VIRTUAL FILE SYSTEM

Nupoor Raj- 19BCE2145

Khushi Agrawal-19BCE0418

Sagar Sethumadhavan-19BCE2460

Kandra Ksheeraj-19BCE0829

S.P.Rushalle Diya-19BCE2375

Submitted to

Dr. Geraldine Bessie Amali, SCOPE

VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

School of Computer Science and Engineering

PROJECT CONTENT

ABSTRACT

For most users, the file system is the most visible aspect of an operating system. It provides the mechanism for on-line storage of and access to both data and programs of the operating system and all the users of the computer system. The file system consists of two distinct parts: a collection of files, each storing related data, and a directory structure, which organizes and provides information about all the files in the system.

Taking its name from the way paper-based information systems are named, each group of data is called a "file". The structure and logic rules used to manage the groups of information and their names is called a "file system".

There are many different kinds of file systems. Each one has different structure and logic, properties of speed, flexibility, security, size and more. File systems can be used on numerous different types of storage devices that use different kinds of media. Some file systems are used on local data storage devices; others provide file access via a network protocol (for example, NFS, SMB, or 9P clients).

Some file systems are "virtual", meaning that the supplied "files" (called virtual files) are computed on request (e.g. procfs) or are merely a mapping into a different file system used as a backing store. The VFS distinguishes local files from remote ones, and local files are further distinguished according to their file-system types. We will also implement a VIRTUAL FILE SYSTEM in this paper.

The file system manages access to both the content of files and the metadata about those files. It is responsible for arranging storage space; reliability, efficiency, and tuning with regard to the physical storage medium are important design considerations.

1. Introduction

File systems provide efficient and convenient access to the disk by allowing data to be stored, located, and retrieved easily. A file system poses two quite different design problems. The first problem is defining how the file system should look to the user. This task involves defining a file and its attributes, the operations allowed on a file, and the directory structure for organizing files. The second problem is creating algorithms and data structures to map the logical file system onto the physical secondary-storage devices.

The file system itself is generally composed of many different levels. The structure shown in the besides figure is an example of a layered design. Each level in the design uses the features of lower levels to create new features for use by higher levels.

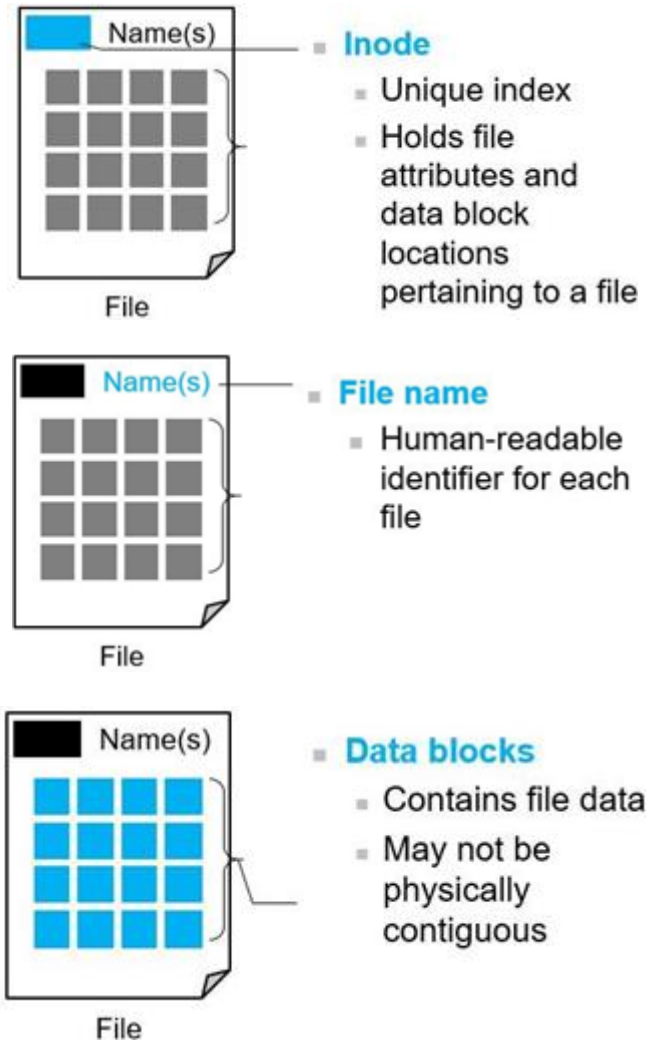
The I/O control level consists of device drivers and interrupt handlers to transfer information between the main memory and the disk system. A device driver can be thought of as a translator. Its input consists of high-level commands such as "retrieve block 146." Its output consists of low-level, hardware- specific instructions that are used by the hardware controller, which interfaces the I/O device to the rest of the system.

