CS 552 (Spring 23): Advanced Operating Systems

LFS Project

Phase 1 due at midnight, Fri 7 Apr 2023
Phase 2 due at midnight, Wed 3 May 2023

1 Overview

In this project, you will be writing a log-based filesystem (see the paper that we read in Week 2). For your backing "disk," you will be using a library that I provide, which pretends to be a Flash drive. You **must not** use any other file or disk to store the data for your filesystem.

Requests to your filesystem will be sent from FUSE; see the "FUSE Intro" that I have also posted on the class website. You will not be required to support every possible FUSE operation, and so your filesystem will be rather limited; however, you will support the most important, core operations.

```
FUSE -> your LFS code -> Flash library -> real file -> OS
        ^^^^^^^^^^^^^^
        | | | | | | |
         write this
```

You will write two programs: `mklfs` and `lfs`. The first program will create a new Flash device (see `Flash_Create()` in `flash.h`) and fill it up with the most basic information about an LFS filesystem; it will then terminate. The second program will open up an already-existing LFS filesystem (`Flash_Open()`) and then connect to FUSE (`fuse_main()`). Your code then will simply wait for the user to make requests, which will come from FUSE using the callbacks that you provided for it. You will read and write the Flash filesystem, and figure out what responses to give to FUSE.

2 Terminology

Sector
In this project, a "sector" is one chunk of data on the Flash disk. It is always the same number of bytes; see `flash.h`.

Block
When we create a new LFS instance with `mklfs`, we will decide how large each block is; this will never change as long as this LFS exists (although other LFS instances might have a different block size).

The block size will always be an integer multiple of the sector size.

Flash Erase Block
While we can read individual sectors on the Flash drive, it is **impossible** to change any already-written sectors without erasing them first. All Erase operations must be aligned on the Flash Erase Block size. This size is always the same, no matter what LFS instance - and you can find its size in `flash.h` .

Segment
An LFS segment is a set of blocks, all in a row. Each LFS instance will have a different LFS segment size, although it will always be a multiple of the Block Size for that instance - and also a multiple of the Flash Erase Block size.

We set the segment size in `mklfs`, and it will not change after that.

Important: Since this is an LFS, you must **never** write any blocks unless you are writing a complete, contiguous segment. You may read single blocks whenever you want - but you **must not** write anything other than a complete segment. (Other than checkpoints, and the disk header than you write in `mklfs`.)

3 Using Flash

A flash disk has strange read/write properties. While it is possible to read individual sectors at any time - and it's also possible to write individual sectors[1], it is **impossible** to modify any sector once you've written it, unless you erase a complete, contiguous section, called an "Erase Block." The size of the Erase Block is usually many sectors.

In your LFS, you will only write complete segments, all at once - it is illegal to write to individual sectors or blocks, or to modify anything that is already written. Instead, your Cleaner will gradually migrate blocks out of an old segment (by rewriting them at the end of the log). Of course, as you migrate the blocks away, it is still impossible to modify the segment on disk; thus, if you want to keep some ongoing metdata about the cleaning process, you must store this metadata elsewhere in the log.

Once a segment is **completely** empty - every single block has been replaced or rewritten later in the log (or, simply deleted because you deleted the file that used the block), then you may re-use the segment. Erase it, and then write a new segment in its place.

4 LFS Requirements

You must use a Log-Based Filesystem to store the user data (and any metadata that you choose to record). You can use the paper as an excellent baseline for how the system could work, but I am giving you **wide lattitude** to change how the system works. So long as it's a log-based filesystem, I'm OK with just about anything.

[1] It's **possible,** but it's illegal for you to do this!

So what does a log-based filesystem mean? Here are a few simple rules. (Talk to me if you think that you're doing something that might violate the letter or the spirit of this spec; I'm happy to discuss it with you.)

- Your system must never modify blocks that are currently in use. You will continually be writing out **new** versions of many blocks (in a new location), but you must not modify the old.
- All writes (except for checkpoints) must be long, contiguous writes, of entire segments.

