Project1:
RLFS: A Parallel File System in Fuse

Due: Aug. 20, 2018

Shenzhen Institutes of Advanced Technology
Email: yang.wang1@siat.ac.cn

July 13, 2018

1 Goal

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

2 Before you start

Understand the concept of user-level file systems and how they work. We will build a simple user-level parallel 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).

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

This tutorial revolves around a simple example user-level file system, called BBFS. You can down- load 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 doesnt 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 doesnt 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.

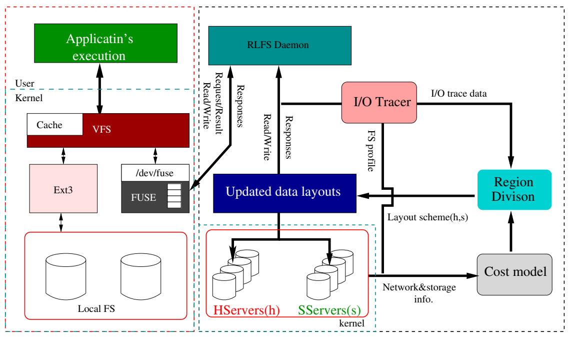

Figure 1: Arachitecture of RLFS.

3 RLFS: A Parallel Filesystem

3.1 Design

RLFS is designed based on the FUSE framework as shown in Fig. 1, where a kernel part (FUSE) and a user-level daemon (RLFS Daemon) of RLFS are illustrated. The kernel part is implemented as Linux kernel module, which registers RLFS with VFS, and in the meantime, creates a `/dev/fuse` block device. The FUSE module acts as a proxy to the RLFS daemon for various file system requests made by the applications while the block device serves as an interface between RLFS daemon and the kernel.

Application can access RLFS by mounting it to its namespace, thereafter all file system calls targeting the mount point are forwarded to the FUSE module via VFS. The FUSE module then relays the calls via a *request queue* to the user-level RLFS daemon through the device `/dev/fuse`, where the appropriate service handlers are invoked to accompolish the file system calls by contacting the metadata server and/or the storage servers. The responses are propagated back along the reverse path through the kernel and eventually to the application, which is usually in a wait state for the reponses after issuing the requests. As with the I/O trapping techniques in PVFS design [1], the RLFS daemon in conjunction with the storage servers should complete all the semantics of a PFS. For example, the `read` handler should first identify which storage servers have the requested data segments, and where in each server stores the corresponding one, and then issue the sub-requests to these servers for parallel accesses.

3.2 Implementation

As with standard PFSs, the implementation of RLFS also consists of three major components as shown in Fig. 2. The *file clients* issue requests on behalf the applications from the *compute severs*. The *storage servers* are responsible for storing and managining the stripped files, andthe *metadata servers* (MDS) contain the description information of the files stored in RLFS. During a file operation, a client first contacts MDS to get the files metadata, then leverages it to interact with the storage servers via the *RLFS Daemon* for data accesses.

In RLFS, a file is stripped across all the storage servers, and each stripe is stored as a separate data file in each storage server. To this end, a *file stripe* table (FST) is maintained by MDS for each physical file in RLFS where each file is recorded in terms of its stripe size for each storage server. The table is created by the module *Region Devision* in Fig. 1 when the file is written into RLFS, and updated when its access pattern is changed. For the sake of efficiency, the RSTs of the files to be read, if possible, could be cached and de-cached in the same directory as the applications when RLFS is mounted and unmounted.

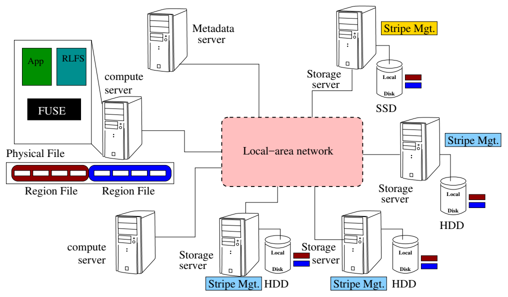

Figure 2: An illustration of RLFS.

3.3 Test

You must implement this deduplication functionality on top of FUSE. You can simply extend the BBFS code, and perform paralle read/write operations on 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, and write them one after the other in parallel fashion via RLFS. That is, copy them into the mount directory that is served by multiple remote storages.
2. Then read the file back from RLFS in parallel fashion.

Implementing a full-blown parallel 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.
- 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 remote directories and reading it back in parallel works fine. You are not required to support other operations like deletions, truncation, and so on.
- You can assume that the total number of blocks in your filesystem is small enough to simplify the design of your block store.

4 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, `group1-name1-name2.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. It should also include the evaluation part when reading/writing files with different sizes of `8MB, 64MB,128MB, and 256MB` with respect to different numbers of storage servers `(1, 4, 8, 16 ,32, 64)`.

- `code` should be commented and stored in a separate directory, including `Readme.txt` showing how to play with it.

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

References

[1] Philip H. Carns, III Walter B. Ligon, Robert B. Ross, and Rajeev Thakur. PVFS: A Parallel Virtual File System for Linux Clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 317–327, 2000.