The basic file system needs only to issue generic commands to the appropriate device driver to read and write physical blocks on the disk. The file-organization module knows about files and their logical blocks, as well as physical blocks. By knowing the type of file allocation used and the location of the file, the file-organization module can translate logical block addresses to physical block addresses for the basic file system to transfer.

Finally, the logical file system manages metadata information. Metadata includes all of the file-system structure except the actual data (or contents of the files). The logical file system manages the directory structure to provide the file-organization module with the information the latter needs, given a symbolic file name.

It maintains file structure via file-control blocks. A file control block (FCB) (an inode in UNIX file systems) contains information about the file, including ownership, permissions, and location of the file contents. The logical file system is also responsible for protection and security.

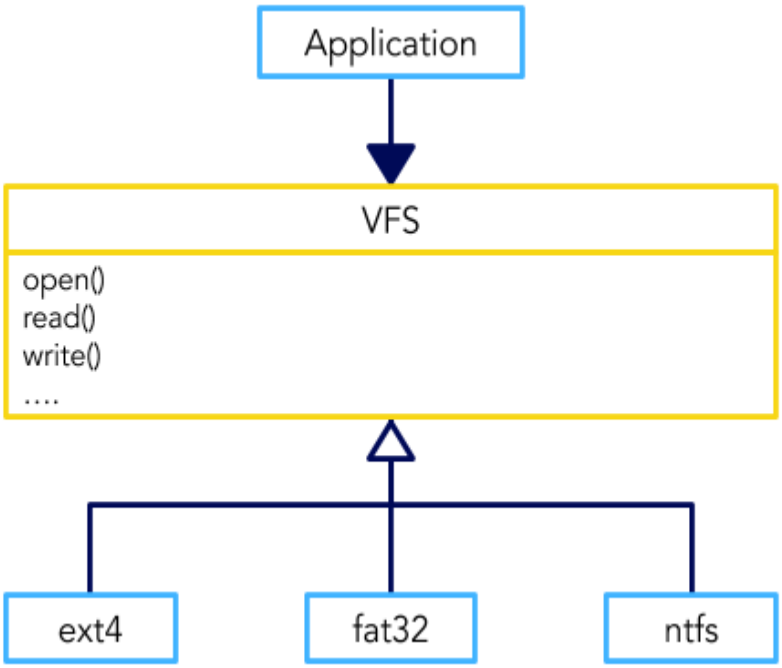
PHYSICAL FILE REPRESENTATION



The above diagrams represent a physical view of the file system. Inodes are the unique indices that hold file attributes and data block locations for a particular file. File name is a user readable identifier for each file. Now, the data blocks contain file data: may be contiguous or not contiguous.

The VFS is sandwiched between two layers: the upper and the lower. The upper layer is the system call layer where a user space process traps into the kernel to request a service (which is usually accomplished via libc wrapper functions) -- thus catalyzing the VFS's processes. The lower layer is a set of function pointers, one set per filesystem implementation, which the VFS calls when it needs an action performed that requires

information specific to a particular filesystem. Taken another way, the VFS could be thought of as the glue between the user space applications' requests for file-related services and the filesystem-specific functions which the VFS invokes as needed. From a high-level view, this can be modeled as a standard abstract interface:



Naturally, the lower layer is where Star Lab's AuthFS (and FortiFS) filesystems hook into the VFS. Therefore it is necessary that we understand the VFS if we want to work in the realm of filesystem implementation. By having the VFS require filesystems to define certain callback functions, such as read and write, we can operate on files without worrying about the implementation details. This way, the user space programmer is only concerned with reading from a file or writing to it and not with how the file is physically stored in a USB flash drive, disk drive, over the network, or on any other conceivable medium.

2. Hardware/Software Requirements

2.1 Hardware Requirements:

No specifications required.

2.2 Software Requirements

C/CPP Code Blocks IDE and OS Windows 10 used for the project.

3. Existing System/Approach:

There are several operating systems and thus multiple file systems available. These file systems provide different levels of user based application programming interfaces for managing the file system by a user based application. This makes the application to be OS dependent. Furthermore, the data can be stored in internal or external disk or may be USB drive. The application running in the user space needs to be independent of the operating system or underlying filesystem.

3.1 Drawback:

The drawback is that the user application needs to be ported for different operating systems which is costly and increases time to market. For example, a Unix provides read/write/execute (3 levels) of access to a file in the file system however windows have more level of access permission to a file. The Virtual File System can solve such a problem by providing a uniform mechanism to store and manage files.

4. Developed Model:

The Virtual File System model is developed to illustrate the capability to solve the problem in managing different operating systems by user based application. In other words, the VFS model can be used by the user application to make the application platform independent.

