**INTRODUCTION TO OPERATING SYSTEM (LAB)**

**UE16CS305**

**MINI PROJECT**

**FUSE FILE SYSTEM**

**01FB16ECS124 - Gundam Satyabhama Reddy**

**01FB16ECS149 - K P Rachita Rao**

**01FB16ECS146 - Jayapriya S**

**CONTENTS**

1. Introduction to FUSE 3
2. History 3
3. Operation and uses 4
4. Design of the file system 4
5. File system implementation using FUSE 4
   * Prototypes of structures used 5-6
   * Functions implemented 7-8
   * Helper functions 9
6. Detailed explanation
   * Phase1 10-15
   * Phase2 16-18
   * Phase3 19
7. Test cases screenshots 20-22
8. References 23

INTRODUCTION TO FUSE

FUSE​ (Filesystem in Userspace) is a simple interface for userspace programs to export a virtual filesystem to the Linux kernel. Fuse also aims to provide a secure method for non privileged users to create and mount their own file system implementations.

HISTORY

The FUSE system was originally part of AVFS, a filesystem implementation heavily influenced by the translator concept of the GNU Hurd.

OPERATION AND USES

To implement a new file system, a handler program linked to the supplied libfuse library needs to be written. The main purpose of this program is to specify how the file system is to respond to read/write/stat requests. The 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.

DESIGN

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| SUPER BLOCK | INODE  BLOCK1 | INODE  BLOCK2 | -------- | --------- | INODE  BLOCK8 | DATA  BLOCK1 | DATA  BLOCK2 | --------- | DATA  BLOCK41 |

FILE SYSTEM IMPLEMENTATION USING FUSE

Below are the prototypes of the data structures used to implement the file system.

#define BLOCK\_SIZE 512 // Individual block size

#define NPB 4 // Number of inodes per block

#define T\_BLOCKS 50 // Total number of blocks

#define IMAX 32 // max files/dir that can be stored in a dir

#define IBLOCK 8 // Number of inode blocks

#define DATA 41 // Total number of data blocks

Total Filesystem size – 50\*512=25600 bytes

Each inode size – 128 bytes

Structure to define filesystem:

typedef struct fs

{

SUPER sup;

INODE inodes[IMAX];

DBLOCK dbs[DATA];

}FS;

Structure to define Super Block:

typedef struct super

{

bool imap[IMAX];

bool bmap[DATA];

}SUPER;

Structure to define Data Block:

typedef struct dblock

{

char data[BLOCK\_SIZE-DP];

int next;

int prev;

}DBLOCK;

Structure to define Inode:

typedef struct inode

{

char name[32];

int inum;

mode\_t mode;

int link;

int uid;

int gid;

int size;

int blksize;

int blkcnt;

time\_t acc\_t;

time\_t mod\_t;

time\_t cre\_t;

int db;

int next;

int prev;

int in;

unsigned int fd;

}INODE;

FUNCTIONS IMPLEMENTED:

|  |  |  |  |
| --- | --- | --- | --- |
| **SL. no** | **Function** | **Prototype** | **Description** |
| 1 | file\_init | void \*file\_init(struct fuse\_conn\_info \*conn) | The first function which is called when the filesystem is mounted.  It initializes the filesystem, super block, inode and datablock bitmaps and inode stat information. It initializes the data and inode blocks. |
| 2 | file\_create | file\_create(const char \*path, mode\_t mode,struct fuse\_file\_info \* fi) | Create File. If file already exists, return error. Else, checks the path to find if the file to be created is in the root or inside a directory. Initialize the file attributes respectively and link it to the data structure. |
| 3 | file\_getattr | int file\_getattr(const char \*path, struct stat \*stbuf) | Get file/directory attributes. Check if given path is a file, or directory or a file within a directory. Fills the stat structure for the file/directory if it exists, else returns noentry. |
| 4 | file\_utime | int file\_utime(const char \* path, struct utimbuf \*buf) | Traverse to a file in the given path and update its access and modified times in its respective inode. |
| 5 | file\_mkdir | int file\_mkdir(const char \*path, mode\_t mode) | Creates a directory. Extract the name of the directory. Initialize the directory structure, and link it to the data structure. If the directory with the same name already exists, return error. |
| 6 | file\_readdir | int file\_readdir(const char \*path, void \*buf, fuse\_fill\_dir\_t filler,off\_t offset, struct fuse\_file\_info \*fi) | Traverse to a directory in the given path and extracts the list of files and directories in the given directory. ( ls implementation) |
| 7 | file\_rmdir | int file\_rmdir(const char \* path) | Remove directory. Traverses to the directory to be removed. Unlink it from the data structure and update the inode bitmap |
| 8 | file\_open | int file\_open(const char \* path, struct fuse\_file\_info \* fi) | Open a file. Increments the pointer count to that file. |
| 9 | file\_read | int file\_read(const char \* path, char \* buf, size\_t size, off\_t off,struct fuse\_file\_info \* fi) | Reading file.  It first checks if the read permissions are allowed and then reads the file.  Reading file from the offset given Large files are read by traversing through the data blocks |
| 10 | file\_write | int file\_write(const char \* path, const char \* buf, size\_t size, off\_t off,struct fuse\_file\_info \* fi) | Write in the file.  It writes the data into a particular file if file exists.if the current blocks are not sufficient to  store the data, a new block is allocated and data is written. |
| 11 | file\_rename | int file\_rename(const char \*old, const char \*new) | Traverses to the old and new files  and updates the new file name by accessing its inode information.  Updates the new info to the binary file. |
| 12 | file\_destroy | void file\_destroy(void \*private\_data) | It is called when the file system is unmounted. Any unsaved changes are written to the binary file. |
| 13 | file\_unlink | int file\_unlink(const char \* path) | Remove a file. First check if the file is in root, or in a directory. Traverse through the directory/root to find the file to be removed.Update the inode bitmap and data block bitmaps as free. |

