
 - When you write a segment, you know the one that came before it, but not necessarily the one that comes after it.
 - You can either link segments backwards or preallocate the next segment.
 - If you preallocate, you need to know if that “next” segment is valid; use timestamps to order segments properly.



LFS Recovery (2)

- Periodically we take checkpoints:
 - Flush all data to disk.
 - Record last segment written in the superblock.
 - Write superblock.
- Overall recovery algorithm:
 - Find most recent superblock.
 - Find last-written-segment from superblock.
 - While more segments follow
 - Parse segment summary and update inodes, ifile, and segment summaries to reflect file system state represented by the segment.
 - Take a checkpoint



LFS Summary

- LFS did two things:
 - Used log to make multiple random I/Os into one large sequential I/O, using the disk more efficiently.
 - Got rid of any other representation of the data other than the log.
- Implications of this second action:
 - No-overwrite storage system.
 - Nice recovery properties, but must garbage collect data.
- The database community backed off no-overwrite strategies versus journaling strategies in the 1970's and early 1980's.
 - The journaling guys won in the DB community (for a long time)!
 - How about in the file system community?