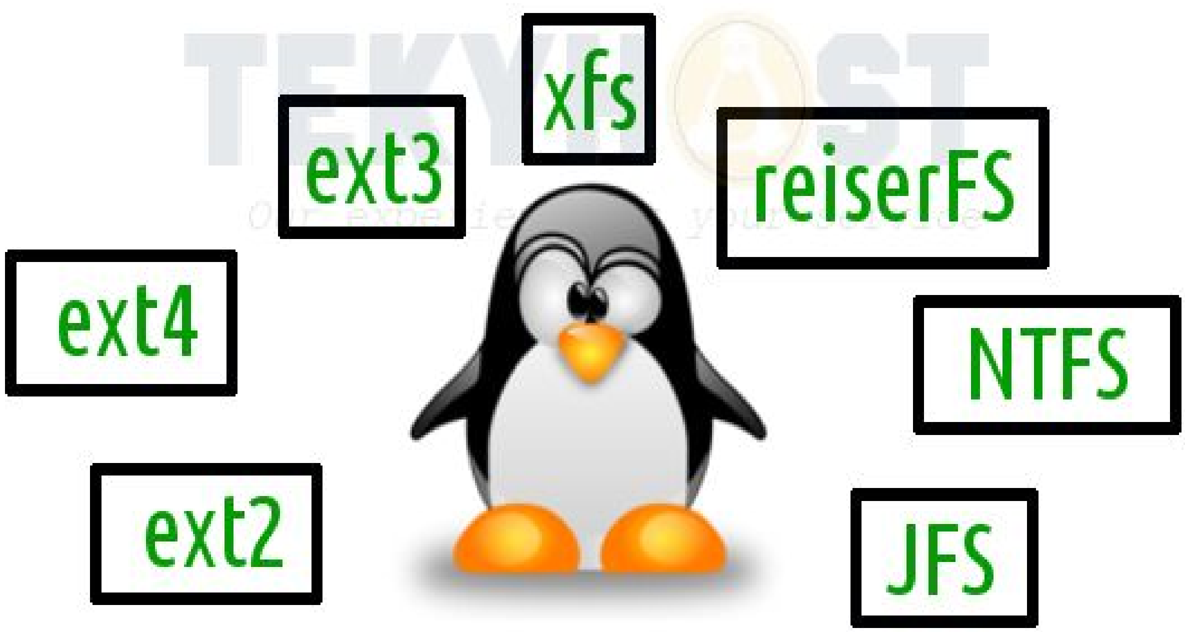
OS MINI-PROJECT
FILE SYSTEMS WITH FUSE

Department of Computer Science Engineering
PES University
Bangalore, India - 85



Members:

1. Nishant Ravi Shankar → 01FB16ECS235
2. Omkar Metri → 01FB16ECS239
3. P. Abhishikta Sai → 01FB16ECS242

Abstract: Open-source local file systems, such as Linux Ext4 , remain a critical component in the world of modern storage. For example, many recent distributed file systems, such as Google GFS and Hadoop DFS, all replicate data objects (and associated metadata) across local file systems. Finally, many desktop users still do not backup their data regularly. In this case, the local file system clearly plays a critical role as sole manager of user data.

Table of Contents:

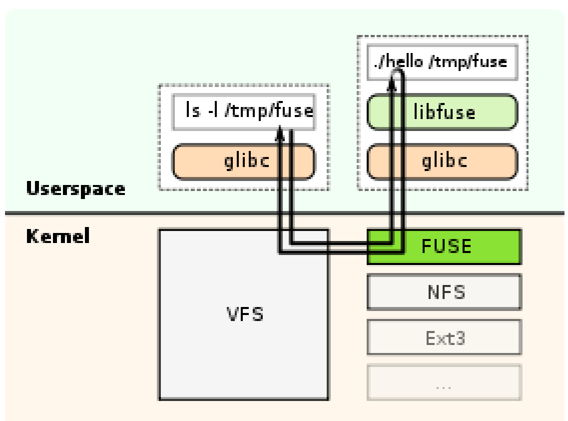
1. Introduction

Overview

Definition - What does *File System* mean?

