Virtual File System

Project Report

Operating Systems(CSE2005)

By:-

Mitarth Jain	**17BCE0765**
Vishal Ladhar	**17BCE0453**
Jyotirmaya Padhy	**17BCE2294**
Potluri Sri Varun	**17BCE0133**

Faculty:- Prof .Vijaya Kumar K

Slot-F1

School of Computer Science &Engineering

VIT®

Vellore Institute of Technology

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

ACKNOWLEDGEMENTS

I sincerely thank Dr.G.Viswanathan - Chancellor, VIT University, for creating an opportunity to use the facilities available at VIT. I also thank Prof. Vijaya Kumar K - Department of Computer Science, VIT University, for giving us the opportunity to do this project. I also thank the Dean and entire department of Computer Science, School of Computer Science and Engineering, for giving us this opportunity.

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.

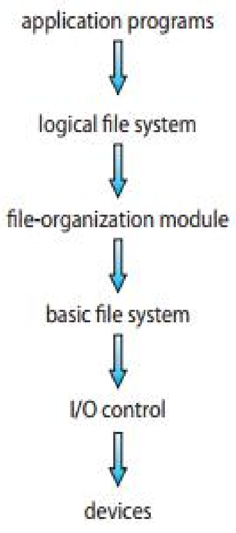
Taking its name from the way paper-based information systems are named, each group of data is called a "file". The structure and logic rules used to manage the groups of information and their names is called a "file system".

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.

FILE SYSTEM STRUCTURE



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.

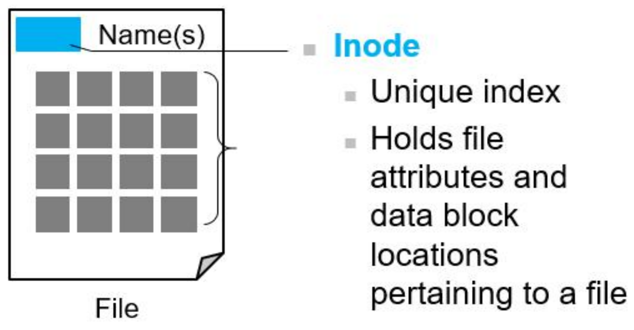
The I/O control level consists of device drivers and interrupt handlers to transfer information between the main memory and the disk system. A device driver can be thought of as a translator. Its input consists of high-level commands such as “retrieve block 146.” Its output consists of low-level, hardware-specific 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.

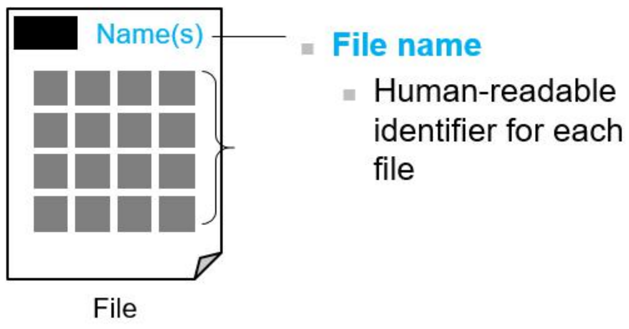
Finally, the logical file system manages metadata information. Metadata includes all of the file-system structure except the actual data (or contents of the files). The logical file system manages the directory structure to provide the file-organization module with the information the latter needs, given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

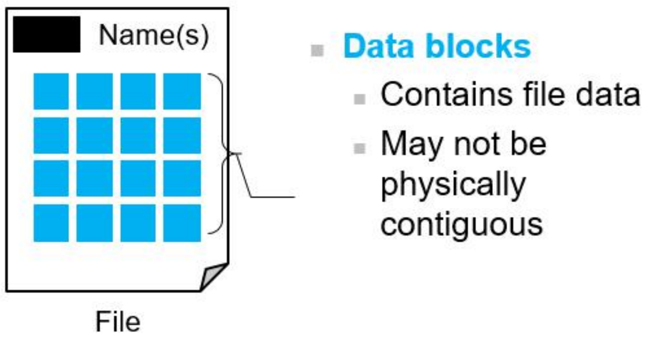
PHYSICAL FILE REPRESENTATION



- **Inode**
  - Unique index
  - Holds file attributes and data block locations pertaining to a file



- **File name**
  - Human-readable identifier for each file



- **Data blocks**
  - Contains file data
  - May not be physically contiguous

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 user readable identifier for each file. Now, the data blocks contain file data: may be contiguous or not contiguous.

