ECE318: Operating Systems 2024-2025
Project 3 - FUSE Filesystem Extension

Contents

1 Setup **2**

2 Introduction **2**

3 Assumptions - Simplifications **2**

4 Our Implementation **2**

5 Functions **5**

6 Testing **6**

6.1 First Test - Test A 6

6.2 Second Test - Test B 8

6.3 Different Files - Test C 9

6.4 Big File support Test- Test D 9

6.5 Non Volatile test 11

1 Setup

1. Go to the `filesystem` folder (`cd fuse-tutorial-2018-02-04/filesystem`) and run:

```
make
```

2. Go to the `experiments` directory:

```
cd ../experiments
```

3. Mount the filesystem with:

```
../filesystem/bbfs rootdir mountdir
```

Note: After completing these steps, you can run `test_script.sh` inside the `fuse-tutorial-2018-02-04` directory for testing, as you will be guided here.

2 Introduction

Our implementation is an extension of the Big Brother File System (BBFS) coded by Dr. Joseph Pfeiffer, to support block-level deduplication using FUSE. The main goal is to minimize disk storage needs using a deduplication technique that stores only unique 4KB blocks, even if multiple files (or different parts of the same file) contain identical data. The system is designed to be simple, reliable, and to follow the assignment's requirements.

3 Assumptions - Simplifications

Given the complexity involved in developing a fully operational file system, the following assumptions and simplifications were made in our implementation:

- Every read and write request happens in blocks of 4KBs and only on files that have a size that is a multiple of 4KBs
- Every hash collision event is treated as if it were the same block. If hashes collide, the different blocks are lost and the oldest block is being kept.
- The filesystem is not designed to be thread safe. Undefined behaviour may occur in the event of many programs executing filesystem operations concurrently.
- Only read, write, file creation and deletion operations are supported. More complex ones such as truncation are not.

There were some simplifications not used. The added functionality is listed below:

- The file system can host more than 10 files that can be more than 64KBs each and in different directories.
- The file system is non-volatile, meaning that unmounting and remounting results in finding the previously stored files present.

4 Our Implementation

We decided that the best course of action would be to "deconstruct" each file into 4KB sized blocks of data, generate their respective SHA-1 hash codes and store them into the root directory (rootdir/), which contains a metadata file called blockstorage.txt as well as the actual unique data blocks being stored in 4KB segments in the blockdata.bin file. Let's see the FSM of the proccess:

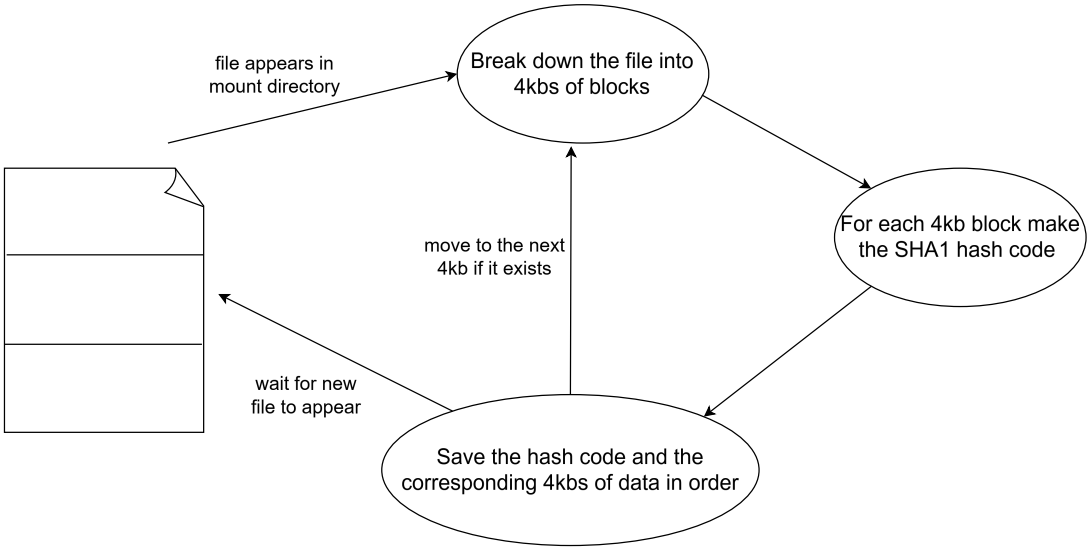

Figure 1: FSM of the deconstruction proccess

Now let's see how the compression works in our implementation:

Let's say we have 2 files, file1 and file2 with a size of 8KB each, and they also share one 4KB block of data. Thus, we initially have 16KB of data, but we aim to reduce that size to 12KB: 4KB of data that are identical for the two files, plus 4+4 KB of data that are unique.

Now, when the first file appears, it's broken down into two 4KB blocks with each block having a corresponding hash code generated from it and stored in blockstorage.txt on a "first-come first-served" basis. The actual file data is stored with the same order in the blockdata.bin.

