CS4410 MP4: Log-Structured Filesystem

You work for a filesystem company at which a mishap has occurred. A programmer named Eleet Haxor was tasked with implementing a log-structured filesystem (LFS). Eleet Haxor was 10 days into the implementation of LFS when he went out to watch a meteor shower, where he was struck and vaporized by a falling meteorite. Your task is to pick up the partially finished code and fill in any missing gaps to yield a fully-functioning filesystem. We know that Eleet's implementation was patterned after the canonical LFS implementation described in this research paper:

> *The Design and Implementation of a Log-Structured File System. Mendel Rosenblum and John K. Ousterhout. ACM Transactions on Computer Systems, 1991.* *PDF*

As with any real software artifact, the original specification is not as detailed as what one may have encountered in CS100, and there are slight variations between the description in this paper and Eleet's implementation. Part of your task will be to reverse-engineer enough of the LFS implementation to understand how it was meant to work, and fix any errors encountered along the way.

LFS Overview

In log-structured file systems, all new data is written on the disk sequentially in a log-like structure. The log structured disk is divided into segments which consist of blocks. Each segment comprises a superblock and a set of data blocks. The superblock keeps track of which data blocks are free, while the data blocks in the segment hold the file and directory contents, as well as Inodes that point to and organize these data blocks.

The main function of Inodes is to point to the blocks the file/directory data spans. Each Inode can point to approximately 100 data blocks directly. When the number of data blocks is greater than an inode can directly point, they are stored in a datablock (called an indirect block) and a pointer to this indirect block is stored in the Inode.

Unlike the traditional Unix filesystem, Inodes in LFS do not have fixed positions on disk. The latest version of an Inode is written to the next available block in the current segment (if the current segment is full, it is flushed to disk, a new segment from the disk is loaded into memory and becomes the current segment). This approach causes the Inodes to be spread throughout

the log, hence an additional data structure, called an InodeMap, is necessary to keep track of the latest position of any given Inode in use. The InodeMap keeps the mapping from virtual inode numbers to physical block addresses for every Inode.

The InodeMap is a critical data structure as it is required to locate any Inode (whose virtual Inode number is obtained by walking the directory hierarchy) for any given file or directory. It is flushed to the disk with a sync() command, which writes the InodeMap to the disk, along with a special Inode that points to all the blocks that house the InodeMap. The physical location of this special Inode is written to the superblock of the segment in which that Inode was written, along with the generation count of the InodeMap. Every flush increments the generation count, making it possible to identify the latest InodeMap on disk following a crash. On filesystem mount, the filesystem goes through every segment and checks every superblock to find and recover the latest InodeMap.

Writes and reads in LFS are very similar to writes and reads in a traditional Unix filesystem, except the position of the Inodes is not fixed. When a new file is being written, LFS first creates an in-memory Inode, writes the data blocks to the next available blocks in the segment, updates the in-memory Inode to point to the data blocks, and writes the Inode to the segment. It then modifies the parent directory block contents to include a directory entry (a name, inodeid tuple spanning 32 bytes, 28 of which hold the file name and 4 of which hold the Inode identifier). It writes the modified data blocks of the parent directory to the segment, followed by the modified Inode for the parent directory. Note that modifications do not have to propagate up the directory hierarchy, as the parent's virtual Inode number remains unaffected, and thus the parent's parent directory contents do not have to be changed.

Locating a desired file on disk is also analogous to the way this operation is performed on a traditional Unix filesystem. Suppose the user is trying to open file "/aaa/bbb/c". A search begins with the Inode for /. Each directory entry in / is examined to see if it matches "aaa". If there is a match, the virtual inode number is recovered from the directory entry. This is translated, through the InodeMap, into a physical address for that Inode. The process continues down the path for "bbb", and so on.

The aforementioned description comes from design documents found on Eleet's desk; there may well be some deviation between this description and the actual implementation. In all such cases, actual code trumps any English description of what is supposed to happen, and your task is to fix up the code to affect the intended action.

Implementation Notes

The LFS implementation is in Python, and comes with a bare bones shell. It can be invoked with "python Shell.py". The shell (should) support the following operations:

- mkfs: Before any command can be issued, a filesystem needs to be either created through formatting, or an existing filesystem needs to be formatted. "mkfs" will create a brand new, empty filesystem. The data for this file will reside in a virtual disk,

implemented as, in turn, another file named "filesystem.bin". "mkfs -reuse" will mount an existing filesystem. So it should be possible to start the shell, mkfs, create some files, exit the shell, restart the shell, mkfs -reuse, at which point the previously created files should be visible.

- create filename size: This command will create a file and fill it in with a repeating string pattern of the specified size. It is useful for quickly testing the write and read paths on the file access path.
- cat filename: This command displays the contents of a given file.
- write filename string: This command writes the specified string starting at offset 0 within the named file.
- ls directoryname: Displays the contents of a given directory.
- mkdir dirname: Creates a new directory.
- cd dirname: Changes the current directory to the named directory. Note that the "current directory" is a fiction maintained by the Shell. The file operations all take absolute paths, and the Shell transparently maintains the current directory and supplies the canonicalized absolute path to all file operations. Also note that the special ".." operator is not supported.
- sync: Flushes all in-memory data structures to the disk.
- exit/quit: Terminate the shell. An exit without a preceding sync will leave the on-disk data structures in a potentially inconsistent state, so typically all exits would be preceded by a sync operation.

Your Tasks

Your primary task is to understand the operation of the current LFS implementation, find and fix all bugs, and add any required functionality to yield a full-fledged log-structured filesystem.

