CS 333 :: Meeting Notes :: Log-structured File Systems (LFS)

LFSes belong to a class of file systems called *no-overwrite* file systems. In an LFS, data is written *out-of-place* (in contrast with FFS's in-place updates). The LFS design and on-disk layout attempts to convert all write operations into sequential I/O.

Unlike many computer science concepts, LFS is descriptively named: the log-structured file system writes data and metadata in an append-only fashion, essentially treating the disk as a circular log.

Learning Objectives

- Be able to describe the trends that motivated the development of LFS
- Be able to describe the way that LFS writes data to disk
- Be able to describe the advantages of logging when compared to update-in-place
- Be able to describe best and worst-case workloads for LFS
- Be able to describe the garbage collection problem and how it is solved in LFS
- Be able to describe the way that segments were used to manage fragmentation

Trends/Realities that Influenced LFS Design

There were two **dominant** motivations for the introduction of LFS.

1. RAM sizes were increasing:
    - More and more reads were able to be satisfied from caches, so write performance was often the bottleneck.
    - **Idea: optimize writes at the (possible) expense of reads**
2. Random I/O had always been slow relative to sequential I/O, and the gap was ever growing
    - Converting common operations/workloads to from random to sequential I/O is a huge performance win.
    - **Idea: rethink disk layouts to remove random I/O from the write path, on both data and metadata operations**

1-sentence Overview of the LFS Idea

Treat the disk as a circular log, and append all updates (data and metadata) to the end of that log.

Challenges:

1. Metadata objects are often smaller than a disk block, so how can we aggregate as many metadata updates as possible into large I/Os?
2. The disk size is limited, so we eventually read the end of our log. When we "wrap around" our circular log, how do we ensure that
    - We don't overwrite "live" data, and
    - We don't revert to "random I/O"

Overview of Techniques

- *write-buffering*: keeping a small pool of pending updates in memory, and writing them out to persistent storage (e.g., a HDD, an SSD) in a batch
    - LFS divides the disk into fixed-sized **segments**, and LFS buffers updates until a segment's worth of data can be written in a batch. The LFS then finds the first free segment and streams the buffered segment to disk in a single sequential I/O (benefit: LFS pays only 1 setup cost to write many blocks).
- *checkpoints* and *dynamic metadata placement*: instead of restricting `inode` placement to a fixed sized `inode` table at a well-known location (recall FFS allocates its `inode` table across several block groups), LFS uses indirection.
    - Inodes can be written anywhere on disk
    - The in-memory *inode map* stores a pointer (disk block number) for each inode's most recently written version.
    - The in-memory inode map is written to disk periodically to form a *checkpoint*
    - The location of the most recent inode map is written to one of two fixed *checkpoint regions* (note that the inode map is logged, but the pointer to the inode map is updated in place because we need to know where to look for it).
        - The system's last *consistent state* can be initialized by reading the inode table (aka a checkpoint).
        - The system alternates between two checkpoint regions, and restores the most recent *successful* checkpoint.
- *garbage collection*: all data is written out-of-place, so each update creates stale on-disk data (and metadata!). Garbage collection reclaims the disk space that was previously allocated to stale resources.

Questions:

1. What are the advantages of using:
    - small segments?
    - large segments?
    - How do you think an appropriate segment size might be determined?
2. How long should the system wait between writing checkpoints?
    - What are some reasons to create checkpoints more frequently?
    - Less frequently?

Garbage Collection

LFS data and metadata is written out-of-place, which means that updates result in older data becoming "stale". When possible, garbage collection runs in the background to reclaim space consumed by stale data/metadata.

LFS performs garbage collection at the granularity of entire segments. The process of GC roughly proceeds as follows:

- Read multiple segments into memory
- Identify the live data in each segment
- Sequentially write the live data to one or more new segments
- Once all live data from an old segment is safely written to a new location, de-allocate the old segment

To simplify the process of identifying live data, each segment keeps additional metadata about the blocks written inside it. For each data block, the **segment summary info** contains:

- the inode number of the file that the data block belongs to
- the offset of that block within the file

Using this info, the segment cleaner can consult the current inode map to determine if the block is "live" or "stale" by comparing the inode's data pointer to the LBA of the block. If they don't match, the block can be cleaned.

Questions:

1. Why does LFS try to prioritize garbage collecting cold segments over hot segments?
2. How does segment size affect garbage collection?
    - What would be the advantage of very large segments?
    - Very small segments?
    - In the extreme, segments of one block could be used. How would this perform?
3. How often should garbage collection be run?
    - What are the advantages of running GC frequently?
    - Of running GC infrequently?
    - Can you "incrementally" perform GC, and if so, what should the increment size be?
4. Does LFS need a defragmenter like FFS?
5. In what scenarios does garbage collection become *foreground* work? In other words, should the system every stop accepting new updates in order to prioritize garbage collection?
6. How many free segments should LFS reserve for GC?

Consistency

File systems crash. Sometimes crashes are caused by software bugs, sometimes by a clumsy professor tripping on the power cord. Either way, it is important that the file system can recover to a **consistent** state. That is, some data may unfortunately be lost during the crash, but the state of the file system data structures should always represent some valid sequence of successful operations.

- LFS uses checkpoints to preserve consistency (as described above, there are two checkpoints that are alternated between, with timestamps to ensure that the latest *complete* checkpoint is used).
- If there have been updates made since the last checkpoint, LFS can attempt to "replay the log" starting at the checkpoint's state to *roll forward* to a more recent state.
    - This is one of the benefits of out-of-place updates: the old version of your data is kept around if you need it, but you can apply changes if the change has been successfully & consistently written.

Questions:

1. How many checkpoint blocks does LFS use, and why is the answer not "1"?
2. How does LFS handle a crash in the middle of a checkpoint?
3. LFS can crash in the middle of GC. When can LFS safely *delete* a GCed segment and reuse it in the system?

Comparing FS Designs

FFS is an *update-in-place* filesystem. LFS is a *no-overwrite* filesystem. Both classes of filesystems have different best and worst case workloads. Without considering long-term "aging", think about the following "simple" reading and writing patterns.

1. Suppose I write a very large file to each filesystem, and later update that file (this is not uncommon, consider a Virtual Machine Disk Image). How efficiently does each filesystem perform the following:
   - sequential overwrites to the file
   - random overwrites to the file
   - appending to the file
2. Suppose I turn on my computer and read a file that I have already written (the cache is initially empty). How efficiently does each filesystem perform the following:
   - sequentially read the entire file
   - randomly read 1 block at a time from 1,000 offsets within the file
3. Based on your answers to the above questions, is one file system clearly better than the other?
4. There are two types of locality that often arise: temporal and spatial. Can you classify the file system types in terms of how they favor each type of locality?