The model is a thin menu based application which simulates any user based application making the VFS API calls to handle the filesystem operations like creating/deleting/reading a file, searching a file or specific contents in a file. The essence of this model is that it stores the data in some logical order to the disk. The logic can be more sophisticated for business but this project showcases the ability of VFS to solve the problem. The data is stored in the disk using OS specific calls however the model provides its own way of handling the file and contents within the file. The user application interacts with VFS based APIs instead of direct OS calls thus VFS provides an abstraction between the user application and OS APIs to make the user application OS independent.

4.1 Design:

There are several design elements in this project. These design elements are listed below.

1. **Menu based application**

   The use case scenario is simulated via menu-based application. User is presented to perform several operations like Create/Delete/List files or search for a file or contents within a file. The function handlers for each of the menu items makes call to virtual file system based APIs. For example, when the user wants to create a file the underlying VFS API 'create_file()' is called. Similarly, for deleting a file 'delete_file()' is called.

2. **APIs**

   There are several APIs provided for user application to operate on a file. These are:

   - Create_file()
   - Delete_file()
   - List_files()
   - Show_file_content()
   - Search_file()
   - Search_keyword()

   An explanation of create_file() API is provided below.

   The create file takes the name of the file and file contents as input arguments. It looks for available memory space in the VFS. If the memory is available, it stores the name and length of the file along with the start position of the pointer into its data structure. The content of the file is written in the orderly manner in a file in the disk. The order of the data is 'name', 'length', 'start position' and lastly the 'content' of the file.

3. Data Structure

There are a couple of data structures used. One of the data structures is to manage the file and other one is for operating on a file system. The file system-based data structure provides public APIs for user to operate on virtual file system.

The data structure is mentioned below.

```
class file{
public:
   char name[64];
   long int len;
   int startpos;

   char* get_file_name();
   long int get_file_length();
   int get_startpos();
};

class filesys
{
public:
file files[MAX_FILES];
void initialize();
void read_from_file();
void write_to_file();
void set_file_system_name();
```

```
filesys(char f_name[]);

char file_system_name[20];

void list_files();

char* show_file_content(char* f_name);

char* search_file(char* f_name);

void search_keyword(char* f_name, char* keyword);

void delete_file(char* f_name);

void create_file(char* f_name, char* file_contents);

};
```

4.2 Module wise description:

The project is made modular to understand and maintain the code base. The various modules for the project is mentioned below.

1. Menu Based Application

2. Data Structure and Public API definitions

3. Internal API

The top level module is user interface which is a menu driven application. The handler for individual menu items makes call to VFS based APIs for a file operation e.g. create/delete/list file or list file contents etc. along with searching file or contents in a file.

The next layer is the VFS API itself which updates its internal data structures, the pointers for the data element like start position pointer of the data content. The API stores the data onto the disk using OS specific call. This OS specific calls is abstracted by the VFS API. The API which used to write the data is Internal to VFS.

The bottom most layer is ‘Internal API’ which actually stores the data into persistent location (e.g. onto the disk or USB etc.). As mentioned above the internal API is making the OS specific calls and thus the VFS provides the abstraction for an application to make OS independent calls to operate on a file.

