PROJECT REPORT

SLOWER FILE SYSTEM

November 14,2017

SHIVAM AGRAWAL (20172074)
RAJAT AGARWAL (20172065)
DIVY SITLANI (20172040)
TEJUS AGARWAL (20172066)

Contents

PROJECT GOAL:

There are two objectives to this project:

1. To understand how file systems work, specifically the directory hierarchy and storage management.
2. To understand some of the performance issues file systems deal with.

Major Objectives:

1. Fully understand the Disk implementations provided in LibDisk.c and LibDisk.h files.
2. Load the disk image in the main memory. If the disk does not exist, then create a file to hold the new disk image and initialise that disk using disk set function.
3. Then according to the main function, call the functions like File_Create, etc.

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".

File and Directory

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.

Information about files is maintained by **Directories**. A directory is a special file that can contain multiple files. It can even have directories inside of them.
In this project, we have build a user-level library, libFS, that implements a good portion of a file system. Our file system is built inside of a library (LibFS) that applications can link with to access files and directories.

There are three parts to the LibFS API: two generic file system calls, a set of calls that deal with file access, and a set of calls that deal with directories, and applications that will link with LibFS in order to test out your file system.

Generic File System API:

- **FS_Boot**
  FS_Boot should be called exactly once before any other LibFS functions are called.

```
int FS_Boot(char *path)
{
    strcpy(disk_name, path);
    cout<<"disk name:" << disk_name<<endl;
    printf("FS_Boot %s\n", disk_name);
    if( access( disk_name, F_OK ) != -1 )
    {
        printf("The Disk already exists\n");
    }
    else
    {
        if (Disk_Init() == -1) {
        printf("Disk_Init() failed\n");
        osErrno = E_GENERAL;
        return -1;
        }
       Disk_Save(disk_name);
       disk_Set();
    }
    FS_Sync();
    return 0;
}
```

- **FS_Sync**
  It is used to make sure the contents of the file system are stored persistently on disk.

File Access API:

- **File_Create**
  Check if file is already present ,then it will return the file if not then it will create a empty file. To write the file we have to call file write function.

```
nextin = free_Inode_list.back();
free_Inode_list.pop_back();
nextdb = free_Db_list.back();
free_Db_list.pop_back();
inode_arr[nextin].pointer[0]=nextdb;


inode_arr[nextin].filesize = 0;

directory_map[string(filename)] = nextin;

strcpy(dir_struct_entry[nextin].file_name,filename);
dir_struct_entry[nextin].inode_num = nextin;
```

- **File_Open**
  It will take one free file descriptor from the free file descriptor list and assign this descriptor to the same file. We are taking maximum of 255 file descriptor.

```
int inode = directory_map[name];
int fd = free_FD_list.back();
free_FD_list.pop_back();
fileDescriptor_map[fd].first=inode;
fileDescriptor_map[fd].second = 0;
openfile_count++;
printf("File descriptor %d\n",fd);

return fd;
```

- **File_Read**
  We will search the filename into map of file directory list. It will give the inode number corresponding to filename, if file exist. From the inode number we can find the block pointers. These block pointer will contain the data block pointer if file is not empty.
  We can access the data blocks and start reading it and finally we will write this file into a file of the same name.

- **File_Write**
  We first read the content of the file which we need to write into disk. It will read the content from the buffer then it will find the size of block needed to write. We will search the free inode block using inode bit vector, which we already loaded into free inode list .
  We will start reading one by one free data block pointer and will set the value of data block into inode structure. We can write the file of maximum size of 30 blocks.
  Then we find the number of block needed to write the data and we will start writing it block by block.

- **File_Close**
  While closing the file, it will just free the file descriptor and the free file descriptor back to file descriptor list.
- **File_Seek**
  We will read the file data till the position of seek and set the pointer at that position.

Directory API:

- **Dir_Create**
  Dir_Create creates a new directory whose path is is given in its argument.

- **Dir_Size**
  Dir_Size() returns the number of bytes in the directory referred by path. It adds the size of all the contents of the directory(files and directories).

- **Dir_Read**
  Dir_Read() is used to read the contents of a directory. It will return a set of directory entries in the buffer.

