**OS Lab Mini-Project**

**Group -**

**Siddharth Kailasam – 01FB16ECS153**

**Gautam Srinidhi – 01FB16ECS140**

**Arjun Rajesh – 01FB16ECS148**

**Jashwanth K – 01FB16ECS145**

**Introduction**

A filesystem in user space built using FUSE, a software interface for Unix-like computer operating systems that lets non-privileged users create their own file systems without editing kernel code. This project was created as a part of the "Unix System Programming" course at PES University, 2018.

FUSE or File System in User Space module provides a "bridge" to the actual kernel interfaces by running file system code in user space. Creating a file system can be achieved by writing a [Virtual File System](https://en.wikipedia.org/wiki/Virtual_file_system) which is nothing but an abstraction layer above a more concrete file system to provide custom access to users.

**Operation**

static struct fuse\_operations file\_oper = {

.getattr = file\_getattr,

.readdir = file\_readdir,

.mkdir = file\_mkdir,

.open = file\_open,

.init =file\_init,

.read = file\_read,

.write = file\_write,

.create =file\_create,

.destroy =file\_destroy,

.rmdir =file\_rmdir,

.chmod =file\_chmod,

.truncate =file\_truncate,

.unlink =file\_unlink,

.rename =file\_rename,

.symlink =file\_symlink

};

**Objective**

The objective is to implement a filesystem using C. It was divided into 3 respective phases -

**Phase 1 -**

In this phase, the implementation details were decided.

**Phase 2 -**

In this phase, the filesystem functions were created. They were however not persistent.

ed.

**Phase 3 -**

In this phase, the filesystem was made persistent.

**Survey**

[https://www.ibm.com/developerworks/topics/linux%20fuse](https://www.ibm.com/developerworks/topics/linux fuse)

We searched the above link, as part of our literature survey to gain insights on how the file system FUSE works and to understand the various ways in which a file system can be implemented.

**Design requirements:**

**Implementation -**

The filesystem was designed to target the devices with low processing capacity and storage capacity devices such as Raspberry Pi (older models), Aurdino etc. Hence, each block was given 2KB in size, to make implementation storage more efficient than storage with 4KB. We had implemented this with 16 blocks (which implies that the total storage capacity was 32KB). We used a linked list structure to hold directories. However, this structure does not pose any sort of limitations on the number of files. Hence, a directory can have as many number of files as it wishes. However, there can never be a directory within another directory. We have a superblock which maintains the list of total number of blocks available, total number of free blocks and the list of free blocks. Each file stores the list of blocks it owns. A file may also own all 16 blocks of the memory. The length of the filename and directoryname is limited to 50 characters.

**Structures used -**

The following is the structures used in our file system -

struct fileinfo

{

//Contains data and info of the data in the file

int fileId;

off\_t offset;

size\_t size;

mode\_t mode;

blkcnt\_t blockcount;

blksize\_t block\_size;

int blockno[T\_BLOCKS];

int uid;

int gid;

};

struct file

{

//Holder of the file information, with a name and fd assigned to it

char fname[50];

struct fileinfo \*fil;

unsigned int fd\_count;

};

struct directry

{

//Directory structure, which can point to other

//directories in the root

char dname[50];

int dirId;

mode\_t mode;

struct file \*\*f;

int filecount;

struct directry \*next;

int uid;

int gid;

};

struct block

{

//Block structure to hold file data

int blockno;

int valid;

void \* data;

size\_t size;

size\_t current\_size;

};

struct superblock

{

//metadata about the blocks

int totalblocks;

int totalfreeb;

int freeblocknos[T\_BLOCKS];

};

static struct fuse\_operations file\_oper = {

//FUSE structure which contains implementation of all our functions

.getattr = file\_getattr,

.readdir = file\_readdir,

.mkdir = file\_mkdir,

.open = file\_open,

.init =file\_init,

.read = file\_read,

.write = file\_write,

.create =file\_create,

.destroy =file\_destroy,

.rmdir =file\_rmdir,

.chmod =file\_chmod,

.truncate =file\_truncate,

.unlink =file\_unlink,

.rename =file\_rename,

.symlink =file\_symlink

};

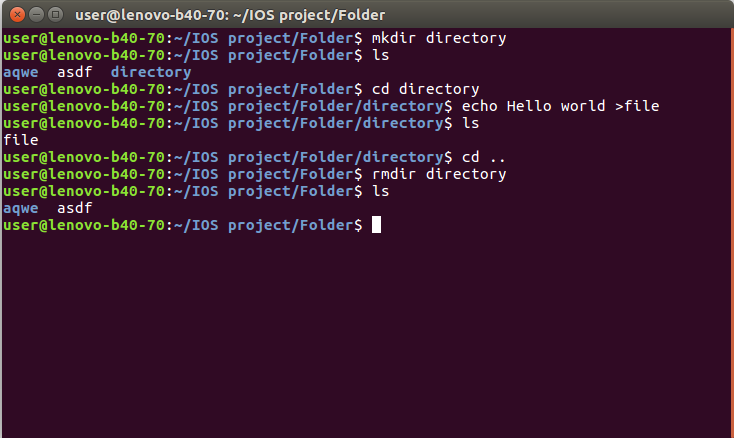
**Features-**

**1. Directory functions-**

**readdir(),mkdir(), rmdir()**

a. The root directory can have any number of directories. However, these directories cannot have further directories within them.

