

## School of Computer Science and Engineering

**OPERATING SYSTEM-CSE2005**

#### PROJECT COMPONENT-REPORT

**Developing a custom File System**

SUBMITTED TO:

Prof. NAGARAJA RAO A

**GROUP MEMBERS:**

1. M SAMPREETH (19BCE0865)
2. CHINTHAM VISHAL REDDY (19BCE2012)
3. MOHAMMED RAYYAN (19BCE2565)

**CONTENTS**

1. **Abstract**
2. **Objective**
3. **Introduction**
4. **System Architecture**
5. **Implementation**
6. **Work Flow Diagrams**
7. **Implementation**
8. **Code and Analysis**
9. **Output and Snapshots**
10. **Conclusions and Scope for future work**
11. **References**

**ABSTRACT**

The File system is a typical and important process in any operating system. Hence a discrete and structured file managing system is essential. The proposed File system project aims at performing basic file managing processes. It uses the regular file partition, for the actual data storage of the files and designing a basic file system structure and implement the format command for it, over the regular file.

**OBJECTIVE**

The project objective is to implement a file system in linux and can add,remove and list all the files in the file system using a command shell interface.

**INTRODUCTION**

* A File System is a collection of files in a partition or disk. This partition might also occupy the entire hard disk
* There nearly 100 different file systems, each file system satisfies unique requirements.
* Hence, we can't conclude that a particular file system is the best.
* Motivation for developing a file system is that it is a challenging task as there is no generic material present for developing a file system.

There are three steps for making a file system:

• Create a file partition system for the physical storage of the input files

• Designing a kernel space or a system structure and implementing the make file system or mkfs command over the regular file.

• Making a graphical user interface where the user can enter the required commands to run the program.

**Creating a file system**

A file system contains two structures, the super block containing the info about the file system, and the file entry structure containing the info about each file in the file system.

**Designing a kernel space**

The basic kernel space would contain the logic for basic commands that can be applied to the input files like creating, deleting, updating. We can also perform functions like reading the input file and writing it.

**Interface**

The interface contains the user visible and executable commands such as to take an input, read or write it and list the available files.

**SYSTEM ARCHITECTURE**

The regular file that is chosen for our purpose is .sfsf (simulating file system file). This is a hidden file as it starts with " . "​

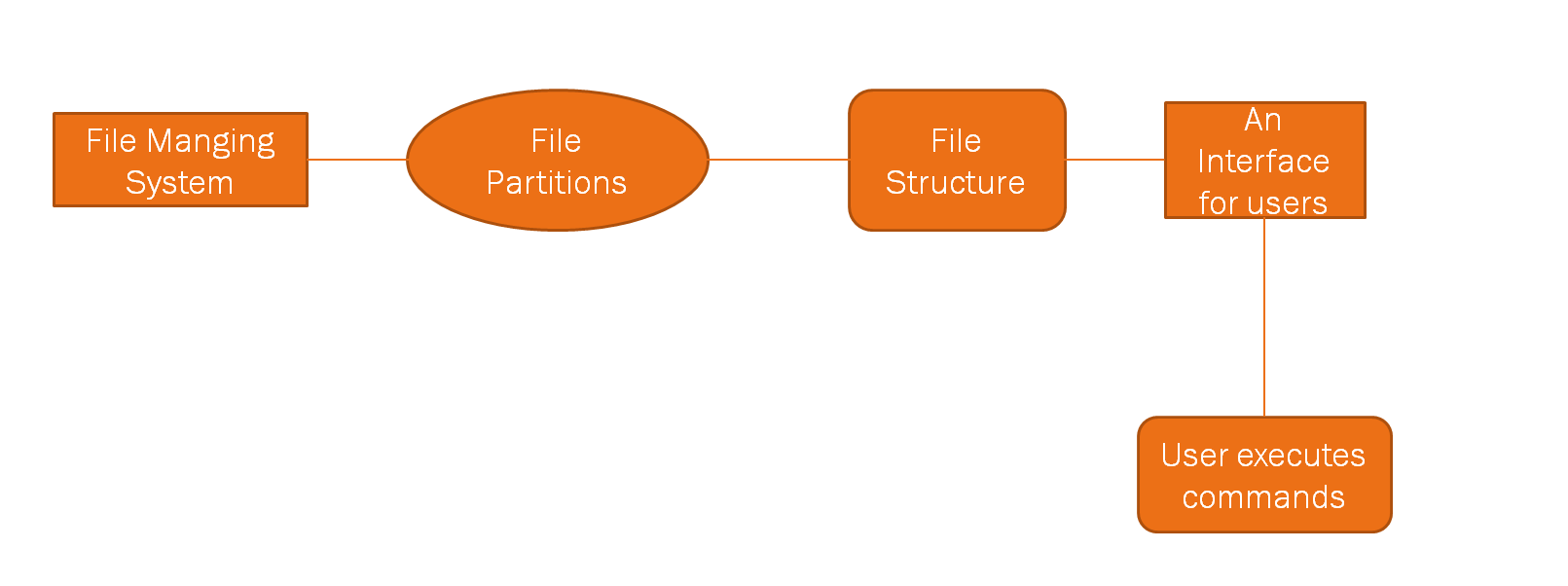
Basic File System consists of two structures:-

