CS552 Fall 2019

LFS Project

January 25, 2019

Phase 1 Due: March 18, 2019 at 11:59pm
Phase 2 Due: May 1, 2019 at 11:59pm

1. Overview

This semester you will implement a log-structured file system (LFS) that allows applications to access files and directories stored in a log. The functionality is similar to that described in the LFS paper, although I've simplified it somewhat. Your LFS will be implemented on a simulated flash drive.

This project will be done in two phases and in groups of two.

2. Architecture

You will implement your LFS as a process that reads and writes flash in response to requests made by FUSE, which in turn are caused by file accesses by application programs. I will provide you with a Flash module that reads and writes a virtual flash drive stored in a regular Unix file. You must use FUSE and you must use the Flash module, what you do between these two is up to you, although you must justify the design. I would suggest the following hierarchical architecture but you are welcome to develop your own. I've sketched out the basic functionality for the layers but not provided all the details.

2.1 Flash

You will be provided with the Flash layer that emulates flash memory. Functions are provided to create, open, read, write, and close flash memory, as well as determine the wear level on individual erase blocks. You should not make any changes to the Flash layer (i.e. do not modify flash.c or flash.h):

```
Flash_Create(filename, wearLimit, blocks)

flash = Flash_Open(filename, flags, *blocks)

Flash_Read(flash, sector, count, buffer)

Flash_Write(flash, sector, count, buffer)

Flash_GetWear(flash, block, &wear)

Flash_Erase(flash, block, count)

Flash_Close(flash)
```

filename is the name of the virtual flash file, *flags* specifies the flash behavior (you may want to specify "silent" and "async" for development, but not when you turn it in), *blocks* returns the number of erase blocks in the flash, *wear* is the current wear level of the specified block, *sector* is the starting offset of the operation in sectors, *block* is the starting offset of the operation in blocks, and *length* is the length of the operation in sectors or blocks, as appropriate. See flash.h for documentation on how to use these functions.

2.2 Log

The Log layer is responsible for creating the log that is stored on the flash.

```
Log_Read(logAddress, length, buffer)

Log_Write(inum, block, length, buffer, &logAddress)

Log_Free(logAddress, length)
```

logAddress indicates the log segment and block number within the segment, *inum* is the inode number of the file, *block* is the block number within the file, *length* is the number of bytes to read/write/free, and *buffer* contains the data to read or write.

2.2.1 Log Format

The log consists of fixed-size segments and each segment consists of fixed-size blocks. You should reserve one or more segments at the beginning of the flash to hold a superblock containing per-file-system metadata such as segment size, block size, number of segments, segment usage table, checkpoint regions, etc. Similarly, you should reserve one or more blocks at the beginning of each segment to hold per-segment metadata, e.g. the segment summary that contains information about the blocks in the segment.

You must write a utility called *mklfs* that creates an empty LFS. It is responsible for initializing all of the on-flash data structures, metadata and creating the root directory and its initial contents.

You must also write a utility called *lfsck* that scans an LFS and reports any errors such as files that do not have directory entries, directory entries that point to unused inodes, wrong segment summary information, etc. You will find this utility invaluable for debugging.

2.2.2 Log I/O

All I/O to the flash must be done in units of segments; that is, the smallest amount of data that LFS can read from or write to the virtual flash is one segment. The file system component will want to read and write in units of file blocks (see the next section), which are smaller than segments. For reads this means that LFS may be forced to read an entire segment from the flash just to access one file block. To reduce the performance penalty,

LFS must implement a segment cache. The cache is stored in memory and contains the N most recently accessed segments (N is specified via a command-line option).

LFS only writes to the end of the log, therefore writes are handled by keeping the log "tail" segment in memory and only writing it to flash when it is full (it should also be added to the segment cache at this time). There are a couple of issues with this. First, blocks may die after they are placed in the tail segment but before the tail segment is written to flash. You should keep track of free space created by these dead blocks and reuse it for new blocks. This means that blocks may not appear in the tail segment in the order they were written, so you will have to record this order in the segment summary if you plan to implement roll-forward. The alternative, leaving dead blocks in the tail segment, causes far too many problems.

Second, during a checkpoint it may be necessary to write the tail segment to flash before it is full. This is ok, although you must ensure that the unused space is noted as such in the segment usage table and segment summary. The next log I/O should use a new segment, leaving the unused space in the partial segment to be reclaimed by the cleaner (this is simpler than trying to fill the unused space).

Third, applications may read blocks from the tail segment before it is written to the flash.