HELPER FUNCTIONS:

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | allocate\_new\_block | int allocate\_new\_block(int block,int file) | Allocates a new block for the file to be written by updating the respective next and prev data blocks.Updates the data block bitmap |
| 2 | disk\_save | void disk\_save() | Writes the unsaved changes into a binary file. |
| 3 | extract | char\* extract(const char\* pa) | Extracts the file name from the given path. |
| 4 | traverse\_to | int traverse\_to(const char\* pa,int\* parent) | Finds the inode of the file, whose path is provided |
| 6 | read\_block | int read\_block(int dblock, char \*buf,int new\_off,int off,int n) | Given the data block number, it reads the data from the respective data block to the memory |
| 7 | write\_block | int write\_block(int dblock,const char \*buf,int new\_off,int off,int n) | Given the data block number, it writes the data from the memory to the respective data block |

DETAILED EXPLANATION:

**Phase 1:**

* We designed the structure for our file system.
* Our structure includes Super block ,Inode blocks and data blocks.Our file system has a total of 50 blocks each of size 512 bytes.
* One block is allocated for super block, 8 for inode block and the rest 41 for data blocks.
  + Super block:

Super block contains the bitmaps to inodes and data blocks of our file system which indicates which of the blocks are empty/full. When a file is created,if first checks if the file already exists and if not it searches for the first free inode block and allocates that block to the file. When a file is deleted the respective inode bitmap and data bitmap are set as free. When the contents of the file are written,it searches for the first data block that is free and allocates that to the file.

* + Inode Block:
    - There are 8 inode blocks.Each inode block is of size 512 Bytes. Therefore we can store around 4 inode informations per one block.So a total of 32 files/directories can be stored in our file system.
    - *Each inode has the following information:*

-Inode number

-Mode - tells if the type of the file is a regular file or a directory. It also gives information about the permissions required by the file

-link - indicates the number of links to a particular file/directory

-uid and gid - indicates the user id and group id

-size - it indicates the size of a file in bytes

- blkcnt - it indicates the number of blocks occupied by the file

- blksize - our block size is fixed. Hence this is 512 bytes

- access time - changes every time a file is accessed or read

-modify time - changes when the contents of the file are modified

-status change time - changes when the permissions or the contents of the file are changed

* + - *Inode also contains:*

- db - indicates the first data block number of the file. And each data block has one more number which indicates the next data block(if the file size is > 512 bytes). And the last data block's "next data block number" will be -1.

- in - indicates the inode number of the first child of the directory . If its a file, then it is -1.

- next - indicates the subsequent children of the same directory. The last child will have its next as -1.

**How is our file system different from UNIX file system?**

1)In Unix file system, the inode also contains the direct and the indirect pointers pointing to the data blocks.We have implemented this in a slightly different manner. We have a db pointer which points to the first data block and if the file grows in size, more and more data blocks are allotted to the file and each data block has a pointer to its next and previous data blocks.