FILE SYSTEM IMPLEMENTATION

Several on-disk and in-memory structures are used to implement a file system. These structures vary depending on the operating system and the file system, but some general principles apply. On disk, the file system may contain information about how to boot an operating system stored there, the total number of blocks, the number and location of free blocks, the directory structure, and individual files.

- A boot control block (per volume) can contain information needed by the system to boot an operating system from that volume. If the disk does not contain an operating system, this block can be empty. It is typically the first block of a volume. In UFS, it is called the boot block. In NTFS, it is the partition boot sector.
- A volume control block (per volume) contains volume (or partition) details, such as the number of blocks in the partition, the size of the blocks, a free-block count and free-block pointers, and a free-FCB count and FCB pointers. In UFS, this is called a superblock. In NTFS, it is stored in the master file table.
- A directory structure (per file system) is used to organize the files. In UFS, this includes file names and associated inode numbers. In NTFS, it is stored in the master file table.
- A per-file FCB contains many details about the file. It has a unique identifier number to allow association with a directory entry. In NTFS, this information is actually stored within the master file table, which uses a relational database structure, with a row per file.

The in-memory information is used for both file-system management and performance improvement via caching. The data are loaded at mount time, updated during file-system operations, and discarded at dismount. Several types of structures may be included.

An in-memory mount table contains information about each mounted volume. An in-memory directory-structure cache holds the directory information of recently accessed directories. The system-wide open-file table contains a copy of the FCB of each open file, as well as other information. The per-process open-file table contains a pointer to the appropriate entry in the system-wide open-file table, as well as other information. Buffers hold file-system blocks when they are being read from disk or written to disk.

To create a new file, an application program calls the logical file system. The logical file system knows the format of the directory structures. To create a new file, it allocates a new FCB.

Now that a file has been created, it can be used for I/O. First, though, it must be opened. The open() call passes a file name to the logical file system. The open() system call first searches the system-wide open-file table to see if the file is already in use by another process. If it is, a per-process open-file table entry is created pointing to the existing system-wide open-file table. This algorithm can save substantial overhead. If the file is not already open, the directory structure is searched for the given file name. Parts of the directory structure are usually cached in memory to speed directory operations. Once the file is found, the FCB is copied into a system-wide open-file table in memory. This table not only stores the FCB but also tracks the number of processes that have the file open.

Next, an entry is made in the per-process open-file table, with a pointer to the entry in the system-wide open-file table and some other fields. These other fields may include a pointer to the current location in the file (for the next read() or write() operation) and the access mode in which the file is open. The open() call returns a pointer to the appropriate entry in the per-process file-system table. All file operations are then performed via this pointer.

The file name may not be part of the open-file table, as the system has no use for it once the appropriate FCB is located on disk. It could be cached, though, to save time on subsequent opens of the same file. The name given to the entry varies. UNIX systems refer to it as a file descriptor; Windows refers to it as a file handle.

When a process closes the file, the per-process table entry is removed, and the system-wide entry's open count is decremented. When all users that have opened the file close it, any updated metadata is copied back to the disk-based directory structure, and the system-wide open-file table entry is removed.

PARTITIONS AND MOUNTING

The layout of a disk can have many variations, depending on the operating system. A disk can be sliced into multiple partitions, or a volume can span multiple partitions on multiple disks.