2.3 File

The file layer is responsible for implementing the file abstraction. A file is represented by an inode containing the metadata for the file, including the file type (e.g. file or directory), size, and the flash addresses of the file's blocks. The flash addresses are stored in a UNIX-like inode structure in which the inode contains four direct pointers to the first four blocks of the file and one indirect pointer to a block of direct pointers.

```
File_Create(inum, type)
File_Write(inum, offset, length, buffer)
File_Read(inum, offset, length, buffer)
File_Free(inum)
```

inum is the inum of the inode of the file to be accessed, *offset* is the starting offset of the I/O in bytes, and *length* is the length of the I/O in bytes.

Your file layer will also need routines for changing a file's metadata such as permissions, owner, group, etc.

2.4 Directory

The Directory layer is responsible for implementing the directory hierarchy.

2.4.1 Ifile

Inodes are stored in a special file called the *ifile*. Note that the ifile is simply a file with a special type -- do not store the inodes outside of the ifile. Storing the inodes in the ifile allows you to grow the number of inodes incrementally, and to access the inodes using the File routines. For example, to read an inode for a particular file the Directory layer calls `File_Read` and passes it the inum for the ifile and the proper offset for the inode of the desired file. The inode for the ifile itself is a bit tricky to handle since you need the inode in order to find the ifile to get the inode. Break the circularity by storing the ifile's inode in the checkpoint (note that you use the File routines to manipulate the ifile itself by passing the ifile's inum to the File routines).

The ifile must be readable by user-level processes via the path `/.ifile` relative to the root of the mounted LFS. Note that the ifile is read-only and cannot be deleted.

2.4.2 Directories

A directory is simply a special kind of file that contains an array of <name, inum> pairs. Every directory must have the standard '.' and '..' entries, and multiple links to the same file must be supported. Do not implement directories any other way.

2.4.3 Special Files

You must support symbolic links, but you do not need to support other types of special files such as devices, named pipes, etc.

2.4.4 Hard Links

You must support hard links.

2.5 Cleaner

The cleaner is responsible for cleaning the log. Since it needs information about both the log and files it will have to use both the File and Log routines. It should use the same segment usage table and cost/benefit calculation as the LFS paper to determine which segments to clean. A segment is cleaned by copying its live data to the end of the log. Cleaning is controlled by two parameters to the cleaner. The first is the minimum number of free segments before cleaning begins. The cleaner starts cleaning when the number of segments falls to this level. The second is the number of free segments at which point the cleaner stops cleaning. Note that if the flash is full the cleaner may not be able to produce the required number of free segments and will simply run forever. You can handle this by having the I/O return "flash full" if the cleaner cannot make progress because the flash is full.

You may make two assumptions to simplify cleaning. First, you may check the number of free segments before performing an operation (e.g. writing to a file) and decide whether or not to clean. In other words, you don't need to clean during an operation. Second, cleaning can be synchronous. You don't need to clean while simultaneously handling file system operations. These two simplifications also imply that the log contents should not change during cleaning so that no synchronization is necessary between the cleaner and other file system operations.

3. Crash Recovery

Your system should recover from a crash by periodically checkpointing its state, including when the file system is unmounted. After a crash the system resumes as of the most recent checkpoint. The checkpoint interval is configurable and is the number of segments written between checkpoints. For example, a checkpoint interval of 1 means that a checkpoint will occur after every segment is written. To simplify things, it is ok to defer a checkpoint until the current write completes, even if that will exceed the current checkpoint interval. For example, if the interval is 1 but the current write spans 10 segments, it is ok to complete the write then do the checkpoint. You should not start any new writes until the checkpoint completes, however.

In the LFS paper checkpoints are written to a fixed location on disk. There are actually two checkpoint locations and LFS alternates between the to avoid corrupting the entire file system if there is a failure while writing the checkpoint. When LFS starts it uses the most recent valid checkpoint. Your LFS stores data on flash, so the checkpoint locations cannot be fixed otherwise the flash sectors that store them will wear out. Instead, the checkpoint locations should be moveable to support wear-leveling. Your superblock should contain the addresses of the current checkpoint locations.

4. FUSE

Your LFS consists of a user-level process that makes use of FUSE to service filesystem operations made by unmodified application programs. You must implement at least the following FUSE routines: *getattr*, *readlink*, *mkdir*,*unlink*, *rmdir*, *open*, *read*, *write*, *statfs*, *release*, *opendir*, *readdir*, *releasdir*, *init*, *destroy*, *create, link, symlink, truncate, rename*, *chmod*, and *chown*.

Make sure you pass the '-s' option to `fuse_main` so that FUSE runs in single-threaded mode. This way you do not need to worry about concurrency in your LFS implementation.

