SLOWER FILE SYSTEM

Report on package Submitted by

Aadhith Narayan, 18PT01
Lakshminarayanan, 18PT20

Subject: Operating System

Department of Applied Mathematics and Computational Sciences
PSG College of Technology
Coimbatore, 641004.

Contents

1.1 Introduction

What is a file system?

A filesystem is the methods and data structures that an operating system uses to keep track of files on a disk or partition; that is, the way the files are organized on the disk.
Data on a system is organized in the form of files. This helps to make the information be partitioned logically hence making it easy to identify and separate it.

A file system is used to control how data is stored and retrieved. Without a file system, information placed in a storage medium would be one large body of data with no way to tell where one piece of information stops and the next begins. By separating the data into pieces and giving each piece a name, the information is easily isolated and identified. 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".

What is a file?

In general, a file is a named collection of related information that is recorded on secondary storage such as magnetic disks, magnetic tapes and optical disks. In general, a file is a sequence of bits, bytes, lines or records whose meaning is defined by the files creator and user.

1.2 Description

We have implemented a miniature file system to help us understand how the file system works, specifically storage management and to understand some of the performance issues the file system we deal with.

The implemented file system is a fixed block contiguous file allocation where data is stored in each block object of fixed size. We have designed the file system to have three classes: Disk, File, Block. There are two parts to our API: A set of calls that deals with file access and an application that links with our file system.

1.3 Tools and Technology

We used C++ to build the API and we are heavily dependent on the OOPS concept as we used class objects to store the data in the file system, and also used Inheritance to manage the stored data using the Disk Class.

1.4 Workflow

The API calls are as follows:

1. **CREATE <FILENAME>**

   "Filename" is a full file name in the form described above.
   If CREATE is given for an existing file, no error should occur, but the file should have deleted and then recreated.

```
void Disk::create(string name)
{
    string original;
    original = name;

        if (isOpen() == true) {
        cout << "Error: Create " << name << " failed because " << sector[openBlock]->getName() << " is opened. Please close it first before crea
    }
    else {

            Block* newBlock;
            newBlock = helpCreate(name);
            if (newBlock != nullptr) {
                cout << "Finished create " << name << endl;
            }
            else {
                cout << "Error: Create " << name << " failed. ALL the sectors are used." << endl;
            }
        }
}
```

Fig 1.4.1 Create Function.

2. OPEN <FILENAME>

Associated with each open file is a pointer to the next byte to be read or written. Opening a file for input or update places the pointer at the first byte of the file, while opening a file for output places the pointer athe the byte immediately after the last byte of the file.

```
void Disk::open(string name)
{
    if (isOpen() == true) {
        cout << "Error: Open " << name << " failed because " << sector[openBlock]->getName() << " is opened. Please close it first before openin
    }

    else {
        Block* file;
        file = findBlock(name);

        if (file == nullptr) {
            cout << "Error: Open " << name << " failed because file name is not valid. Please try again." << endl;
        }
        else {
            openBlock = file->getNumber();
            cursor = 0;
            openMode = 2;
            seek(-1, 0);
        }
    }

}
```

Fig 1.4.2 Open Function.

3. CLOSE

This command causes the last opened or created file to be closed. No filename is given.

4. READ n

This command may only be used between an OPEN (in input or update mode) and the corresponding CLOSE.
If possible, "n" bytes of data should be read and displayed. If fewer than "n" bytes remain before the end of file, then those bytes should be read and displayed with a message indicating that the end of file was reached.

```
void Disk::read(int count)
{
    if (isOpen() == false) {
        cout << "Error: Read failed because no file is opened." << endl;
    }
    else {
        int current, blockNum;
        Block* next;

        current = cursor;
        blockNum = openBlock;
        next = sector[openBlock];

        while (current >= FILE_SIZE && next->getFrwd() != nullptr) {
            next = next->getFrwd();
            openBlock = next->getNumber();
            current -= FILE_SIZE;
        }

        if (current >= FILE_SIZE) {
            cout << "Error: Read failed because the cursor is getting beyond the size of " << sector[openBlock]->getName() << endl;
        }
        else {
            int totalRead, readed;
            totalRead = current + count;
            next = sector[openBlock];
            readed = 0;

            while (totalRead >= FILE_SIZE) {
                current = current % FILE_SIZE;
                if (FILE_SIZE > count) {
                    readed = count - current;
                }
```