4.3 Implementation:

Virtual File System Code:

```
#include <iostream>
#include<fstream>
#include<sstream>
#include<string.h>
#include<string>
#include<math.h>

#ifndef fi
#define fi
class file{
public:
char name[64];
long int len;
int startpos;
char* get_file_name();
long int get_file_length();
int get_startpos();
};
#endif // fi

#define MAX_FILE_LEN 1000
#define MAX_FILES 15
```

```
#ifndef fs
#define fs
#include<fstream>
using namespace std;
class filesys
{
public:
file files[MAX_FILES];
void initialize();
void read_from_file();
void write_to_file();
void set_file_system_name();
filesys(char f_name[]);
char file_system_name[20];
void list_files();
char* show_file_content(char* f_name);
char* search_file(char* f_name);
void search_keyword(char* f_name, char* keyword);
void delete_file(char* f_name);
void create_file(char* f_name, char* file_contents);
};
#endif // fs
#include<iostream>
using namespace std;
char* file::get_file_name()
```

```
{
return name;
}
long int file::get_file_length()
{
return len;
}
int file::get_startpos()
{
return startpos;
}

using namespace std;
void filesys::set_file_system_name()
{
cout << "Enter file system name\n";
char temp[80];
cin >> temp;
strcpy(file_system_name, temp);
initialize();
}
filesys::filesys(char f_name[])
{
if(f_name == NULL)
set_file_system_name();
```

```
else{
cout << "Existing File System\n";
strcpy(file_system_name, f_name);
read_from_file();
}
}
void filesys::initialize()
{
fstream myfile(file_system_name, ios::out);
int i;
myfile.seekp(0, ios::beg);
for(i = 0; i < MAX_FILES; i++){
strcpy(files[i].name, "\0");
files[i].len = 0;
files[i].startpos = 0;
}
myfile.close();
write_to_file();
}
void filesys::list_files()
{
for(int i = 0; i < MAX_FILES; i++){
if(!strcmp(files[i].get_file_name(), "\0"))
break;
cout << files[i].get_file_name() << endl;
```

```
}
}
char* filesys::show_file_content(char* f_name){
fstream myfile(file_system_name, ios::in);
int i;
char* file_contents = new char[MAX_FILE_LEN];
cout << f_name << "\n";
for(i = 0; i < MAX_FILES; i++){
if(!strcmp(files[i].get_file_name(), f_name))
{
myfile.seekg(files[i].get_startpos(), ios::beg);
myfile.read(file_contents, files[i].get_file_length());
*(file_contents + files[i].get_file_length()) = '\0';
myfile.close();
return file_contents;
}
}
cout<<"File not found!";
myfile.close();
return NULL;
}
char* filesys::search_file(char* f_name){
int i;
for(i = 0; i < MAX_FILES; i++){
if(!strcmp(files[i].get_file_name(), f_name)){
```

```
cout<<"File found\nFile name:";
return files[i].get_file_name();
}
}
cout<<"File not found!";
return NULL;
}
void filesys :: search_keyword(char* f_name, char* keyword){
fstream myfile(file_system_name, ios::in);
for(int i = 0; i < MAX_FILES; i++){
if(!strcmp(f_name, files[i].get_file_name())){
char* file_content = new char[MAX_FILE_LEN];
myfile.seekg(files[i].get_startpos());
myfile.read(file_content, files[i].get_file_length());
char* p = strstr(file_content, keyword);
if(p == NULL){
cout<<"\nKeyword not Found!\n";
myfile.close();
return;
}
int pos = p - file_content + 1; // The difference between the address of substring in the
//string and
cout<<"\nKeyword Found!\nPosition of keyword:\t"<<pos;
myfile.close();
return;
```

```
}
}
cout<<"\nFile not found!\n";
myfile.close();
return;
}
void filesys::delete_file(char* f_name)
{int i,j;
char* file_content = new char[MAX_FILE_LEN];
for(i = 0; i < MAX_FILES; i++){
if(!strcmp(files[i].get_file_name(), f_name)){
int del_len;
strcpy(files[i].name, "\0");
del_len = files[i].get_file_length() ;
files[i].len = 0;
files[i].startpos = 0;
write_to_file(); //Will skip the content of the file to be deleted
for(j = i + 1; j < MAX_FILES ; j++){
strcpy(files[j - 1].name, files[j].get_file_name());
files[j - 1].len = files[j].get_file_length();
files[j - 1].startpos = files[j].get_startpos() - del_len;
}
write_to_file();
cout << "\nFile deleted!\n";
return;
```

```
}
}
cout<<"File not found!";
}
void filesys::create_file(char* f_name, char* file_contents){
int i;
for(i = 0; i < MAX_FILES; i++){
if(!strcmp(files[i].get_file_name(), "\0"))
break;
}
if(i == MAX_FILES)
cout<<"No space";
else{
fstream myfile(file_system_name, ios::out | ios::app);
strcpy(files[i].name, f_name);
files[i].len = strlen(file_contents);
myfile.seekp(0, ios::end);
files[i].startpos = myfile.tellp();
myfile.write(file_contents,sizeof(char) * strlen(file_contents));
myfile.close();
write_to_file();
}
}
void filesys::read_from_file()
{
```

```
fstream myfile(file_system_name, ios::in);
int i;
myfile.seekg(0 , ios::beg);
for(i = 0; i < MAX_FILES; i++) //read already created files till null string is
//encountered or max limit
{myfile.read((char*)&(files[i].name) , sizeof(files[i].name));
if(!strcmp(files[i].name,"\0"))
break;
myfile.read((char*)&files[i].len , sizeof(long int));
myfile.read((char*)&files[i].startpos , sizeof(int));
}
while(i < MAX_FILES){
strcpy(files[i].name,"\0");
files[i].len = 0;
files[i].startpos = 0;
i++;
}
myfile.close();
}
void filesys::write_to_file(){
fstream myfile(file_system_name, ios::in);
fstream newfile("temp.txt", ios::out);
int i;
char file_content[MAX_FILE_LEN];
newfile.seekp(0 , ios::beg);
```

```
for(i = 0; i < MAX_FILES; i++){
newfile.write((char*)&files[i].name , sizeof(files[i].name));
newfile.write((char*)&files[i].len , sizeof(long int));
newfile.write((char*)&files[i].startpos , sizeof(int));
}
for(i = 0; i < MAX_FILES; i++){
if(files[i].get_file_length()){
myfile.seekg(files[i].get_startpos(), ios::beg);
myfile.read((char*)&file_content, sizeof(char) * files[i].len);
newfile.seekp(0 , ios::end);
newfile.write((char*)&file_content, sizeof(char) * files[i].len);
}
}
newfile.close();
myfile.close();
remove(file_system_name);
rename("temp.txt", file_system_name);
}
using namespace std;
int main(int argc, char* argv[])
//Enter command line arguments to open a previously created file system // or leave blank to create a new one
{
char name[80], content[200], name1[80], keyword[60], *c = NULL;
cout<< "--------------------------------------------------------------------------\n";
cout<< "******** Virtual File System **********\n";
```

```
cout<< "--------------------------------------------------------------------------\n";
int choice;
filesys f1(argv[1]);
while(1){
cout << "\nEnter your choice:\n";
cout << "1. List files in the file system\n";
cout << "2. Show file content\n";
cout << "3. Search a file\n";
cout << "4. Search for a keyword in a file\n";
cout << "5. Create new file\n";
cout << "6. Delete a file\n";
cin >> choice;
switch(choice){
case 1:
f1.list_files();
break;
case 2:
cout << "Enter file name\n";
cin >> name;
c = f1.show_file_content(name);
if(c != NULL)
cout << c;
break;
case 3:
cout << "Enter file name\n";
```

```
cin >> name;
c = f1.search_file(name);
if(c != NULL)
cout << c;
break;
case 4:
cout << "Enter the file name\n";
cin >> name;
cout << "Enter the keyword to be searched\n";
cin >> keyword;
f1.search_keyword(name, keyword);
break;
case 5:
cout << "Enter the name of new file\n";
cin >> name;
cout << "Enter the content of the file\n";
cin.clear();
fflush(stdin);
cin.getline(content, sizeof(content));
f1.create_file(name, content);
break;
case 6:
cout << "Enter the name of the file to be deleted\n";
cin >> name;
f1.delete_file(name);
```

```
break;
default:
cout << "Enter a valid option!\n";
}
}
}
```

