Operating Systems – Fall 2020 Project 3

1. Introduction

In project 3 and 4, we will be building a file system that mounts and works on the real Linux. In project 3, you will get yourself familiar with the tools and implement basic functionalities of the file system, and then in project 4, you will complete the file system. The goal of the project, in addition to letting you learn more about file system structure, is allowing you to practice some software engineering methods in a small-scale project.

Particularly, we will build a log structured file system (LFS), as we discussed in lecture, that use the log as the first-class citizen to store all data and metadata in the file system. The original LFS paper (https://web.stanford.edu/~ouster/cgi-bin/papers/lfs.pdf) provides lots of details on the design, but you will only need to implement a subset of all the functionalities listed as below.

1.1 Introduction to FUSE

In order to enable mounting the filesystem on real Linux without hacking into the kernel, we will use a popular framework called Filesystem in User Space (FUSE).

From Wikipedia: FUSE is a software interface for Unix and Unix-like computer operating systems that lets non-privileged users create their own file systems without editing kernel code. This is achieved by running file system code in user space while the FUSE module provides only a "bridge" to the actual kernel interfaces.

To implement a new file system, you will need to write a handler program, and link the handler to the libfuse library (part of FUSE). The main purpose of this program is to specify how the file system is to respond to read/write/stat requests. The program is also used to mount the new file system. At the time the file system is mounted, the handler is registered with the kernel. If a user now issues read/write/stat requests for this newly mounted file system, the kernel forwards these IO-requests to the handler and then sends the handler's response back to the user.

FUSE is widely used to implement many file systems and distributed file system clients. It also has many language bindings, in addition to its original C implementation, such as Java, Go and Python. You can choose any of these language bindings in your project. However, TA support level would vary based on their own language ability.

- C version (including Python binding): https://github.com/libfuse/libfuse
- Java binding: https://github.com/SerCeMan/jnr-fuse
- Go binding: https://bazil.org/fuse/

To best learn to program with FUSE, you can look at the examples in the library, read its API documentation or search for instruction videos available online at a lot of places, just like learning any software library in the real world. The first task below allows you to get familiar with FUSE.

1.2 General Functionality Requirements

Like many software engineering projects, first you will need to understand the requirements from the user side. In particular, the users require your file system to offer the following functionalities, at the end of project 3:

1) You need to provide a `buildfs` script will generate the file system. It does so by creating a 100MB file, and all your file system content will be stored in that file as persistent storage, your "block size" is always 1KB in the file. (1KB = 1024B and 1MB = 1024KB)
2) The file system structure should be a log-structured file system, i.e. all data and metadata should be in a single log when you write it.
3) You can mount the file system to any Linux directory, using standard FUSE mount mechanism.
4) The users can create, list, delete files, using standard Linux shell command and libraries. You should offer all the Linux standard file stats, such as sizes and timestamps.
5) The users can replace file content, either part of the file or the entire file, or appending to an existing file, using standard Linux tools and libraries. In these operations, you need to make sure that the standard stats should be updated accordingly.
6) Your file system should support *hard link*s, and files are deleted only when all hard links are deleted. You can hard link directories and normal files. We do NOT require symbolic links, for simplicity.
7) The file system should support standard file system permissions, i.e. being able to configure the owner, group of the file, and their access rights (read, write, execute), just like other Linux file systems. However, we do NOT need to support special permissions, such as `setuid`.
8) In project 3, the file system should be *thread safe*, i.e. we can allow multiple processes / threads to access the file system without explicit locks in the applications / scripts, but we do NOT require any concurrency support. In other words, you can put a global lock to the entire file system allowing only a single operation at any given time. However, we will ask you to support concurrency for project 4, so if you want, you can support concurrency (by designing finer granularity locks) in this project to save you time for the next project.
9) You need to support Linux's `sync` operation, which flushes all data in memory buffer (in user space or system kernel) onto persistent storage (in this case, the file you created in 1)). You can design a write-back memory cache to accelerate file system operations, but in project 3, it is not required (will be a task for project 4), so everything can be synchronous for now, e.g. the cache can be write-through, or no cache at all. Note that you still need to call the Linux system `sync` to really flush to your disk file from system buffer cache.
10) Given the log structure and the synchronous operations, your file system should survive random crashes in the middle of any operation. All operations, especially metadata operations, should be atomic. After restarting back from a crash, when you mount the file system again, it should fix the persistent storage to a consistent state (i.e. no broken structure, directory etc.). Of course, all in-memory states should be cleaned after the crash.
11) In project 3, you do NOT need to actually delete any data and recycle the space (i.e. the "compression"

operation in the LFS paper). We will leave that part to project 4. You can assume that under no test cases in project 3, your file system will be full.