  You are allowed to read individual blocks as often as you like, however!
- You must write periodic Checkpoints, to a well-known location on disk. You **must** use at least two checkpoint buffers, so that it is possible to overwrite one without corrupting the other. During LFS init, you will read both, and accept the valid one; if both are valid, then you will accept the newer one.
- You must have a Cleaner, which finds old, partial segments, and rewrites blocks at the end of the log, thus emptying the segments. Make sure to **never rewite an out-of-date version of a block** - always rewrite the newest version.

  Not required until Phase 2.
- Your code **must not** blindly clean a segment; if every single block in a segment is still valid, then you must not clean it.

  On the flip side, a segment that has only one invalid block must be cleaned **eventually** (if you can't find something better). It might not get cleaned for a long time, but it must not stay partially-full forever.

  Not required until Phase 2.
- You must re-use cleaned segments; the same physical space will be used to hold new segments that you compose in the future.

  Not required until Phase 2.
- All metadata that you store in the system must be stored in the log; just like ordinary blocks, these can never be modified - simply replaced with newer versions, and then the space re-used. (Some metadata can be stored in a checkpoint, but beware - you want to keep checkpoints as small as possible!)

  You may, if you wish, use ordinary files to store your metadata. Or (as suggested in the paper), you may keep your metadata and user data separate, recording them as different things in the log.

5 Things You Don't Have to Do

To simplify the project:

- You are not required to implement "roll-forward." That is, when you open up an existing LFS instance, you may use the latest checkpoint as the "end of log," and ignore any segments written after that.

  If you wish to support roll-forward, you are allowed to do so. But I won't require it.

- You are not required to ensure that complex operations (such as creating a new file) are atomic (since we don't have roll-forward). Our checkpoints are our units of atomicity.

- You are not required to implement the enhanced Cleaner algorithm discussed in the paper (that takes into account both how many blocks are free, and also how recently they've been modified). While a Cleaner is required, feel free to be relatively naive about how you choose which segment to clean.

- When loading your LFS instance from disk, it is **permissible** to do **many** reads (such as scanning all of the segment metadata in all segments). This would not be acceptable in a real system (users hate it when it takes a long time to mount a filesystem), but we'll allow it here.

  But if you can store the metadata in the log, and read it quickly - that's nice!

6 FUSE

You must support the following FUSE operations. Some are required only for Phase 2:

- `readdir` - Lists the set of files in a directory. You are not required to support sub-directories until Phase 2, but in Phase 1, this should work just fine for the root directory.

- `getattr` - Gets basic information about a file or directory. You must fill in the following fields for directories:
  `mode, uid, gid, nlink`

  You must fill in the following fields for files:
  `mode, uid, gid, nlink, atime, mtime, size`

  You are not required to give real, correct values for the owner, group, or modification times; I suggest that you call `getuid()`, `getgid()`, and `time(NULL)`.

- `utimens` - Linux expects to be able to update the access time for files, but you are not required to store it. If you wish, you may hard-code this function to always return 0, and simply **ignore** whatever Linux tells you.
- `create` - Creates a new ordinary file (empty).
- `unlink` - Removes a file from a directory. Cannot be used to remove directories. (Note that you will need to keep track of `nlink` for each file, so that you can handle hard links.)

  Not required until Phase 2.
- `link` - Creates a hard link, between a file that already exists and a new directory entry. They must share the same inode, and you must not delete the file from disk until all of the hard links have been `unlink()`ed.

  Not required until Phase 2.
- `open` - Opens a file for later reading or writing. You must confirm that the path is valid; return `-ENOENT` or `-ENOTDIR` if appropriate.