To display the block address and FAT table code:

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
char *buffer[500]; //Memory created-500 bytes
int s_b[10][2],empty[10]; //Stores size and number of blocks
int mem=500; //keeps track of free memory
char *block[10]; //store address of the blocks
int *address[10]; //stores address of files
char name[10][30]; //stores name of the file
void open_createdfile() //to open the file
{
char fname[30],c;
int size=1,i;
printf("Enter txt file name\n");
scanf("%s",fname);
FILE *inputf;
inputf = fopen(fname,"r");
```

```
if (inputf == NULL)
{
printf("\nFile unable to open ");
exit(0);
}
rewind(inputf);
c=fgetc(inputf);
while(c!=EOF)
{
c=fgetc(inputf);
size=size+1;
}
printf("The size of given file is : %d\n", size);
if(mem>=size)
{
int n=1,parts=0,m=1;
while(address[n]!=0)
n++;
strcpy(name[n],fname);
s_b[n][1]=size;
int bnum=size/50;
if(size%50!=0)
bnum=bnum+1;
s_b[n][2]=bnum;
mem=mem-(bnum*50);
```

```
int *bfile=(int*)malloc(bnum*(sizeof(int)));
address[n]=bfile;
printf("Number of blocks required: %d\n",bnum);
rewind(inputf);
c = fgetc(inputf);
while(parts!=bnum && c!=EOF)
{
int k=0;
if(empty[m]==0)
{
char *temp=block[m];
while(k!=50)
{
*temp=c;
c=fgetc(inputf);
temp++;
k=k+1;
}
*(bfile+parts)=m;
parts=parts+1;
empty[m]=1;
}
else
m=m+1;
}
```

```
printf("File info displayed\n");
printf("\n");
fclose(inputf);
}
else
printf("Not enough memory\n");
}
int filenum(char fname[30])
{
int i=1,fnum=0;
while(name[i])
{
if(strcmp(name[i], fname) == 0)
{
fnum=i;
break;
}
i++;
}
return fnum;
}
void blocks()
{
int i;
printf(" Block address empty/free\n");
```

```
for(i=1;i<=10;i++)
printf("%d. %d - %d\n",i,block[i],empty[i]);
printf("\n");
}
void file()
{
int i=1;
printf("File name size address\n");
for(i=1;i<=10;i++)
{
if(address[i]!=0)
printf("%s %d %d\n",name[i],s_b[i][1],address[i]);
}
printf("\n");
}
void print()
{
char fname[30];
int i=1,j,k,fnum=0;
printf("Enter the file name: ");
scanf("%s",fname);
fnum=filenum(fname);
if(fnum!=0&& address[fnum]!=0)
{
int *temp;
```