1.3 Documentation and Testing Requirements

You should provide a short design and introduction document listing at least the following:

1) The main data structures of your file system

2) Code organization, which source file contains which set of functionalities?

3) Some details on how you satisfy requirements 7-10.

4) Explanation of test cases, especially non-trivial or hard to understand ones.

5) A user manual, listing any non-standard operations for the TA's when grading your project, such as you need to pass a special flag to some command etc., to make their grading a little easier.

6) Known bugs and limitations. Things you know that do not work yet. Things listed here will results in less deduction of points than bugs you do not know, but found by TA's.

The document should be succinct, I think two pages should be enough, but in any cases, it should NOT exceed four pages.

In your source code, you should provide a directory called `tests/`, containing all test cases, both *unit tests* – testing single functions, and *integration tests*, testing the file system when it mounts on the Linux OS). You should provide a *build target* called `test` (whether you use makefile or ant or bazel or whatever build tool you like), so the TA's can run all tests with a single command.

2. Specific Tasks

We have the following step-by-step tasks guiding you to build the LFS from the ground. Following this order will help you get partial credits even if you cannot get the entire thing working.

1. (20%)

The first task allows you to get familiar with FUSE, its interface and setup your source code build and testing environment.

Implement an *echo file system*, which simply print out the name of the file system operation, with the parameter that passed to it, but do nothing with the file. For example, calling

```
read(int fd, void *buf, size_t count);
```

should print out

```
READ, <fd>, <buf>, <count>
```

All integers should be as decimal numbers, and all pointers (if you use C) should be printed as hexadecimal numbers.

There is only one tricky part here: for any file name, you should print out the full path, even if the user passed in a relative path. For example, your file system mounts at /mnt/testfs/ , and the user's current directory is `/mnt/testfs`, and she executes `open("aaa", "r")`, then you should print out

```
OPEN, /mnt/testfs/aaa
```

You should be able to build the echo file system and mount it correctly. Also, it should support all operations mentioned in the requirements 3-7.

Make sure that you write some test cases using standard Linux commands (`cat, ls, cp, ln, mv`) and write assertions on expected print-outs. In order to make certain tools to work, you can return some fake data to the tool, for example, on a directory listing call.

2. (30%)

Implement the log structured file system metadata and the underlying operations. Specifically, you should implement:

1) The file as persistent storage, and initialize the file with superblocks, empty block table (if your design uses it), etc.

2) Data structure for any file system on-disk or in-memory metadata.

3) A data structure representing the disk block (always 1KB in size), and operations to append / read blocks to/from the file.

4) An *inode* data structure, indirection block data structure, etc. and related operations.

5) A *directory* data structure and related operations.

6) Other data structures you think would be necessary, for example, the segment in LFS paper, but it is not required.

7) Hint: it is a good practice to provide a `pretty_print` function (thinking of the java toString function) for each of these data structures. It is useful for debugging, and useful in the required tests in Step 4, too.

You should write some unit test cases showing how these data structures / operations would work

3. (30%)

Using *echo file system* as code skeleton (make a copy), using the data structures developed in step 2, and implement all LFS operations as detailed in requirements 1-7. If you have done step 1 and 2 properly, the implementation should be quite straightforward, if you find anything missing, feel free to go back to step 1 and 2 and fix it.

Then you should be able to run the test cases (using Linux commands) in 1) again, to test if your file system actually works. You should add some additional test cases, such as `open-after-read, chown, chmod, ln, ls -l` etc. to test the new functionality.

4. (20%)

This step only asks you to write some tests to ensure that your system satisfies requirements 8-11. You should test your system's thread safety, and crash recovery capability. Specifically,

1) You should be able to run an arbitrary number of concurrent processes accessing the file system (both data and metadata).

2) Though we do not test the access speed, but the file content should be correct in this situation. Also,

you will write some test cases covering crashing, umount and recovery of the file system.

3) Write a `block_dump` class, allowing printing out all metadata blocks in your file system. All dumped blocks should be in the `pretty_print` format in Step 2.

5. (10% extra credit)

   Allow the Linux command, `tree`, to correctly run on your file system. The command recursively lists all files in the directory. The tricky part to consider is that there might be some other FS hard linking to your FS, or other FS mounting at some directory at your FS, and *vice versa*.