In our implementation, there is a pointer from each 4KB block that points towards its own SHA1 hash code, as well as pointer that points from the hash code towards the actual data. Now the second file appears, which shares one 4KB block with the first file.

We search for the hash code generated using the shared block within the blockstorage.txt file. The hash already exists so we modify the pointer from the second file so that it points towards that shared hash code in the block storage, and subsequently we have the pointer from the hash code to the original data (blockdata.bin).

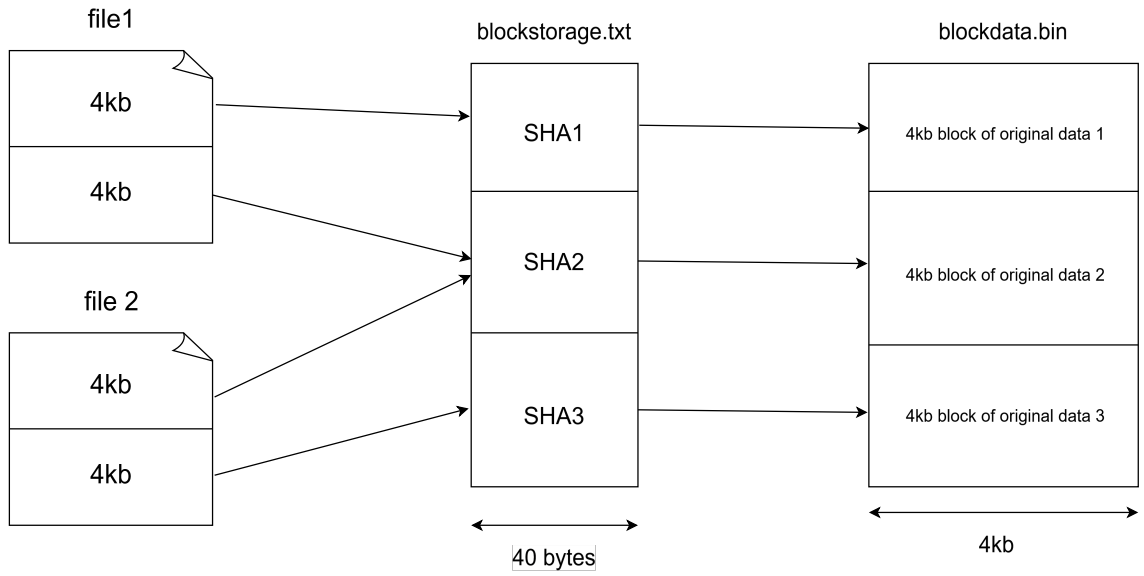

Figure 2: Deconstruct Process

After this proccess we end up with 2 files containing the 2 hash codes each and being in total size of 160 bytes, three 40 bytes of hash codes, so total 120 bytes and three 4KB of block data being total of 12KB.

In summary for this example:

- **Hashes stored in files:** $2 \times 2 \times 40 = 160$ bytes
- **Unique hashes:** $3 \times 40 = 120$ bytes
- **Block data:** $3 \times 4\text{KB} = 12\text{KB}$

The original file size without compressions would have been 16KB. The space savings can be expressed as a percentage:

$$\text{Space Usage Percentage} = \frac{\text{Total Storage}}{\text{Original File Size}} \times 100\% = \frac{12.28\text{KB}}{16\text{KB}} \times 100\% \approx 76.75\%$$

The compression process resulted in using 76.75% of the original storage space. This means we have a space saving of approximately 23.25% for two files that share one block of data.

So our structure after the compression looks like this now:

