# CS380L Advanced Operating Systems Project: 380LFS

Campbell Sinclair

December 2020

## 1 Introduction

In 1992, Rosenblum and Ousterhout published the design and implementation of the first log-structured file system, Sprite LFS, which they developed at the University of California, Berkeley for the distributed operating system Sprite [1]. They propose the log-structured file system design as an improvement over the conventional file system paradigms of the era. Speculating that systems' growing memory caches would reduce the number of read accesses to disk on average, they expected to achieve high performance by grouping small non-sequential data writes and metadata updates into large sequential writes. This strategy, they argued, had compelling advantages over the Unix FFS, as it greatly improved write bandwidth, and its layout provides more natural crash recovery. It also introduces new challenges; for example, there must be a way to locate files' data and metadata when their positions are changing on the on the log with every update, and there must be a mechanism for discovering and creating contiguous free space on the circular log so it can keep expanding to meet the capacity of the underlying storage medium.

Upon reading the paper, I was interested in the comparison between the bandwidths of Sprite LFS and Unix FFS for various access patterns. I built 380LFS, a log-structured file system that mimics many aspects of Sprite LFS's design, with Linux's FUSE interface. The goal of 380LFS is to investigate whether the original performance tradeoffs described by Rosenblum and Ousterhout still exist in a direct comparison to ext4, a modern descendant of the original Unix FFS.

## 2 Background

Sprite LFS was designed for a system with a large memory capacity, but a slow disk [1]. Reads can often be serviced from the memory cache, but writes must propagate to the underlying disk eventually, so write bandwidth is the dominant performance bottleneck. Where Unix FFS may issue multiple disk writes to implement a single file system operation, Sprite LFS maintains a circular log to which it can append all new data and metadata from an operation with one large disk write. As a result, none of the files, directories, or their metadata have a permanent home on the disk, so Sprite LFS keeps additional metadata in a fixed superblock that allows it to trace the most recent location of any data block or file inode. As files are created and altered on the system, previous versions of the files are not overwritten; they are simply invalidated. The log continuously expands as users operate on the file system, but eventually it must circle back to where it started, and a fragmentation problem arises. Sprite LFS creates contiguous free space by partitioning the log into fixed-size segments and invoking a segment cleaner when too many segments hold live data. The segment cleaner consolidates live data from separate dirty segments together to increase the number of clean segments available on the log. While the log structure introduces complexity in locating files and free space, it provides a simpler solution to crash consistency than Unix FFS. There is no need for a separate journal because the file system can restore to the latest checkpoint region and roll-forward guarantee consistency.

380LFS is a similar log-structured file system built with FUSE, the Filesystem in Userspace interface. It is a user-level application that can be mounted onto an existing file system [2]. When requests for file system operations are issued to the mounted file system, FUSE invokes handler functions in user space that implement the desired functionality. As a virtual file system, 380LFS does not reside in its own disk partition or access the disk directly. The data persists in a log file on the host file system, so 380LFS's operation handlers service requests by reading and writing to the log file.

## 3 Design

### 3.1 File Location and Reading

Like the original Sprite LFS, 380LFS writes all file data to a circular log in fixed block-size chunks (4KB). It also writes an inode, or a metadata block, for each file, which points to the file's active data blocks. To keep track of all files' inodes, 380LFS also writes inode maps to the log, which map inumbers (unique identifiers for files) to the offset into the log where the file's inode currently resides. Inode maps are also written onto the log, so their offsets are recorded in the superblock, which is a block-sized data structure that occupies a fixed position at the start of the log file. Inode 0 is reserved for the root directory, which is created when the log is initialized and written to the log. To locate a file given its name, 380LFS must first read in inode map 0, which points to the current offset of inode 0, which in turn points to the data blocks that contain root directory entries. The directory maps file names to inumbers, so if the file exists, 380LFS must read another inode map (or the same inode map as before) to locate the file's inode and its data.