2)The directories in unix file system have a block allocated to them indicating all its children files/directories. We used ‘in’ pointer to point to the first child of the directory and a ‘next’ pointer to the subsequent files/directories in the same folder.

3) Unix uses a block size of 4096 bytes and we used a block size of 512 bytes as ours is a simple and small file system.

Functions implemented in phase 1:

1. Init - initialization. When a FUSE file system is mounted. *This function is called and all the file system structures are initialized*( the super block , inode blocks and the data blocks)
2. Create - Creation of a file

*Traverse to the path specified and return the file and parent directory inode number.*

Base cases:

-If file exists then -EEXIST is returned

-If the path is not proper(ex: parent directory not present) then -ENOENT is returned

Procedure:

a) Check the inode bitmaps to get the first free inode.Set that bitmap to 1.

b) Set the in pointer of the parent to this inode if it is the first child of that directory.

Else traverse to the last child of the directory and set its next pointer to this inode.

c) Set the metadata(all the inode details of the file) in the inode.

d)change the access and mod time of the parent directory.

e) Save the changes to the disk.

3) mkdir - Creation of a directory

*Traverse to the path specified and return the directory and parent directory inode number.*

Base cases:

-If directory exists then -EEXIST is returned

-If the path is not proper(ex: parent directory not present) then -ENOENT is returned

Procedure:

a) Check the inode bitmaps to get the first free inode.Set that bitmap to 1.

b) Set the in pointer of the parent to this inode if it is the first child of that directory. Else traverse to the last child of the directory and set its next pointer to this inode.

c) Set the metadata(all the inode details of the directory) in the inode.

d) Change the link count, access and mod time of the parent directory.

e) Save the changes to the disk.

4) readdir - Read contents of the directory.

*Traverse to the path specified.*

if the path is incorrect return -ENOENT

add .(current) and ..(parent) and traverse through all the children and add

all to the buffer.

change the access time of the directory.

5) Unlink - Delete a file.

*Traverse to the path specified and return the file and parent directory inode*

*number.*

Base cases:

-If file does not exist then -ENOENT is returned

-If the path is not proper(ex: parent directory not present) then -ENOENT is returned

Check if the number of links is equal to 1 and if not return .

Set the respective inode bitmaps and data block bitmaps to 0

Change the modify and status change time of the parent directory.

Save unsaved changes to the disk.

6) rmdir - Delete a directory.

*Traverse to the path specified and return the file and parent directory inode*

*number.*

Base cases:

-If directory does not exist then -ENOENT is returned

-If the path is not proper(ex: parent directory not present) then -ENOENT is returned

Check if the directory is empty(by checking the link count or the in pointer)

Set the respective inode bitmap of the directory to 0

Change the modify and status change time of the parent directory.

Save unsaved changes to the disk.

7) getattr - get the statistics/metadata of the file.

This the foremost functions to be implemented.

*Traverse to the path specified and return the file/directory and parent*

*directory inode number.*

Base cases:

-If directory does not exist then -ENOENT is returned

-If the path is not proper(ex: parent directory not present) then -ENOENT is returned

The details of the file or the directory are returned as the stat structure.

**Phase 2:**

System calls implemented:

1) Open - opens a file

*Traverse to the path specified and return the file and parent directory inode*

*number.*

Base cases:

-If file does not exist then -ENOENT is returned

-If the path is not proper(ex: parent directory not present) then -ENOENT is returned

open the file and increment the file descriptor count.

change the access time of the file.

2) Write - writing data into the file.

*Traverse to the path specified and return the file and parent directory inode*

*number.*

Base cases:

-If file does not exist then -ENOENT is returned

-If the path is not proper(ex: parent directory not present) then -ENOENT is returned

- If the write permissions are not set the return -EACCES

a) If no data block is allocated to the file, get the first free data block number from the bitmap

and allot that to this file.

b)If a block is already allocated, then continue write on the same block.

c) If more blocks are needed then a and b are repeated.

The contents are written based on the offset and the size of the buffer.

The modify time and status change time are modified.

the number of bytes written are returned.

Save the changes to the disk.

**Small files:**

As the block is only 512 bytes, internal fragmentation is comparatively lesser when compared to a block size of 4096 bytes. We used this implementation as,if multiple files are stored on the same block, then when a file is deleted , we have keep track of these small holes in the memory which is not very efficient as pointers to every hole must be stored

**Large files:**

We are allotting extra blocks when the file’s size increases.