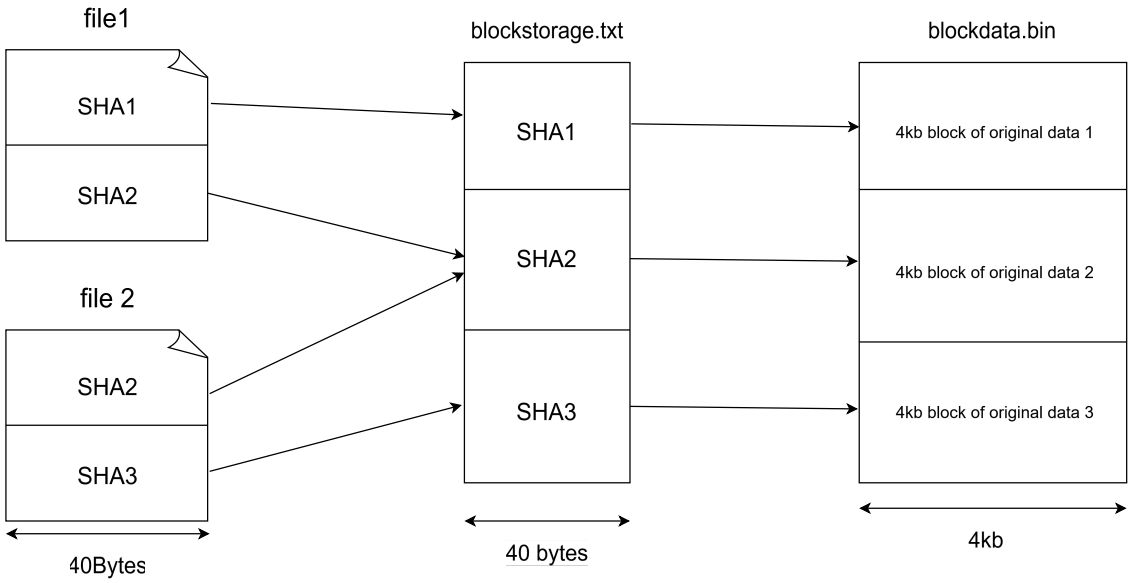

Figure 3: Structure after compression

The whole point of this implementation is having a simple and robust structure for reading, writing and deleting files. We only use two core concepts, search for the hash in blockstorage.txt and finding the offset from the start of the file and this offset corresponds to the 4KB data block which is in the position blocksize * offset in the blockdata.bin to retrieve the actual data.

Based on these simple concepts we avoid unexpected behavior and we only have to deal with the size of the file and the position of the file pointers in blockstorage and blockdata files.
Example showcase of reading the file:

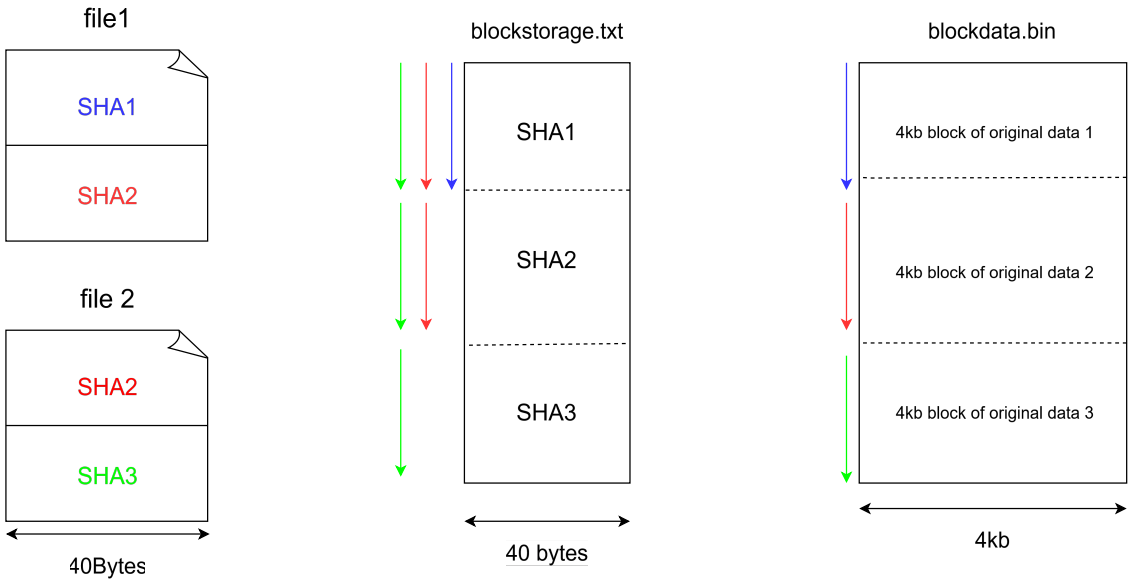

Figure 4: File pointers logic

Let's say we want to read file1 and file2. We read the first 40 bytes of file1 and generate the first hash code. Then we scan the blockstorage.txt file and we find it at the start of the file, so the index or offset is 0 from the start of the file, or we can say that we have to check from the start until the first block (thus reading the first block) and to find the corresponding data in the blockdata.bin we have to check from offset = 0 * 4KB until 1 * 4KB so the first block.

For the second hash code, we check in file1 the 40 until 80 bytes and we get the 2nd hash code, we look into blockstorage from the start of the file, the first 40 bytes of the file are not a match so we check for the next 40 bytes (that's why we put 2 red arrows), and we find it on the second place of the file, so the corresponding data in the blockdata.bin starts from offset 1 until offset 2 * 4kbs and we grab the data.

It becomes apparent that with this implementation, it's very simple to store the hash codes that constitute each file and then just calculating the offsets instead of saving them, because it makes deleting files just a reorganizing procedure of the blockstorage and blockdata "list".

For writing, we just add everything new to the end of the file if the hash code is unique or we find the matching hash code and save the file pointer.