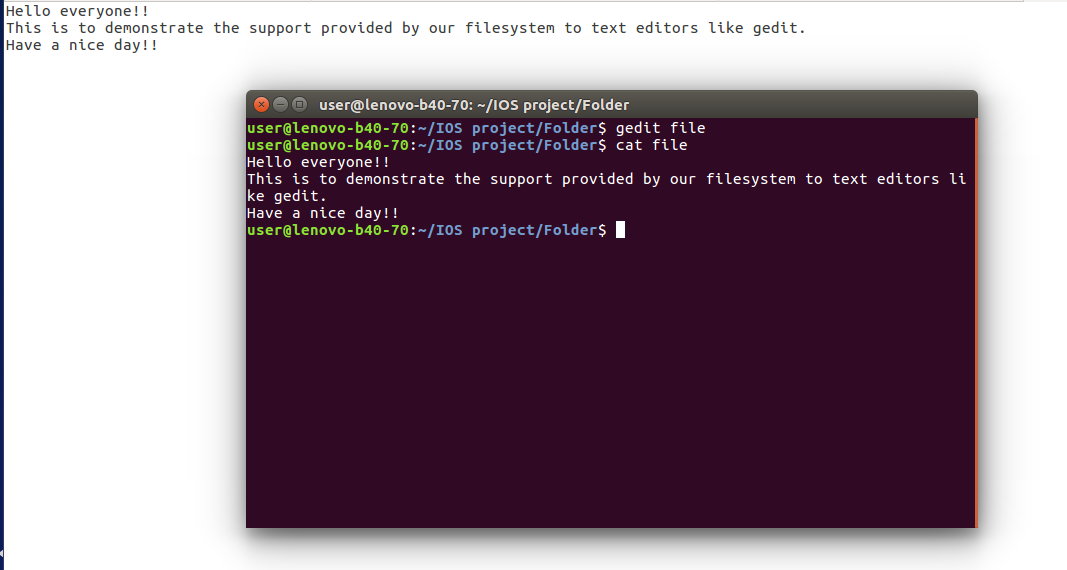
b. The file contents in a directory is removed along with the directory on deleting a directory with a file inside.



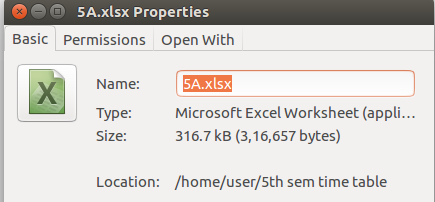
**2. File handling functions**

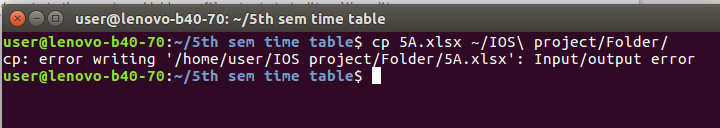
**open(),read(),write(),unlink(),truncate(), create()**

a. works with text editors like gedit, notepad, notepad++ etc when the particular file has write permissions.



b. makes sure that the writing operation does not make exceed memory limit of the filesystem (32KB with all files put together and memory limit can be changed by changing the header file if needed). The filesystem deletes the file which exceeded the memory limit and hence causes I/O error.



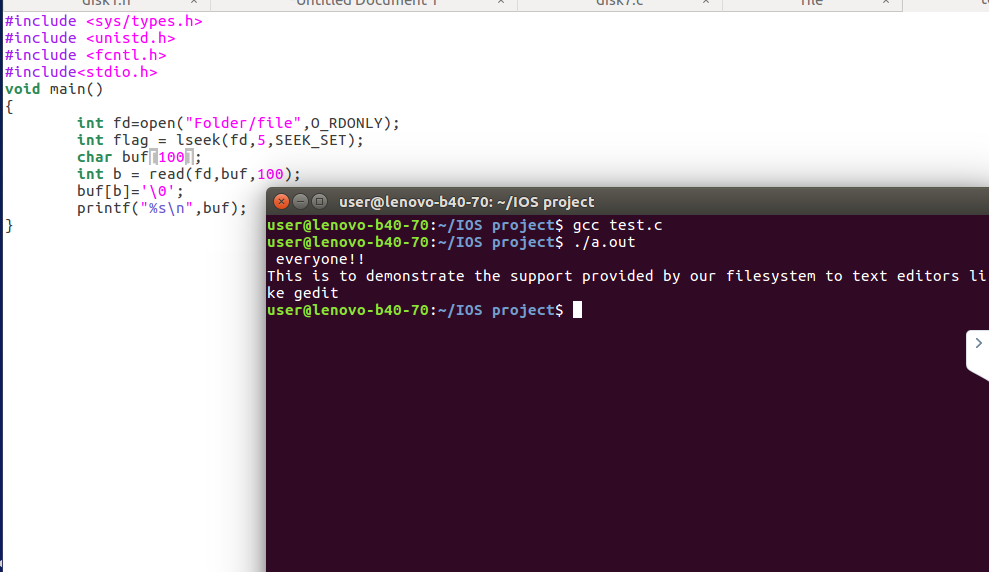


c. Allows deletion of files.

d. Opening and closing of files.

e. Allows lseek.