A file system is a process that manages how and where data on a storage disk, typically a hard disk drive (HDD), is stored, accessed and managed. It is a logical disk component that manages a disk's internal operations as it relates to a computer and is abstract to a human user.

Filesystem in Userspace (**FUSE**) is a software interface for Unix-like computer operating systems that lets non-privileged users create their own file systems without editing kernel code. This is achieved by running file system code in user space while the FUSE module provides only a "bridge" to the actual kernel interfaces.



Goals

- Main purpose of this program is to specify how the file system is to respond to read/write/stat requests.
- Program is also used to mount the new file system. At the time the file system is mounted, the handler is registered with the kernel. If a user now issues read/write/stat requests for this newly mounted file system, the kernel forwards these IO-requests to the handler and then sends the handler's response back to the user.
- FUSE is particularly useful for writing virtual file systems. Unlike traditional file systems that essentially work with data on mass storage, virtual filesystems don't

actually store data themselves. They act as a view or translation of an existing file system or storage device.

2. Requirements

Hardware:

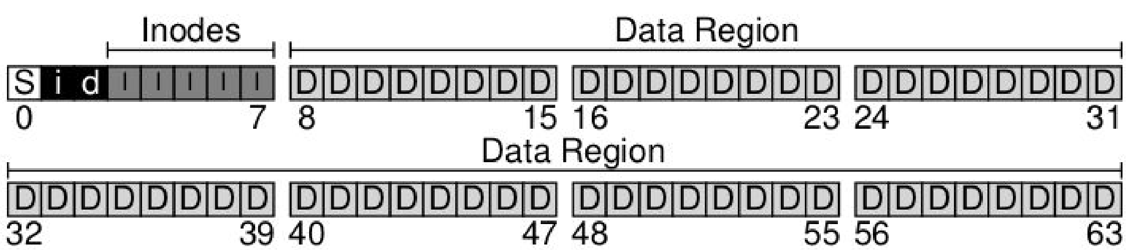
- Disk space
- Mount point

Software:

- Linux OS
- Fuse

3. Design

Block Diagram



File System View

1. File system consists of two types of data - user data and metadata, hence persistence was made to accommodate both.
2. ‘fsData’ stores file metadata (directories and files)
3. 56 blocks of size 512 bytes. Each directory or file has a dedicated block
4. In case of directory, first 44 bytes are used for the storage of metadata followed by the user given name and path
5. List maintaining block numbers and element types are also stored
6. All the blocks are loaded onto memory at the start which is stored in ‘Blockdata’
7. Each file can use a max of 8 blocks, thereby having size 4096 bytes.

4. Implementations

Phase 1: Basic file I/O operations referencing only to memory

- We have implemented a hierarchical data structure (multi-tree)

Structures declarations:

File Info

```
struct fileInfo
{
    int fileId;
    nlink_t st_nlink;
    size_t size;
    mode_t mode;
    blkcnt_t blockcount;
    blksize_t block_size;
    int blockno[8];
    char *path;
    int path_length;
    char *name;
    int name_length;
    int uid;
    int gid;
};
```

Directory

```
struct directory {
    mode_t mode;
    int block_number;
    int offset;
    int index;
    int path_length;
    char *path;
    int name_length;
    char *name;
    struct directory **children;
    struct directory *parent;
    int *fileBlockNumbers;
    int n_children;
    int n_link;
    int filecount;
    int uid;
    int gid;
};
```

SuperBlock

```
#define N_BLOCK_DATA 56
#define N_DATA  80
#define BLOCK_SIZE 4096
#define DATA_SIZE 528
#define DATA_BLOCK_SIZE 512
#define N_CHILDREN_OFFSET 40
#define PATH_OFFSET 44

struct superBlock {
    int totalblocks;
    int totalfreeb;
};
```

Block

```
struct block {
    int blockno;
    int valid;
    void *data;
    int size;
    int current_size;
};
```

Fuse Operations:

1. Do_getattr → Collects all the attributes, i.e, metadata of the data required for for operations on the file
2. Do_readdir → reads the contents of the directories
3. File_init → creates a Data BitMap and initialises the updates along with metadata.
4. File_mkdir → creates a directory at the specified location
5. File_rmdir → remove the directory if is empty
6. File_create → creates a file and stores the metadata
7. File_open → given the path, searches for file and returns descriptor
8. File_read → read the contents from the specified file descriptor
9. File_write → writes the contents to the specified file descriptor

Helper Functions:

1. Update_file → updates the content and metadata file
2. Find_file → checks for the presence of the file
3. checkIfDelimiterInPath → checks for the '/' notation in the path
4. createChild → creates a child in the directory and updates the metadata
5. checkPath → checks for the existence of the path

Phase 2: File System Abstractions

File-system has two types of data:

1) Meta data – This consists of all the inode equivalent information of a file or directory.