5 Functions

We changed the following functions:

bb_getattr: We use this function to get the file size and attributes. We modified it so that the function now uses the number of hashes inside the backing file blockstorage.txt and now it calculates the size of the original file by multiplying the number of hashes * 4KB, which is our block size, so it produces the original file size for the user to see.

bb_read: This function is responsible for reading the file data. The original file data must be presented as it is when opening the file using the unmodified filesystem . We achieve this by opening the backing file that contains the constituting hashes of the original file (acting like a file pointer), searching within the blockstorage.txt file, which contains the mapping of hashes to actual data block, keeping a local counter to find the offset of the hash from the start of the file, and using that counter to read the offset*block size data from the blockdata.bin. Finally we copy the actual contents of the requested file from the blockdata into a buffer and display it to the user.

bb_write: This function is responsible for writing data to a file. This is where the "deconstruct" process happens. The function splits the file's data into 4KB blocks, generating a SHA1 hash code for each block, then searching for an existing hash code within the blockstorage.txt file using the hash_exists

function. If the hash does not exist already, it is appended to the blockstorage.txt file and the actual data is appended at blockdata.bin, then finally appends the hash code into the backing file (file in the root directory that share the same name as the original file). If the hash already exists, then it just appends the hash code in the file.

bb_unlink: This function is called when we delete a file from the filesystem. It calls our deleteFile function to keep the logistics of the blockstorage and blockdata files. If it's successful, it removes the file using the unlink system call.

Also it's worth mentioning our helper functions:

hash_generate: This functions uses as input the data of the original file in 4KB blocks and the size we want to hash. For the purpose of extendability for the project, we allowed a variable block size to be acceptable. For our implementation, the function uses 40 bytes to generate a uniqie hash based on the 4KB of the data block,represented as a hexadecimal string.

hash_exists: This function requires the file descriptor and the hash code as inputs. It checks if the hash already exists in the blockstorage.txt file by reading each line from the start of the file and comparing it to the input hash code. If it exists then it returns 1. Otherwise, it returns 0, which means that the hash code is unique.

set_NewHash: This function uses as input the file descriptor and the hash code. It opens blockstorage.txt file in order to append the hash code, plus a newline character, then flushes the file to ensure that the data is actually writen.

deleteFile: This function is called inside bb_unlink and is used to keep the logistics of the blockstorage and blockdata.bin in order. It operates in four steps: First it checks all the files except the one we want to delete and collects all the hashes. Second, we want to find which hashes we want to keep, so it reads all the hashes in the blockstorage. Third, it creates a new temporary file for the blockstorage and databin and copies only the hashes and data we want to keep. For each hash that we want to keep, we use the hash position in the blockstorage file and we find the corresponding block offset by `offset = index * BLOCK_SIZE` in the databin file, then we copy that 4KB block of data in the new temporary file. Finally, it replaces the original files with the temporal versions of them.

6 Testing

First, we have to mount the filesystem.
After we have succesfully mounted our system, we will run the first test.

Or for your convenience you can just run the test_script.sh STRICTLY inside the "fuse-tutorial-2018-02-04" directory.

6.1 First Test - Test A

In this test we want to see if the compression works, if we can read the file and if the deletion works. We begin by copying the files from the `testA` directory into the mount directory using the command
`cp -R ../experiments/testA/file1.txt mountdir/.`
`cp -R ../experiments/testA/file2.txt mountdir/.`

In the Test A directory are two files, file1.txt and file2.txt. Both of them are 8KBs and they share one 4KB block with identical data.

```
vm@vm:~/fuse-tutorial-2018-02-04/example$ ls -alR

./mountdir:
total 16
-rw-rw-r-- 1 vm vm 8192 Jun 10 15:33 a1.txt
-rw-rw-r-- 1 vm vm 8192 Jun 10 15:33 a2.txt

```

```
./rootdir:
total 32
-rw-rw-r-- 1 vm vm    80 Jun 10 15:33 a1.txt
-rw-rw-r-- 1 vm vm    80 Jun 10 15:33 a2.txt
-rw-rw-r-- 1 vm vm 12288 Jun 10 15:33 blockdata.bin
-rw-rw-r-- 1 vm vm   123 Jun 10 15:33 blockstorage.txt

vm@vm:~/fuse-tutorial-2018-02-04/example$ cat rootdir/blockstorage.txt
608cabc0f2fa18c260cafd974516865c772363d5
59682bddd4b2a31b08bc6977169691c10db7a501
6e6ea08294ea261bf0ca6598cd7b23b988a47cc8
```

Listing 1: Test A directory list

From the terminal we see that the files appear to be 8KB in the mountdir and we can confirm that get_attr works correctly meaning that we see the original file size, also in the backing directory rootdir we see that each file is 80 bytes long which are the hashes and in .blockstorage we see the file blockstorage.txt that contains the 3 hashes as it should. We can also see in the rootdir/blockdata.bin that all the contents of the original files are stored in the correct order(it's too long to include here).
Finally, we see that in our storage file (line 12) the size is 12KBs instead of 16KBs that are stored in our system, so we can confirm that the compression works.

```
-rw-rw-r-- 1 vm vm 12288 Jun  9 21:32 blockdata.bin
-rw-rw-r-- 1 vm vm   123 Jun  9 21:32 blockstorage.txt
```

Listing 2: Close look at data and hash file directory info

Also if we run `cat mountdir/file1.txt` we can see the original contents of the file, same for the file2.

