Operating Systems
Project 3

Tsogkas Panagiotis Nikas Ioannis Iason Tsiantos Dimitrios

June 2025

Contents

1 Description **3**

2 Basic File Operations **3**
2.1 Write . . . 3
2.2 Read . . . 3
2.3 Delete . . . 4

3 File and Directory Hierarchy **5**

4 Functions **5**

5 Experiments **6**
5.1 Block Structure Tests . . . 6
5.2 Directory and File Hierarchy Structure Tests . . . 7

6 Execution Instructions **7**
6.1 Running the File System . . . 7
6.2 Running Experiments . . . 7

1 Description

In this report, we present the implementation of a non-volatile file system with block-level compression. This mechanism can reduce reserved disk space in cases where files share identical blocks, though it increases the cost of write, read, and delete operations compared to a simple file system.

The goal of the file system is the reuse of common blocks, the successful reconstruction of the original file during **reading**, the correct update of metadata during **writing** and **deletion**, as well as the preservation of the **file** and **directory** structure.

For reasons of simplicity and ease, the file system implementation is based on the Big Brother File System (BBFS), which uses the FUSE library, utilizing the Virtual File System.

2 Basic File Operations

2.1 Write

To achieve file compression during writing, it suffices to: i) identify identical blocks being written to the disk, ii) write them only once, and iii) maintain pointers for each file that point to these blocks.

For the first step, we use the SHA-1 hashing algorithm to identify common data blocks. It should be noted that the hash collision phenomenon may occur, which we ignore in this implementation. Once a new block is identified, we write it to the block repository in contiguous positions and place a new entry in a hashmap with the hash as the key and the block offset as the value. This hashmap maintains metadata for the blocks.

Hash (SHA-1)	**Block offset**
01b307acba4f54f55aafc33bb06bbbf6ca803e9a	3
bd5e5eb049f3907175f54f5a571ba6b9fdea36ab	1
64ea0dc7dadd49a337f1ef14815bd3f428141c7d	8
...	...

Table 1: Example hashmap mapping block hashes to block offsets in the repository.

We select 4KB as the fixed size for each block and 4 bytes for the size of the block offset field. This means that in this file system, we can write up to 2^{32} unique blocks. Thus, the maximum capacity of unique blocks on the disk is:

$$\#max_blocks = 2^{block_offset_bits} = 2^{32} = 4,294,967,296\ blocks$$

$$max_disk_size = \#max_blocks \times sizeof_block = 2^{32} \times 2^{12} = 2^{44} = 16TB$$

2.2 Read

To read a file, we need to reconstruct its individual blocks. This can be achieved by storing the sequence of blocks in a virtual file during the write process.

To support reading dynamic sizes, and not just multiples of 4 kilobytes, it is necessary to store the size of the last block, as only the last block can have a size smaller than 4KB. Dynamic file sizing may cause internal fragmentation if the number of bytes is not a multiple of 4096, but in the worst case, this fragmentation will be 4095 bytes per system file and, on average, about 2048 bytes per file.

Last Block Size	50
Hashes	01b307acba4f54f55aafc33bb06bbbf6ca803e9a 01b307acba4f54f55aafc33bb06bbbf6ca803e9a bd5e5eb049f3907175f54f5a571ba6b9fdea36ab 64ea0dc7dadd49a337f1ef14815bd3f428141c7d 64ea0dc7dadd49a337f1ef14815bd3f428141c7d ...

Table 2: Example of a virtual file used to reconstruct the original file using the last block size and block hashes.

2.3 Delete

During deletion, issues may arise regarding the hashmap and the repository. After deleting a file (i.e., removing the virtual file), it is necessary to delete the corresponding blocks it used. However, other virtual files may still need these blocks. The solution to this problem is to add a value to the blocks hashmap representing the number of references from other files to each block. With this change, we delete a block only if the references to it are zero.