Each partition can be either "raw," containing no file system, or "cooked," containing a file system. Raw disk is used where no file system is appropriate. UNIX swap space can use a raw partition, for example, since it uses its own format on disk and does not use a file system. Likewise, some databases use raw disk and format the data to suit their needs. Raw disk can also hold information needed by disk RAID systems, such as bit maps indicating which blocks are mirrored and which have changed and need to be mirrored. Similarly, raw disk can contain a miniature database holding RAID configuration information, such as which disks are members of each RAID set.

Boot information can be stored in a separate partition. Again, it has its own format, because at boot time the system does not have the file-system code loaded and therefore cannot interpret the file-system format. Rather, boot information is usually a sequential series of blocks, loaded as an image into memory. Execution of the image starts at a predefined location, such as the first byte. This boot loader in turn knows enough about the file-system structure to be able to find and load the kernel and start it executing. It can contain more than the instructions for how to boot a specific operating system. For instance, many systems can be dual-booted, allowing us to install multiple operating systems on a single system. How does the system know which one to boot? A boot loader that understands multiple file systems and multiple operating systems can occupy the boot space. Once loaded, it can boot one of the operating systems available on the disk. The disk can have multiple partitions, each containing a different type of file system and a different operating system.

The root partition, which contains the operating-system kernel and sometimes other system files, is mounted at boot time. Other volumes can be automatically mounted at boot or manually mounted later, depending on the operating system. As part of a successful mount operation, the operating system verifies that the device contains a valid file system. It does so by asking the device driver to read the device directory and verifying that the directory has the expected format. If the format is invalid, the partition must have its consistency checked and possibly corrected, either with or without user intervention. Finally, the operating system notes in its in-memory mount table that a file system is mounted, along with the type of the file system. The details of this function depend on the operating system.

Microsoft Windows–based systems mount each volume in a separate name space, denoted by a letter and a colon. To record that a file system is mounted at G: , for example, the operating system places a pointer to the file system in a field of the device structure corresponding to F:. When a process specifies the driver letter, the operating system finds the appropriate file-system pointer and traverses the directory structures on that device to find the specified file or directory. Later versions of Windows can mount a file system at any point within the existing directory structure.

On UNIX, file systems can be mounted at any directory. Mounting is implemented by setting a flag in the in-memory copy of the inode for that directory. The flag indicates that the directory is a mount point. A field then points to an entry in the mount table, indicating which device is mounted there. The mount table entry contains a pointer to the superblock of the file system on that device. This scheme enables the operating system to traverse its directory structure, switching seamlessly among file systems of varying types.

VIRTUAL FILE SYSTEMS

Data structures and procedures are used to isolate the basic system call functionality from the implementation details. Thus, the file-system implementation consists of three major layers, as depicted schematically here. The first layer is the file-system interface, based on the open (), read (), write (), and close () calls and on file descriptors. The second layer is called the virtual file system (VFS) layer. The VFS layer serves two important functions:

- It separates file-system-generic operations from their implementation by defining a clean VFS interface.
- It provides a mechanism for uniquely representing a file throughout a network. The VFS is based on a file-representation structure, called a vnode that contains a numerical designator for a network-wide unique file.

Thus, the VFS distinguishes local files from remote ones, and local files are further distinguished according to their file-system type. The VFS activates file-system-specific operations to handle local requests according to their file-system types and calls the NFS protocol procedures for remote requests. File handles are constructed from the relevant vnodes and are passed as arguments to these procedures. The layer implementing the file-system type or the remote-file-system protocol is the third layer of the architecture.

Virtual File System supports file systems based on local storage (like block-based, File systems in userspace: FUSE, Specialized storage files, and Memory file systems), network storage (like NFS, Coda, AFS, CIFS, NCP), and Special file systems (procfs, sysfs).

The most common file system interface involves topmost layer to be applications which communicate with virtual file system. The virtual file system is the abstraction layer over a more concrete file system. The file system interacts with multi-device drivers which in turn access the disk drivers to open, read, or write.

