AHMEDABAD UNIVERSITY

Global Education at Local Cost, Context and Ethos™

Operating System Project Report

Group No. 10
Date:17.11.2017

1. Title of the project:

File System Management using EXT approach

2. Team members (Student ID and Name):

Member 1:

Student ID: 201501008
Name: Ashna Jain

Member 2:

Student ID: 201501072
Name: Janvi Patel

3. Brief description:

- **“On a system, everything is a file; if something is not a file, it is a process”**. The file system is an important part of any operating system. After all, it’s where users keep their stuff. The organization of the **file system plays an important role** in helping the user find files.

- A Linux system, just like UNIX, makes no difference between a file and a directory, since a directory is just a file containing names of other files. **Input and output devices**, and generally all devices, are considered to be **“files”** according to the system.

- We have used this similar approach in our project. We have also **considered directories as files** and implemented our own file system.

- Our file system has the capability to **multitask and support multiuser** which makes it efficient to maintain security.

- Our project is based on the **I-node concept**. Each file or directory is uniquely identified by its name, the directory in which it resides and a unique identifier, typically called an I-node. We have given I-node number 0 to our root directory and I-node 1 is reserved for system files.

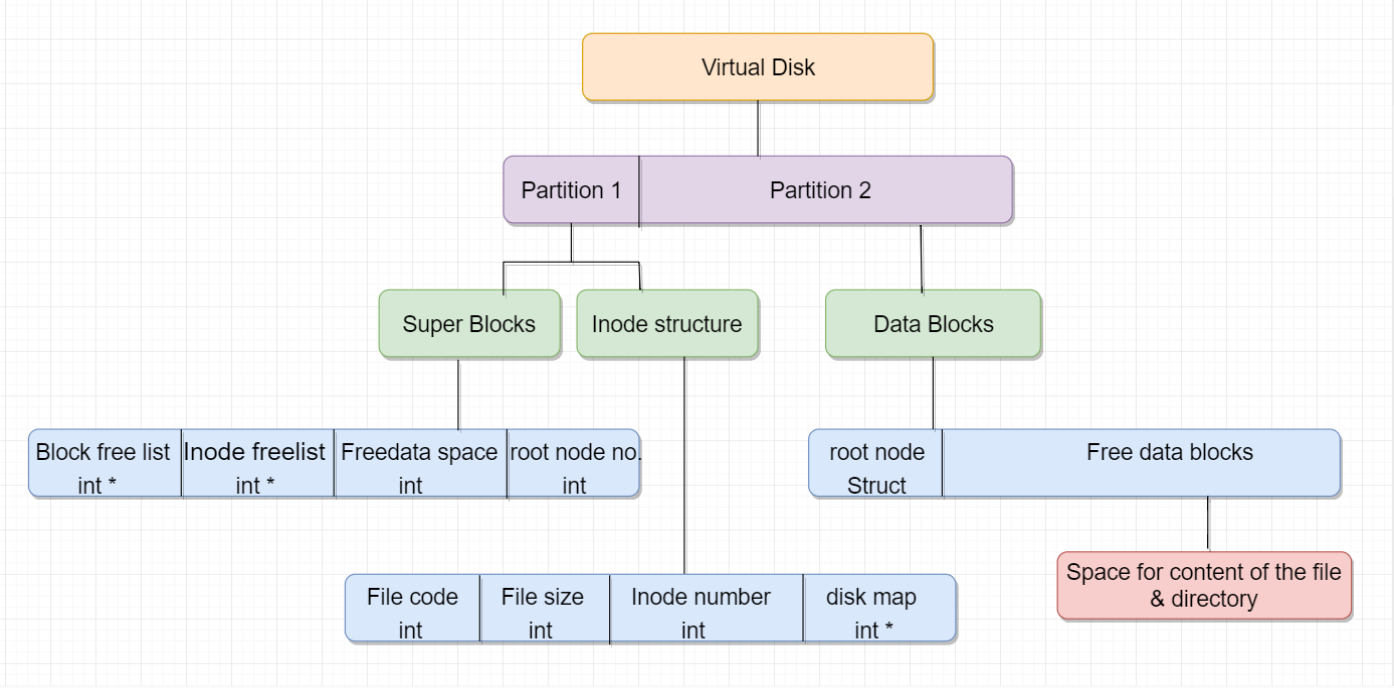
- The basic objective of a good filing system is to be able to find the record you need quickly and economically, regardless of its format. The goal of a good filing system is to provide **quick access to information**. By using efficient data structures we have tried to make the system run as quick as possible.

- Features which our file system supports are:
    - mkdir (Creating a new directory)
    - touch (Creating an empty file)
    - VI (Inserting data into a file)
    - cat (Display the file contents)
    - update (update the content of the file)
    - pwd (Shows the path to the current directory)
    - ls (List the directories & files)
    - copy (Copying the contents of one file to other file)
    - file (shows the type of the file)
    - rename (Renaming a file)
    - remove (Delete a file)
    - Implementing the tree feature of windows (tree command).

- Detailed features of our project:
    - Initially when we compile our code a virtual disk named as “disk file” is created if the disk file is not present, otherwise the existing disk file is modified. The disk file is allocated 1MB (1000 blocks) space.
    - Our disk file is divided into 2 different proportions – 5% of the Fisk is allocated to Super Block and the I-node struct list and 95% of the Fisk is allocated to data blocks.
    - Our project supports multi user by allowing many users to log in to their own file system.
    - Just like UNIX File system it can create and manage files and directories by different command and I-node approach.
    - This file system works as a virtual file system by storing the data in the same manner as it is stored in a hard disk.
    - A very important feature that our project implements is the graceful exit of the system.