2) User data – This consists of all the data written by the user in files.

Block Data: In our implementation of the file system, metadata and user-data are dealt in different files. The meta data is stored in a file called fsData and the user-data is stored in a file called BlockData.

fsData File: This file stores the meta data of the file. The file is divided into blocks of size 512 bytes. Each directory or file in the file system has a dedicated block that stores all the related meta data of that directory or file. Since files and directories meta-data are different, the block structure of files and directories have been implemented differently.

The block organization of a directory is :

1. offset - The total number of bytes of metadata currently written in the block
2. block number - The block number of the current block, i.e, The root has a default block number of 0.

3. uid - The user id of the directory
4. gid - The group id of the directory
5. mode - The permissions mode of the directory
6. path length - The length of the path (string length) of the directory from the root of the file system.
7. name length - The string length of the name of the directory.
8. filecount - The total number of files in the directory.
9. number of children - The number of sub-directories within that directory.
10. path - The string path of the directory from the root of the file system.
11. name - The name of the directory.
12. variable length:

- block number - The block number of the child of the directory (file or directory)
- type - This is to denote the type of child for the directory. 'd' for sub-directory and 'f' for file

The block organization of a file is:

1. file_id - The inode number of the file
2. name length - The string length of the name of the file
3. name of the file
4. uid - The user id of the file in the file system
5. gid - The group id of the file in the file system
6. mode - The permission mode of the file in the file system
7. The string length of the path of the file from the root
8. The total number of datablocks associated with the file
9. The block size of each data-block
10. The total size of the file - (total number of bytes already written into the file)
11. The variable part of the block consists of a set of data blocks numbers. While opening and reading the file, each datablock number is read and is loaded onto the memory and displayed on the terminal.

Helper Functions:

1. **persistence** → This function is called while initializing the file system. This function first checks to see if the persistence files already exist and if not returns -1. If the files exist, then it loads the meta-data bitmap and calls initialize tree. If the persistence files don't exist (when the file system is mounted for the first time), it creates the persistence files and writes default values into them.

2. **initialize tree** → This function reads a directory block and writes all the meta data to a node of the tree. It is implemented recursively. The process of construction a tree is:

- The fsData file is used while mounting the file system and in construction of an in-memory tree. The root directory has a default block number of 0. When the file system is mounted, the root directory is first loaded from block 0 and constructed.
- The initializeTree function first reads the first 44 bytes of the block of a directory and stores the read information in the specific members of the tree node structure.
- As seen in fig xx, after an offset of 44, a directory can have variable number of bytes – which depends on the number of files of sub-directories with that directory. The initializetree function looks at these block numbers and their types. If the type is ‘f’, then the block number is appended into the fileBlockNumbers array in the node. If the type is ‘d’, which means a sub-directory, then a recursive call is made to the initializeTree function and the same process takes place for that sub-directory.

3. **Write node** → Writes a new block into the fsData file. This helper function is called in the mkdir function, right after the node has been inserted into the tree. The block number of this new block is appended into the block of the parent directory, denoting that this new directory is the child of the parent directory.

4. **Remove node** → Clears the block in the fsData file. This is done while removing a directory or a file from the file system. Once the block is cleared, it means the file/directory does not exist in its parent directory. Hence, we go to the block of the parent directory of the deleted file, and remove it from set of children block numbers and reduce the offset of the parent block by the required amount.

5. **LoadDataBitmap**→ Reads the inode bitmap, if "databitmap" is present which basically contains metadata of files and directories.

6. **CreateDatabitMap** → Creates the "databitmap" file that contains inode bitmap initialised with 0's and later updated to 1 based on the user commands.