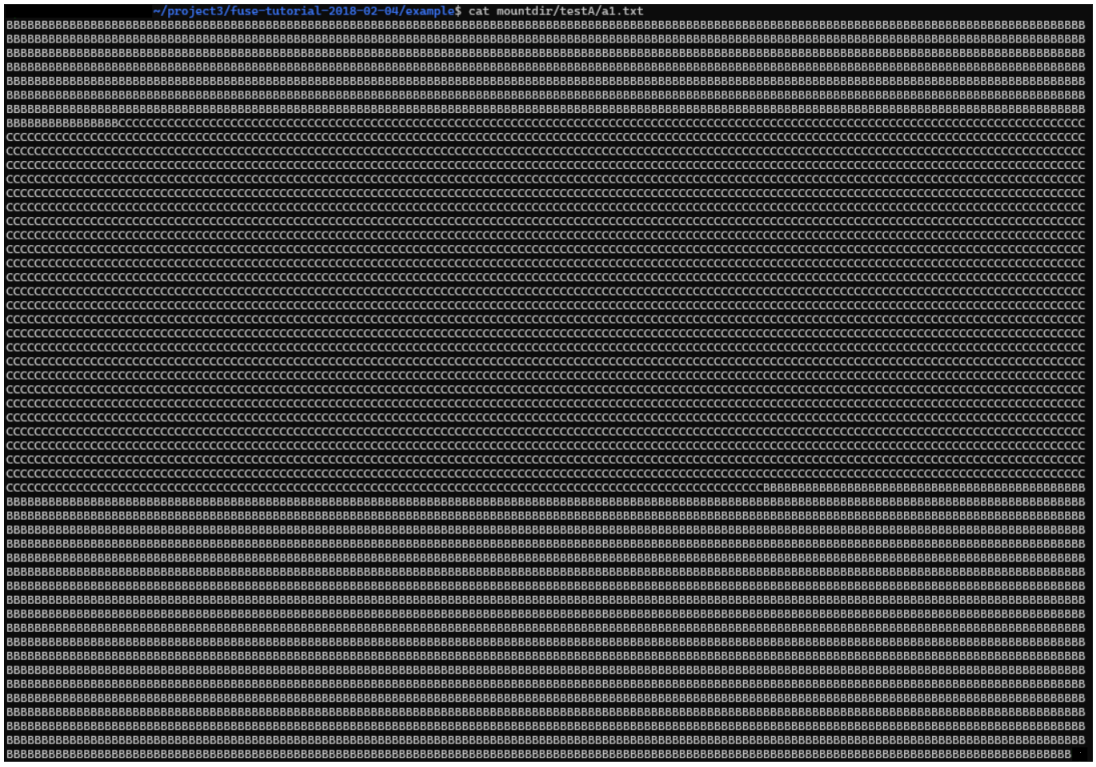

Figure 5: file1.txt contents display with cat command

Regarding the deletion: We remove one file and we expect the blockstorage and blockdata to be updated as well as the mountdir and rootdir directory.

```
./mountdir:
total 12
-rw-rw-r-- 1 vm vm 8192 Jun 10 15:33 a2.txt

./rootdir:
```

```
total 24
-rw-rw-r-- 1 vm vm   80 Jun 10 15:33 a2.txt
-rw-rw-r-- 1 vm vm 8192 Jun 10 15:36 blockdata.bin
-rw-rw-r-- 1 vm vm   82 Jun 10 15:36 blockstorage.txt

vm@vm:~/fuse-tutorial-2018-02-04/example$ cat rootdir/blockstorage.txt
59682bddd4b2a31b08bc6977169691c10db7a501
6e6ea08294ea261bf0ca6598cd7b23b988a47cc8
```

As we can see, the rootdir and mountdir are updated, blockdata.txt file is now 80 bytes (2 bytes are for the newline character) and blockdata which contains the actual data is 8KBs as it now only has the contents of the file1.txt file. Shown more clearly in table below:

Before Deletion (ls -alR)	**After Deletion (ls -alR)**

`./mountdir:`
`total 16`
`drwxrwxr-x 2 vm vm 4096 Jun 10 15:39 .`
`drwxrwxr-x 4 vm vm 4096 Jun 10 15:20 ..`
`-rw-rw-r-- 1 vm vm 8192 Jun 10 15:39 a1.txt`
`-rw-rw-r-- 1 vm vm 8192 Jun 10 15:39 a2.txt`
	`./mountdir:`
	`total 12`
	`drwxrwxr-x 2 vm vm 4096 Jun 10 15:40 .`
	`drwxrwxr-x 4 vm vm 4096 Jun 10 15:20 ..`
	`-rw-rw-r-- 1 vm vm 8192 Jun 10 15:39 a2.txt`