When data is written to a file, several blocks of metadata must be written to the log. The data blocks of the file have new offsets, so a new inode must be written to overwrite the old offsets. That means the location of the file's inode changes too, so a new inode map must be appended and the superblock must be updated as well. Each inode in 380LFS points to up to 10 direct blocks, and it may also point to a double indirect block, which can hold references to up to 512 indirect blocks that can each refer to up to 512 additional direct blocks. This means that when writing to large files, 380LFS may need to append updated indirect and double indirect blocks as well. Each update to the file system makes two writes to the log file: the first is a buffer of all the new data and metadata that results from the update, appended to the tail of the log, and the second is an update to the superblock written to offset 0.

380LFS borrows an important property from Sprite LFS in that the live data on the log is always read-only and the free space is always write-only. Neither file system ever goes back and updates data on the log in-place. New data is appended to the log, and references to the new data are appended with it, so the old data becomes garbage.

### 3.2 Segment Cleaner

Rosenblum and Ousterhout described free space management as "the most difficult design issue for log-structured file systems" [1]. Sprite LFS divides the log into segments and tracks each segment's metadata in a segment summary block. Information about how much live data each segment holds and when each was last modified is stored in segment usage tables, which are also appended to the log. The segment cleaner cleans segments by copying their live data out to other segments. It chooses which segments to clean by consulting the segment usage tables, reads them into memory, and writes the data back out to a smaller number of segments, thereby compacting the data and creating more contiguous free space. Threading between segments is also used to take advantage of existing contiguous free space; as it circulates, the log skips over dirty segments that might be fragmented.

380LFS implements a similar segment cleaner, which is a function invoked if appending to the log causes the number of remaining clean segments to drop below a fixed threshold (20 segments). It reads 20 segments at a time, chosen with the same heuristic of cleaning "cold" segments first that Rosenblum and Ousterhout suggested in their paper: maximum benefit / cost ratio, (1-u)*age/(1+u) [1]. However, unlike Sprite LFS, 380LFS does not append segment information to the log. It stores the equivalent of segment usage tables (whose entries include segment summaries) for each segment in memory and serializes them into a "checkpoint region" in segment 0 when the file system is unmounted so they can be restored upon remounting. Every time a block is written on the log, the corresponding segment usage tables are updated for both the segment of the block's new offset and the segment of its old offset if there is an older version of the block that is no longer valid. As a result, functions that need to append to the log in 380LFS must keep track of this information.

### 3.3 Crash Recovery and Consistency

Consistency and crash recovery are significant advantages that Sprite LFS had over Unix FFS. The log-structured system ensures consistency with three guarantees: a structure is fully initialized before any other structure points to it, a resource is never reused unless every pointer to it is nullified, and a live resource always has a pointer pointing to it. Sprite LFS maintains checkpoints, which are points on the log where the file system is in a consistent state, by periodically writing information about the file system's state to redundant checkpoint regions [1]. If the system crashes, Sprite LFS restores the most recent checkpoint and rolls forward to redo any operations committed to the log.

Unfortunately, 380LFS does not provide as strong crash recovery guarantees as the original Sprite LFS. To persist its state between mounts, it serializes its segment summaries, segment usage tables, and other runtime information to a fixed offset in the log file upon unmount, but if a crash occurred that prevented this, the file system would recover its state from data that is outdated or otherwise wrong. With the current design, 380LFS could scan the log and rebuild this state from scratch, or it could conform to Sprite LFS's approach instead.

## 4 Implementation

380LFS is implemented by reading and writing to a backing log file on the host file system to service requests for file system operations. It handles the following FUSE operations:

