

Outline for Today's Lecture

Administrative:

- Assignment 5 – sign up for demos

Objective:

- Revisiting log structured file systems
- Recovering a file system after a crash

Log-Structured File Systems

- Assumption: Cache is effectively filtering out reads so we should optimize for writes
- Basic Idea: manage disk as an append-only **log** (subsequent writes involve minimal head movement)
- Data and meta-data (mixed) accumulated in large segments and written contiguously
- Reads work as in UNIX - once inode is found, data blocks located via index.
- Cleaning an issue - to produce contiguous free space, correcting fragmentation developing over time.
- Claim: LFS can use 70% of disk bandwidth for writing while Unix FFS can use only 5-10% typically because of seeks.

LFS logs

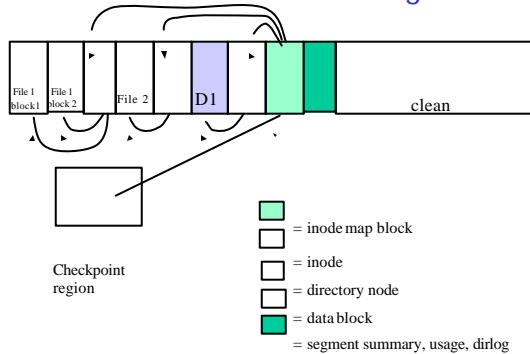
In LFS, **all** block and metadata allocation is log-based.

- LFS views the disk as "one big log" (logically).
- *All* writes are clustered and sequential/contiguous.
 - Intermingles metadata and blocks from different files.
- Data is laid out on disk in the order it is written.
- No-overwrite allocation policy: if an old block or inode is modified, write it to a new location at the *tail* of the log.
- LFS uses (mostly) the same metadata structures as FFS; only the allocation scheme is different.
 - Cylinder group structures and free block maps are eliminated.
 - Inodes are found byindirecting through a new map

LFS Data Structures on Disk

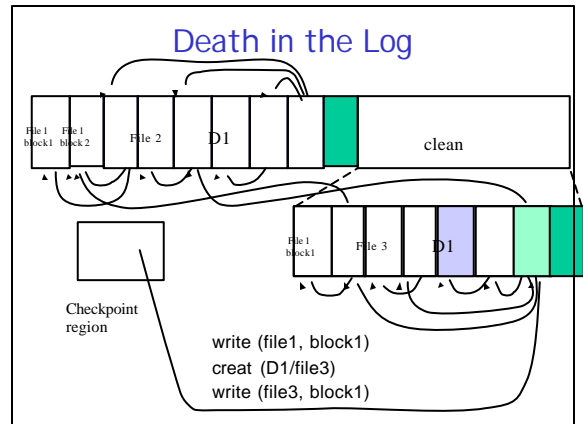
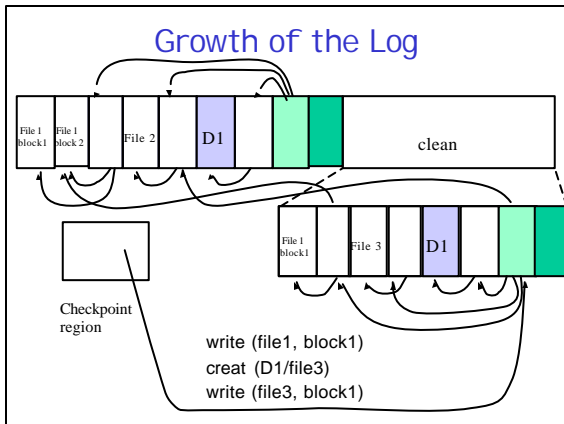
- Inode – in log, same as FFS
- Inode map – in log, locates position of inode, version, time of last access
- Segment summary – in log, identifies contents of segment (file#, offset for each block in segment)
- Segment usage table – in log, counts live bytes in segment and last write time
- Checkpoint region – fixed location on disk, locates blocks of inode map, identifies last checkpoint in log.
- Directory change log – in log, records directory operations to maintain consistency of ref counts in inodes

Structure of the Log



Writing the Log in LFS

1. LFS "saves up" dirty blocks and dirty inodes until it has a full *segment* (e.g., 1 MB).
 - Dirty inodes are grouped into block-sized clumps.
 - Dirty blocks are sorted by (*file, logical block number*).
 - Each log segment includes summary info and a checksum.
2. LFS writes each log segment in a single burst, with at most one seek.
 - Find a free segment "slot" on the disk, and write it.
 - Store a back pointer to the previous segment.
 - Logically the log is sequential, but physically it consists of a chain of segments, each large enough to amortize seek overhead.