3) Read - read the contents of the file;

*Traverse to the path specified and return the file and parent directory inode*

*number.*

Base cases:

-If file does not exist then -ENOENT is returned

-If the path is not proper(ex: parent directory not present) then -ENOENT is returned

- If the read permissions are not set the return -EACCES

Read the contents of the file from the offset and store it in the buffer.

If it is a large file then the blocks are traversed using next and previous

pointers.

change access time of the file.

return the number of bytes read.

4) utime - change the modify and status change times of a file

5) chmod- Change permissions of a file/directory

Traverse to the path specified and return the file/directory and parent

directory inode number.

Base cases:

-If file does not exist then -ENOENT is returned

Change the permissions as specified in mode argument.

change the status change time of the file/directory

save the changes to the disk

6) Rename - Change the file/directory name.

7) Destroy - this function is called when the file system is unmounted.

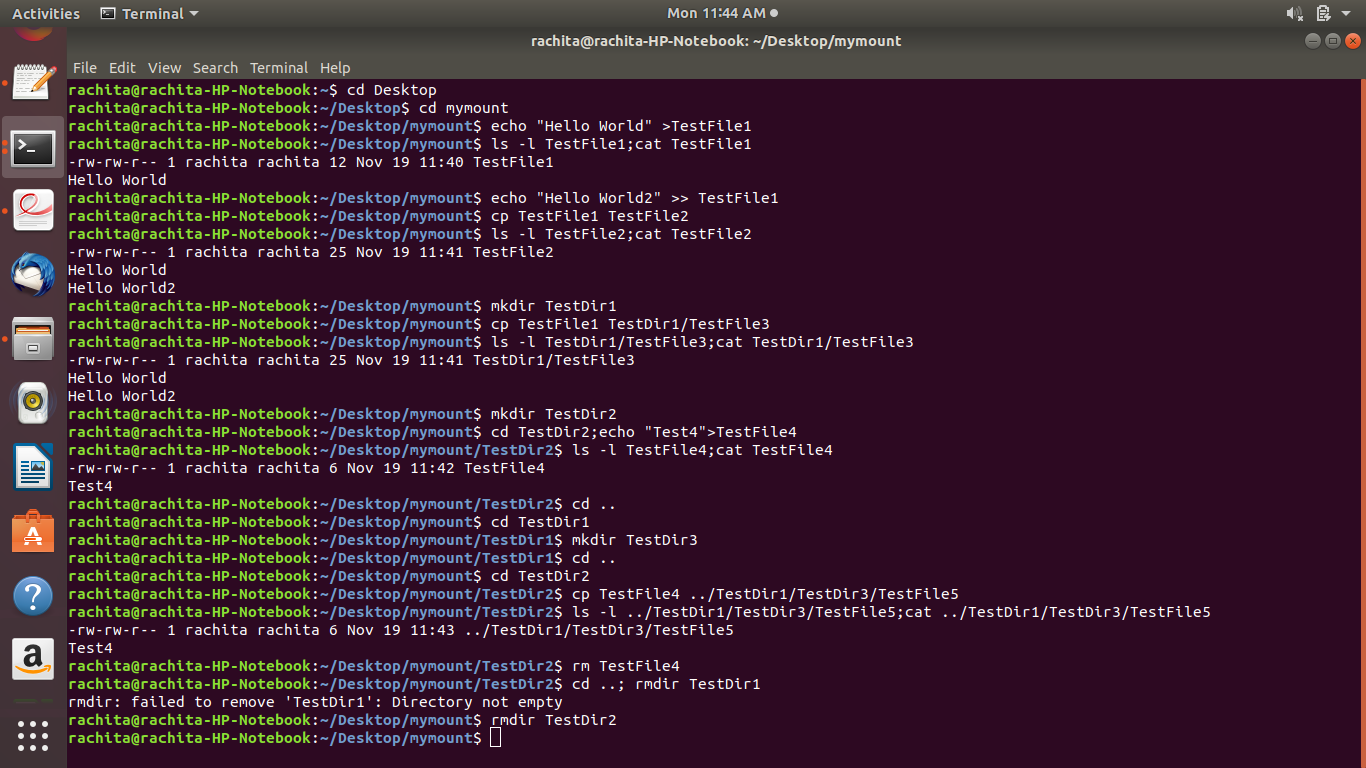
**Phase 3:**

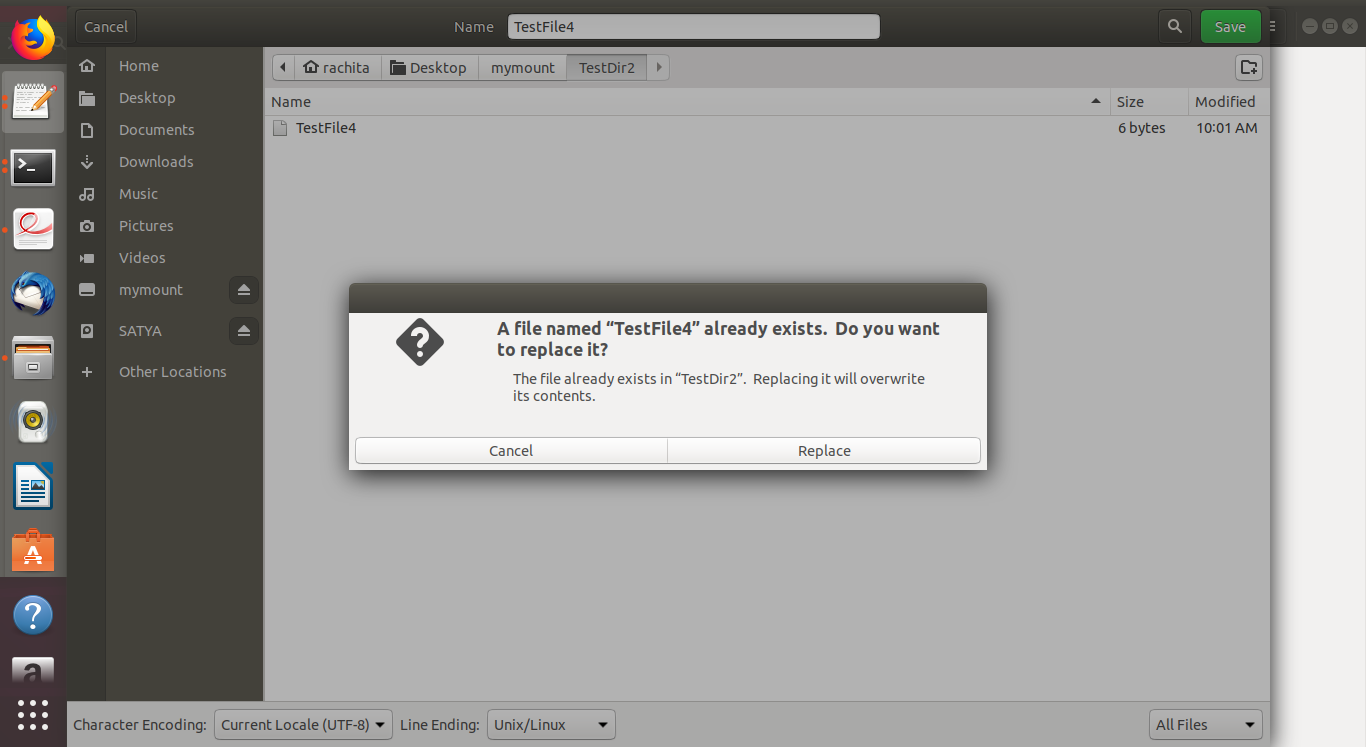
Persistence:

* We achieved persistence by creating a binary file outside the filesystem and everytime we mount the file system , it checks if this file exists and if it does, then it reads the contents of the file and loads it into the structures.Any changes in the filesystem are updated to this file.
* This file contains the entire file system details (including super block, inode blocks and data blocks).
* When the filesystem is unmounted or when the system is rebooted the files and directories are not deleted nor are recreated. They are just loaded back from this binary file.