Let’s briefly examine the VFS architecture in Linux. The four main object types defined by the Linux VFS are:

1. The inode object, which represents an individual file
2. The file object, which represents an open file
3. The superblock object, which represents an entire file system
4. The dentry object, which represents an individual directory entry

For each of these four object types, the VFS defines a set of operations that may be implemented. Every object of one of these types contains a pointer to a function table. The function table lists the addresses of the actual functions that implement the defined operations for that particular object. For example, an abbreviated API for some of the operations for the file object includes:

- int open(. . .) – Open a file.
- int close(...) – Close an already-open file.
- ssize t read(. . .) – Read from a file.
- ssize t write(. . .) – Write to a file.
- int mmap(. . .) – Memory-map a file.

An implementation of the file object for a specific file type is required to implement each function specified in the definition of the file object.

Thus, the VFS software layer can perform an operation on one of these objects by calling the appropriate function from the object's function table, without having to know in advance exactly what kind of object it is dealing with. The VFS does not know, or care, whether an inode represents a disk file, a directory file, or a remote file. The appropriate function for that file's read() operation will always be at the same place in its function table, and the VFS software layer will call that function without caring how the data are actually read.

CODE FOR IMPLEMENTATION OF VFS

1. **file.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
```

2. **filesys.h**

```
#define MAX_FILE_LEN 1000
#define MAX_FILES 15
#ifndef fs
#define fs
#include "file.h"
#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
```

3. **file.cpp**

```
#include "file.h"
#include "filesys.h"
#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;
}
```

4. **filesys.cpp**