Fig 1.4.3 Read Function.

```
                else {
                    readed = FILE_SIZE - current;
                }

                ((File*)sector[openBlock])->readFile(count, current);
                totalRead -= readed;

                next = sector[openBlock]->getFrwd();
                if (next != nullptr) {
                    openBlock = next->getNumber();
                }
                else {
                    break;
                }
            }

            if (totalRead < FILE_SIZE && next != nullptr) {
                current = current % FILE_SIZE;
                ((File*)sector[openBlock])->readFile(count, current);
                cout << "(EOF)" << endl;
                if (count > 0) {
                    cout << "\nEnd of file is reached." << endl;
                }
            }
            else {
                cout << "(EOF)" << endl;
                cout << "\nEnd of file is reached." << endl;
            }
        }
        openBlock = blockNum;
    }
}
```

5. WRITE n ‘data’

This command causes the first "n" ata bytes from 'data'(actually enclosed in single quotes in the command line) to be written to the file.
If fewer than "n" bytes are given, then append sufficient blanks to 'data' to make "n" bytes.

If it is impossible to write "n" bytes(because the disk is full) then an appropriate message should be issued, but the command should be otherwise treated as if the largest possible value of "n" was specified. (That is, the remaining available disk space should be filled.

```
void Disk::write(int count, string input)
{
    if (isOpen() == false) {
        cout << "Error: Write failed because no file is opened." << endl;
    }
    else {
        int current, blockNum;
        File* next;

        current = cursor;
        blockNum = openBlock;
        next = (File*)(sector[openBlock]->getFrwd());


        while (current >= FILE_SIZE && next != nullptr) {
            openBlock = next->getNumber();
            current -= FILE_SIZE;
            next = (File*)(sector[openBlock]->getFrwd());
        }

        if (current >= FILE_SIZE) {
            cout << "Error: Write failed because the cursor is getting beyond the size of " << sector[openBlock]->getName() << endl;
        }
        else {
            helpWrite(count, input, current);
        }
        openBlock = blockNum;
    }
}
```

Fig 1.4.4 Write Function.

6. SEEK base offset

"base" is either -1, 0, or +1 indicating the beginning of the file, the current position in the file, or the end of file.
"offset" is an integer indicating the number of bytes from the "base" that the file pointer should be moved.

```
void Disk::seek(int base, int offset)
{
    if (isOpen() == false) {
        cout << "Error: Seek failed because no file is opened." << endl;
    }
    else {

        if (base == -1 && offset >= 0) {
            cursor = 0;
            cursor += offset;
        }
        else if (base == -1 && offset < 0) {
            cout << "Error: Seek failed. Can't go backward when reach the beginning of the file." << endl;
        }

        else if (base == 0) {
            if (cursor + offset < 0) {
                cout << "Error: Seek failed. Can't go backward when reach the beginning of the file." << endl;
            }
            else {
                cursor += offset;
            }
        }

        else if (base == 1) {
            if (((File*)sector[openBlock])->getEnd() + offset < 0) {
                cout << "Error: Seek failed. Can't go backward when reach the beginning of the file." << endl;
            }
            else {
                cursor = ((File*)sector[openBlock])->getEnd() + offset;
            }
        }
    }
}
```

Fig 1.4.5 Seek Function.