`./rootdir:`
`total 32`
`drwxrwxr-x 2 vm vm 4096 Jun 10 15:39 .`
`drwxrwxr-x 4 vm vm 4096 Jun 10 15:20 ..`
`-rw-rw-r-- 1 vm vm 80 Jun 10 15:39 a1.txt`
`-rw-rw-r-- 1 vm vm 80 Jun 10 15:39 a2.txt`
`-rw-rw-r-- 1 vm vm 12288 Jun 10 15:39 blockdata.bin`
`-rw-rw-r-- 1 vm vm 123 Jun 10 15:39 blockstorage.txt`
	`./rootdir:`
	`total 24`
	`drwxrwxr-x 2 vm vm 4096 Jun 10 15:40 .`
	`drwxrwxr-x 4 vm vm 4096 Jun 10 15:20 ..`
	`-rw-rw-r-- 1 vm vm 80 Jun 10 15:39 a2.txt`
	`-rw-rw-r-- 1 vm vm 8192 Jun 10 15:40 blockdata.bin`
	`-rw-rw-r-- 1 vm vm 82 Jun 10 15:40 blockstorage.txt`

6.2 Second Test - Test B

We begin by copying the files from the `testB` directory into the mount directory using the command `cp -R ../experiments/testB/file1.txt mountdir/`.
`cp -R ../experiments/testB/file2.txt mountdir/`.

Now we are going to test that if two files are identical, we expect to have 2 hashes in the blockstorage.txt and 2 4KBs of data in blockstorage.bin.

```
./mountdir:
-rw-rw-r-- 1 vm vm 8192 Jun 10 16:28 file1.txt
-rw-rw-r-- 1 vm vm 8192 Jun 10 16:28 file2.txt

./rootdir:
total 28
-rw-rw-r-- 1 vm vm 8192 Jun 10 16:28 blockdata.bin
-rw-rw-r-- 1 vm vm   82 Jun 10 16:28 blockstorage.txt
-rw-rw-r-- 1 vm vm   80 Jun 10 16:28 file1.txt
-rw-rw-r-- 1 vm vm   80 Jun 10 16:28 file2.txt

vm@vm:~/fuse-tutorial-2018-02-04/example$ cat rootdir/blockstorage.txt
608cabc0f2fa18c260cafd974516865c772363d5
59682bddd4b2a31b08bc6977169691c10db7a501
vm@vm:~/fuse-tutorial-2018-02-04/example$ ls -l rootdir/blockdata.bin
-rw-rw-r-- 1 vm vm 8192 Jun 10 16:28 rootdir/blockdata.bin
```

We see that we have 2 identical files that are 8KBs each but in our blockdata, the place where we store the data is 8KBs and in the hash file only 2 hashes appear. This demonstrates that our compresion works.

6.3 Different Files - Test C

This test aims to demonstrate the case where all files hosted contain different/unique blocks, which will result in files that won't be compressed. Four files of 128KBs each will be used.

Since every file has unique blocks, the result should be a blockdata.bin file of 512KBs (or 524.288 bytes as shown below) and there will be 128 hashes produced.

```
kperidis@oslabserver:~/project3/fuse-tutorial-2018-02-04/example$ cp -R ../experiments/
    test_unique_blocks/ mountdir/
kperidis@oslabserver:~/project3/fuse-tutorial-2018-02-04/example$ ls -alR
.:
total 76
drwxr-xr-x 4 kperidis kperidis  4096 Jun 10 15:39 .
drwxr-xr-x 6 kperidis kperidis  4096 May 30 17:20 ..
-rw-rw-r-- 1 kperidis kperidis 55211 Jun 10 18:05 bbfs.log
-rw-r--r-- 1 kperidis kperidis   185 Feb  4  2018 Makefile
drwxrwxr-x 3 kperidis kperidis  4096 Jun 10 18:04 mountdir
drwxrwxr-x 3 kperidis kperidis  4096 Jun 10 18:04 rootdir

./mountdir:
total 12
drwxrwxr-x 3 kperidis kperidis 4096 Jun 10 18:04 .
drwxr-xr-x 4 kperidis kperidis 4096 Jun 10 15:39 ..
drwxrwxr-x 2 kperidis kperidis 4096 Jun 10 18:04 test_unique_blocks

./mountdir/test_unique_blocks:
total 24
drwxrwxr-x 2 kperidis kperidis   4096 Jun 10 18:04 .
drwxrwxr-x 3 kperidis kperidis   4096 Jun 10 18:04 ..
-rw-rw-r-- 1 kperidis kperidis 131072 Jun 10 18:04 file1.txt
-rw-rw-r-- 1 kperidis kperidis 131072 Jun 10 18:04 file2.txt
-rw-rw-r-- 1 kperidis kperidis 131072 Jun 10 18:04 file3.txt
-rw-rw-r-- 1 kperidis kperidis 131072 Jun 10 18:04 file4.txt

./rootdir:
total 532
drwxrwxr-x 3 kperidis kperidis   4096 Jun 10 18:04 .
drwxr-xr-x 4 kperidis kperidis   4096 Jun 10 15:39 ..
-rw-rw-r-- 1 kperidis kperidis 524288 Jun 10 18:04 blockdata.bin
-rw-rw-r-- 1 kperidis kperidis   5248 Jun 10 18:04 blockstorage.txt
drwxrwxr-x 2 kperidis kperidis   4096 Jun 10 18:04 test_unique_blocks

./rootdir/test_unique_blocks:
total 24
drwxrwxr-x 2 kperidis kperidis 4096 Jun 10 18:04 .
drwxrwxr-x 3 kperidis kperidis 4096 Jun 10 18:04 ..
-rw-rw-r-- 1 kperidis kperidis 1280 Jun 10 18:04 file1.txt
-rw-rw-r-- 1 kperidis kperidis 1280 Jun 10 18:04 file2.txt
-rw-rw-r-- 1 kperidis kperidis 1280 Jun 10 18:04 file3.txt
-rw-rw-r-- 1 kperidis kperidis 1280 Jun 10 18:04 file4.txt
```