7. **Getfreeblock** → Fetches the index of first inode block available by traversing through databitmap and finding the first 0.

8. **getfreedatablock** → Fetches the index of the first data block available by traversing through blockbitmap and finding the first 0.

Phase 3: Persistent Storage

Task: Write to secondary storage management such that it implements persistence and preserves the file system state across machine roots

Report: Achieved shutting the file system either through umount or reboot and still have all the files remain intact and in the same state when the file system is remounted.

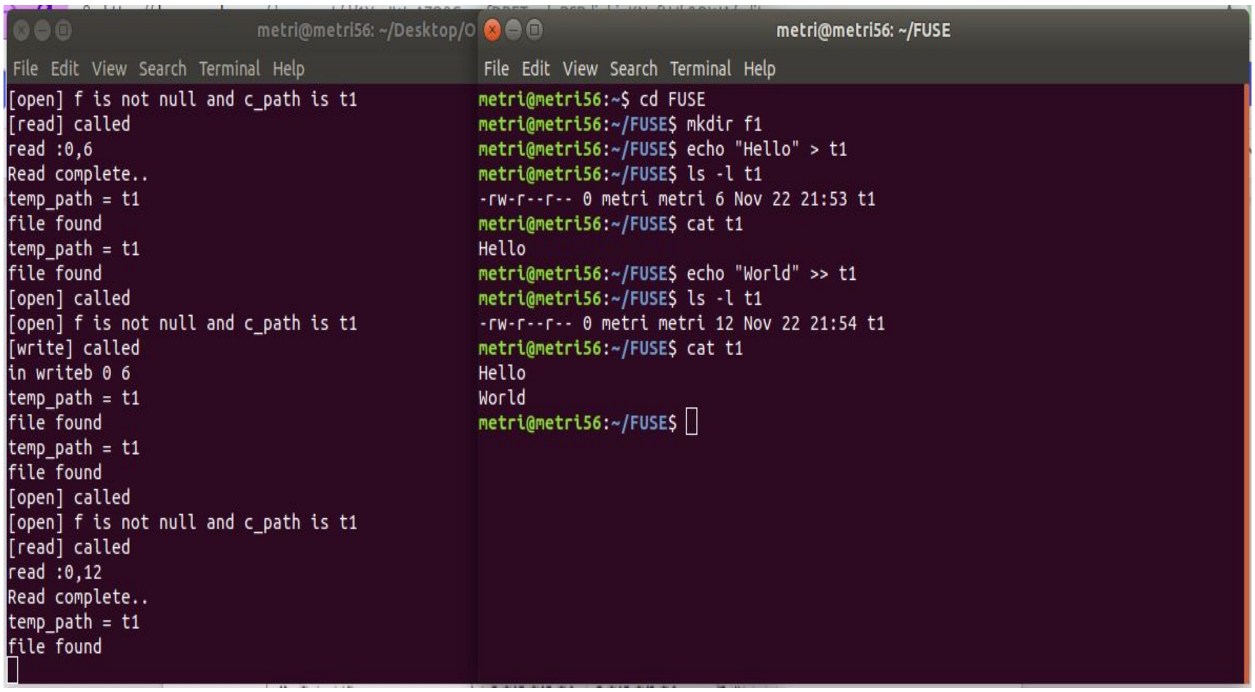
5. Test Cases

Following test cases were experimented

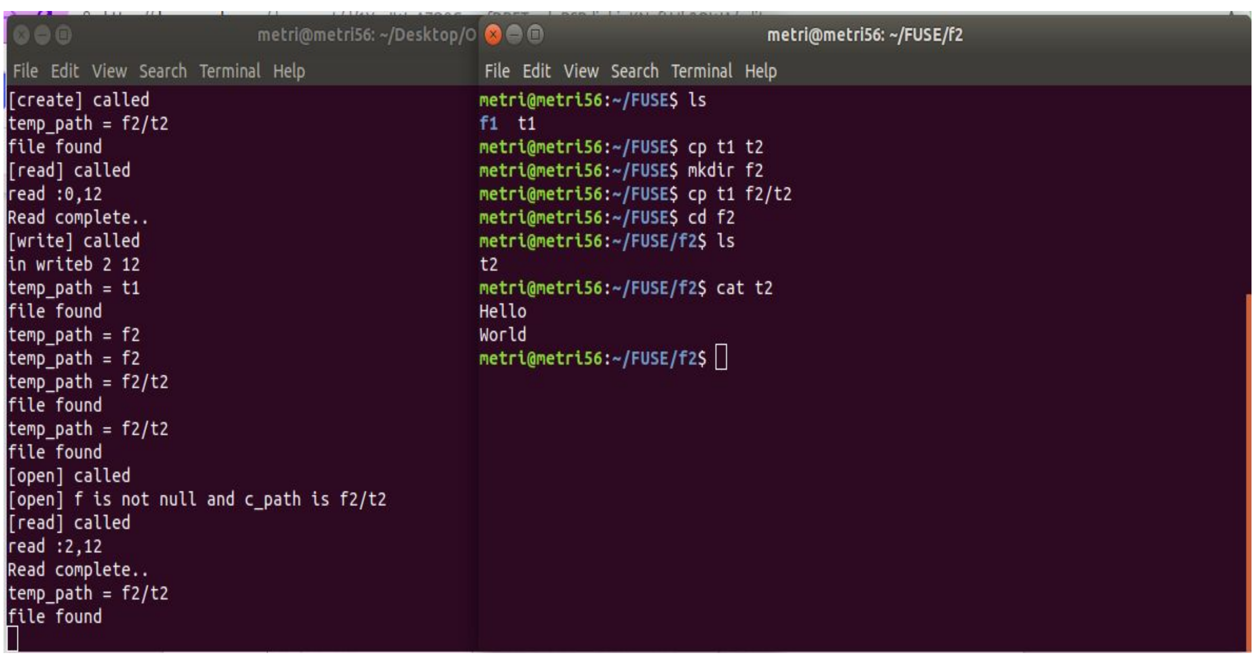
S. No	Desciption	Commands	Expected Output
1	Create Directories		
2	Create File	echo "Hello World1" > TestFile1	TestFile1 created
3	Check opened file	ls -l Testfile1; cat TestFile1	"Hello World1" Text output
4	Append to an existing file	echo "Hello World2" >> TestFile1	Text Appended
5	Copy file	cp TestFile1 TestFile2	File copied
6	Check copied file	ls -l TestFile2; cat TestFile2	Same as TestFile1