1.5 Result

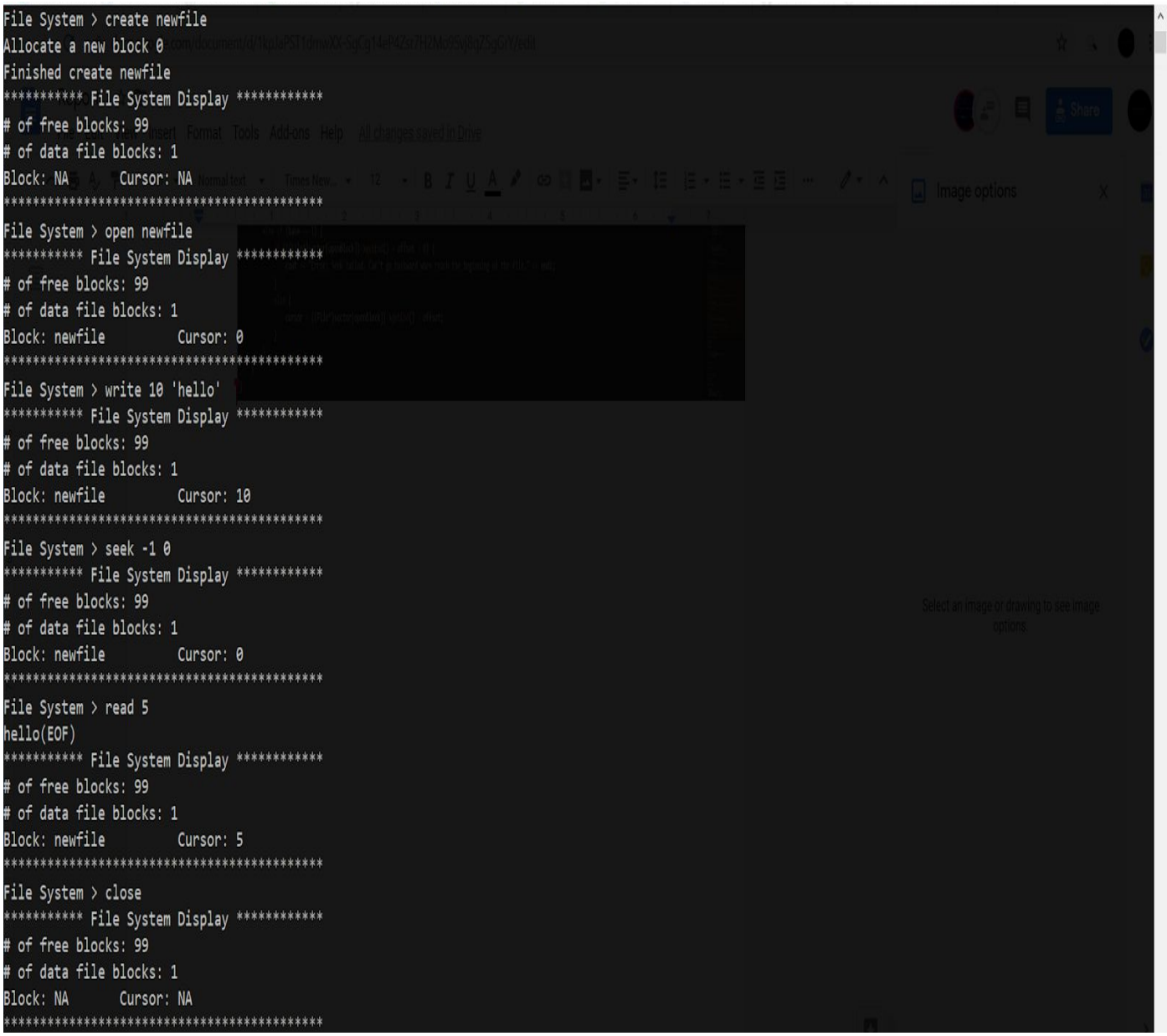

Fig 1.5.1 File System Output.

1.6 Conclusion

File management is one of the important parts in OS functionality.
From this project, we have successfully learned to implement a miniature file system and understood the practical implementation of file allocation methods.

1.7 Bibliography

The following web references have been used while working on this project.

1. http://pages.cs.wisc.edu/~dusseau/Classes/CS537-F07/Projects/P4/i ndex.html
2. http://www.tldp.org/LDP/intro-linux/html/sect_03_01.html
3. http://www.tldp.org/LDP/sag/html/filesystems.html
4. https://en.wikipedia.org/wiki/File_system