```
#include "filesys.h"
#include "file.h "
#include<iostream>
#include<fstream>
#include<string.h>
#include<stdio.h>
#include<string>
#include<math.h>
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:";
```

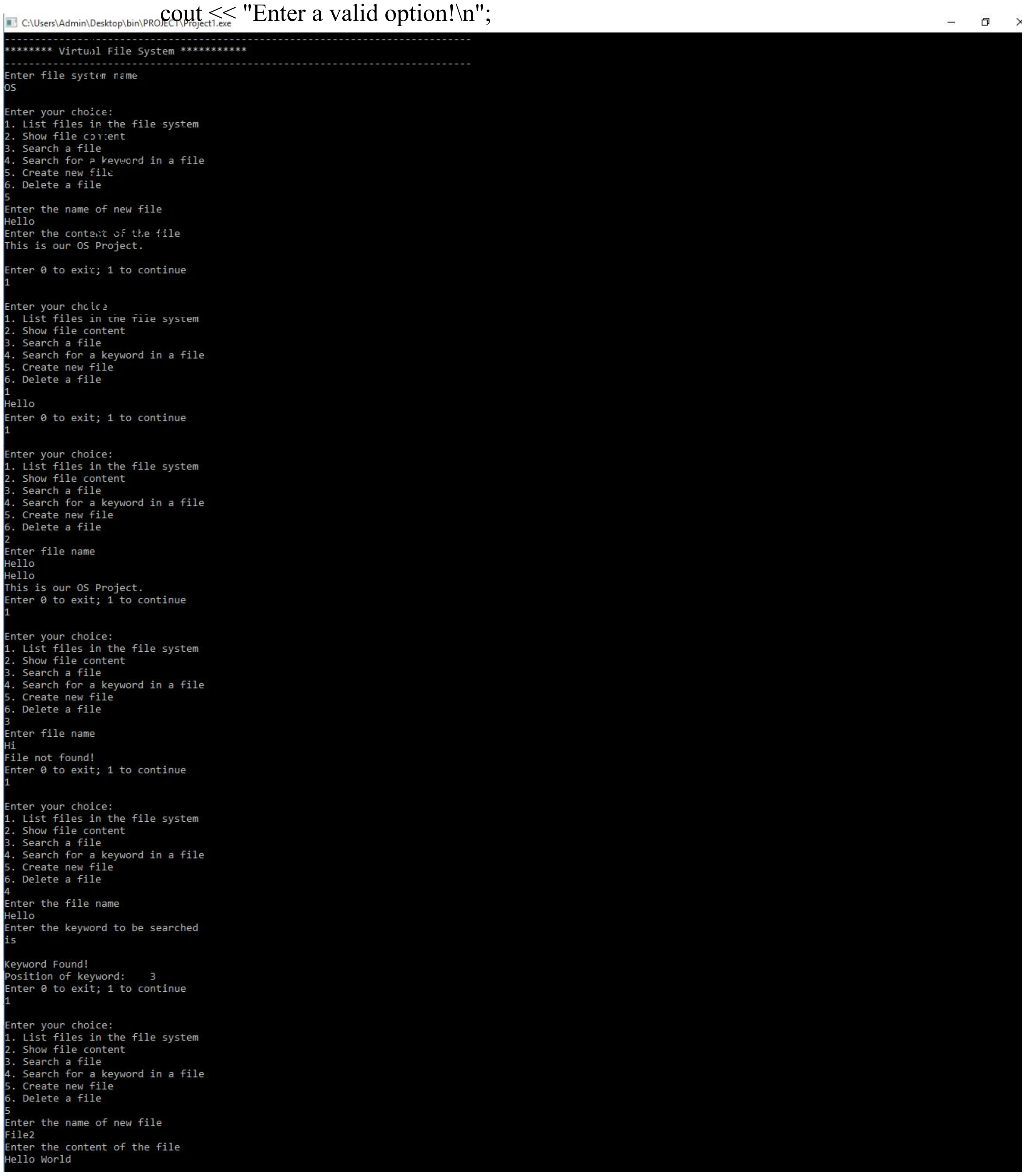
```
return files[i].get_file_name();
}
}
cout<<"File not found!";
return NULL;
}
void filesys :: search_keyword(char* f_name, char* keyword){
fstream myfile(file_system_name, ios::in);
for(int i = 0; i < MAX_FILES; i++){
if(!strcmp(f_name, files[i].get_file_name())){
char* file_content = new char[MAX_FILE_LEN];
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);
}
```

5. main.cpp

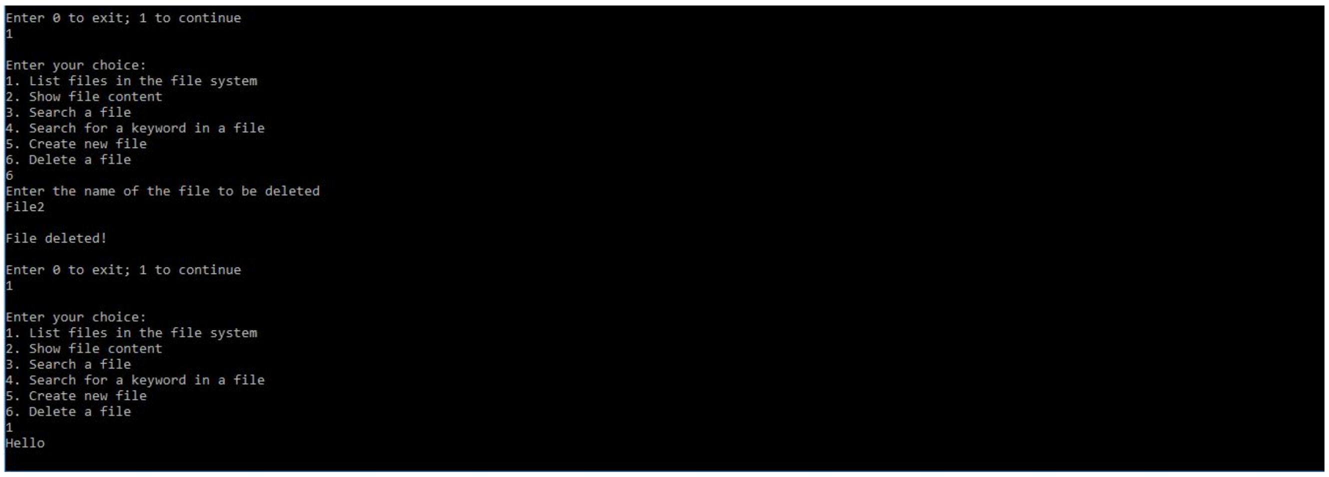
```
#include <iostream>
#include<fstream>
#include "filesys.h"
#include "file.h"
#include<string.h>
using namespace std;
int main(int argc, char* argv[]) //Enter command line arguments to open a previously
created file system // or leave blank to create a new one{
char name[80], content[200], name1[80], 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";
```



C:\Users\Admin\Desktop\bin\PROJECT\Project1.exe

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

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
Hello
Enter the content of the file
This is our OS Project.

Enter 0 to exit; 1 to continue
1

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
Hello
Enter 0 to exit; 1 to continue
1

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
Hello
Hello
This is our OS Project.
Enter 0 to exit; 1 to continue
1

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
Hi
File not found!
Enter 0 to exit; 1 to continue
1

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
Hello
Enter the keyword to be searched
is

Keyword Found!
Position of keyword:    3
Enter 0 to exit; 1 to continue
1

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
File2
Enter the content of the file
Hello World
```



```
Enter 0 to exit; 1 to continue
1

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
File2

File deleted!

Enter 0 to exit; 1 to continue
1

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
Hello
```