- **lfs_init**: Creates the provided log file with the given size if it doesn't already exist. It writes the initial superblock to the log file, as well as the root entries, its inode, and inode map 0.
- **lfs_getattr**: Searches for a file in the root directory and returns information about it, such as its inumber and size. Returns -ENOENT to tell FUSE that the given file doesn't exist [2].
- **lfs_access**: Returns whether or not the current user has the given permissions to access (read, write, and/or execute) the given file. It is implemented with a call to lfs_getattr.
- **lfs_create**: Creates a new file. It allocates a new inumber, adds an entry to the root directory, and appends its new inode and updated inode map to the log.
- **lfs_utime**: Updates the given file's access time. It must read in the inode append it back to the log, along with an updated inode map.
- **lfs_truncate**: Updates the given file's size. If the new size is larger, it uses a call to lfs_write to append zeroes until the file is large enough.
- **lfs_unlink**: Deletes the given file from the namespace. It uses lfs_truncate to shorten the root by one entry after fixing the contents of the root directory. It also calls lfs_truncate to truncate the deleted file to zero so its data blocks are freed.
- **lfs_open**: Opens the given file for reading and writing. It reads in the inode and stores a reference to it as FUSE's file handle, so subsequent calls to read, write, and close can quickly find their given path's inumber without reading the root directory.
- **lfs_read**: Reads bytes from the given file.
- **lfs_write**: Writes bytes to the given file. This can be used to extend a file by writing past its end. The call appends all of the file's modified data blocks ot the log, as well as any indirect blocks and the double indirect block if applicable, the file's inode, and the corresponding inode map, so it appends at least 3 blocks for a non-empty write.
- **lfs_release**: Simply frees the file handle to close the given file.
- **lfs_statfs**: Reports some information about the file system itself, including the block size, file count, and max filename length.
- **lfs_destroy**: Writes some private data (such as the current tail and the file count) and the segment usage tables to segment 0. The next time 380LFS is mounted with this log file, lfs_init will read this information back into memory and restore the state of the file system.

## 5 Evaluation

### 5.1 Environment

The following experiments were performed in a Ubuntu 20.04.1 virtual machine running in VMware. My host machine is an HP Spectre x360 Convertible 13-ac0XX laptop running Windows 10 with an Intel Core i7-7500U 2.70 GHz CPU with 2 cores. The virtual machine runs on an equivalent processor with 4GB of RAM and about 107GB of disk capacity.

To compare the performance of 380LFS and the virtual machine's native ext4 file system, an instance of 380LFS with 1GB capacity was mounted to a mount point on ext4. The benchmark programs were executed from the mount point, where they invoked 380LFS's FUSE calls, and from outside the mount point, where their system calls were handled normally by the kernel. For both file systems, each measurement is an average of the results of 10 trials.

### 5.2 Benchmarks

Both benchmarks are modeled after the Micro-Benchmarks outlined in Section 5.1 of Rosenblum and Ousterhout's original LFS paper.

- **Small File Benchmark**: The Small File Benchmark tests performance for handling small files. The program creates 10000 files and writes 1KB of random data to each. Then, it reads back the data from each file. Finally, it deletes all 10000 files.
- **Large File Benchmark**: The Large File Benchmark tests performance for different access patterns on a large file. The program creates a file and writes 100MB of random data to it, before reading the data back sequentially. Next, it writes another 100MB of random data, but now in a random order, 4KB at a time, until all 256000 blocks have been replaced. Then, in a separate random order, it reads all 256000 blocks back into memory. Finally, it makes one last sequential read through the file's 100MB.

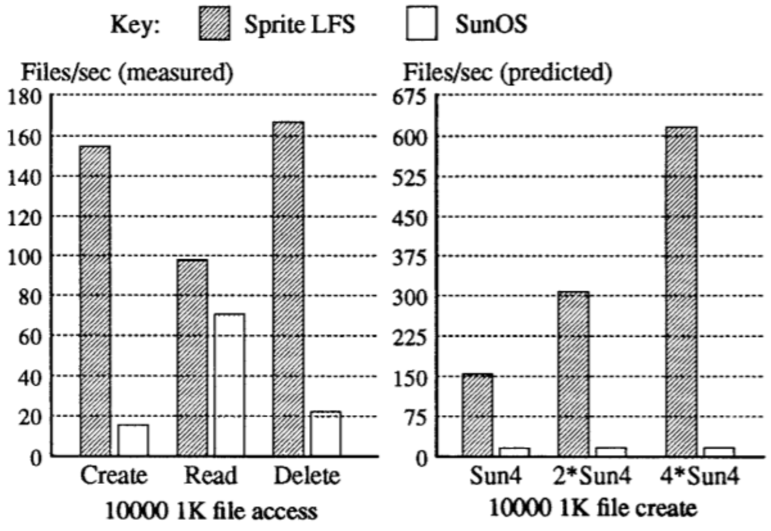

Figure 1: Rosenblum and Ousterhout's comparison of the performance of Sprite LFS and Sun OS on the Small File Benchmark

Note the very different scales for the y-axes of both benchmarks' graphs; in all cases, ext4 vastly outperforms 380LFS in bandwidth, and forunately, 380LFS outperforms the original Sprite LFS's 1992 measurements.