**Super Block** – contains the information about the file system.

**File Entry Structure** – contains the information of each file present in the file system.

File structure is defined by the header file sfs\_ds.h.

**WORK FLOW DIAGRAM:**

****

**IMPLEMENTATION**

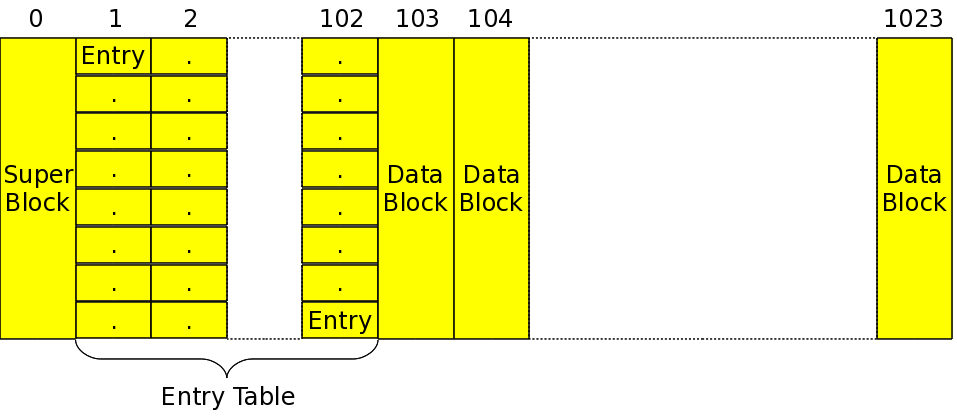
There are basically 3 parts of making the file managing system.

* Use a regular file for a partition, that is for the actual data storage of the file system (Hardware space equivalent)
* The Basic file system will be containing two structure, The Super Block which contains all the information about the file system, and the file entry structure containing the info about each file in the file system.
* The regular file that is going to be used here is the .sfsf(simulating file system file) in the current directory which acts as a header file which stores the information

Design a basic file system structure (Kernel space equivalent) and implement the format/mkfs command for it, over the regular file

* Each file coming to the entry contains
* Name of the file
* Size of the file in bytes
* Time stamp
* Permissions
* Ex: ./format\_sfs 1024 # Partition size in blocks of 512-bytes

**Example of file partition**



* With the above design of simulating file system (sfs.h), along with the implementation for its format command file
* We thus created the empty file system over the simulated partition .sfsf. Now, the final step in simulating the user space file system is to create the interface/shell to type commands and operate on the empty file system just created on .sfsf.

Provide an interface/shell to type commands and operate on the file system, similar to the usual bash shell. This step is achieved by the shell along with the corresponding file system driver in kernel space. But here that translation is embodied/simulated into the user space interface

To complete the understanding of a file system, the final system is to browse and play around with the file system created.

The created shell program primarily reads the super block from the partition file which has been created at first and then gets into browsing the file system based on the information using a user defined function.