From what we could understand from the incomplete code, this subtask can be broken down into several sub-tasks, and the missing parts are identified with XXX's in the code:

1. Fix sync: The function sync is responsible for writing all the contents of the memory to the disk. It has to make sure that the inode map is updated properly and then flush the segment to the disk.
2. Fix File Descriptor Write: The write function in the file descriptor is supposed to handle the write to a file or directory. It just makes the necessary calls to the inode of the file or directory, and handles essential changes to the file descriptor.
3. Fix file search: The function "searchfiledir" is intended to return the virtual inode number for a given absolute path if the path exists and None otherwise. To locate a file, a

directed search needs to be performed, starting from the root directory. So a lookup for "/aaa/bbb/ccc/ddd" should find the Inode for /, read the data blocks containing directory entries in /, locate the Inode for aaa, read the data blocks containing directory entries in aaa, locate the Inode for bbb, read the data blocks containing directory entries in bbb, locate the Inode for ccc, read the data blocks containing directory entries in ccc, and finally locate the Inode for ddd. It should return None if any component along the way is missing.

4. Write block to segment: The function write_to_newblock is used to write a given block's worth of data to an unused block in the current segment. It should mark that block as being in-use in the superblock, and return the physical block location of the block that was used to house the data. If the segment in memory fills up during the write, the full segment should be flushed to the disk. If the disk is completely full, this function should throw a file system exception saying that the disk is out of space.
5. Implement Segment Cleaner: Blocks frequently go dead/stale in LFS as new blocks are written to the segment. LFS collects and reuses such stale blocks with the help of a background thread that periodically scans the segments, the InodeMap and the associated inodes to determine which blocks are in use. It marks the unused blocks appropriately in their segment superblocks so they can later be reused. The details about the Cleaner are given in the LFS paper. There are two different approaches described in the paper regarding the Cleaner, feel free to implement either of these two (copy-and-compact or threaded-log) or any other scheme that collects and recycles the no-longer-used blocks.
6. Test and play: Once you fill in the missing pieces above, you should be able to create files, make directories, read files, ls directories, and write over existing files. Play around with these until you become comfortable with their internal operation. There are no deliverables from this subtask, but you'll find it critical for the next few operations.
7. Deletion: The delete function removes a specified file, if it exists. This involves finding the file's parent, iterating through its directory entries, finding the target to be deleted, and then writing over that directory entry. You may choose to leave a hole in the directory by writing over the deleted entry with a special pattern, or you may compact the directory and trim down its size. Note that the blocks spanned by the data are not deleted from the memory or the disk; this is periodically done by the cleaner.
8. Indirect Blocks: You should add support for indirect blocks into this LFS implementation. Make sure that files that span more than 100 blocks can be properly written to and read from the disk.
9. Extra Credit: Multithread support: The current implementation does not support concurrent access at all. Add all requisite synchronization to make it possible for multiple threads to perform simultaneous LFS operations. You should strive to support as much concurrency between filesystem operations as possible and feasible. A single giant lock

around the entire filesystem is one possible starting point, but finer grain locking is required to allow non-conflicting operations to proceed in parallel.

10. Extra Credit: Mailserver Integration: Integrate your working LFS with the mail server from the previous MiniProject. An LFS is ideally suited for holding a mailbox! But make sure that your implementation is reliable enough to hold your precious emails.

Quirks

The Eleet LFS has a very minimalistic feel, similar to the early Unix filesystem. You'll find that the filesystem makes no distinction between files and directories; it is possible to treat a directory as if it were a file. This makes reading the directory simple, as no special code is necessary. But it also makes it possible to write over a directory and mess up its contents. So don't do that, or if you do, live with the consequences!

Evaluation

Software artifacts of this level of complexity are sufficiently difficult to make manual grading infeasible. Your solutions will be graded against a battery of automatic tests. Such automated evaluation leaves no wiggle-room -- if the system crashes for any reason, no matter how small, it's as if it doesn't work at all. So be sure to test your code thoroughly, and turn in only a coherent set of files that fulfill the requisite functionality.

What to Turn In

Please turn in your code along with a README.txt file that describes how much of the required and extra-credit tasks you completed. Also turn in all code you used to test your implementation.

Your submission should be packaged into a single zip file, MP4.zip. This zip should be structured so that when we unzip your submission we should get a single folder named MP4 containing all of the python files needed to run your submission. We will deduct points if your submission does not match the described structure or is missing files necessary to run your solution out of the box.

Getting Started and Help

Start by running Eleet's code and creating a new disk by typing "mkfs". You can then try to create a file named /a of size 10, with "create a 10". You will see that this will fail, as the file creation code will look for the root directory to see if a already exists, fail to find the root, and thus return an error. You may want to turn on the DEBUG variables in Disk.py and Segment.py to get some insight into what is happening inside the filesystem. You will then probably want to start in the shell, and insert print statements extensively to see how the code is navigating through the different components. Once you figure out a way to make searchdir and write_new_block to work (and there are many such ways), you should have a mostly functioning system.

While the amount of code you need to write to complete this assignment is not much, keep in mind that writing those lines will require, like most realistic tasks in the real world, reverse engineering and understanding an existing, intricate code base. This may seem frustrating at first; start by inserting print statements along a particular code path (e.g. file create), and expand from there. If you feel so inclined, you can delete as much of the existing code as you like and rewrite it from scratch; however, we strongly advise against this, as matching the same functionality will take a lot of time.