One interesting observation from Figures 2 and 3 is that for the small file benchmark, ext4's similar to Sun OS's in that reads are much faster than deletes and creates. Both Sprite LFS and 380LFS were faster at deleting files

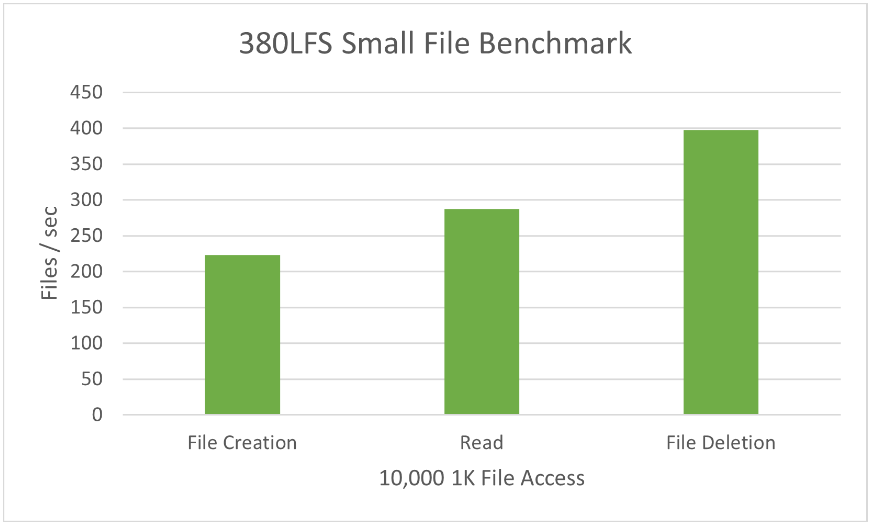

Figure 2: 380LFS's performance on the Small File Benchmark in files/second

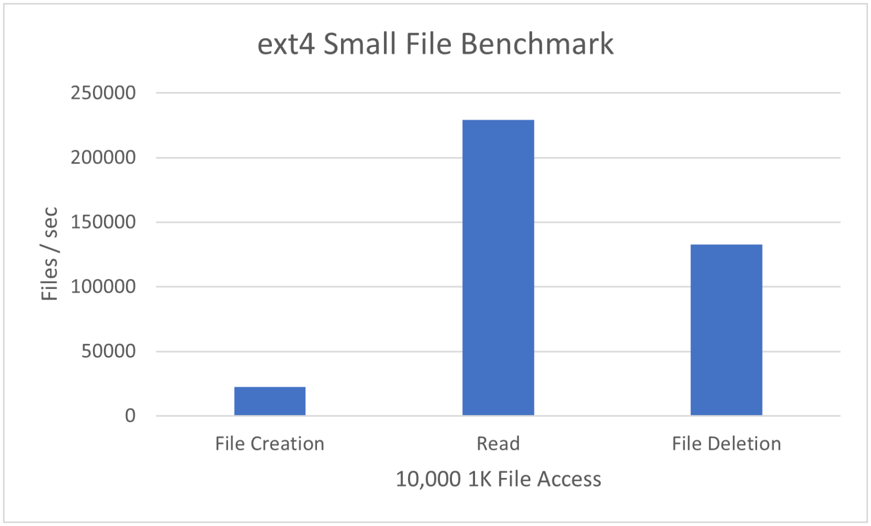

Figure 3: ext4's performance on the Small File Benchmark in files/second

than creating them or servicing 1-block reads. However, Sprite LFS seemed to benefit a lot more when creating files relative to reading them. Sprite LFS's write bandwidth was generally higher than its read bandwidth, but this is not the case for 380LFS and ext4. Even though 380LFS is log-structured, its write bandwidth compared to its read bandwidth is probably a performance bottleneck, as Rosenblum and Ousterhout predicted for systems with high memory capacity. As file creation involves both reading and writing to directories, the write bottleneck likely hurt 380LFS's create performance relative to its pure read performance.