  I encourage you to store some data about the open file in the `fh` field of the `struct fuse_file_info` struct. For instance, you might record the inode number of the file that you found, so that you wouldn't need to search for it again.
- `release` - Closes a file. Remember to clean things up, if you allocated anything in `open()`.
- `read` - Reads bytes from a file. The parameters give the starting point, and the number of bytes to read; read into the buffer provided. If the request takes you past the end of the file, then read as many bytes as you can, and return the number of bytes read. (Return 0 if you cannot read anything, as negative numbers indicate errors.)
- `write` - Same parameters as `read()`. Extend the length of the file, if it takes you past the current end of the file.
- `mkdir` - Creates a new directory.

  Not required until Phase 2.
- `rmdir` - Removes a directory (it must be empty).

  Not required until Phase 2.

7 Required LFS Flexibility

When we create an LFS instance, there are two parameters that we provide to `mklfs`, which can affect the geometry of your LFS. I will do most of my testing with the **default values** - and so if you don't support this flexibility, you can still earn most of the credit. But to earn full credit, you need to support these.

In addition to the two listed below, `mklfs` is given a third parameter: the size of the entire disk, in segments. You **must** support me setting this to any value ≥ 16.

Block Size
The block size is the smallest size that your LFS can write; although files can be only a few bytes long, you must always write data to the log in blocks. (Small user files may only take up part of a block, and similarly with the end of a user file.)

The default block size is 2 Flash sectors (1024 bytes). In all circumstances, the block size will be an integer multiple of the Flash sector size.

Segment Size
The segment size determines the size of your long, contiguous writes - and for your Cleaner. The default segment size is 32 blocks. (Thus, with the default block size and the default segment size, a segment is exactly 32KB.)

The segment size will always be an integer multiple of the block size.

The segment size will always be an integer multiple of the Flash erase block size. The segment size will always be (at least) 3 Flash erase blocks long.

8 The Tools

You are required to write two tools: `mklfs` and `lfs`.

8.1 `mklfs` [opts] image_file

`mklfs` creates a new LFS image, and stores it into `image_file`. Each of the `opts` has a default value; if the option is not given, then use the default.

- `-b size`
  Size of a block, in sectors. Default is 2.
- `-l size`
  Size of a segment, in blocks. Default is 32. Report an error if the size of a segment is not an integer multiple (≥ 3) of the Flash erase block size.
- `-s segs`
  Size of the flash disk, in segments. The default is 100.
- `-w limit`
  Wear limit for erase blocks.[2] Default is 1000.

[2] See `Flash_Create()` in `flash.h`

9 `lfs` [opts] image_file mount_point

The `lfs` program loads up an existing LFS image from disk. It discovers basic parameters (block size, segment size, num segments) from the disk itself (and the Disk Header that you write). It then starts up a FUSE filesystem, which will stay online until you kill the program.

Like `mklfs`, you should accept several arguments (each has a default):

- `-i num`
  Checkpoint interval, in segments. Default is 4.
- `-c num`
  Threshold (number of free segments) at which cleaning starts. Default is 4.
- `-C num`
  Threshold (number of free segments) at which cleaning stops. Default is 8.

You should start the FUSE filesystem using the `fuse_main()` call (see my example program, and the documentation I've provided); however, its `argv` will be different than yours. Construct one to make FUSE do what you need it to you.

You should pass (at minimum) the following flags to FUSE:

- `-s`

  Tells FUSE to run single-threaded
- `-f`

  Tells FUSE to run in the foreground; don't kick off a background server (a.k.a. daemon).
- `-o auto_cache` (Recommended, not required)

  Tells FUSE to be a little smarter about how it caches file contents (so that we can test your `read()` code better). If you don't pass this flag, then FUSE will often not call your `read()` function.
- `mount_point`

  The last parameter must always be the name of a directory (it must already exist, but be empty) where the filesystem will be mounted.

10 Hints, Tips, and Tricks

None of the following statements are binding on you; feel free to ignore them. But I think that they might be helpful to you:

- Be single threaded! A real LFS would be multi-threaded and asynchronous, to maximize performance. You don't want to deal with that complexity. Make your code simple, by making all of the operations synchronous (you block until they happen). And make sure to pass the `-s` argument to FUSE, so that it will only send you requests on a single thread, too.