4. Architecture/model/diagrams:

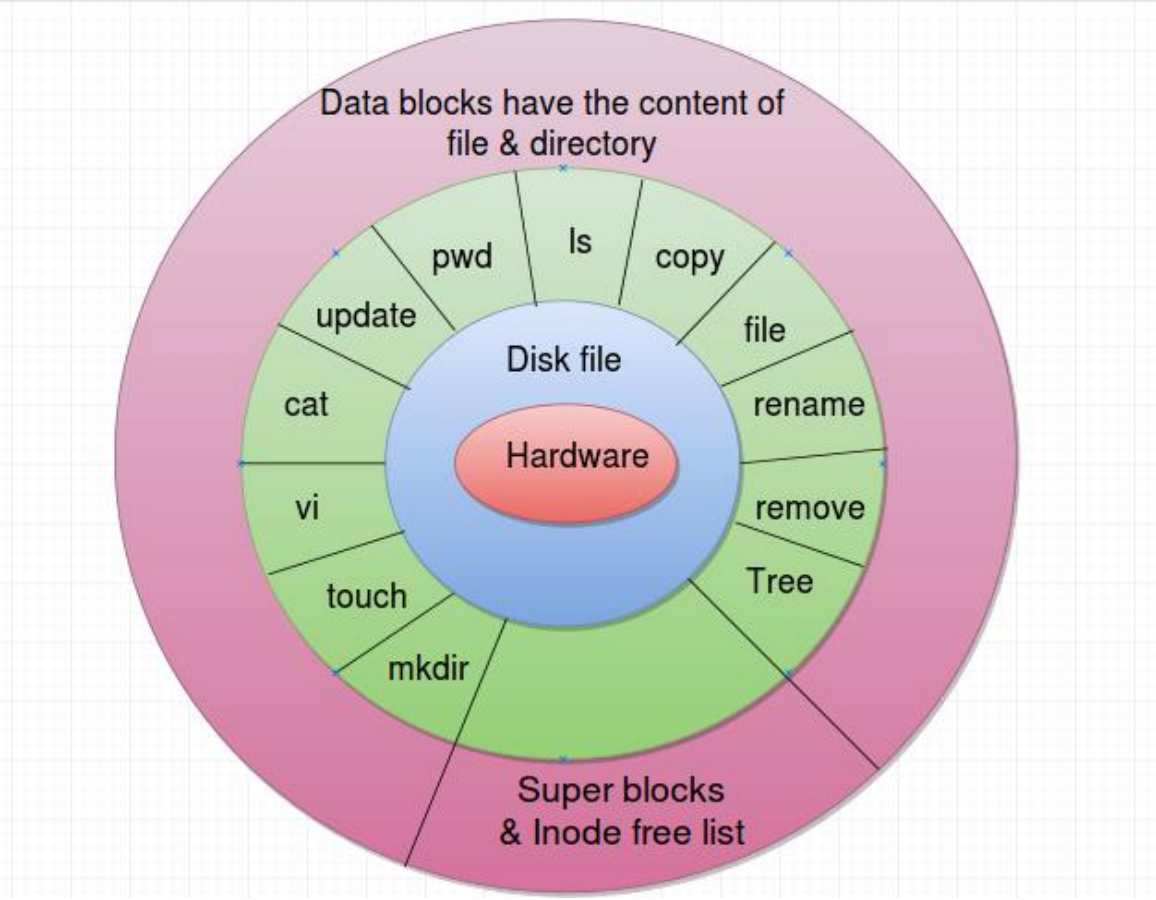
1. Architecture of the project:



- Interpretation of the above figure:

  The virtual disk is divided in two partition. The first partition contains the superblock and the I-node structure which constitute 5% of the total space. The second partition contains the data blocks which constitute 95% of the total space allocated. The structure of superblock constitute block free list, I-node free list, free data space and root node number. The structure of I-node constitute file code, file size, I-node number and disk map. The data blocks contain root node struct and free data blocks where contents of file and directories are stored.

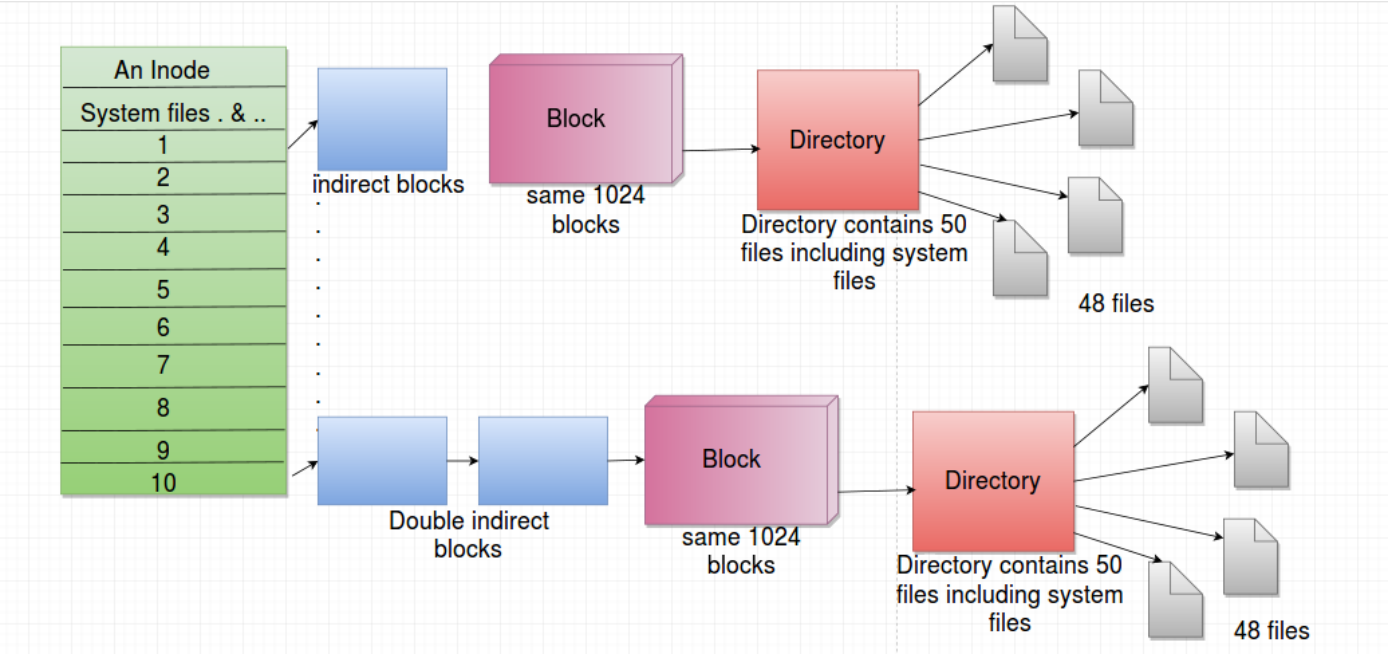
2. <u>Interpretation at kernel level:</u>



- Interpretation of the above figure:

  The above figure shows the architecture of the project and logical analysis of the features implemented. The disk file created works at logical level in the same way as kernel does. The various commands (tree, remove, rename etc.) logically work like the kernel level file management system. After the various command then there are data block for storing and managing files.

3. Architecture showing the storage mechanism:



- Interpretation of the above figure:

  An I-node contains many pointers which store the data of the file. When the allocated data block becomes full then indirect blocks are created and those blocks are pointed by direct blocks. A directory is allocated a block and a directory can contain at most 48 files.

5. Technical specifications/details:

➢ **Technical specifications of the structure:**

1. A hypothetical disk size of 1 MB (1000 blocks) is created. This facilitated the number of blocks that Superblock and I-node struct list would span on the Fisk, and helps to determine the beginning of data blocks.

2. The disk is divided into 2 different proportions – 5% of the Fisk is allocated to Super Block and the I-node struct list and 95% of the Fisk is allocated to data blocks.

   i. Hypothetical Fisk Size = 1000 blocks (1 MB)
   ii. Super Block + I-node struct list = 5% = 50 blocks
   iii. Data Blocks = 95% = 950 blocks

3. We created a structure for Super Block, which contains the below-mentioned components and calculated the Super Block size. Super Block contains a list of integers to represent information. An integer is 4-bytes; hence, multiplication by four. I-node number for root is stored in Super Block.

- Calculating Super Block size (total = 5808 bytes):
  - Block free list = 950 * 4 (integer is 4-bytes) = 3800 bytes
  - I-node free list = 500 * 4 = 2000 bytes
  - Free data space = 4 bytes
  - Root I-node # = 4 bytes

4. Created a struct for I-node, which contains the below-mentioned components and calculated the individual I-node size. Disk map is an array of 15 integers, each array index points to a data block on Fisk, which is used to store contents of a file/directory. File code has a number representation for a file or a directory.

