Operating Systems – Project4 Report

1. TITLE

File System

NTEMKAS CHRISTOS, AEM: 02984

TOLOUDIS PANAGIOTIS, AEM: 02995

GAROUFALI MARIA NEFELI, AEM: 03129

2. ABSTRACT

The purpose of this assignment is to comprehend how Linux's FUSE file system is implemented and functions. To put it more precisely, we are required to put in place a file system that seeks to "compress" the data by looking for opportunities to share blocks between files (or even inside the same file): It is not essential to save blocks of other files (or even different blocks of the same file) to disk more than once if their contents are identical.

3. INTRODUCTION

We use FUSE to construct a straightforward file system that seeks to reduce the number of disk blocks required for storing multiple files by identifying and reusing any blocks with similar content. As we all know, files are made up of blocks, and applications read and record these blocks. If any current block on the disk occurs to have the same content as a record, there would be no need to write a new block in that scenario. We can store a hash for each existing block on disk so that we don't have to compare entire blocks (because it would be slow). When a new block is requested to be written, its matching hash is calculated, and it is compared to the hashes of the previous blocks. When two hashes match, we presume that the corresponding blocks' contents match as well. Only distinct (new) blocks are written to the disk in this manner. Naturally, subsequent reads for files that tried to write blocks that are copies of already-existing ones should provide the right content (thus the file should somehow acquire a "pointer" to the already-existing block). If there is a considerable amount of content overlap between the files, using this strategy will significantly minimize the amount of disk space needed. We modified the Big Brother Filesystem (BBFS) code to "compress" the storage needed for the files serviced by BBFS in the way previously mentioned.

4. IMPLEMENTATION

First of all, we have the root and the mount directories. This is the logic of the bbfs code that we edited. The root directory is the actual folder where the file management happens. This is the real storage of the file system. The mount directory is the folder that the user sees. This folder is a mirror of the root.

In the root directory, there are 2 folders: the .metadata and the .blocks folder. Inside the .metadata folder there are files, using the filenames of the actual files stored in the filesystem. They are not the files that will be represented to the user, however each one of them contains the hash values of all the blocks that create that file. The .blocks folder contains files that represent all the blocks that exist in the file system. The name of each block is its hash value. In this way, we can easily find it.

The functions we changed from the bbfs code are the bb_init, bb_getattr, bb_read, bb_write, bb_unlink, and (I don't remember). The changes we made will be explained below.

5. CODE DEVELOPMENT & CHANGING IN THE GIVEN

Firstly, regarding the code we have implemented a struct node. This struct has two fields, a char* hash_value, and an int exist. We then create a struct node* root, which is an array of struct nodes. The node represents a block since it has the char* hash_value field that is useful as the name of the block. The int exist refers to the number of times each block is used. The purpose of this struct and this data structure is the easiness for information of the times a block exists that it provides. Another implementation was searching inside each file and counting but this would be much slower. Below we will analyze the functions we edited to make the system work as expected. We also have implemented some functions to handle this data structure.

BB_INIT

This method was altered to create 2 folders inside the root directory. The first is .blocks, and the second is .metadata. Additionally, we read for the .load file, which is essential for storing the details of the blocks that we keep in the array data structure before unmounting the filesystem. After mounting the system, we carry out that action to restore this data. In other words, the system is non-volatile, which means that the files in the system will remain the same no matter how many times we open and close the file system.

BB_WRITE

To meet our expectations, we modified the bbfs' write function. In the metadata folder, we directly navigate to the file we wish to change. We identify the block that is most suitable for modification and calculate its hash value. The block we want to write's hash value is then taken. We go to the folder containing the blocks, and if the block already exists, we open it with the updated hash value; otherwise, we create it. The user-requested information is subsequently written to this block. The hash value of the new block is then compared to the previous one to see if it matches. If so, the only action required is to update—more specifically, to increase by one—the array that stores the frequency of use for each block. We raise the number for the brand-new block while lowering the number for the previous one. If the previous block was solely used in this file, it must now be deleted. If we need to write at the file's end without changing any existing blocks, we simply build a new block based on the updated hash value and update the existence count. The last section of code allows us to write blocks smaller than 4 KB, but it can only be used at the conclusion of the file.

BB_READ

The goal of this function is to read a file. So, first of all, we open the file from the .metadata folder. We compute from which block and beyond the user wants to read. Then we read the names of each block till the end of the file. Having the name of each block we move to the blocks folder and open each block and read.

BB_GETATTR

This function is to give the attributes of a file. To do that we open the file we want the attributes in the .metadata folder. Then read one by one all the hash values that are included in this file. After reading a hash value we move to the .blocks folder where the blocks are. There as we have each block saved as a file we call lstat to get the size of the block. We then sum the size of each block that composes the file to find the total size of the file.

BB_UNLINK

This function is to delete a file. We changed it so we go to the .metadata folder, find the file and see the blocks that compose it. After reading each block we go to the folder of the blocks (.blocks) and to delete the block. Since a block can be used several times we can not delete it until we make sure that it is not useful anymore. So instead of deleting the blocks we simply go to our array and update their existence to be one less than before. Only if a block's existence is equal to 0 we will delete it by calling the unlink function.

BB_DESTROY

In this function we create a new file named .load in the metadata folder, where we keep information from the data structure in order to have it when mounting the system. If we did not do so this information would be lost. Furthermore, in this function we destroy our data structures.

6. TESTING

Five tests:

- test1.sh : Mount the system, create a file with size 4096, copy it to /mountdir and see it in the /rootdir and then delete it.
- test2.sh : Mount the system, create two files, size 69KB and 57KB respectively, copy them to the /mountdir and make some mounts and unmounts to the system to test the non-volatility of the system and the support for big files
- test3.sh : Mount the system, create two files, size 19KB and 20KB respectively, copy them to the /mountdir to the system. The files are similar and we test the space saved by not having blocks with the exact same contents
- test4.sh : Mount the system, create 13 different files of medium size, copy them to the /mountdir to the system. Here, we test the support for more than 10 files in the filesystem and they are at least 16KB

- test5.sh : Mount the system, create 1 file with size 12KB, copy them to the /mountdir to the system. Then a block is changed in the middle of it. So, overwrite is supported in our system

7. RESULTS

As a general result, we confirm that blocks with the same contents are not created multiple times. Also it is confirmed that all the functions work as expected. The results of each given test is located in the testX/out_X.txt file

8. RUN THE TESTS

a. Copy the file into the example directory.
b. Go to the testX folder.
c. Make the testX.sh executable, with the command chmod 777 testX.sh
d. Run the test with command ./testX.sh
e. Then print the result in the terminal.
   !!! The X is the number of the test.

9. GENERALIZATIONS - IMPROVEMENTS

Some improvements were made on account of the assumptions:

- Non volatile filesystem
- It can host more than 10 files
- The files does not have to be multiple of 4096 bytes in size

10. CONCLUSION

All in all, this homework is about implementing the FUSE file system in a linux vm. The key of this homework is to compress the data of the files saved in the file system by not saving the blocks that show up more than one time. This happens due to having an array of structs having as attributes block_id and exist(times that show up), which help to not store the same blocks again and again. Consequently, less space is consumed for the files inside the file system in the case of duplicated pieces of data.