7	Copy one directory below	cp TestFile1 TestDir1/TestFile3	Copied
8	Check copied file	ls -l TestDir1/TestFile3; cat TestDir1/TestFile3	Same as TestFile1
9	Create TestFile4	cd TestDir2; echo "Test4" > TestFile4	file created
10	Check created file	ls -l testFile4; cat TestFile4	"Test4" text output
11	Create duplicate file	use an editor to create TestFile4	Error Message
12	Copy File	cp TestFile4 ../TestDir1/TestDir3/TestFile5	File copied
13	Check copied file	ls -l ../TestDir1/TestDir3/TestFile5; cat ../TestDir1/TestDir3TestFile5	"Test4" text output
14	Remove a file	rm TestFile4	File removed
15	Try to remove a nonempty directory	cd ..; rmdir TestDir1	Error Message
15	Remove an empty directory	rmdir TestDir2	Directory removed
16	Create a file spanning 4 blocks	Write 1 to 512 using C/Python to TestFile2	File created
17	Check block usage	du -s TestFile2	4 blocks should be used
18	Reduce file size	Remove lines 101 onwards and save the file	
19	Check block size	du -s TestFile2	Should be still 4 blocks
20	Add additional blocks	Add 101 to 1024 to TestFile2	
21	Check block size	du -s TestFile2	Block size should still be 4