Hash (SHA-1)	**Block offset**	**References**
01b307acba4f54f55aafc33bb06bbbf6ca803e9a	3	2
bd5e5eb049f3907175f54f5a571ba6b9fdea36ab	1	1
64ea0dc7dadd49a337f1ef14815bd3f428141c7d	8	2
...	...	...

Table 3: Example hashmap mapping block hashes to offsets and reference counts.

Following file deletions, and consequently block deletions, gaps are created within the repository which we would ideally like to refill by placing new blocks from subsequent writes.

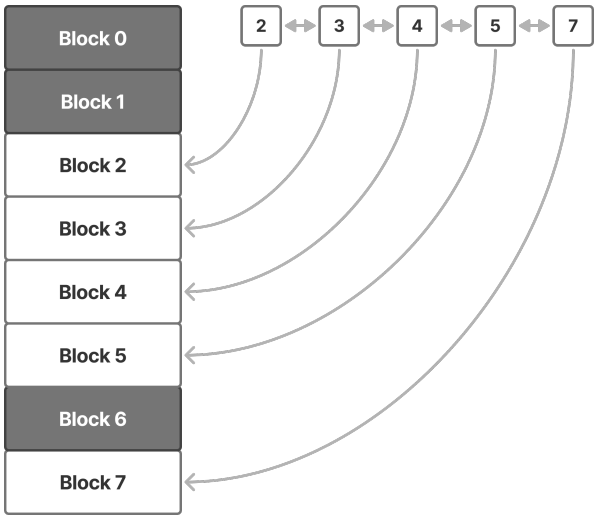


Figure 1: Block repository and free list.

For the system to know which blocks are free, we need to store a sorted doubly linked circular list with a sentinel that holds the offsets of free blocks as values. Thus, during writing, we remove the first element from the list and use the offset to write to the corresponding block.

Another problem that may arise from deletion is external fragmentation in the block repository. In the worst case, in a system full of unique blocks, if all are deleted except the last one, a new write would fail due to lack of space for metadata, as the block repository would maintain its initial fixed size. This is solved by block defragmentation, moving the last blocks to the first available free positions.

Ideally, the function executing this process should be called under specific conditions, for example, if the disk begins to fill up. For testing purposes, in this implementation, we perform partial defragmentation when external fragmentation exceeds 10%, continuing until it drops below 10%, ensuring the percentage stays under this limit at any given time. The issue with this tactic is that it exhibits the classic "thermostat" problem, where the system tends to oscillate around the selected threshold limit.

3 File and Directory Hierarchy

Another fundamental responsibility of the file system is to present the user with a structured tree-like hierarchy of directories and files. Simulating such a structure requires storing metadata for the directories. Each directory will have its own hashmap containing its contents: the names of the nodes it contains, the node type (file or directory), the unique ID of each node, and, in the case of a directory, a pointer to another hashtable.

ID	Type	Name Length	Name
2	F	9	file1.txt
3	D	9	settings
4	F	5	bogus
...	...	...	...

Table 4: Example of directory metadata on disk.

We can store this data on the disk, and when the filesystem is initialized, we recursively build the directory hierarchy in memory for better performance. When the file system is unmounted, this metadata must be written back from memory to the disk.

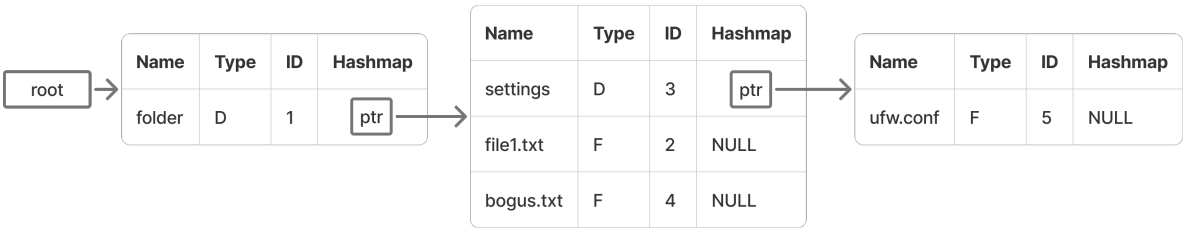


Figure 2: Example of directory hierarchy in memory.