Your LFS process has the following command-line syntax:

lfs [options] *file mountpoint*

Where the options consist of:

-f

Pass the '-f' argument to `fuse_main` so it runs in the foreground.

-s *num*, --cache=*num*

Size of the cache in the Log layer, in segments. Default is 4.

-i *num*, --interval=*num*

Checkpoint interval, in segments. Default is 4.

-c num, --start=num

Threshold at which cleaning starts, in segments. Default is 4.

-C num, --stop=num

Threshold at which cleaning stops, in segments. Default is 8.

The *file* argument specifies the name of the virtual flash file, and *mountpoint* specifies the directory on which the LFS filesystem should be mounted.

5. Utilities

5.1 mklfs

You must write a command to create and format a flash for LFS. It should use Flash_Create to create a flash memory, then initialize all your on-flash data structures so that that your LFS process can access it properly. The initial filesystem should be empty, consisting of only the root directory and the '.', '..', and '.ifile' entries. The *mklfs* command has the following syntax:

mklfs [options] *file*

where *file* is the name of the virtual flash file to create and the options consist of:

-b *size*, --block=*size*

Size of a block, in sectors. The default is 2 (1KB).

-l *size*, --segment=*size*

Segment size, in blocks. The segment size must be a multiple of the flash erase block size, report an error otherwise. The default is 32.

-s *segments*, --segments=*segments*

Size of the flash, in segments. The default is 100.

-w *limit*, --wearlimit=*limit*

Wear limit for erase blocks. The default is 1000.

The *mklfs* command should exit with a return status of '0' if there are no errors, '1' otherwise. The *mklfs* must use the Flash library.

5.2 lfsck

You must write a command that checks an LFS for consistency. The *lfsck* command has the following syntax:

lfsck *file*

where file is the name of the virtual flash file to check. Note that *lfsck* doesn't actually fix any errors, it simply reports them. *lfsck* must check for at least the following errors:

- in-use inodes that do not have directory entries
- directory entries that refer to unused inodes
- incorrect segment summary information

As an implementation suggestion you might split lfsck into two parts -- the first in C that uses the Flash layer to convert the LFS data structures into JSON, and the second in a high-level language that processes the JSON and performs the consistency checks.

6. Turnin

6.1 Phase 1

Write *mklfs*, *lfsck*, prototypes of the Log and File layers, and a simplified Directory layer. The simplified Directory layer supports a single root directory (i.e. all files must be in the root directory). The Log layer need not do cleaning, the File layer should only implement direct blocks, and inodes should be stored in the ifile. The Log layer need not do periodic checkpointing, however it should create a checkpoint when the file system is unmounted.

6.2 Phase 2

Implement the remainder of the assignment.

6.3 Logistics

This project will be done in teams of no more than two. Since working in groups means that there is a danger of one person not carrying his or her load, I'm likely to quiz each group member orally on any part of the system during the final demos. Each group member must be familiar with the overall design and structure of their group's project.

Turn in your phases using GradeScope. You must include a design document (PDF preferred) for both phases. Be sure to include what does and doesn't work, as well as any cool features you implemented. I will test your Phase 1 on lectura so make sure it compiles and runs there. You will demo your Phase 2 to me at the end of the semester.

6.4 Testing

Do NOT write the entire project then try to test it. It will never work. One of the benefits of a hierarchical organization as proposed by Dijkstra is it allows for independent testing of layers. You can, for example, extensively test your Log layer and make sure it works properly before trying it with the File layer. Similarly, you can extensively test your File layer without the Log layer by stubbing it out -- create a stube Log layer that implements `Log_Read`, `Log_Write`, and `Log_Free` by simply reading and writing files or even memory. Make sure each layer functions properly on its own before composing it with another layer. DO THE FUSE INTEGRATION LAST. FUSE significantly complicates things so give yourself plenty of time to get it working, but don't use FUSE from the beginning.

7. Extra Credit

Implement roll-forward. Roll-forward requires that you store additional information in the log so that the metadata can be updated properly. For example, a file block may have information that indicates to which file it belongs so its inode can be updated. One of the difficulties in supporting rollforward is maintaining consistency between directory entries and the inodes to which they refer. The LFS paper describes a directory log that serves this purpose. As an alternative you can add two fields, *action* and *links*, to each directory entry to support directory logging. Instead of writing out a separate directory log entry, write the directory block itself before the inode to which it refers. Use the *action* field in the entry to indicate the action that occurred, and the *links* field to indicate the new number of links to the file. Unchanged entries use a "nop" action and have an invalid *links* field.