- Calculating I-node size:
  - File code = 4 bytes
  - File size = 4 bytes
  - I-node number = 4 bytes
  - Disk map = 15 * 4 = 60 bytes
  - Total size of each I-node = 72 bytes
  - Total number of I-nodes = 5% of Fisk * 10 = 5% * 1000 * 10 = 500

5. Created a directory struct – the directory struct has a table that contains a list of I-node numbers and the corresponding file/directory names. Filename can be a maximum of 9 characters (10th character being null). Maximum number of files per directory is set to 50, which spans one data block on the Fisk.

- Calculating Directory size:
  - File count = 4 bytes
  - 50 I-node numbers = 50 * 4 = 200 bytes
  - 50 filenames of 10 characters each = 50 * 10 = 500 bytes

➢ Fisk structure:

	Superblock -free data block list -free I-node list -free data space -root I-node #	I-node Blocks -file code -file size -I-node number -disk map	Data Blocks
Block number	0 to 5	6 to 49	50 to 1000

After conceptualizing the Fisk structure, the next step involved initializing the a) root directory, b) I-node Blocks, and c) Superblock. The function initialize () is used to call other functions such as initialize_super_blocks(), initialize_inode_blocks() and initialize_root_dir() in order to initalize SuperBlock, Inode Block, and root directory respectively.

a) Initializing the Root Directory:
Root directory is created by default when initialize() is called, and given an I-node number of 0. It is the top-level directory. According to the design, it can contain 50 files or subdirectories (spans 1 data block). It contains a table of I-node numbers and their corresponding file/directory names. The table is stored on the data block on Fisk. Since hypothetical Fisk size is 1000 blocks, the offset is calculated as (0.05 *1000 = 50), so the table will be written to the 50th data block on the Fisk.

b) Initializing the I-node Block:
The I-node struct contains information such as I-node number, file code(file or directory), file size and disk map. By default, file size is set to 1024 bytes for all I-nodes, and file code is set to directory. For I-node number 0 (hypothetical fisk size 1000 blocks), which is the I-node for the root, disk map [0] = 50. This means, that the table that the root directory contains with I-node numbers and file/directory names will be written to data block 50 on Fisk. Disk map [1 to 14] is set to 0, since by design, directories can only contain 50 files, which fits in one data block (disk map [0]). When I-node are written to Fisk, the offset is calculated as the size of the Superblock, since the I-node blocks begin directly after the Superblock.

c) Initializing the Superblock:
Since hypothetical Fisk size is 1000 blocks, the free data block list is an array of 1000 integers, each representing whether the data block number on the Fisk is free to put the contents of the file/directory. The I-node free list is an array of 500 integers, as there are 500 I-nodes (given the Fisk size is 1000 blocks). The first data block is taken by the root directory, so data_block_free_list [0] = 1337. If a data block is taken, the free list indicates that by switching the bit to 1, but 1337 was chosen to check that Fisk is only initialized once. The rest of the bits in data block free list are set to 0 (free). For the I-node free list, the index 0 is set to bit 1 (occupied), since I-node number 0 belongs to the root directory. The rest of the I-nodes have the bits set to 0 (free). The Superblock starts at block 0 on Fisk and spans 6 blocks (Fisk size= 1000 blocks).

➢ Technical specifications of the code:

❖ Description of the structures created:
- **typedef struct superblock**-This structure describes the internal structure of a superblock.
- **typedef struct directory**-This structure describes the internal structure of a directory.
- **typedef struct I-node**-This structure describes the internal structure of a I-node.

- Functions in disk file:

- **openDisk () –** This function uses the C library open() to open the Fisk file, which is to emulate a disk. If the file does not exist, the function creates a file using lseek() to move the file pointer to the end of the file and uses the C library write() to write a null character. This action creates a blank file with a null character written at the end of the file so that a specific size of Fisk may be created.
- **closeDisk () –** This function uses the C library close() to close the file (Fisk).
- **readBlock () –** This function uses lseek() to move the file pointer to a given data block on Fisk and uses the C library read() to read a block into the buffer, which is situated in the memory.
- **writeBlock() –** This function uses lseek() to move the file pointer to a given data block on Fisk and uses the C library write() to write the contents of the buffer into a data block on Fisk.
- **syncDisk() –** This function uses fsync() to force any outstanding writes from the buffer into a data block on Fisk.

- Functions in fileSystem file:

- **void tree()**- Prints the directory and files in hierarchical manner similar to tree structure.
- **void print_curr_dir()**- Prints the contents of current directory.
- **int effs_initialize(int numblocks)**- initialize function called by shell.
- **static void initialize_super_blocks()**- initializes superblocks and writes its information to fisk.
- **static void initialize_inode_blocks()**- initializes all inode blocks, setting all to default except root.
- **static void initialize_root_dir()**-initializes the root directory writing out to its data blocks
- **static void read_super_blocks()**-reads current superblock information from fisk into superblock struct.
- **static void write_super_blocks()**-writes all super block information to fisk.
- **static int get_free_inode()**-grabs the next free inode from the inode freelist, setting it to occupied.
- **static void read_inode(int inode_index)**-reads inode specified by inode_index into the node struct.
- **static int add_inode(int inode_index, int filecode, int filesize)**-adds a free new inode for a newly created file and reserves data blocks based on filesize and sets them to the new inode's diskmap.
- **static int remove_inode(int inode_index)**-zeroes out an inode's diskmap and sets the inode specified by inode_index to unused default.
- **static int get_free_data_block()**-grabs a free data block from the data block free list and sets it to occupied.
- **void file_to_dir(char *file,char *dir,int inode_file)**-Function to map a file to a specific directory.

- **int create_file(int filesize, char* filename,char* oldDir)** -creates a new file, setting up its I-node and reserving data blocks for it.
- **int create_dir(char* dirname)-**creates a new directory, setting up its I-node and reserving data blocks for it.
- **int effs_create(char* name, int flag, int filesize,char* oldDir)**-general create function that calls either create_file() or create_dir() depending on flag.
- **void change_dir(char* dirname)**-changes current directory to a directory in current directory's (I-node #, filename) tuples list
- **int get_curr_dir(char* filename,int inode_file)-** Gets the current working directory and prints it to stdout.
- **int fs_delete(char* filename)-**Deletes file specified by filename. Nulls out the disk map of I-node of file to be deleted and updates file's parent directory (I-node#, fname)tuple chart.
- **int fs_open(char* filename)-**opens a file specified by filename for writing.
- **int fs_ls()**-Displays the contents of current directory.
- **int fs_write(int inode_index, void *file_content, int filesize)**-Writes content from file_content to the data blocks in the disk map of I-node_index. The amount to write, file size, cannot be greater than that file's size.
- **int fs_read(char* filename, void *file_content, int filesize)**-Reads content from the data blocks in the disk map of I-node_index to data_buffer and prints it. The amount to read, filesize, cannot be greater than that file's size.
- **int fs_sync(int fildes)**-syncs open file descriptor fildes to disk.
- **int fs_close(int fildes)**-syncs file descriptor fildes and closes it (done in shell).