The large file benchmark results in Figure 5 is surprising because for the access pattern that Rosenblum and Ousterhout described as Sprite LFS's worst case scenario, the sequential reread of randomly written data, 380LFS is not relatively slower than ext4. Like ext4, 380LFS has better read bandwidth than write bandwidth in all cases, and its performance is significantly slower for the random read than either sequential read.

380LFS was designed to imitate Sprite LFS and reproduce its performance results, but there are several possible reasons that its performance is more closely aligned with Unix FFS and ext4 than SpriteLFS. A major reason for this outcome, I suspect, is that 380LFS is implemented as a large log file on ext4. It makes system calls to request I/O instead of accessing the disk directly as a kernel-level file system does. Small random writes to the log file are

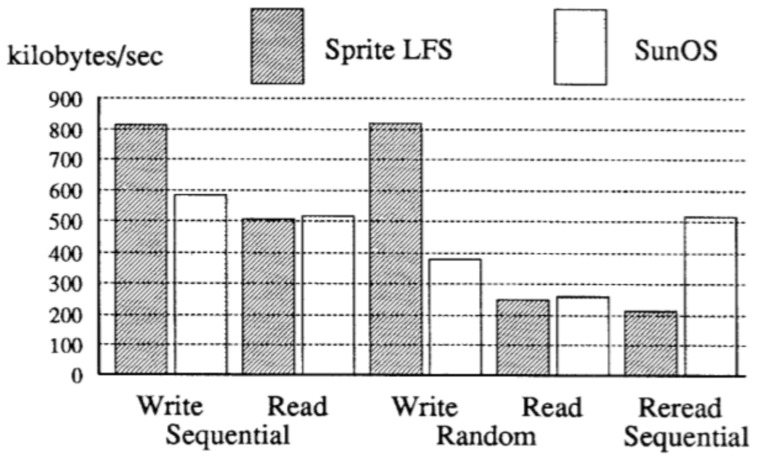

Figure 4: Rosenblum and Ousterhout's comparison of the performance of Sprite LFS and Sun OS on the Large File Benchmark

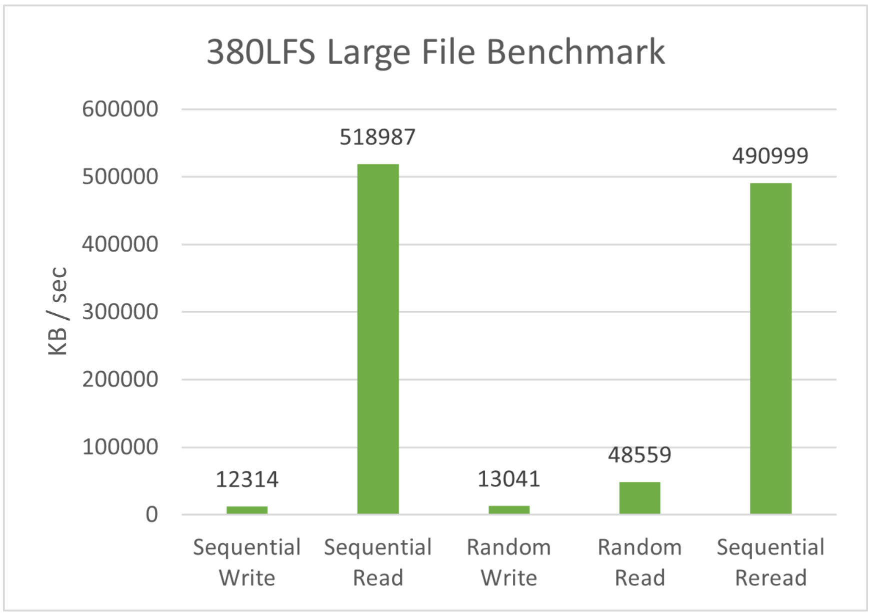

Figure 5: 380LFS's performance on the Large File Benchmark in KB/second

probably grouped into larger sequential writes to the underlying storage medium anyway. Also, 380LFS is likely not as purely "log-structured" and optimized as Sprite LFS. As a result, the conceptual benefits of a log-structured file system did not materialize in 380LFS as I envisioned.