The Final commands that can be executed in the thus created file system are:

* quit – to quit the file system browser
* list – to list the current files in the file system
* create “FileName”  – to create a new file in the file system
* remove “FileName” -– to remove an existing file from the file system

**CODE & ANALYSIS**

**Defining the super block structure and file entry in sfs\_ds.h**

#ifndef SFS\_DS\_H

#define SFS\_DS\_H

#define SIMULA\_FS\_TYPE 0x13090D15 /\* Magic Number for our file system \*/

#define SIMULA\_FS\_BLOCK\_SIZE 512 /\* in bytes \*/

#define SIMULA\_FS\_ENTRY\_SIZE 64 /\* in bytes \*/

#define SIMULA\_FS\_DATA\_BLOCK\_CNT ((SIMULA\_FS\_ENTRY\_SIZE - (16 + 3 \* 4)) / 4)

#define SIMULA\_DEFAULT\_FILE ".sfsf"

typedef unsigned int uint4\_t;

typedef struct sfs\_super\_block

{

uint4\_t type; /\* Magic number to identify the file system \*/

uint4\_t block\_size; /\* Unit of allocation \*/

uint4\_t partition\_size; /\* in blocks \*/

uint4\_t entry\_size; /\* in bytes \*/

uint4\_t entry\_table\_size; /\* in blocks \*/

uint4\_t entry\_table\_block\_start; /\* in blocks \*/

uint4\_t entry\_count; /\* Total entries in the file system \*/

uint4\_t data\_block\_start; /\* in blocks \*/

uint4\_t reserved[SIMULA\_FS\_BLOCK\_SIZE / 4 - 8];

} sfs\_super\_block\_t; /\* Making it of SIMULA\_FS\_BLOCK\_SIZE \*/

typedef struct sfs\_file\_entry

{

char name[16];

uint4\_t size; /\* in bytes \*/

uint4\_t timestamp; /\* Seconds since Epoch \*/

uint4\_t perms; /\* Permissions for user \*/

uint4\_t blocks[SIMULA\_FS\_DATA\_BLOCK\_CNT];

} sfs\_file\_entry\_t;

#endif

**ANALYSIS SUPER BLOCK STRUCTURE**

* **Type**: It is the magic number for a file system which is unique to every file system.​
* Block\_size: It is unit of allocation that is writable by the file system.​
* Partition\_size: It is the size of the entire file system (defined in number of blocks).​
* Entry\_size: Entry size is the size of the smallest unit in the entry table (where the files are stored)​
* Entry\_table\_size:  Total size of the entry table (in number of blocksx)​
* Entry\_table\_block\_start: This is the block number from which the entry table starts.​
* Entry\_count: It is the number of enties present in the entry table.​
* Data\_block\_start: The Block number from which data blocks start.​
* Reserved[SIMULA\_FS\_BLOCK\_SIZE / 4 - 8]: ​

**ANALYSIS FILE ENTRY STRUCTURE**

* Name: It holds the name of the  file.​
* Size: It holds the size of the file.​
* Perms: It holds the permissions of the file ​
* Timestamp: It holds the date of file creation (in epoch time)​
* Blocks[SIMULA\_FS\_DATA\_BLOCK\_CNT]:holds the array of block numbers for upto 9 data blocks.​

So every file contains name, size, timestamp of its creation, permissions and blocks.

**Allocation of the Super Block and clear\_file\_entries in format\_sfs.c**

#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <unistd.h>

#include "sfs\_ds.h"

#define SFS\_ENTRY\_RATIO 0.10 /\* 10% of all blocks \*/

#define SFS\_ENTRY\_TABLE\_BLOCK\_START 1

sfs\_super\_block\_t sb =

{

.type = SIMULA\_FS\_TYPE,

.block\_size = SIMULA\_FS\_BLOCK\_SIZE,

.entry\_size = SIMULA\_FS\_ENTRY\_SIZE,

.entry\_table\_block\_start = SFS\_ENTRY\_TABLE\_BLOCK\_START

};

sfs\_file\_entry\_t fe; /\* All 0's \*/

void write\_super\_block(int sfs\_handle, sfs\_super\_block\_t \*sb)

{

write(sfs\_handle, sb, sizeof(sfs\_super\_block\_t));

}

void clear\_file\_entries(int sfs\_handle, sfs\_super\_block\_t \*sb)

{

int i;

for (i = 0; i < sb->entry\_count; i++)

{

write(sfs\_handle, &fe, sizeof(fe));

}

}

void mark\_data\_blocks(int sfs\_handle, sfs\_super\_block\_t \*sb)

{

char c = 0;

lseek(sfs\_handle, sb->partition\_size \* sb->block\_size - 1, SEEK\_SET);

write(sfs\_handle, &c, 1); /\* To make the file size to partition size \*/

}

int main(int argc, char \*argv[])

{

int sfs\_handle;

if (argc != 2)

{

fprintf(stderr, "Usage: %s <partition size in 512-byte blocks>\n",

argv[0]);

return 1;