Listing 3: Directory tree after copying of our test files

Regarding the hashes, we can see that they are actually 128, as expected.

```
kperidis@oslabserver:~/project3/fuse-tutorial-2018-02-04/example$ wc -l rootdir/
    blockstorage.txt
128 rootdir/blockstorage.txt
```

6.4 Big File support Test- Test D

The simplicity of our design allows us to host more than 10 files and each file being more than 64KBs. We will test for 10 files being 68KBs each and a pdf file with size 416KBs in a different directory. The way we made the files, each file contains 17 4KBs blocks of data, where the 10 of them are the same for each file and the 7 rest are unique to each file. So we expect to see 10 shared hashes plus 7*10 unique

hashes = 10 + 70 = 80 hashes in total for the txt files plus 416KBs so in total 184 different hashes in the blockstorage.txt file and the blockdata.bin being 733KBs. After we copied the files into the directory the output is the following:

```
vm@vm:~/fuse-tutorial-2018-02-04/example$ ls -alR
.:
total 108
drwxrwxr-x 4 vm vm  4096 Jun 10 15:20 .
drwxrwxr-x 5 vm vm  4096 Jun 10 15:16 ..
-rw-rw-r-- 1 vm vm 93322 Jun 10 15:59 bbfs.log
drwxrwxr-x 4 vm vm  4096 Jun 10 15:59 mountdir
drwxrwxr-x 4 vm vm  4096 Jun 10 15:59 rootdir

./mountdir:
total 16
drwxrwxr-x 4 vm vm 4096 Jun 10 15:59 .
drwxrwxr-x 4 vm vm 4096 Jun 10 15:20 ..
drwxrwxr-x 2 vm vm 4096 Jun 10 15:59 files_txt
drwxrwxr-x 2 vm vm 4096 Jun 10 15:59 pdf

./mountdir/files_txt:
total 48
drwxrwxr-x 2 vm vm  4096 Jun 10 15:59 .
drwxrwxr-x 4 vm vm  4096 Jun 10 15:59 ..
-rw-rw-r-- 1 vm vm 69632 Jun 10 15:59 test_file10.txt
-rw-rw-r-- 1 vm vm 69632 Jun 10 15:59 test_file2.txt
-rw-rw-r-- 1 vm vm 69632 Jun 10 15:59 test_file3.txt
-rw-rw-r-- 1 vm vm 69632 Jun 10 15:59 test_file4.txt
-rw-rw-r-- 1 vm vm 69632 Jun 10 15:59 test_file5.txt
-rw-rw-r-- 1 vm vm 69632 Jun 10 15:59 test_file6.txt
-rw-rw-r-- 1 vm vm 69632 Jun 10 15:59 test_file7.txt
-rw-rw-r-- 1 vm vm 69632 Jun 10 15:59 test_file8.txt
-rw-rw-r-- 1 vm vm 69632 Jun 10 15:59 test_file9.txt
-rw-rw-r-- 1 vm vm 69632 Jun 10 15:59 test_file.txt

./mountdir/pdf:
total 16
drwxrwxr-x 2 vm vm   4096 Jun 10 15:59 .
drwxrwxr-x 4 vm vm   4096 Jun 10 15:59 ..
-rw-rw-r-- 1 vm vm 425984 Jun 10 15:59 shattered-1.pdf

./rootdir:
total 760
drwxrwxr-x 4 vm vm   4096 Jun 10 15:59 .
drwxrwxr-x 4 vm vm   4096 Jun 10 15:20 ..
-rw-rw-r-- 1 vm vm 750115 Jun 10 15:59 blockdata.bin
-rw-rw-r-- 1 vm vm   7544 Jun 10 15:59 blockstorage.txt
drwxrwxr-x 2 vm vm   4096 Jun 10 15:59 files_txt
drwxrwxr-x 2 vm vm   4096 Jun 10 15:59 pdf

./rootdir/files_txt:
total 48
drwxrwxr-x 2 vm vm 4096 Jun 10 15:59 .
drwxrwxr-x 4 vm vm 4096 Jun 10 15:59 ..
-rw-rw-r-- 1 vm vm  680 Jun 10 15:59 test_file10.txt
-rw-rw-r-- 1 vm vm  680 Jun 10 15:59 test_file2.txt
-rw-rw-r-- 1 vm vm  680 Jun 10 15:59 test_file3.txt
-rw-rw-r-- 1 vm vm  680 Jun 10 15:59 test_file4.txt
-rw-rw-r-- 1 vm vm  680 Jun 10 15:59 test_file5.txt
-rw-rw-r-- 1 vm vm  680 Jun 10 15:59 test_file6.txt
-rw-rw-r-- 1 vm vm  680 Jun 10 15:59 test_file7.txt
-rw-rw-r-- 1 vm vm  680 Jun 10 15:59 test_file8.txt
-rw-rw-r-- 1 vm vm  680 Jun 10 15:59 test_file9.txt
-rw-rw-r-- 1 vm vm  680 Jun 10 15:59 test_file.txt

./rootdir/pdf:
total 16
drwxrwxr-x 2 vm vm 4096 Jun 10 15:59 .
drwxrwxr-x 4 vm vm 4096 Jun 10 15:59 ..
-rw-rw-r-- 1 vm vm 4160 Jun 10 15:59 shattered-1.pdf
vm@vm:~/fuse-tutorial-2018-02-04/example$
```