## 6 Limitations and Future Work

380LFS is missing several key file system features, but it could be expanded to include them by implementing more FUSE operations. Currently, the most glaring omission from 380LFS is its lack of support for subdirectories and the readdir system call. I originally planned to record an experiment about path traversal, but I did not implement the FUSE readdir callback in time. This is a crucial feature for any file system, so I intend to add it in the future. Other useful features that could reasonably be built on top of the current 380LFS design are hard/soft links and

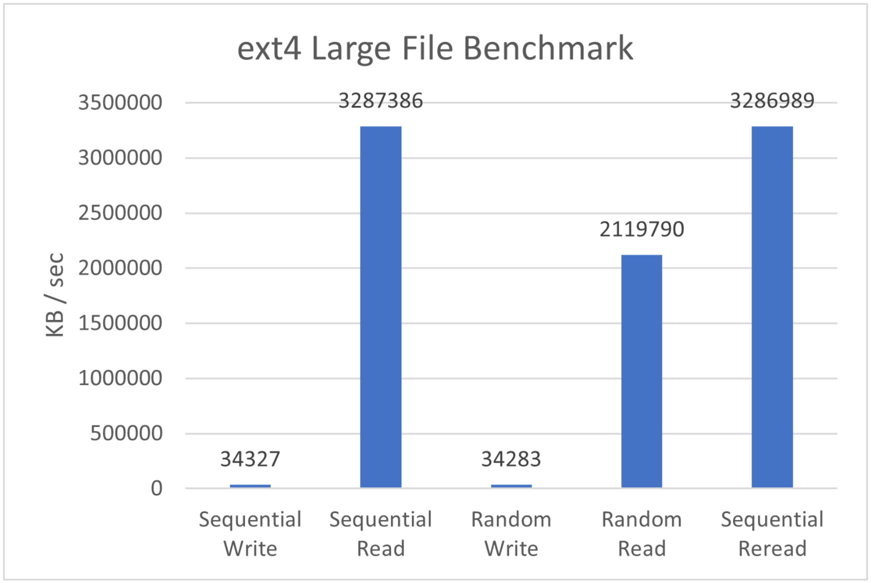

Figure 6: ext4's performance on the Large File Benchmark in KB/second

chmod.

Sprite LFS includes some features and optimizations that are missing from 380LFS, such as its better crash recovery system. Sprite LFS caches inode maps in memory to avoid reading and writing them to disk for every operation, and while 380LFS does this for segment summaries and segment usage tables, its performance would benefit greatly from this optimization to cut down on writes to the underlying file. Sprite LFS's inode maps also include version numbers for each inumber that are incremented whenever a file with that inumber is deleted or truncated to 0. This allows the segment cleaner to quickly mark live blocks as unused if their corresponding inumbers have older version numbers than their respective inode map entries.

## 7 Conclusion

The original log-structured file system had a creative design that diverged from the existing file system design paradigms of the time. It sought to provide better fault tolerance without any sacrifice to performance. Between the LFS and FFS file layout schemes, it is interesting to theorize about situations in which one outperforms the other. I built 380LFS to replicate Rosenblum and Ousterhout's results, and while it did not achieve that, it did demonstrate that their prediction about increasing memory capacity making write bandwidth a file system performance bottleneck came to fruition.

:Total time: About 150 hours

## 8 References

1. Rosenblum, M., and Ousterhout, J. K. The Design and Implementation of a Log-Structured File System. TOCS, 1992 https://web.stanford.edu/ouster/cgi-bin/papers/lfs.pdf

2. libfuse API Documentation. https://libfuse.github.io/doxygen/index.html