```
temp=address[fnum];
printf("Content of the file %s is:\n",name[fnum]);
int b=(s_b[fnum][2]);
for(j=0;j<b;j++)
{
int s=*(temp+j);
char *prt=block[s];
for(k=0;k<50;k++)
{
printf("%c",*prt);
prt++;
}
}
printf("\n");
printf("\n");
}
else
printf("File not available:\n");
}
void remove1()
{
char fname[30];
int i=1,j,k,fnum=0;
printf("Enter the file name: ");
scanf("%s",fname);
```

```
fnum=filenum(fname);

if(fnum==0)

printf("File not available:\n");

else

{

int *temp=address[fnum];

int b=(s_b[fnum][2]);

mem=mem+b*50;

for(j=0;j<b;j++)

{

int s=*(temp+j);

empty[s]=0;

}

address[fnum]=0;

}

printf("\n");

}

int main()

{

printf("\n\t\t _____");

printf("\n\t\t|\t\t\t     |");

printf("\n\t\t|\tFAT TABLE AND\t
|\t\t_____");

printf("\n\t\t|    BLOCK ADDRESS VIEWER    |\t\t          #ADD-ON FOR VFS");

printf("\n\t\t|_____|\t\t_____
_____\n");
```

```
int choice,i;
char *temp;
if (buffer == NULL)
{
fputs ("Memory error",stderr);
exit(2);
}
temp=buffer;
block[1]=temp;
empty[1]=0;
for(i=2;i<=10;i++)
{
block[i]=block[i-1]+50;
empty[i]=0;
}
menu:while(1)
{
printf("1.Open existing file\n");
printf("2.Delete a file\n");
printf("3.Print a file \n");
printf("4.Display FAT table\n");
printf("5.Display Block Details\n");
printf("6.Exit the program\n");
printf("Enter your choice: ");
scanf("%d",&choice);
```

```
switch(choice)
{
case 1:
open_createdfile();
break;
case 2:
remove1();
break;
case 3:
print();
break;
case 4:
file();
break;
case 5:
blocks();
break;
case 6:
exit(1);
default:printf("Invalid option entered. Try again!\n\n");
goto menu;
}
}
return 0;
}
```

5. Results and Discussion

5.1 Implementation of VFS Program

5.1.1 Creating new files in the file system

```
--------------------------------------------------------------------------------
************************* Virtual File System ******************************
--------------------------------------------------------------------------------
Enter file system name
OSproject


Enter your choice:
1. List files in the file system
2. Show file content
3. Search a file
4. Search for a keyword in a file
5. Create new file
6. Delete a file
5
Enter the name of new file
Monday
Enter the content of the file
First day of the week


Enter your choice:
1. List files in the file system
2. Show file content
3. Search a file
4. Search for a keyword in a file
5. Create new file
6. Delete a file
5
Enter the name of new file
Sunday
Enter the content of the file
It is a holiday


Enter your choice:
1. List files in the file system
2. Show file content
3. Search a file
4. Search for a keyword in a file
5. Create new file
6. Delete a file
5
Enter the name of new file
Friday
Enter the content of the file
Last working day of the week
```

5.1.2 List all the files in file system

```
Enter your choice:
1. List files in the file system
2. Show file content
3. Search a file
4. Search for a keyword in a file
5. Create new file
6. Delete a file
1
Monday
Sunday
Friday
```

5.1.3 Display file content

```
Enter your choice:
1. List files in the file system
2. Show file content
3. Search a file
4. Search for a keyword in a file
5. Create new file
6. Delete a file
2
Enter file name
Sunday
Sunday
It is a holiday

Enter your choice:
1. List files in the file system
2. Show file content
3. Search a file
4. Search for a keyword in a file
5. Create new file
6. Delete a file
2
Enter file name
Tuesday
Tuesday
File not found!
```

5.1.4 Search for a file in the file system

```
Enter your choice:
1. List files in the file system
2. Show file content
3. Search a file
4. Search for a keyword in a file
5. Create new file
6. Delete a file
3
Enter file name
Friday
File found
File name:Friday

Enter your choice:
1. List files in the file system
2. Show file content
3. Search a file
4. Search for a keyword in a file
5. Create new file
6. Delete a file
3
Enter file name
Saturday
File not found!
```

5.1.5 Search for a Key word in content of file

```
Enter your choice:
1. List files in the file system
2. Show file content
3. Search a file
4. Search for a keyword in a file
5. Create new file
6. Delete a file
4
Enter the file name
Monday
Enter the keyword to be searched
week