What we want see for the hashes is this:

```
vm@vm:~/fuse-tutorial-2018-02-04/example$ wc -l rootdir/blockstorage.txt
184 rootdir/blockstorage.txt
```

As we see here there are 184 unique hashed in the blockdata.bin file, and:

```
vm@vm:~/fuse-tutorial-2018-02-04/example$ ls -l rootdir/blockdata.bin
-rw-rw-r-- 1 vm vm 750115 Jun 10 15:59 rootdir/blockdata.bin
```

We see that blockdata.bin is 733KBs as expected(we round up the number for easier comprehension). So we see that our implementation works for bigger files.

6.5 Non Volatile test

We designed our implementation to be non volatile, meaning when files are created or copied in the mounted file system when we unmount and remount we can still see our files. Let's see an example with the first test case, we are going to copy the files from testA in our mountdir, ountmount and remount it and we expect to see the same contents in the files:

```
vm@vm:~/fuse-tutorial-2018-02-04/example$ ls -alR
./mountdir:
total 16
-rw-rw-r-- 1 vm vm 8192 Jun 10 16:14 file1.txt
-rw-rw-r-- 1 vm vm 8192 Jun 10 16:14 file2.txt

./rootdir:
total 32
-rw-rw-r-- 1 vm vm 12288 Jun 10 16:14 blockdata.bin
-rw-rw-r-- 1 vm vm   123 Jun 10 16:14 blockstorage.txt
-rw-rw-r-- 1 vm vm    80 Jun 10 16:14 file1.txt
-rw-rw-r-- 1 vm vm    80 Jun 10 16:14 file2.txt

----copied files in the mountdir and now we unmount and remount

vm@vm:~/fuse-tutorial-2018-02-04/example$ fusermount -u mountdir
vm@vm:~/fuse-tutorial-2018-02-04/example$ ../src/bbfs rootdir mountdir
Fuse library version 2.9
about to call fuse_main
vm@vm:~/fuse-tutorial-2018-02-04/example$ ls -alR
.:
total 20
-rw-rw-r-- 1 vm vm 1489 Jun 10 16:18 bbfs.log
drwxrwxr-x 2 vm vm 4096 Jun 10 16:14 mountdir
drwxrwxr-x 2 vm vm 4096 Jun 10 16:14 rootdir

./mountdir:
total 16
drwxrwxr-x 2 vm vm 4096 Jun 10 16:14 .
drwxrwxr-x 4 vm vm 4096 Jun 10 15:20 ..
-rw-rw-r-- 1 vm vm 8192 Jun 10 16:14 file1.txt
-rw-rw-r-- 1 vm vm 8192 Jun 10 16:14 file2.txt

./rootdir:
total 32
-rw-rw-r-- 1 vm vm 12288 Jun 10 16:14 blockdata.bin
-rw-rw-r-- 1 vm vm   123 Jun 10 16:14 blockstorage.txt
-rw-rw-r-- 1 vm vm    80 Jun 10 16:14 file1.txt
-rw-rw-r-- 1 vm vm    80 Jun 10 16:14 file2.txt
```

and we run:

```
vm@vm:~/fuse-tutorial-2018-02-04/example$ cat mountdir/file1.txt
AAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

(this file contains 4KBs of As and 4KBs of Bs, thats why we will not include it here, but we include this smaller version to demonstrate that it works).