Functions in main file:

- **int create_fd(char* test, char* fname, int flag, int filesize,char* oldDir)-**creates a file or directory by calling effs_create function of effs file.
- **int open_fd(char* test, char* fname)**-opens the file with filename as fname.
- **int remove_fd(char* test, char* fname)**-Function to remove a file whose name is the string stored in fname string.

6. Algorithms/Flow charts, list of programs and test data sets:

Algorithms:

1: Start
2: Read all the variables.
3: Read the username and password.

4: for i=0 to 2 do
        If user[i] = username and pass[i]=password then
                flag=1
5: If flag=0 then
        Write “Invalid Username or password.”
        Return.
6: Allocate space to buffer,c_buffer and i_buffer.
7: Set size = FISK_SIZE*1024.
8: If effs_initialize(size) =0
        Write “Test initialize disk of size 10000 blocks Passed”.
  else
        Write “Test initialize disk of size 10000 blocks Failed”.
9: While 1 do the following steps.
10: Write 1. mkdir (Create New Directory)
        2. touch (Create New file)
        3. vi (Enter the data into file)
        4. cat (Display the data from file)
        5. Update (Update the content of file)
        6. pwd (Display the current path)
        7. ls (List the directories & files)
        8. Copy (Create copy of the file)
        9. file (Shows the type of file)
        10. rename (Rename the file & directory)
        11. remove (Delete the file)
        12. Tree (Display the structure of the disk)
        0. Exit.
11: Get the value of Choice from user.

12: If Choice =1 then
    Write “Format: mkdir 'directory name”.
        Get the value of result and dirName.
        If result=mkdir then
                If effs_create(dirName, DIRECTORY, 0, dirName) = -1
                        Write “Please enter different name of directory"
                        break
                else
                        Write “Directory created successfully created.”
                        Write “Task Completed!!!"
                        Increment errors by 1.
                Else
                        Write "Invalid Format!!! Please try again”.

13: If Choice =2 then
    Write “Format: mkfile 'File name”

```
Get the value of result and dirName.
If result=mkdir then
        Write “Enter the directory in which you want to create file”.
        Get the value of oldDir.
        Set newName_buffer= newName.
        if call effs_create(newName_buffer, FILE, 0,oldDir) = -1
                Write “File already exists”.
                break
        else
                Set size = 0.
                Write “File created successfully”
else
        Write Invalid Forma!!! Please try again.

14: If Choice=3 then
        Write “Format: vi 'File name' “
        Get the value of result and newName.
          If result=vi then
                Set newName_buffer=newName
                Set size = 11;
                Write "ENTER THE DATA:"
                for i = 0 to size:
                    Read the value of c.
                    Set c_buffer[i] = c
           Open newName_buffer file.
           if fmain > 0 then:
                    Call fs_write(fmain, buffer, size)

15: If choice=4
        Write “Format: cat 'File name’”
        Get the value of result and newName
        If result=cat then
                Set newName_buffer = newName
                Set fmain = open_fd("test", newName_buffer)
                Set size=11.
                Call fs_read(newName_buffer, buffer, size)

16: If choice=5
        Write “Format: update 'File name'”
                Get the value of result and newName
                If result=update then
                        Read the value of oldDir and data.
                        Set fmain = open_fd("test",newName)
```

if(fmain>0) then

Call remove_fd(test, newName)

Read the data and store it in c_buffer.

Set newName_buffer= newName;

Call fs_create(newName_buffer, FILE, 0,oldDir) and then call fs_write(fmain, buffer, size).

17.If choice=6

Write “Format: pwd'”

Get the value of result.

If result=pwd then

Set fmain = open_fd("test",newName)

Call get_curr_dir(newName,fmain)

18.If choice=7

Write “Format: ls'”

If result=ls then

Get the value of result.

Call print_curr_dir().

19.If choice =8

Write ”Format: copy 'filename' 'Name of other file'”

Get the value of result,newName,newName1

if result= copy then

Read the value of oldDir

Set newName_buffer = newName

Set newName_buffer1 = newName1.

Read the contents from newName_buffer and store it in buffer.

Create newName_buffer1 file .

Set fmain = open_fd("test",newName1)

Write the contents of buffer in it by calling fs_write(fmain, buffer, size).

20.If Choice=9 then

Write “Enter the name of the file: “

Get the value of newName.

For i=0 to 10

If newName[i] = '.' then

Break.

Store the value after the . in store string.

Compare the value of store with each extension and print the respective statement.

21.If Choice=10 then

Write “Format: rename 'File name'”

Get the value of newName and result.

if result = rename

Get oldDir and newName1

Set newName_buffer = newName

Set newName_buffer1 = newName1

Set size=11

Read from newName_buffer and store it in buffer.

Remove the file newName.

Create newName_buffer1 file by calling

Write the contents of buffer in newName_buffer1.

22.If Choice=11 then

Write “Format: remove 'File name'”

Get the value of newName and result.

if result = remove

Remove the file by calling remove_fd(test, newName).

23..If Choice=12 then

Write “Format: Make Tree'”

Call the Tree() function.

Algorithm of the functions used in the main file:

- **fs_create function :** Function to create file and directory.

1.If flag = “FILE” then

Call create_file(filesize, name,oldDir).

2.if flag = “DIRECTORY”

Call create_dir(name).

- **Algorithm of create_file function:**

1.Check if file already exists.

2. If file exists then return -1 and exit.

3. If file size is negative then write “File name cannot be negative” and exit.

4. Check if the maximum file per directory has reached or not. If maximum number of files have reached then print “Maximum number of files reached!” and exit.

5. Allocate a inode to the file.

6. Add inode to inode table.

7. Map the file to the specific directory.

- **Algorithm of create_dir function:**

1. Check if file already exists.

2. If file exists then return -1 and exit.

3. Allocate a inode to the directory.
4. Add inode to inode table.
5. Reserve data blocks for the directory.

- **fs_write function:**
  1. Read I-node Meta data from specified file index.
  2. Check if the file content will fit into the available data space. If the file content doesn't fit then print error.
  3. Write file content to read I-node node's disk map.
  4. Write first 12 data blocks of current node (direct).
  5. If more data block are needed then use a single indirection, double indirection and so on.

- **fs_read function:**
  1. Check whether the file exists or not.
  2. If the file doesn't exist then print "File not found" and exit.
  3. Read inode Meta data from specified file index.
  4. Read file content node's disk map to data_buffer and print.
  5. Read first 12 data blocks of current node (direct).
  6. If single indirection used then read from pointed blocks.