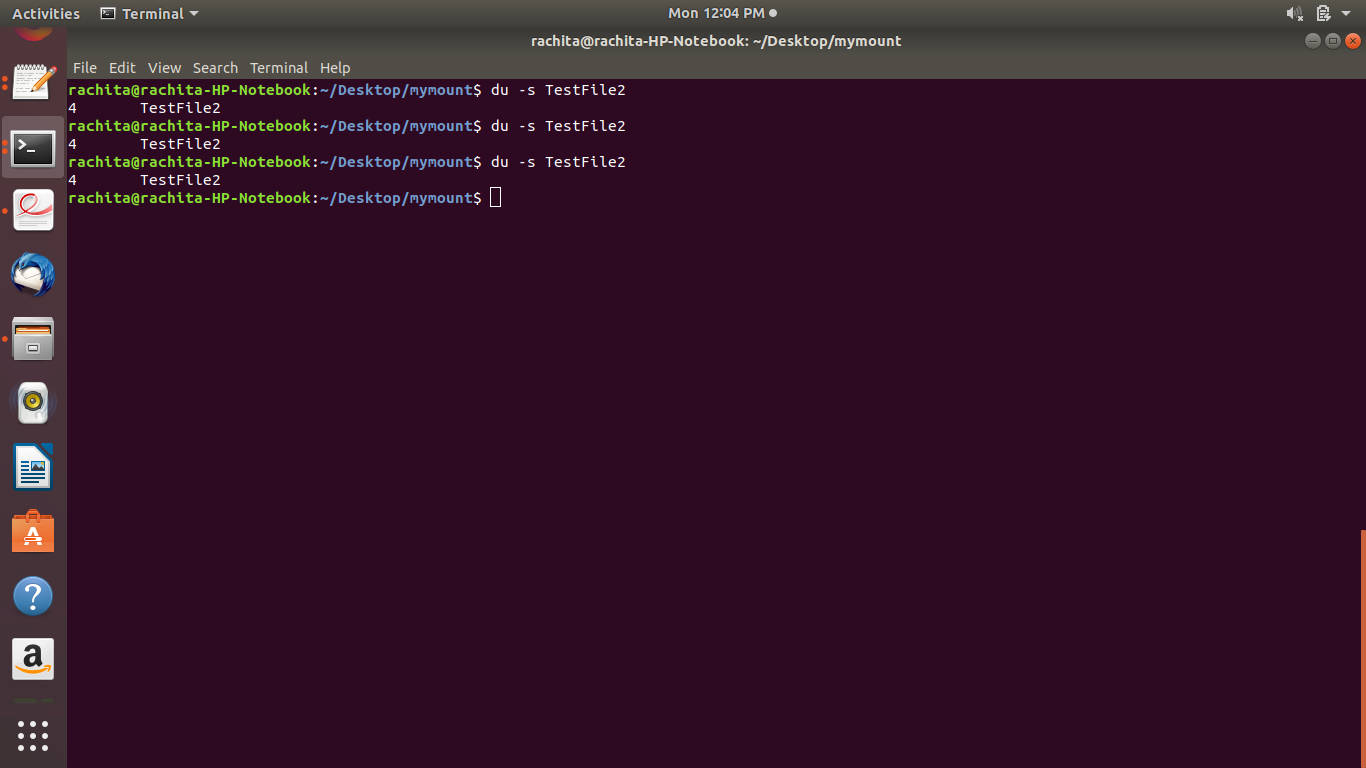
TEST CASES SCREENSHOTS

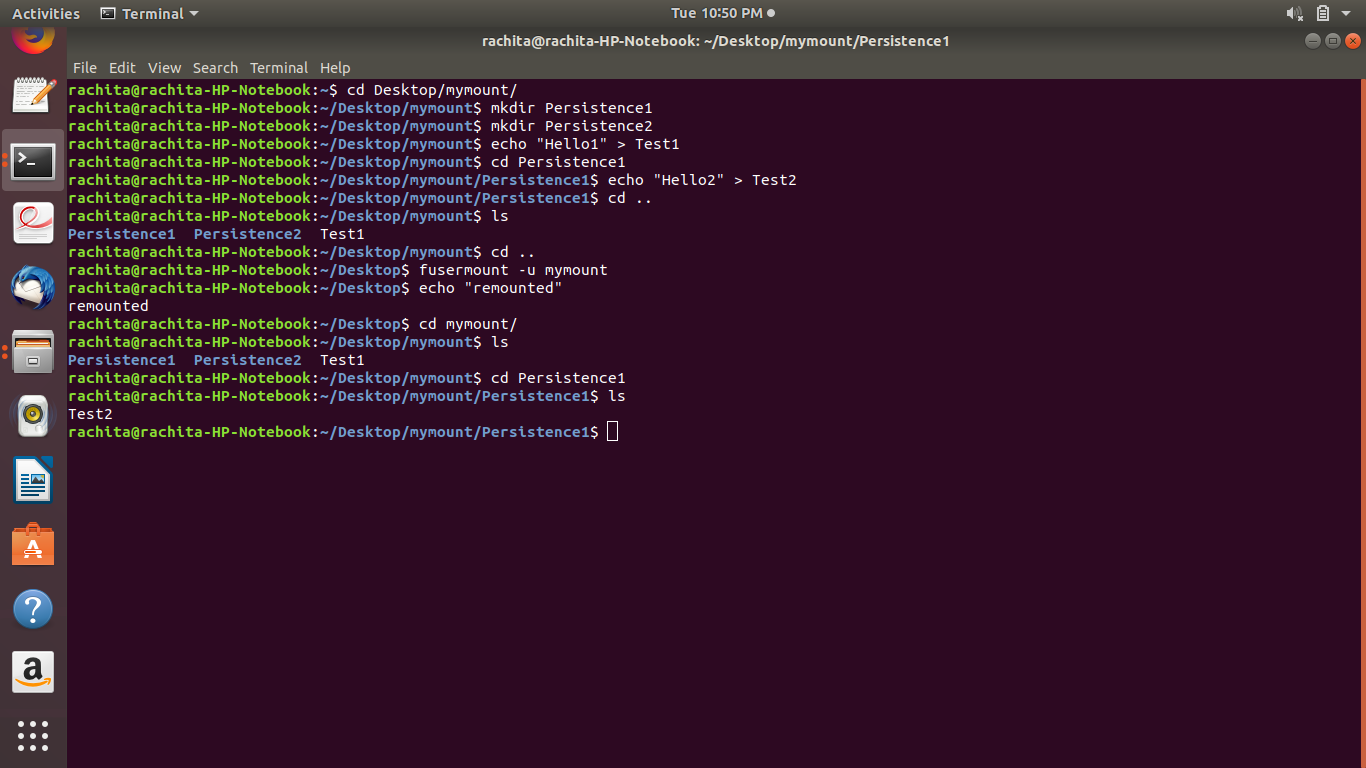
**Phase1:**

****

****

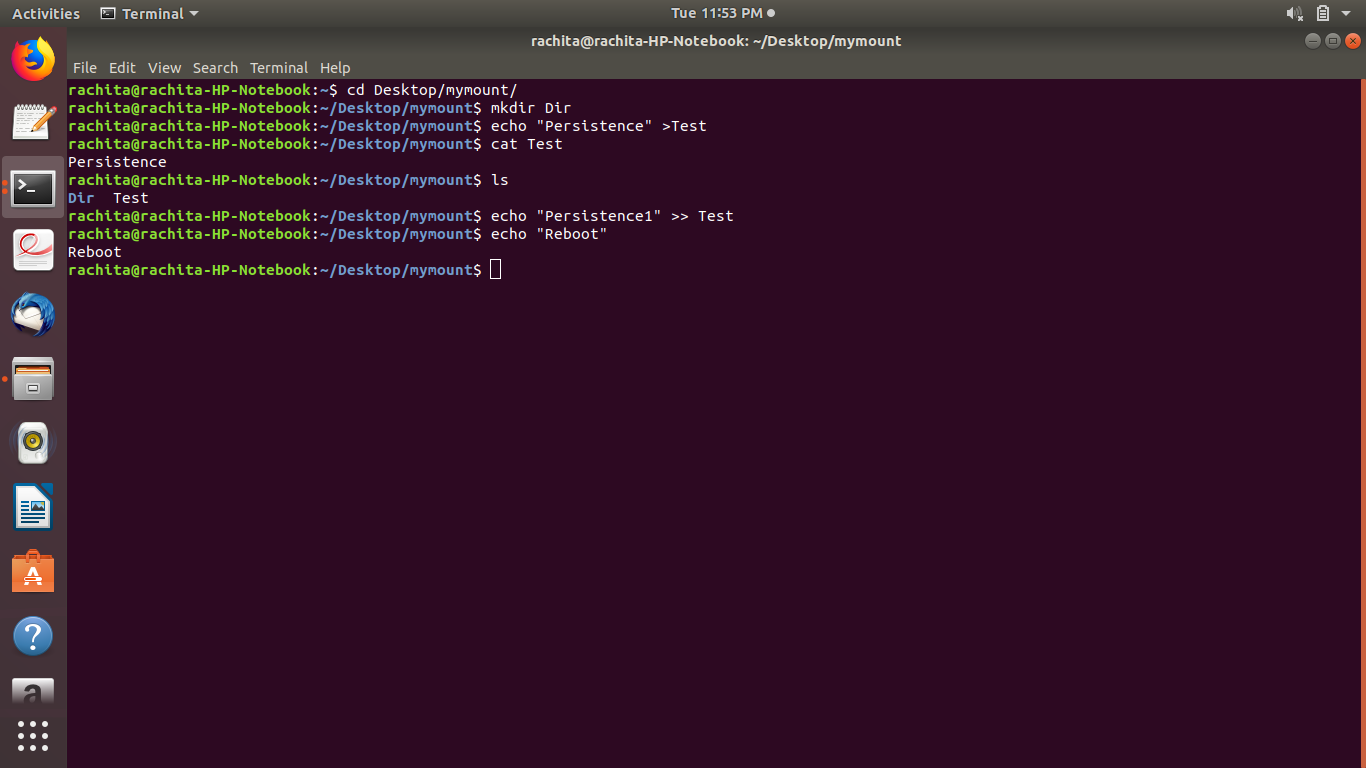
**Phase2:**

****

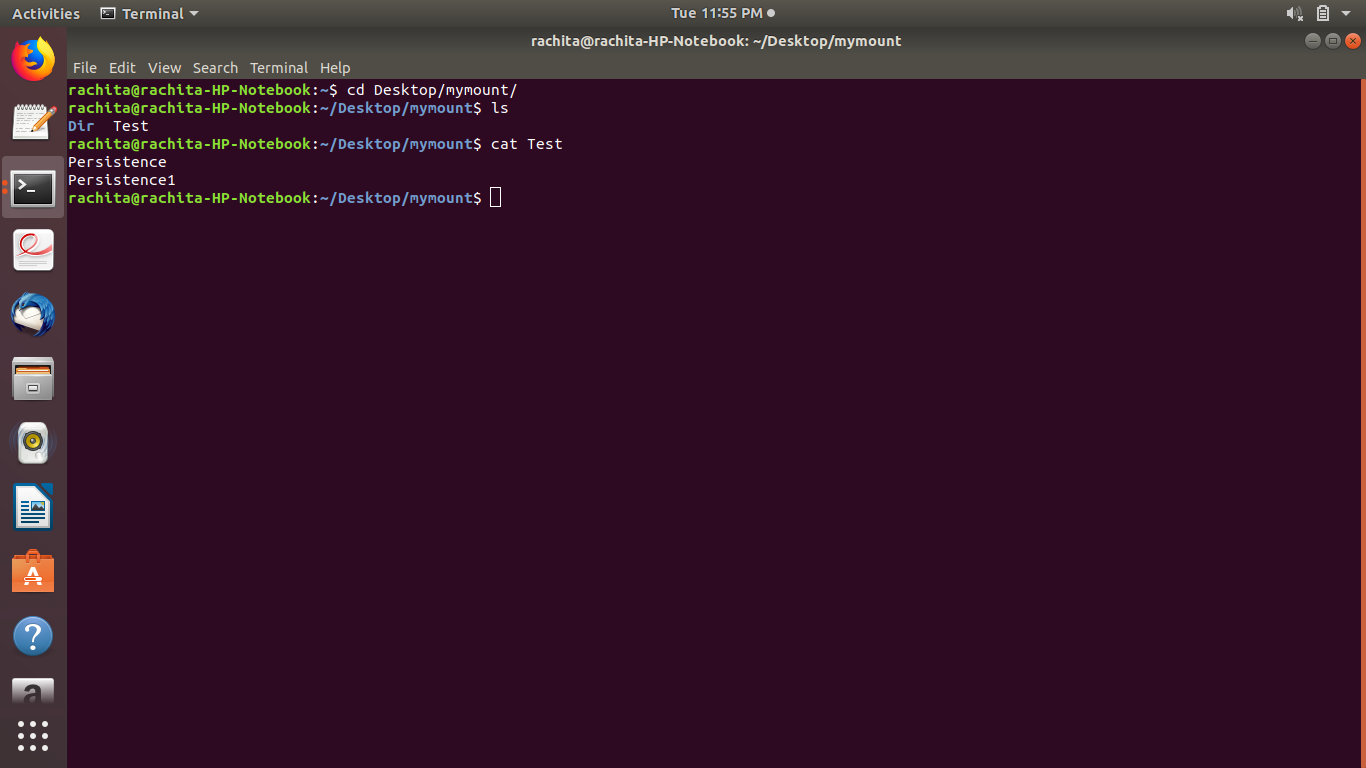
****

**Phase3:**

Before reboot -



After reboot -



**REFERENCES**

1. The following website was very useful in order to write our system calls

<https://www.cs.hmc.edu/~geoff/classes/hmc.cs135.201109/homework/fuse/fuse_doc.html>

2) Our lab instructors guided us through the project and helped us think about how we can achieve internal and external fragmentation and also the editing of a file using an editor like gedit.