Keyword Found!
Position of keyword:    18
```

Data validation for this case

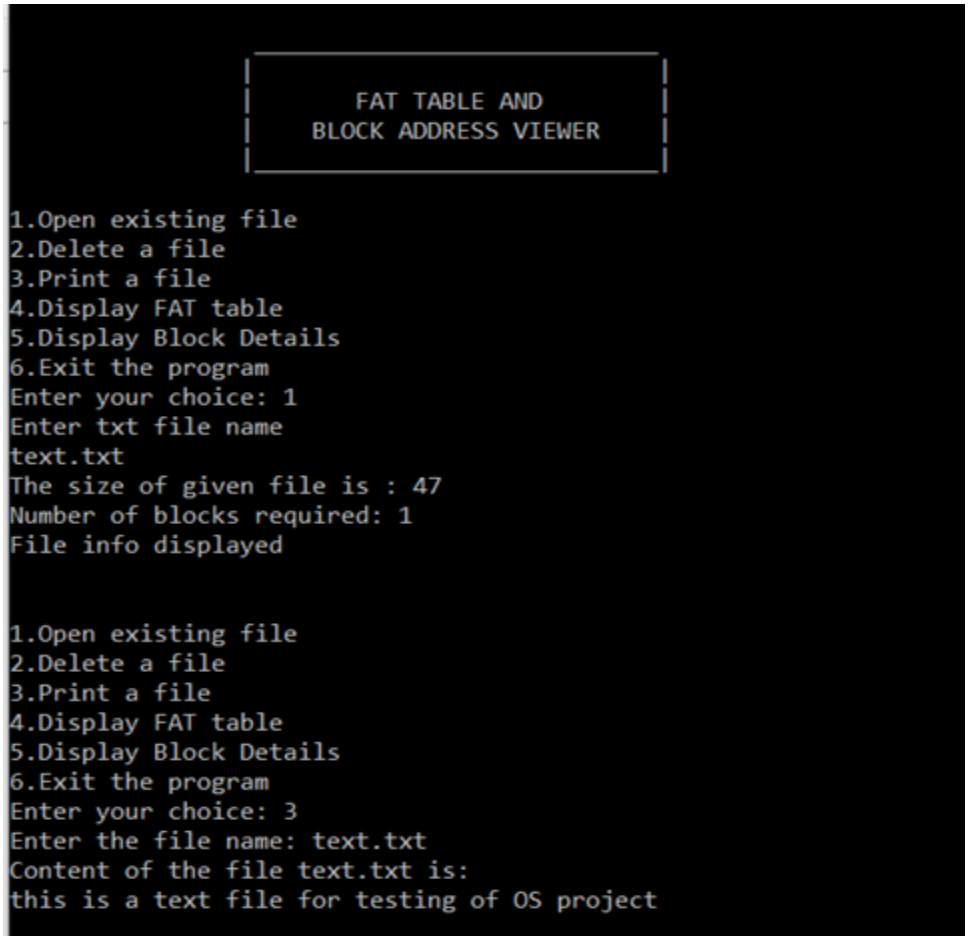
```
Enter your choice:
1. List files in the file system
2. Show file content
3. Search a file
4. Search for a keyword in a file
5. Create new file
6. Delete a file
4
Enter the file name
Monday
Enter the keyword to be searched
working

Keyword not Found!
```

5.1.6 Delete a file from the file system

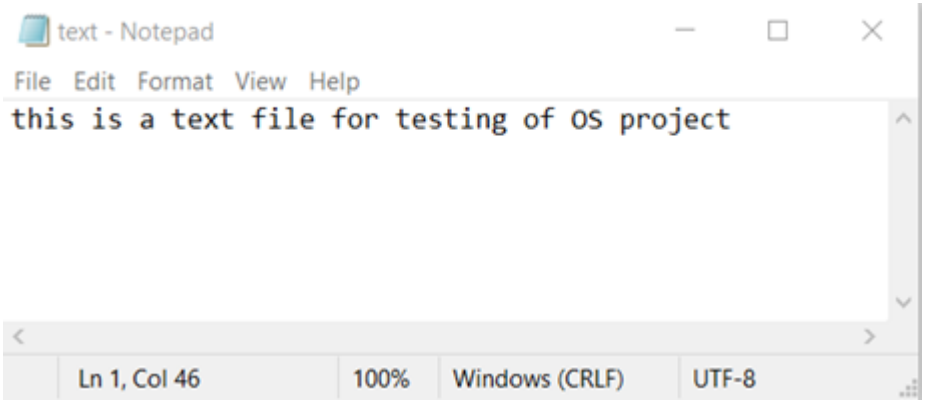
```
Enter your choice:
1. List files in the file system
2. Show file content
3. Search a file
4. Search for a keyword in a file
5. Create new file
6. Delete a file
6
Enter the name of the file to be deleted
Friday

File deleted!