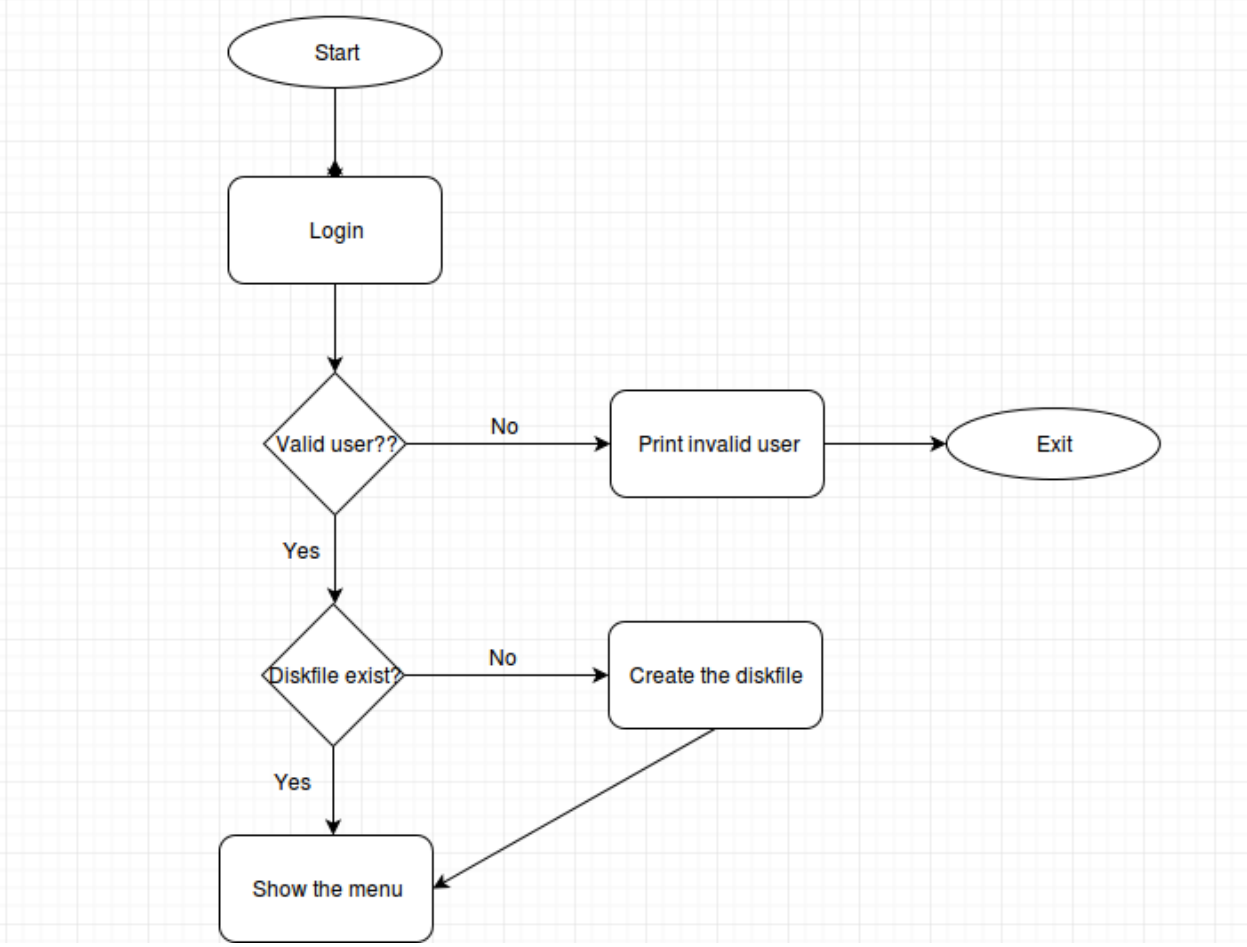
- **fs_delete function:**
  1. Check whether the file exists or not.
  2. If the file does'nt exist then print "File not found" and exit.
  3. Find inode number of the file.
  4. Load and remove the inode number of the file.
  5. Update the current directory's table.

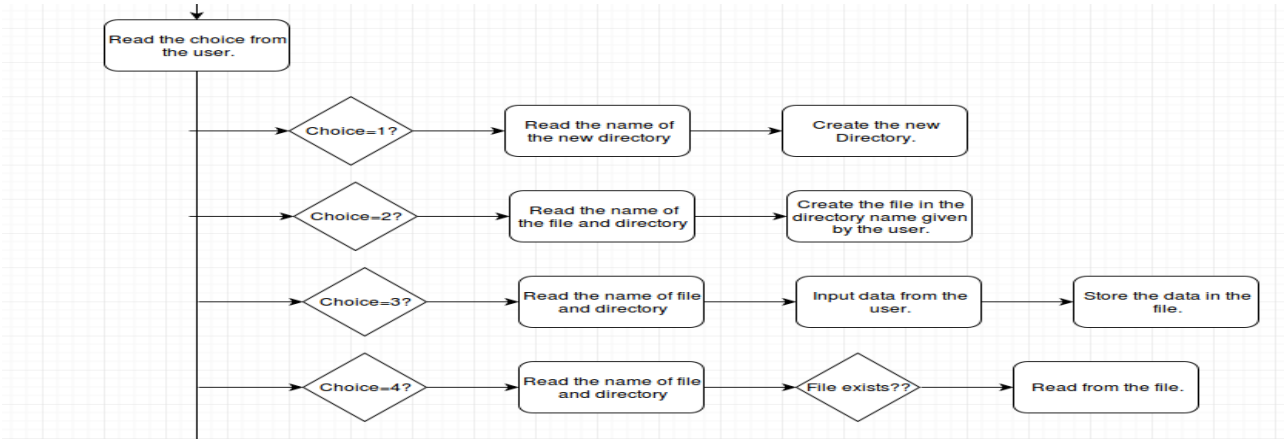
- **print_curr_dir function:**
  1. Search for curr dir. starting at inode 2 because of . and ..
  2. Run a loop from i=2 to curr_dir->inode_num[i]+2
     Print curr_dir->filename[i].