}

sb.partition\_size = atoi(argv[1]);

sb.entry\_table\_size = sb.partition\_size \* SFS\_ENTRY\_RATIO;

sb.entry\_count = sb.entry\_table\_size \* sb.block\_size / sb.entry\_size;

sb.data\_block\_start = SFS\_ENTRY\_TABLE\_BLOCK\_START + sb.entry\_table\_size;

sfs\_handle = creat(SIMULA\_DEFAULT\_FILE,

S\_IRUSR | S\_IWUSR | S\_IRGRP | S\_IROTH);

if (sfs\_handle == -1)

{

perror("No permissions to format");

return 2;

}

write\_super\_block(sfs\_handle, &sb);

clear\_file\_entries(sfs\_handle, &sb);

mark\_data\_blocks(sfs\_handle, &sb);

close(sfs\_handle);

return 0;

}

**ANALYSIS OF format\_sfs.c**

* Type : The magic number of the filesystem​
* Block size: Size of the blocks of the filesystem​
* Entry Size: size of each entry in the entrytable​
* Entry\_table\_block\_start: 1 (as the super block takes the block 0)​
* ​

Write\_super\_block() :​

Arguments: ​

1) sfs\_handle (used to identify the file system)​

2) sfs\_block\_t \*sb (It contains the information that needs to be stored in the super block)​

​

Write function:​

The in-built write function takes the handle, the given super block and its size as arguments and writes this data into the file system and hence the super block of the file system has been created.​

Arguments:​

1) sfs\_handle (uniquely identifies a file system)​

2) sfs\_super\_block\_t  \*sb ( super block that has been created already is also passed as the entry count of the super block is required)​

​

Write function :​

It takes the sfs\_handle, the pointer to the empty fe object of sfs\_file\_entry (it contains all 0's as it's data) and also the size of fe object as the arguments.​

​**Main function**

It clears all the file entries of entry table as a part of initialization of the file system.

* sb.partition\_size will be set to the number of bytes that is given as an command-line argument when the file is executed.​
* Sb.entry\_table\_size,sb.entry\_count,sb.data\_block\_start can be subsequently calculated using the given details.​
* Sfs handle is found by the in-built creat function.​
* Super block is created by calling the write\_super\_block()​
* Enty table is cleared (initialized) by calling clear\_file\_entries()

**With the above code**

* The above design of simulating file system (sfs\_ds.h), along with the execution of the format\_sfs.c file​.
* We thus created the empty file system over the simulated partition .sfsf . Now, the final step in simulating the user space file system is to create the interface/shell to type commands and operate on the empty file system.

**CODE FOR BROWSER.C**

#include <stdio.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <unistd.h>

#include <string.h>

#include <time.h>

#include "sfs\_ds.h"

sfs\_super\_block\_t sb;

void sfs\_list(int sfs\_handle)

{

int i;

sfs\_file\_entry\_t fe;

lseek(sfs\_handle, sb.entry\_table\_block\_start \* sb.block\_size, SEEK\_SET);

for (i = 0; i < sb.entry\_count; i++)

{

read(sfs\_handle, &fe, sizeof(sfs\_file\_entry\_t));

if (!fe.name[0]) continue;

printf("%-15s %10d bytes %c%c%c %s",

fe.name, fe.size,

fe.perms & 04 ? 'r' : '-',

fe.perms & 02 ? 'w' : '-',

fe.perms & 01 ? 'x' : '-',

ctime((time\_t \*)&fe.timestamp)

);

}

}

void sfs\_create(int sfs\_handle, char \*fn)

{

int i;

sfs\_file\_entry\_t fe;

lseek(sfs\_handle, sb.entry\_table\_block\_start \* sb.block\_size, SEEK\_SET);

for (i = 0; i < sb.entry\_count; i++)

{

read(sfs\_handle, &fe, sizeof(sfs\_file\_entry\_t));

if (!fe.name[0]) break;

if (strcmp(fe.name, fn) == 0)

{

printf("File %s already exists\n", fn);

return;

}

}

if (i == sb.entry\_count)

{

printf("No entries left\n", fn);

return;

}

lseek(sfs\_handle, -(off\_t)(sb.entry\_size), SEEK\_CUR);

strncpy(fe.name, fn, 15);

fe.name[15] = 0;

fe.size = 0;

fe.timestamp = time(NULL);

fe.perms = 07;

for (i = 0; i < SIMULA\_FS\_DATA\_BLOCK\_CNT; i++)