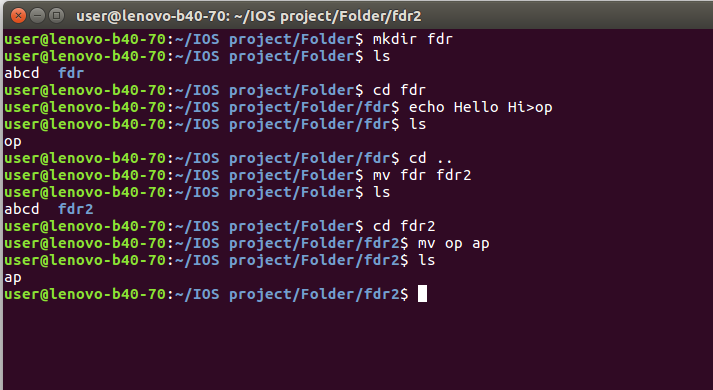
f. If the number of characters is beyond the size, it appends how many ever characters are there in the file.



Here, we seeked 5 characters and asked for 100 characters in file.txt. However, the file “file” doesn’t have 100 characters proceeding it, as we saw in making of file.txt. Hence, it reads only given number of characters.

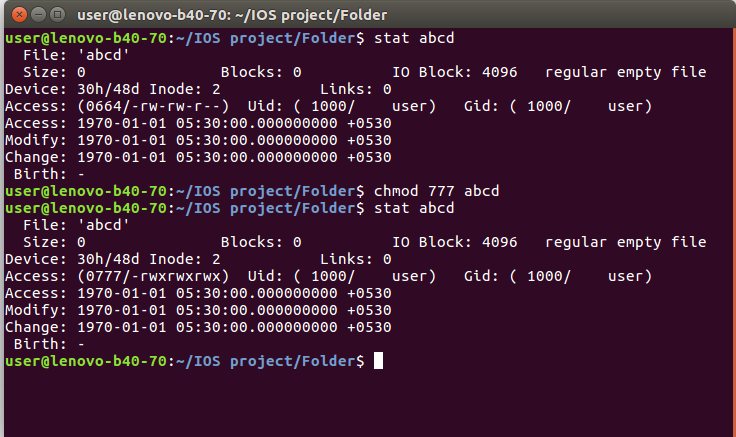
**3. Renaming files and directories -**

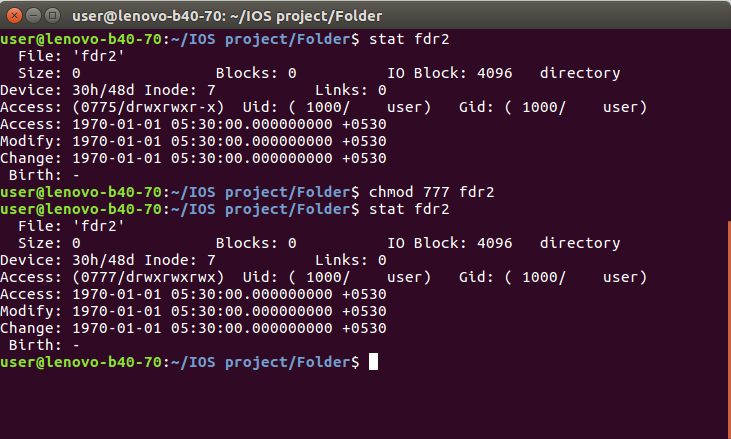
Any file or directory can be renamed. We can also move the file or directory using mv.



**4. Changing permissions of file/directory-**

Permissions of a file/directory can be changed using chmod.





**5. Persistence**

Each time mounting is called for, the filesystem checks for the file “file.txt” outside the mount directory. If this is empty, then it initializes the directory on its own. Before closing, it writes all blocks into the file in a binary form. So the next time, when opened, the filesystem reads the file “file.txt” and restores its blocks to as it was. Hence, persistence is implemented.

**6.Other functions implemented -**

**getattr()**

**Mounting a file**

**1. Open terminal in the directory which contains the code.**

**2. Type make.**

**3. Type ./ssfs -f <Folder to be mounted>**

**4. In case of abnormal termination you can type -**

**sudo umount <Folder to be unmounted>**

**To unmount the folder.**

**Areas of improvement desired -**

We would like to make our symlink function executable, so as to make this filesystem a complete one.

**Result -**

Hence we made a filesystem, which had most of our executable functions working in an almost perfect manner like a normal filesystem. This filesystem as already mentioned could be used by Aurdino, older models of Raspberry Pi etc to make the most of our filesystem.