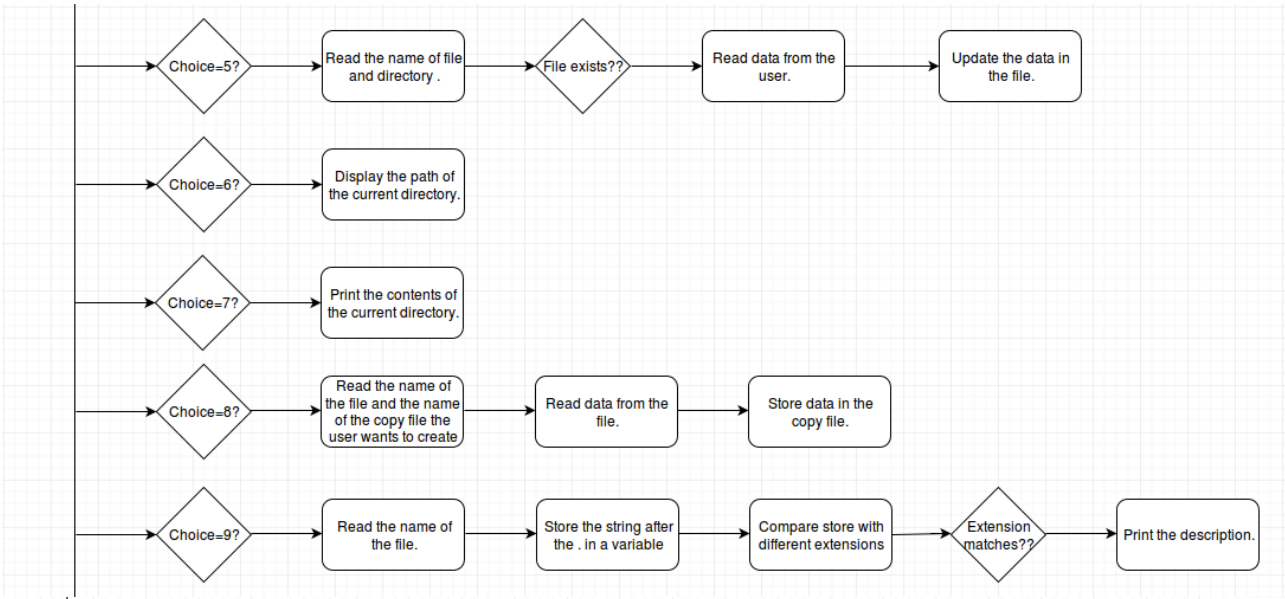
- **get_curr_dir function:**Prints the path to current directory.
  1. Copy curr_dir struct to temp_dir struct.
  2. If the 1st inode number is 0, then current directory is root.Print root and exit.
  3. Read the parent of current directory into global node struct.
  4. Find out where the parent node stores its directory table.
  5. Read the block where the parent directory table is stored.
  6. Load the directory struct into the dummy directory struct.
  7. Store the path in st1.

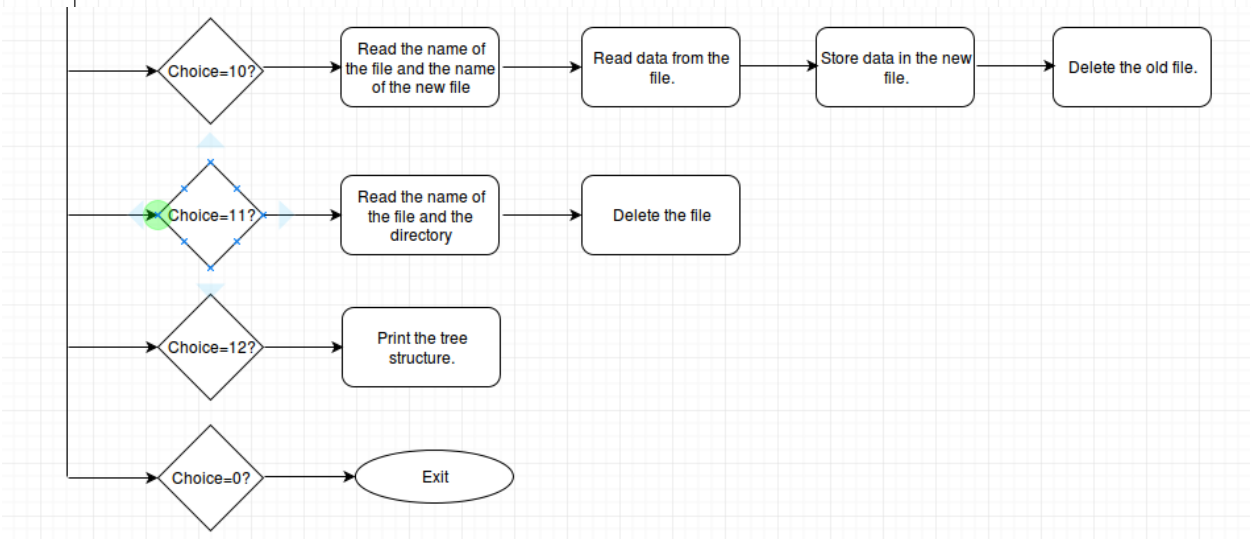
- **Tree function:**
  1. Print root.
  2. Search for curr dir. starting at inode 2 because of . and ..
  3. Run a loop from i=2 to curr_dir->inode_num[i]+2
     read_inode(curr_dir->inode_num[i]).
     If node->filecode = DIRECTORY
     Print curr_dir->filename[i]
  4. Run a loop from i=2 to curr_dir->inode_num[i]+2
     read_inode(curr_dir->inode_num[i]).
     If node->filecode = FILE
     Print curr_dir->filename[i]

Flowchart:




Read the choice from the user.
Choice=1?
Read the name of the new directory
Create the new Directory.
Choice=2?
Read the name of the file and directory
Create the file in the directory name given by the user.
Choice=3?
Read the name of file and directory
Input data from the user.
Store the data in the file.
Choice=4?
Read the name of file and directory
File exists??
Read from the file.


Choice=5?
Read the name of file and directory .
File exists??
Read data from the user.
Update the data in the file.
Choice=6?
Display the path of the current directory.
Choice=7?
Print the contents of the current directory.
Choice=8?
Read the name of the file and the name of the copy file the user wants to create
Read data from the file.
Store data in the copy file.
Choice=9?
Read the name of the file.
Store the string after the . in a variable
Compare store with different extensions
Extension matches??
Print the description.

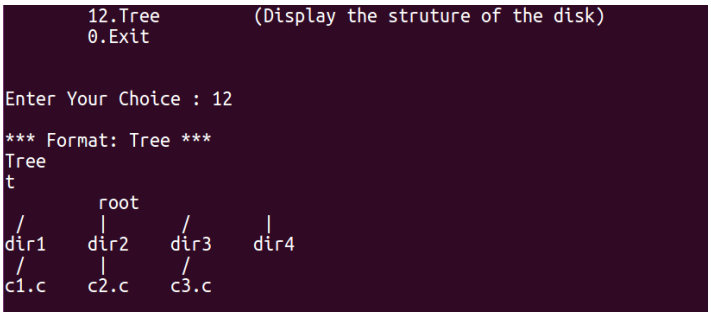

Choice=10?
Read the name of the file and the name of the new file
Read data from the file.
Store data in the new file.
Delete the old file.
Choice=11?
Read the name of the file and the directory
Delete the file
Choice=12?
Print the tree structure.
Choice=0?
Exit

List of programs:

- disk.c : The disk.c file contains structure of virtual disk.
- disk.h: The disk.h API simulates hard disk drivers used by the Filesystem (FS), fileSystem.c.
- fileSystem.c: The File System (EFFS), fileSystem.c simulates a file system with a file, i.e. a "fisk", as disk and separate blocks of the file as "fisk files".
- fileSystem.h: The fileSystem.h contains initialization of functions.
- main.c: The code which begins execution of the file system.
- Delete.mk: This file is for format the disk drive.
- Makefile.mk: This file contains code for how to compile the code.