{

fe.blocks[i] = 0;

}

write(sfs\_handle, &fe, sizeof(sfs\_file\_entry\_t));

}

void sfs\_remove(int sfs\_handle, char \*fn)

{

int i;

sfs\_file\_entry\_t fe;

lseek(sfs\_handle, sb.entry\_table\_block\_start \* sb.block\_size, SEEK\_SET);

for (i = 0; i < sb.entry\_count; i++)

{

read(sfs\_handle, &fe, sizeof(sfs\_file\_entry\_t));

if (!fe.name[0]) continue;

if (strcmp(fe.name, fn) == 0) break;

}

if (i == sb.entry\_count)

{

printf("File %s doesn't exist\n", fn);

return;

}

lseek(sfs\_handle, -(off\_t)(sb.entry\_size), SEEK\_CUR);

memset(&fe, 0, sizeof(sfs\_file\_entry\_t));

write(sfs\_handle, &fe, sizeof(sfs\_file\_entry\_t));

}

void browse\_sfs(int sfs\_handle)

{

int done;

char cmd[256], \*fn;

int ret;

done = 0;

printf("Welcome to SFS Browsing Shell v1.0\n\n");

printf("Block size : %d bytes\n", sb.block\_size);

printf("Partition size : %d blocks\n", sb.partition\_size);

printf("File entry size: %d bytes\n", sb.entry\_size);

printf("Entry tbl size : %d blocks\n", sb.entry\_table\_size);

printf("Entry count : %d\n", sb.entry\_count);

printf("\n");

while (!done)

{

printf(" $> ");

ret = scanf("%[^\n]", cmd);

if (ret < 0)

{

done = 1;

printf("\n");

continue;

}

else

{

getchar();

if (ret == 0) continue;

}

if (strcmp(cmd, "?") == 0)

{

printf("Supported commands:\n");

printf("\t?\tquit\tlist\tcreate\tremove\n");

continue;

}

else if (strcmp(cmd, "quit") == 0)

{

done = 1;

continue;

}

else if (strcmp(cmd, "list") == 0)

{

sfs\_list(sfs\_handle);

continue;

}

else if (strncmp(cmd, "create", 6) == 0)

{

if (cmd[6] == ' ')

{

fn = cmd + 7;

while (\*fn == ' ') fn++;

if (\*fn != '')

{

sfs\_create(sfs\_handle, fn);

continue;

}

}

}

else if (strncmp(cmd, "remove", 6) == 0)

{

if (cmd[6] == ' ')

{

fn = cmd + 7;

while (\*fn == ' ') fn++;

if (\*fn != '')

{

sfs\_remove(sfs\_handle, fn);

continue;

}

}

}

printf("Unknown/Incorrect command: %s\n", cmd);

printf("Supported commands:\n");

printf("\t?\tquit\tlist\tcreate <file>\tremove <file>\n");

}

}

int main(int argc, char \*argv[])

{

char \*sfs\_file = SIMULA\_DEFAULT\_FILE;

int sfs\_handle;

if (argc > 2)

{

fprintf(stderr, "Incorrect invocation. Possibilities are:\n");

fprintf(stderr,

"\t%s /\* Picks up %s as the default partition\_file \*/\n",

argv[0], SIMULA\_DEFAULT\_FILE);

fprintf(stderr, "\t%s [ partition\_file ]\n", argv[0]);

return 1;

}

if (argc == 2)

{

sfs\_file = argv[1];

}

sfs\_handle = open(sfs\_file, O\_RDWR);

if (sfs\_handle == -1)

{

fprintf(stderr, "Unable to browse SFS over %s\n", sfs\_file);

return 2;

}

read(sfs\_handle, &sb, sizeof(sfs\_super\_block\_t));

if (sb.type != SIMULA\_FS\_TYPE)

{

fprintf(stderr, "Invalid SFS detected. Giving up.\n");

close(sfs\_handle);

return 3;

}

browse\_sfs(sfs\_handle);

close(sfs\_handle);

return 0;

}

**ANALYSIS OF browser.c**

**Browser\_sfs() function in browser.c**

* Sfs handle is found by the in-built open function.​
* brower\_sfs() function is immediately called when browser.c is executed, this displays the file system information and commands which the user can input.​
* The browser\_sfs() function takes sfs\_handle as the argument.
* The browser.c file contains a function called browser\_sfs() which returns void and prints the  information about file system like block size,partition size,file entry size,entry count etc. when it is called.​
* The user must give an input for the read,write,list,quit operations.​
* When the input is "?" then the available operations list is printed​
* When quit is typed comes out of while loop exits, and the function execution is completed.​
* When the input is "list" by the user then the sfs\_list() function is called with sfs\_handle as the argument.