6. Snapshots

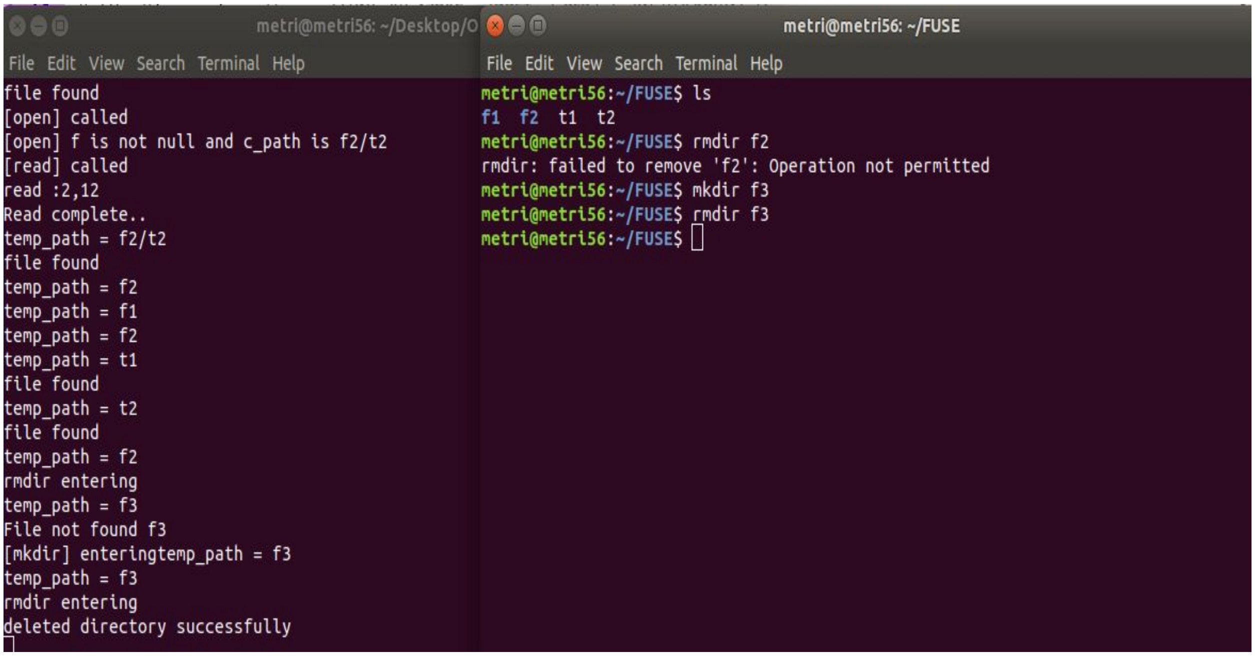
1. Creating a directory, creating a file, appending content to the file



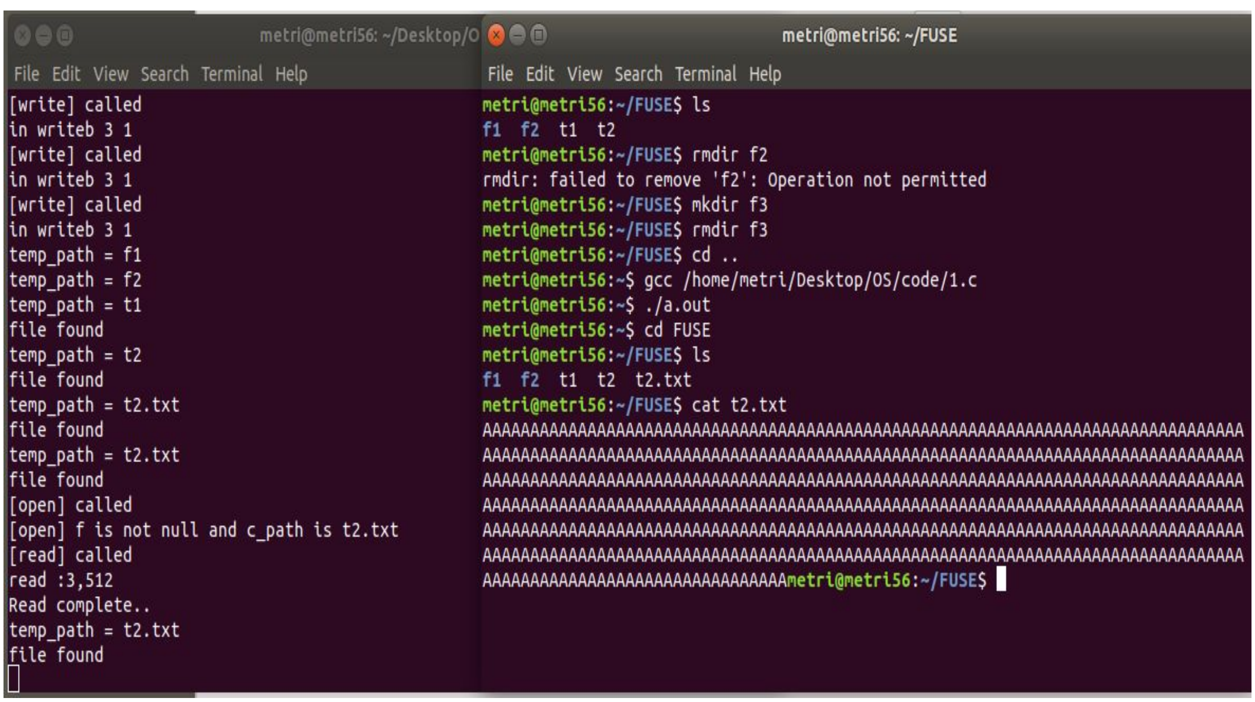
2. Copying a file in the same directory, different directory and displaying the contents