Test data sets:

- mkdir dir1 : mkdir command will create directory called dir1.
- touch c1.c: touch command will create empty file c1.c
- vi c1.c & Enter data: I am janvi. : vi command will insert the data to the c1.c file
- cat c1.c : using cat command the content of c1 file will be shown.
- update c1.c : using update command we can update the data.
- pwd : pwd command will show the result exactly current working directory
- Ls : list out all the directories and files
- Copy c1.c c4.c : copy the content of c1 file to c4 file
- file c1.c : file command will show the type of file
- rename c1.c to c5.c : rename command is used to rename the file c1 to c5
- remove c1.c : remove command is for removing the file c1
- mkdir dir2 : mkdir command will create directory called dir2.
- touch c2.c: touch command will create empty file c2.c
- mkdir dir3 : mkdir command will create directory called dir3.
- touch c3.c : touch command will create empty file c2.c
- mkdir dir4 : mkdir command will create directory called dir3.
- tree : it will make tree to show hierarchical structure of system shown in below screenshot.



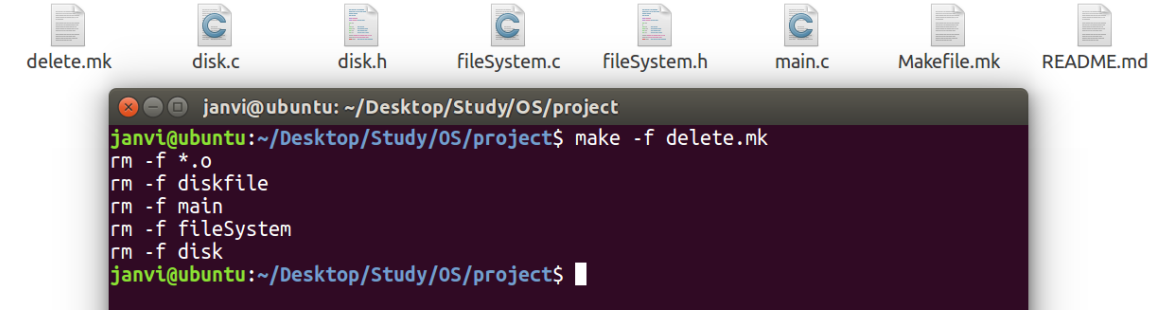
7. Implementation: Source code:

Please click on the link to see the code:
https://drive.google.com/open?id=1rkhRBusYiKY1W7ciDYgkCy7PEMufjbmO

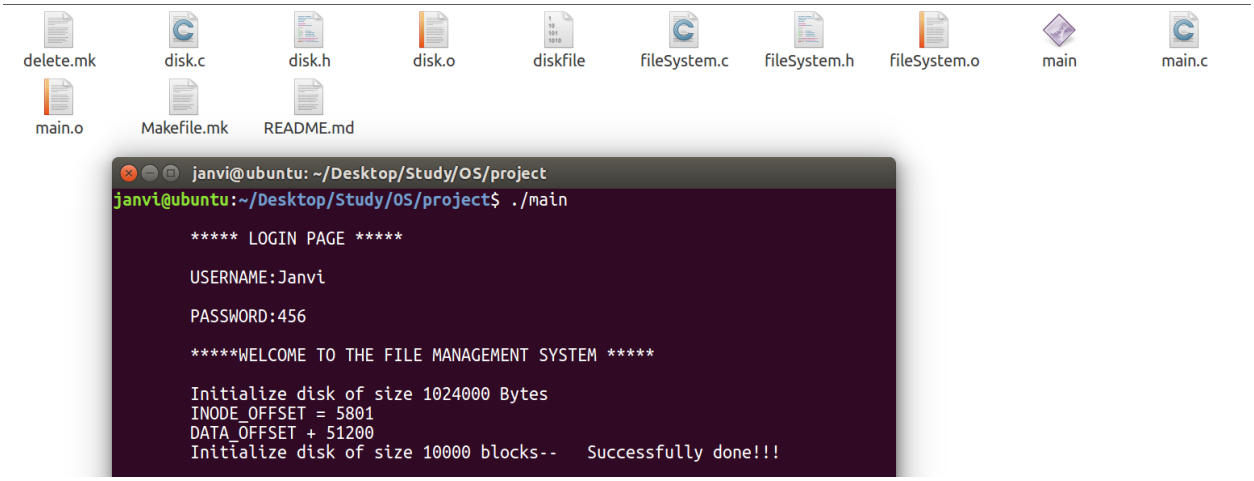
8. Test Results:

Here I am attaching the screen shots of result. Which contains main menu, all 12 cases and other compilation.

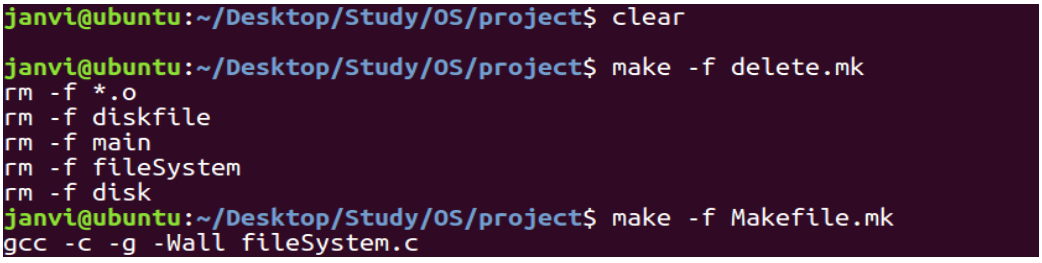
Result 1:



Result 2:



Result 3:



Result 4:

```
gcc -g -Wall fileSystem.o disk.o main.o -o main
janvi@ubuntu:~/Desktop/Study/OS/project$ ./main

        ***** LOGIN PAGE *****

        USERNAME:Janvi

        PASSWORD:456

        *****WELCOME TO THE FILE MANAGEMENT SYSTEM *****

        Initialize disk of size 1024000 Bytes
        INODE_OFFSET = 5801
        DATA_OFFSET + 51200
        Initialize disk of size 10000 blocks--   Successfully done!!!


        ******* MENU FOR THE FILE SYSTEM *******

        1.mkdir         (Create New Directory)
        2.touch         (Create New file)
        3.vi            (Enter the data into file)
        4.cat           (Display the data from file)
        5.Update        (Update the content of file)
        6.pwd           (Display the current path)
        7.ls            (List the directories & files)
        8.Copy          (Create copy of the file)
        9.file          (Shows the type of file)
        10.rename       (Rename the file & directory)
        11.remove       (Delete the file)
        12.Tree         (Display the struture of the disk)
        0.Exit


Enter Your Choice :
```

Result 5:

```
Enter Your Choice : 1

*** Format: mkdir 'directory name' ***
mkdir project

Directory project created successfully...
Task Completed!!!
```

Result 6:

```
Enter Your Choice : 2

*** Format: touch 'File name' ***
touch os.txt

Enter the directory in which you want to create file:project
num_blocks_needed = 1

New File os.txt of size 0 bytes created successfully
```

Result 7:

```
        ******* MENU FOR THE FILE SYSTEM *******

        1.mkdir         (Create New Directory)
        2.touch         (Create New file)
        3.vi            (Enter the data into file)
        4.cat           (Display the data from file)
        5.Update        (Update the content of file)
        6.pwd           (Display the current path)
        7.ls            (List the directories & files)
        8.Copy          (Create copy of the file)
        9.file          (Shows the type of file)
        10.rename       (Rename the file & directory)
        11.remove       (Delete the file)
        12.Tree         (Display the struture of the disk)
        0.Exit


Enter Your Choice : 3

*** Format: vi 'File name' ***
vi os.txt
ENTER THE DATA:I am janvi.
File @ inode 2
test Opened Successfully...
Writing to file desc. inode[2] 11 Bytes
wrote block 51202
```