Each record requires 8 bytes for the ID, 1 byte for the type, 256 bytes for the name, and 8 bytes for the pointer to another hashmap. Therefore, a total of 273 bytes is needed for each node. For example, if we store the Linux Kernel (v6.8.0) code in our filesystem, the total space for node metadata would be $273 \times 88982 = 24292086 \approx 23.1667MB$.

4 Functions

The modified functions are as follows:

`bb_fullpath`, `bb_getattr`, `bb_mknod`, `bb_mkdir`, `bb_unlink`, `bb_rmdir`, `bb_truncate`, `bb_open`, `bb_read`, `bb_write`, `bb_opendir`, `bb_readdir` & `bb_init`.

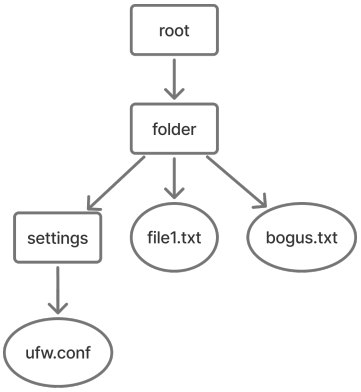


Figure 3: The node hierarchy described in Figure 2.

5 Experiments

To ensure the file system operates correctly, we use automated **functional tests** that check the file system's functionality. We have implemented two different categories of tests: the first checks block structures, and the second checks structures in the node hierarchy. The Python framework "unittest" is used for testing.

5.1 Block Structure Tests

Experiment	**Purpose**
Write one block	Verify correct writing and reading of a single block in a file.
Write ten blocks	Verify correct writing and reading of multiple blocks in a file.
Write 10.5 blocks	Verify correct writing and reading of sizes that are not multiples of the block size.
Overwrite block data	Verify that writing to a block correctly replaces previous data.
Data Compression	Verify that data is compressed during write and correctly decompressed during read.
ftruncate to shrink file	Verify that the file can be shrunk using ftruncate and maintain correct content.
Add padding with ftruncate	Verify that when a file is grown via ftruncate, new bytes are filled with zeros.
Block repository Defragmentation	Verify that blocks are correctly redistributed to avoid gaps after deletions.

5.2 Directory and File Hierarchy Structure Tests

Experiment	Purpose
Create and delete directory	Verify the correct creation and deletion of a simple directory.
Create multiple directories	Verify the creation of multiple directories within the same directory.
Create/delete directory and file	Verify creation and deletion of files and directories within the same directory.
Create many files in dir	Verify correct management of many files within a single directory.
Create nested dirs and file	Verify creation of a multi-level directory hierarchy and adding a file inside.
Create many dirs and files	Verify creation of a complex structure with many directories and files at various levels.

6 Execution Instructions

Instructions for running the file system and experiments can be found in the Readme files of the `filesystem` and `experiments` folders respectively. These instructions are summarized in sections 6.1 and 6.2.

6.1 Running the File System

To run the filesystem on a Linux system, fuse and libssl must first be installed:

```
sudo apt update
sudo apt install pkg-config libssl-dev libfuse-dev
```

Next, inside the `filesystem` folder, execute the command:

```
make
```

The filesystem requires two directories to run: one where metadata will be stored and one for the user to use.

```
mkdir -p rootdir
mkdir -p mountdir
```

Then, execute the executable with the metadata directory and user directory as arguments:

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

After using the filesystem, finally unmount it with the command:

```
fusermount -u ./mountdir
```

6.2 Running Experiments

The experiments are executed with the following command in the `experiments` folder:

```
python3 test.py
```