**sfs\_list() function in browser.c**

* When the input is "create" by the user then sfs\_create()  is called with sfs\_handle and fn(contains the name of the file) as arguments.​​
* When the input is "remove" by the user then sfs\_remove()  is called with sfs\_handle and fn(contains the name of the file) as arguments. ​
* If the user gives any other input which is not listed in the opertions list, then an error message is printed.

**sfs\_create() ​function in browser.c**

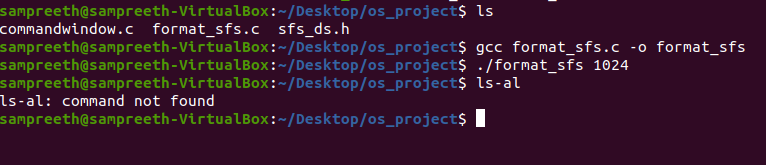
* Sfs\_create function takes sfs\_handle and file name as parameters.​
* In this function a loop is iterated from 0 to entry count and it checks if the name already exists in the file system if it exists then it gives a warning.​
* Else it creates a new file with the given name.​
* Then write function is called and it writes the file information (that is created) to the file system.

**sfs\_remove() ​function in browser.c**

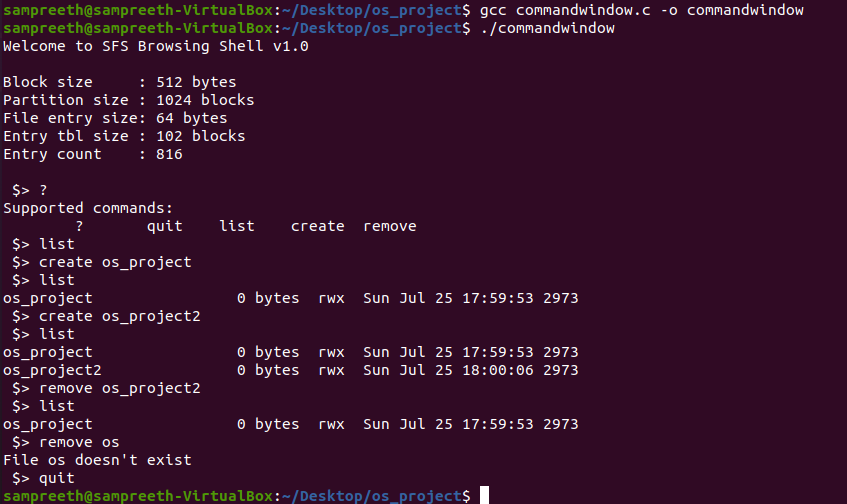
* Sfs\_remove function takes sfs\_handle and file name as arugments.​
* In this fucntion also a loop is iterated from 0 to entry count and it checks whether the file name already exists in the entry table ,if it doesn't exist then a warning is printed.​
* If file name exists, then the loop is stopped and the memset function is called which overwrites the given file with all 0's .​
* Then the modifed file is written to the file system by  calling write and giving argumnets of sfs\_handle,file information.​
* As the file contains only 0's, it is termed to be an empty file. Therfore, the given file is removed from the file system.

**OUTPUT**

CREATING FILE SYSTEM



CREATING,REMOVING FILE IN FILE SYSTEM



**CONCLUSION**

Finally using the above mentioned approach for making the file managing system. The user will be able to interact with the file system which is capable of all the basic tasks of creation, updation , Reading and deleting files using a bash or command shell which takes in the mentioned commands and returns with the required operations completed.

**FUTURE SCOPE OF WORK**

The project can still be developed adding and running scripting and c programming code.

**REFERNCES**

Modern Operating System(Andrew S)

Operating System Principles(Gavin)

<http://pages.cs.wisc.edu/~ll/papers/fsstudy.pdf>

<https://opensource.com/article/19/3/virtual-filesystems-linux>

<https://www.semanticscholar.org/paper/On-Journaling-File-Systems-in-Linux-%3A-An-Empirical-Kofahi-Al-Mubarak/e65b1f9dbfadb75da91b20895e658369e3c58808?p2df>