Result 7:

```
Enter Your Choice : 4

*** Format: cat 'File name'***
cat os.txt
File @ inode 2
test Opened Successfully...
Reading from file desc. inode[2] 11 Bytes
reading block 51202

I am janvi
@ Data successfully read from the file
```

Result 8:

```
        5.Update        (Update the content of file)
        6.pwd           (Display the current path)
        7.ls            (List the directories & files)
        8.Copy          (Create copy of the file)
        9.file          (Shows the type of file)
        10.rename       (Rename the file & directory)
        11.remove       (Delete the file)
        12.Tree         (Display the struture of the disk)
        0.Exit


Enter Your Choice : 5

*** Format: update 'File name' ***
update os.txt

Enter the directory:project
File @ inode 2
test Opened Successfully...
@ removing file os.txt
inode # of file = 2
filesize is 1024
freed block 2
freed indirection data blocks = 0
total freed blocks = 1
@--      Removed Successfully...
Input the Data:I am ashna.
num_blocks_needed = 1
Updating fileFile @ inode 2
test Opened Successfully...
Test 4b: test writing to file desc. inode[2] 0 Bytes
wrote block 51202
 Replacement did successfully..!!
```

```
Enter Your Choice : 4

*** Format: cat 'File name'***
cat os.txt
File @ inode 2
test Opened Successfully...
Reading from file desc. inode[2] 11 Bytes
reading block 51202

I am ashna
```

Result 9:

```
        ******* MENU FOR THE FILE SYSTEM *******

        1.mkdir         (Create New Directory)
        2.touch         (Create New file)
        3.vi            (Enter the data into file)
        4.cat           (Display the data from file)
        5.Update        (Update the content of file)
        6.pwd           (Display the current path)
        7.ls            (List the directories & files)
        8.Copy          (Create copy of the file)
        9.file          (Shows the type of file)
        10.rename       (Rename the file & directory)
        11.remove       (Delete the file)
        12.Tree         (Display the struture of the disk)
        0.Exit


Enter Your Choice : 6

*** Format: pwd ***
pwd
File @ inode 2
test Opened Successfully...
2/root/project
```

Result 10:

```
Enter Your Choice : 7

*** Format: ls ***
ls
        DIRECTORY-----project
        FILE----------os.txt
```

Result 11:

```
        8.Copy          (Create copy of the file)
        9.file          (Shows the type of file)
        10.rename       (Rename the file & directory)
        11.remove       (Delete the file)
        12.Tree         (Display the struture of the disk)
        0.Exit


Enter Your Choice : 8

*** Format: copy 'filename' 'Name of other file'***
copy os.txt osCode.txt

Enter the directory:project
The content of the file :
reading block 51202

@ Passed
num_blocks_needed = 1
File @ inode 3
test Opened Successfully...
Test 4b: test writing to file desc. inode[3] 0 Bytes
wrote block 51203
@ Passed
```

```
Enter Your Choice : 7

*** Format: ls ***
ls
        DIRECTORY-----project
        FILE----------os.txt
        FILE----------osCode.txt
```

Result 12:

```
        9.file          (Shows the type of file)
        10.rename       (Rename the file & directory)
        11.remove       (Delete the file)
        12.Tree         (Display the struture of the disk)
        0.Exit


Enter Your Choice : 9

Enter the name of the file: os.txt
os.txt:ASCII text
```

Result 13:

```
        10.rename       (Rename the file & directory)
        11.remove       (Delete the file)
        12.Tree         (Display the struture of the disk)
        0.Exit


Enter Your Choice : 10

***Format: rename 'File name'***
rename os.txt

Enter the directory:project

Enter the new name for file: file.txt
reading block 51202

@ Renamed successfully
@ removing file os.txt
inode # of file = 2
filesize is 1024
freed block 2
freed indirection data blocks = 0
total freed blocks = 1
@--      Removed Successfully...
num_blocks_needed = 1
File @ inode 2
test Opened Successfully...
wrote block 51202
```

```
Enter Your Choice : 7

*** Format: ls ***
ls
          DIRECTORY-----project
          FILE----------osCode.txt
          FILE----------file.txt
```

Result 14:

```
        11.remove       (Delete the file)
        12.Tree         (Display the struture of the disk)
        0.Exit


Enter Your Choice : 11

*** Format: remove 'File name'***
remove file.txt
removing file removing file file.txt
inode # of file = 2
filesize is 1024
freed block 2
freed indirection data blocks = 0
total freed blocks = 1
removing file--  Removed Successfully...
```

```
Enter Your Choice : 7

*** Format: ls ***
ls
        DIRECTORY-----project
        FILE----------osCode.txt
```

Result 15:
Created new directories and files to display Tree structure

```
Enter Your Choice : 7

*** Format: ls ***
ls
        DIRECTORY-----dir1
        FILE----------c1.c
        DIRECTORY-----dir2
        FILE----------c2.c
        DIRECTORY-----dir3
        FILE----------c3.c
        DIRECTORY-----dir4
```

```
        12.Tree         (Display the struture of the disk)
        0.Exit


Enter Your Choice : 12

*** Format: Tree ***
Tree
t
         root
 /        |        /        |
dir1     dir2     dir3     dir4
 /        |        /
c1.c     c2.c     c3.c
```

Result 16:

```
janvi@ubuntu:~/Desktop/Study/OS/project$ ./main

        ***** LOGIN PAGE *****

        USERNAME:Janvi

        PASSWORD:123
Invalid username or password....!!!
```

Result 17:

```

Enter Your Choice : 1

*** Format: mkdir 'directory name' ***
mkdir dir1
Cannot create directory 'dir1': Directory exists

Please enter different name of directory...!!!
```

Result 18:

```
Enter Your Choice : 2

*** Format: touch 'File name' ***
Touch demo.txt
Invalid Format !!! Please try again
```

9. References:Books, research papers, articles, web sites referred:

1) Operating system: Internals and design principles by William Stallings.
2) http://nptel.ac.in/courses/106108101/pdf/PPTs/Mod_2.pdf
3) Chapter 3 -File Systems and the File Hierarchy- http://www.compsci.hunter.cuny.edu/~sweiss/course_materials/unix_lecture_notes/chapter_03.pdf
4) https://www.slideshare.net/harleen-johal/file-system-12292068.
5) http://heather.cs.ucdavis.edu/~matloff/UnixAndC/Unix/FileSyst.pdf
6) File Systems: http://www.tldp.org/LDP/sag/html/filesystems.html
7) General overview of Linux file system: http://www.tldp.org/LDP/intro-linux/html/sect_03_01.html
8) I-node and its structure in unix:https://www.slashroot.in/inode-and-its-structure-linux.
9) Windows file system: https://web.cs.wpi.edu/~cs4513/b05/week2-windowsfs.pdf
10) https://www.le.ac.uk/oerresources/bdra/unix/page_34.htm

THANK YOU…!!!