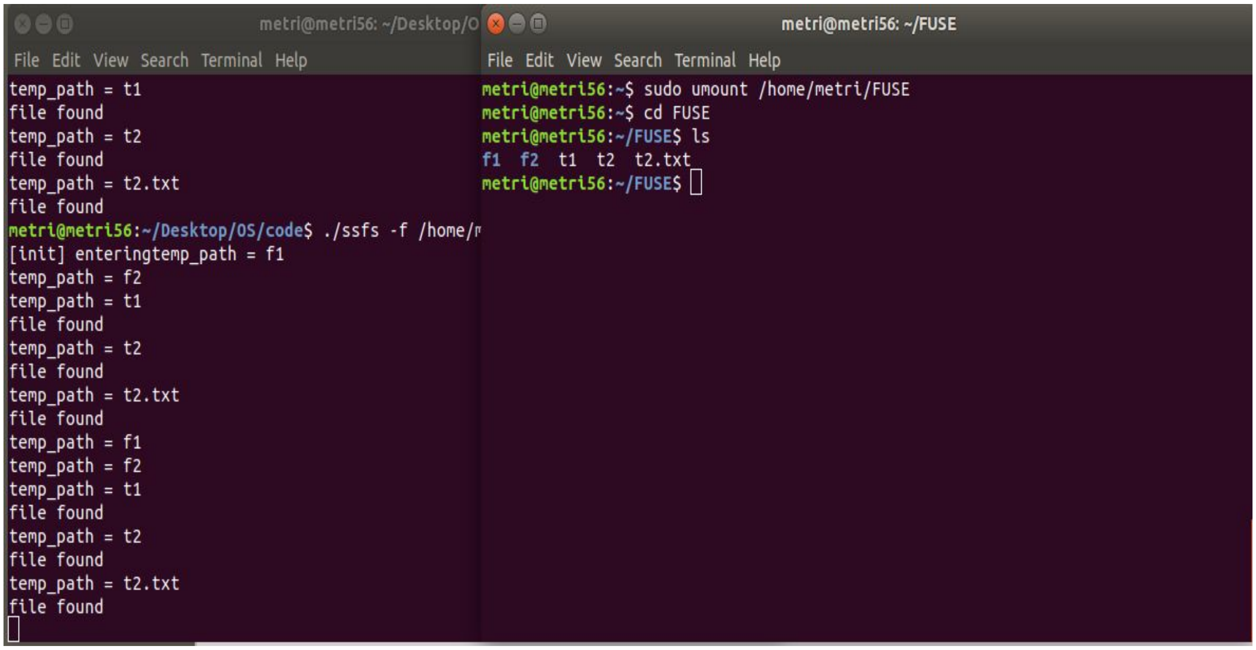
3. Remove a empty directory and a non-empty directory



4. Write 1 to 512 bytes in a file using C/Python code



5. Persistence



Conclusion: Successfully implemented file systems with fuse.

THANK YOU