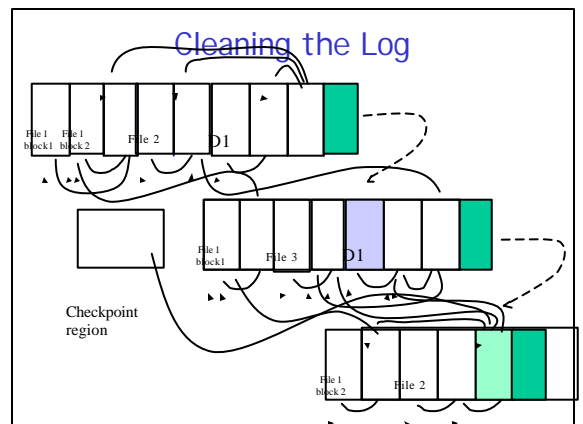
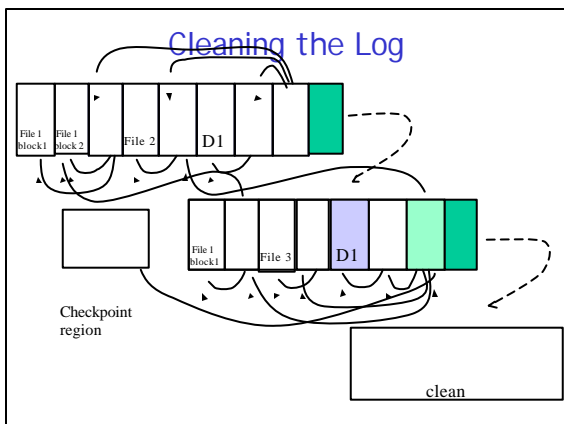
Writing the Log: the Rest of the Story

1. LFS cannot always delay writes long enough to accumulate a full segment; sometimes it must push a *partial segment*.
 - fsync, update daemon, NFS server, etc.
 - Directory operations are synchronous in FFS, and some must be in LFS as well to preserve failure semantics and ordering.
2. LFS allocation and write policies affect the buffer cache, which is supposed to be filesystem-independent.
 - Pin (*lock*) dirty blocks until the segment is written; dirty blocks cannot be recycled off the free chain as before.
 - Endow *indirect blocks with permanent logical block numbers suitable for hashing in the buffer cache.

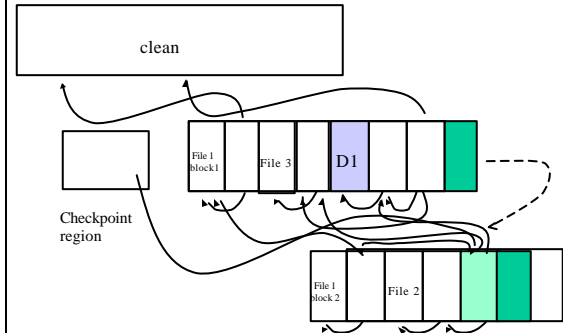
Cleaning in LFS

What does LFS do when the disk fills up?

1. As the log is written, blocks and inodes written earlier in time are superseded ("killed") by versions written later.
 - files are overwritten or modified; inodes are updated
 - when files are removed, blocks and inodes are deallocated
2. A cleaner daemon compacts remaining live data to free up large hunks of free space suitable for writing segments.
 - look for segments with little remaining live data
 - benefit/cost analysis to choose segments
 - write remaining live data to the log tail
 - can consume a significant share of bandwidth, and there are lots of cost/benefit heuristics involved.



Cleaning the Log



Cleaning Issues

- Must be able to identify which blocks are live
- Must be able to identify the file to which each block belongs in order to update inode to new location
- Segment Summary block contains this info
 - File contents associated with uid (version # and inode #)
 - Inode entries contain version # (incr. on truncate)
 - Compare to see if inode points to block under consideration

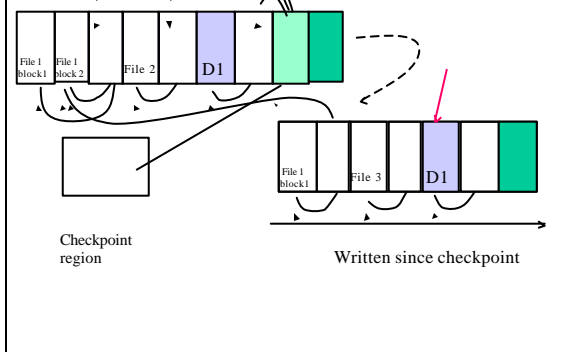
Policies

- When cleaner cleans – threshold based
- How much – 10s at a time until threshold reached
- Which segments
 - Most fragmented segment is not best choice.
 - Value of free space in segment depends on stability of live data (approx. age)
 - Cost / benefit analysis
 - Benefit = free space available (1-u) * age of youngest block
 - Cost = cost to read segment + cost to move live data
 - Segment usage table supports this
- How to group live blocks

