CS333: Operating Systems Lab

Lab 8: Building Your Own Filesystem

Goal

In this lab, we will understand how file systems work in Linux, by building a new user-level file system.

Before you start

Understand the concept of user-level file systems and how they work. We will build a user-level file system using the FUSE framework. FUSE is a library that lets you easily build user-level file systems for Linux. You must first install the library on your machine, and then use the library to build your file system.

Below is an excellent tutorial on how to build user-level file systems using FUSE (after installing the library).

`http://www.cs.nmsu.edu/~pfeiffer/fuse-tutorial/`

This tutorial revolves around a simple example user-level file system, called BBFS. You can download the latest version of BBFS filesystem from this link (also linked from the tutorial above).

`http://www.cs.nmsu.edu/~pfeiffer/fuse-tutorial.tgz`

While you can read and understand BBFS in detail from the tutorial, here is a very high-level overview. When you run BBFS, you will provide it with two directories. One directory, called the *root* directory, is where regular files reside. The other directory called the *mount* directory is what BBFS is responsible for. When you read and write files in the root directory, your requests are served by the regular Linux file system. When you access the mount directory, your requests are routed via BBFS. The BBFS code given to you doesn't do anything special, except to execute your request on the root directory itself. For example, when you type `ls` in the mount directory, BBFS simply performs `ls` in the root directory, and returns the result. Therefore, it would appear to you that the mount directory is a mirror of the root directory, even though in reality, it is only an empty directory. While this simple example really doesn't do much, you can extend BBFS to do several interesting things, as we will do in this lab. It is highly recommended that you spend some time familiarizing yourself with BBFS before you proceed further. Pay particular attention to the VFS function calls that are made to implement each system call (e.g., open, read, write), as logged in `bbfs.log`.

Deduplicating Filesystem

You must now use FUSE to build a userspace filesystem that does block-level deduplication for disk storage. Assume files are made up of blocks, and read and write requests are issued for these blocks. Now, when you get a request to write a file, you needn't store the complete contents of the block on the disk, as some other file block with the same contents may already be present in your filesystem. Instead, you can hash the contents of this block, compare with hashes of all previous blocks for duplicates, and store only unique blocks on disk. For blocks that are duplicates, you can store some sort of a pointer to the previous block content. This deduplication saves disk space when there is significant overlap in file contents. Now, when you read a file, you must correctly follow the pointers to deduplicated blocks, reconstruct the original content, and display the original file written by the user.

There are several design choices you will have to make here. For example, how do you keep track of all the blocks, their hashes and their contents? One option is to have some sort of a *block store* that stores all deduplicated block contents and their hashes. New requests to write blocks can be checked against this database of existing blocks, and files written on disk can only store pointers to unique deduplicated blocks in this block store. Read requests can reconstruct a file from the pointers to the contents in the block store. This is, of course, only a suggested design option, and you are free to come up with your own design.

You must implement this deduplication functionality on top of FUSE. You can simply extend the BBFS code, and perform deduplication for files served from BBFS. Please describe the detailed design, and the BBFS functions you modified, in your report. Further, you must describe at least one correctness test you have performed on your implementation. For example, here is one way to test your implementation (we will use a similar test case in our evaluation).

1. Create two (or more) files with highly similar content, and write them one after the other via BBFS. That is, copy them into the mount directory that is served by BBFS.

2. BBFS should deduplicate the files and store only a smaller compressed version of the files in the root directory of the native filesystem. So, if you check the sizes of the files written in the root directory, you should see a significantly smaller size on the disk when there is overlap in content. Further, the contents of the files in the root directory may not match the original contents written (since these may just be pointers to blocks in the block store, for example).

3. Separately, BBFS may store the master copy of deduplicated contents in some other file in the native filesystem (e.g., /tmp/blockstore).

4. When the files written into the mount directory are read via BBFS from the mount directory itself, you should see the complete original content that was written, with all the duplicate blocks. That is, BBFS must retrieve full block contents (from the block store, for example) and serve the files as they were written in their original form.

Implementing a full-blown deduplicating filesystem is very challenging, so we ask you to make the following simplifying assumptions to make the lab tractable.

- Assume all read and write requests are made at offsets that are multiples of 4KB. Further, the size of data read/written is also a multiple of 4KB. You can assume that all filesizes are multiples of 4KB. Therefore, you can always check for duplicates on fixed 4KB size blocks. There is no need to support deduplication on variable sized blocks.

- It is enough to support simple read and write operations on files. Your test cases can be very simple, as illustrated above. All you need to ensure is that copying a file into a BBFS directory and reading it back works fine. You are not required to support other operations like deletions, truncation, and so on.
- You may ignore hash collisions. That is, you may assume that two blocks that have the same hashes also have the same contents. You can use a hash like SHA1.
- You can assume that the total number of blocks in your filesystem is small enough to simplify the design of your block store. We will not test with more than 10 files, each not larger than 64KB.

Submission and Grading

You may solve this assignment in groups of one or two students. You must submit a tar gzipped file, whose filename is a string of the roll numbers of your group members separated by an underscore. For example, `rollnumber1_rollnumber2.tgz`. The tar file should contain the following:

- `report.pdf` should describe the design and implementation of your filesystem, and how you tested its correctness.
- A patch of your code to apply to the BBFS filesystem. Please start with the latest version of FUSE and BBFS, so that the patch can work on our code. Please create the patch after the Makefile is generated by the automake system. If you take a patch against a directory which does not have a Makefile generated, any changes you make to the Makefile in your patch may be lost. Further, do `make clean` in your directory before you create the patch. This will ensure that any compiled binaries and such do not appear in your patch.

  To make the procedure clear, have a clean unmodified version of BBFS in *OldDir*; this directory should have a Makefile generated by running `configure`, but must not have any further changes made. Next, work on your changes in another *NewDir*. Then, do `make clean` in *NewDir*, and take a diff between *OldDir* and *NewDir* to create your patch.

Evaluation of this lab will be as follows.

- We will read your report and check that your design makes sense.
- We will install your patch and check that it is doing what it is supposed to do.
- We will read through your code for correctness.