To display the block address and FAT table:-

```
#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 create() //to create a 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;
```

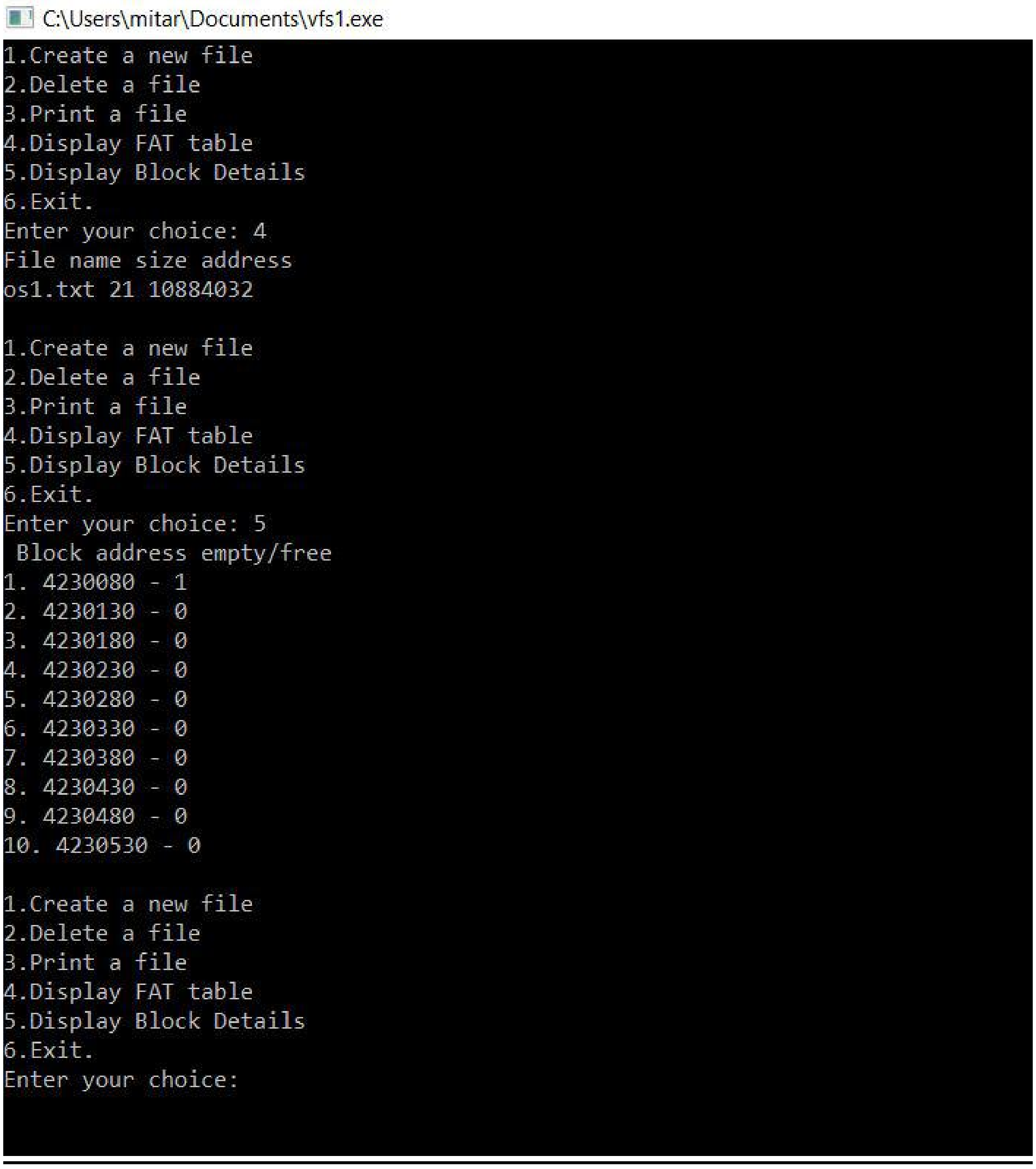
```
}
printf("The size of given file is : %d\n", size);
if(mem>=size)
{
int n=1,parts=0,m=1;
while(address[n]!=0)
n++;
strcpy(name[n],fname);
s_b[n][1]=size;
int bnum=size/50;
if(size%50!=0)
bnum=bnum+1;
s_b[n][2]=bnum;
mem=mem-(bnum*50);
int *bfile=(int*)malloc(bnum*(sizeof(int)));
address[n]=bfile;
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 created\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()
{
int choice,i;
char *temp;
if (buffer == NULL)
{
fputs ("Memory error",stderr);
exit (2);
}
temp=buffer;
block[1]=buffer;
empty[1]=0;
for(i=2;i<=10;i++)
{
block[i]=block[i-1]+50;
empty[i]=0;
}
while(1)
{
```

```
printf("1.Create a new 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.\n");
printf("Enter your choice: ");
scanf("%d",&choice);
switch(choice)
{
case 1:
create();
break;
case 2:
remove1();
break;
case 3:
print();
break;
case 4:
file();
break;
case 5:
blocks();
break;
case 6:
exit(1);
}
}
return 0;
```

}



```
C:\Users\mitar\Documents\vfs1.exe
1.Create a new file
2.Delete a file
3.Print a file
4.Display FAT table
5.Display Block Details
6.Exit.
Enter your choice: 4
File name size address
os1.txt 21 10884032

1.Create a new file
2.Delete a file
3.Print a file
4.Display FAT table
5.Display Block Details
6.Exit.
Enter your choice: 5
 Block address empty/free
1. 4230080 - 1
2. 4230130 - 0
3. 4230180 - 0
4. 4230230 - 0
5. 4230280 - 0
6. 4230330 - 0
7. 4230380 - 0
8. 4230430 - 0
9. 4230480 - 0
10. 4230530 - 0

1.Create a new file
2.Delete a file
3.Print a file
4.Display FAT table
5.Display Block Details
6.Exit.
Enter your choice:
```

CONCLUSION

Here we implemented our own Virtual File System in C++ Programming Language. We created our own *header* files to perform the file-system functions. Thus, the given paper helps you to learn theoretically as well as practically the concepts of Virtual File System.

File-system research continues to be an active area of operating-system design and implementation. Google created its own file system to meet the company's specific storage and retrieval needs, which include high performance access from many clients across a very large number of disks. Another interesting project is the FUSE file system, which provides flexibility in file-system development.

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