Recovering Disk Contents

- Checkpoints – define consistent states
 - Position in log where all data structures are consistent
 - Checkpoint region (fixed location) – contains the addresses of all blocks of inode map and segment usage table, ptr to last segment written
 - Actually 2 that alternate in case a crash occurs while writing checkpoint region data
- Roll-forward – to recover beyond last checkpoint
 - Uses Segment summary blocks at end of log – if we find new inodes, update inode map found from checkpoint
 - Adjust utilizations in segment usage table
 - Restore consistency in ref counts within inodes and directory entries pointing to those inodes using Directory operation log (like an intentions list)

Recovery of the Log



Recovery in Unix

fsck

- Traverses the directory structure checking ref counts of inodes
- Traverses inodes and freelist to check block usage of all disk blocks

Evaluation of LFS vs. FFS

1. How effective is FFS clustering in "sequentializing" disk writes? Do we need LFS once we have clustering?
 - How big do files have to be before FFS matches LFS?
 - How effective is clustering for bursts of creates/deletes?
 - What is the impact of FFS tuning parameters?
2. What is the impact of file system age and high disk space utilization?
 - LFS pays a higher cleaning overhead.
 - In FFS fragmentation compromises clustering effectiveness.
3. What about workloads with frequent overwrites and random access patterns (e.g., transaction processing)?

FFS Cylinder Groups

- FFS defines *cylinder groups* as the unit of disk locality, and it factors locality into allocation choices.
 - typical: thousands of cylinders, dozens of groups
 - Strategy place "related" data blocks in the same cylinder group whenever possible.
 - seek latency is proportional to seek distance
 - Smear large files across groups:
 - Place a run of contiguous blocks in each group.
 - Reserve inode blocks in each cylinder group.
 - This allows inodes to be allocated close to their directory entries and close to their data blocks (for small files).
 - Fixed locations on disk: superblock describing key parameters of disk layout; for each cylinder group: inode space, bit map, copy of superblock



FFS Allocation Policies

1. Allocate file inodes close to their containing directories.
 - For *mkdir*, select a cylinder group with a more-than-average number of free inodes.
 - For *creat*, place inode in the same group as the parent.
2. Concentrate related file data blocks in cylinder groups.
 - Most files are read and written sequentially.
 - Place initial blocks of a file in the same group as its inode.
 - How should we handle directory blocks?
 - Place adjacent logical blocks in the same cylinder group.
 - Logical block $n+1$ goes in the same group as block n .
 - Switch to a different group for each indirect block.

Creates a *logical* locality

Clustering in FFS

- *Clustering* improves bandwidth utilization for large files read and written sequentially.
 - Allocate clumps/clusters/runs of blocks contiguously; read/write the entire clump in one operation with at most one seek.
 - Typical cluster sizes: 32KB to 128KB.
- FFS can allocate contiguous runs of blocks "most of the time" on disks with sufficient free space.
 - This (usually) occurs as a side effect of setting *rotdelay* = 0.
 - Newer versions may relocate to clusters of contiguous storage if the initial allocation did not succeed in placing them well.
 - Must modify buffer cache to group buffers together and read/write in contiguous clusters.

Creates spatial locality on a per-file basis
(for multi-block files)

LFS supports temporal locality – order of operation

Which kinds of locality on disk are better?

Benchmarks and Conclusions

1. For bulk creates/deletes of small files, LFS is an order of magnitude better than FFS, which is disk-limited.
 - LFS gets about 70% of disk bandwidth for creates.
2. For bulk creates of large files, both FFS and LFS are disk-limited.
3. FFS and LFS are roughly equivalent for reads of files in create order, but FFS spends more seek time on large files.
4. For file overwrites in create order, FFS wins for large files.

The Cleaner Controversy

Seltzer measured TP performance using a TPC-B benchmark (banking application) with a separate log disk.

1. TPC-B is dominated by random reads/writes of account file.
2. LFS wins if there is no cleaner, because it can sequentialize the random writes.
 - Journaling log avoids the need for synchronous writes.
3. Since the data dies quickly in this application, LFS cleaner is kept busy, leading to high overhead.
4. Claim: cleaner consumes 34% of disk bandwidth at 48% space utilization, removing any advantage of LFS.