Enter your choice:
1. List files in the file system
2. Show file content
3. Search a file
4. Search for a keyword in a file
5. Create new file
6. Delete a file
1
Monday
Sunday
```

5.2 Implementation of FAT Table and Block Address

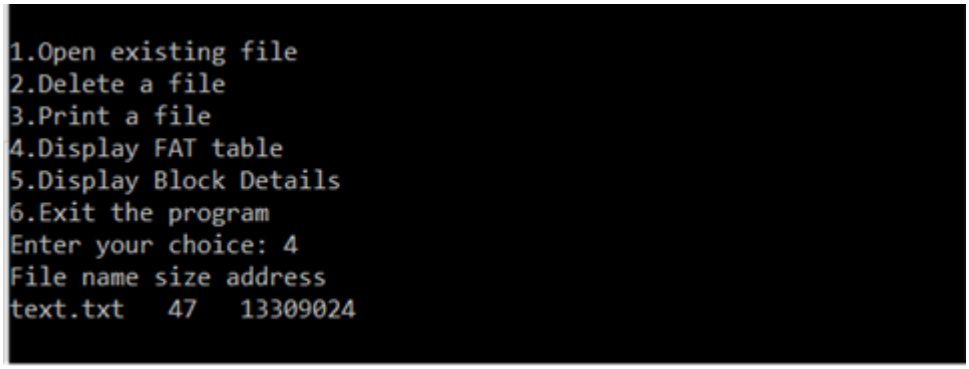
5.2.1 Opening a existing text file and printing file content



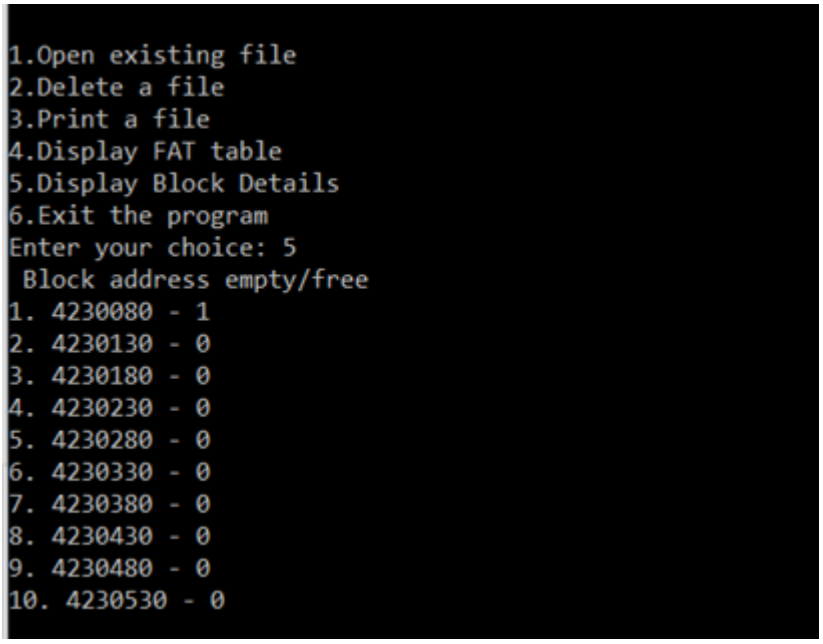
5.2.2 Text file opened in Windows using Notepad



5.2.3 Display of File Allocation Table(FAT Table)



5.2.4 Display of Block Address



5.3 Future Work

Thus, through our project, we have been able to successfully implement a virtual file system and its operations in C/C++. However, there are a few shortcomings. We would also like to enhance our project by adding some more features in the future and make it deployment ready.

Current command line UI can be replaced with a more user friendly and easy to navigate GUI. More VFS features will be added to support needs of wide variety of users. We also plan to have a better integration with the OS (Windows 10). A more intuitive file explorer will be made to access all files present in the system in a better way. Provision for an automated clean up function at a specific time interval can be added to save space. We would like to expand our project for the Linux OS also.

Github Link For This Project:

https://github.com/ksheeraj1161/Virtual-File-System

6. Conclusion

File systems like other components of an operating system need to evolve. In this work, we argued that this is because of the fact that hardware technology keeps on advancing and hence changing. As a consequence, user requirements also change. Hence, to cope up with these changes, file systems need modifications. However, modifying design of file systems to overcome these challenges has its limitations. This work was aimed to argue that the file systems can still evolve even with keeping design and source intact.

REFERENCES

- https://www.cse.iitb.ac.in/~puru/courses/autumn14/cs695/downloads/vmfs.pdf
- http://pages.cs.wisc.edu/~ll/papers/fsstudy.pdf Abraham Silberschatz, Peter B. Galvin, Greg Gagne-Operating System Concepts, Wiley (2012)
- https://www.techopedia.com/definition/27103/virtual-file-system-vfs