- Don't write your code to take checkpoints in the **middle** of operations! Instead, at the end of each FUSE operation, check to see if you have hit the "checkpoint threshold," and take a checkpoint then.

- **Avoid caching.** I know, it's really tempting to go around, caching everything all the time. And there are a few things (like all of the Inodes) which I think make sense. But in my experience with this project, the more things that you cache, the harder it gets to handle all of the corner cases.

  Your LFS will be slow, that's OK. Live with it.

- Initializing a new LFS is surprisingly complex! I found it easier, in my code, to have `mklfs` start up a new LFS instance in memory, and then **take a checkpoint.** Since I have to write checkpoint code anyway, for the regular `lfs` program, why not use the same logic for `mklfs`?

- Store directories as files; the contents of a directory can just be an array of simple structs, that map names to inodes.

- Use a Disk Header - that is, have something, stored in Sector 0 of the disk, which gives the basic parameters of the system, such as the block size and segment size. **Never change this sector, no matter how long the LFS instance exists.**

- Use Segment 0 to hold basic disk data, including the Disk Header, and two checkpoints. To make sure that you have space for the checkpoints, I will ensure that the segment size is never smaller than 3 Erase Blocks - so use the first Erase Block for the Disk Header, and the next two for Checkpoints.

- Use a Segment Header - a chunk of data, at the head of each segment, which describes what is in the segment. This should include basic data (such as a Sequence Number, which allows us to tell in what order the segments were written), and also information that allows you to know which blocks are stored in the segment.

- Make your checkpoints as **tiny as possible** by putting your LFS metadata into the log, instead of into the checkpoint. My checkpoint only has two fields: the seqNum of the most recently flushed segment, and the physical location of Inode 0 inside that segment. Can you make yours as small, and still access all of the metadata you need?

- Store metadata in files, as much as is practical. Why implement special code to handle (for example) Cleaner metadatat, when you can just read/write a special file? (No need to go crazy with this, though - try to avoid things that cause cyclic dependencies.)
- Annotate key data structures with "magic" strings. My Disk Header, Checkpoint, Segment Header, and Inode all have small `char[]` buffers at the start of them, and I put standard strings into them. It makes it a lot easier to read the disk, if you are looking at it with a hexdump!
- Use hexdump to see your disk contents!

```
hexdump -C your_lfs_disk
```

11 Testing

You are not expected to perform "whitebox" testing of your code (that is, connecting to small components and testing them individually) - although I'm very supportive if you choose to do so. Most of your testing however, will probably be using the Linux filesystem directly - try to do simple things, like

```
touch foo
```

and see if your filesystem can create a new file. Try to write to a file; try to read from it. Use commands like `cp, cat, diff, sha256sum, echo,` and any others you can think of to exercise your file in a variety of different uses.

For more advanted testing, write a C program that uses things like `fseek()` to jump around in a file.

I do not plan to release my own testcases to the class - but I **encourage** you to share your own testcases with other students! Please post them to Discord, so that everybody can get the benefit of the testing code that you've written.

12 Teams

You may work alone, or in a team of two. Note that you cannot switch teams between Phase 1 and Phase 2.

13 Turning in Your Solution

You must turn in your code using GradeScope. For both phases, you must turn in:

- A design document (PDF preferred). Be sure to describe what works and what doesn't, as well as any cool features you've implemented, or any noteworthy problems that you've encountered.

- Your entire code (except for `flash.c, flash.h`).
- A Makefile which will build your code (both `lfs` and `mklfs`). If I need to type anything other than `make` to build them, include instructions how to do so.

14 Final Demo

Near the end of the semester, each team will set up appointments with me, where they will give me a demo of their code running. I will then discuss the project with both partners, to see if they both understand how the code works (I don't want one student to do most of the work, and the other to just ride along).