The Disk Abstraction

A real file system would store all the file system data on disk, but since we are writing this all at user-level, we will store it all in a "fake" disk which is already provided to us in the form of two files with disk implementation as LibDisk.c LibDisk.h .The model of the disk is quite simple: in general, the file system will perform disk reads and disk writes to read or write a sector of the disk. In actuality, the disk reads and writes access an in-memory array for the data;

We have the following disk API's:

- Disk_Init
- Disk_Load
- Disk_Save
- Disk_Write
- Disk_Read

DATA STRUCTURE DESIGN

- **Directory Map**
  map <string , int> - it will map the filename to the inode number

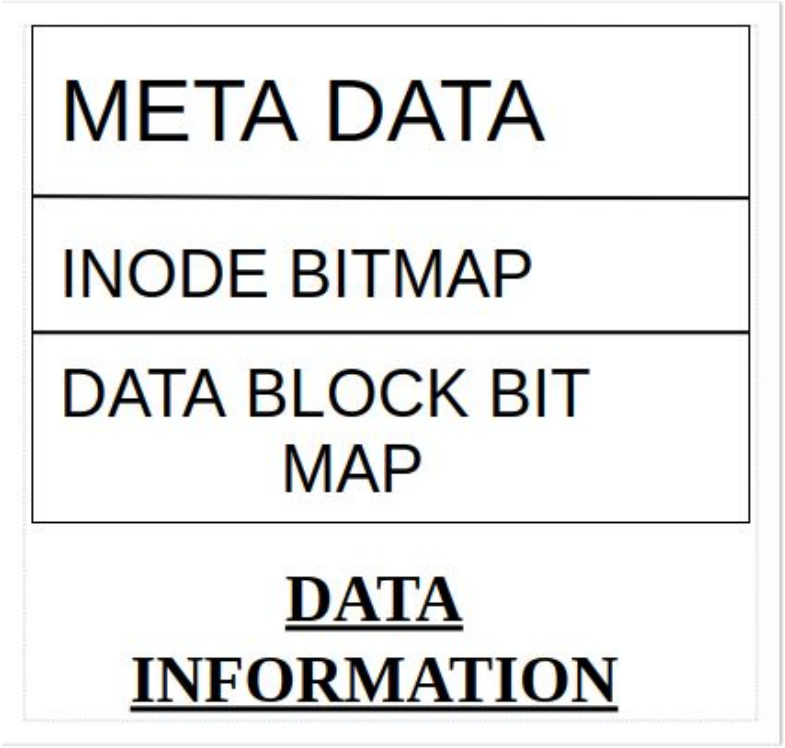
- **free_Db_List**
  Vector <int> - it will store all the free disk block number

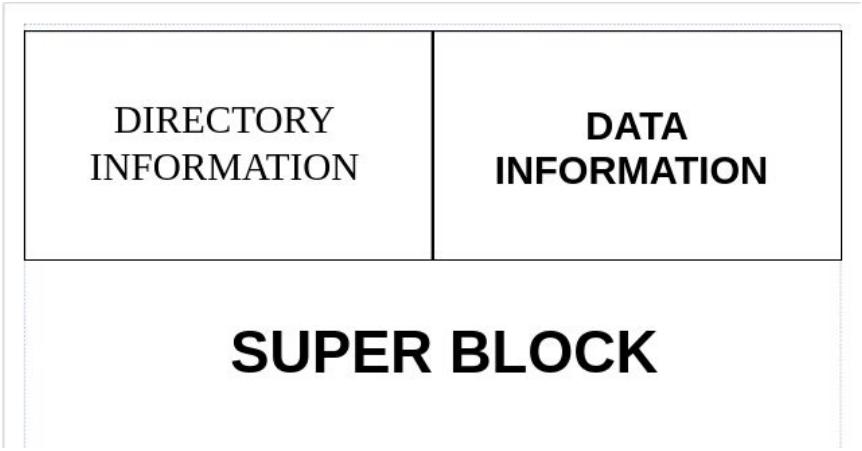
- f**ree_Inode_List**
  vector<int>- it will store all the free inode numbers available

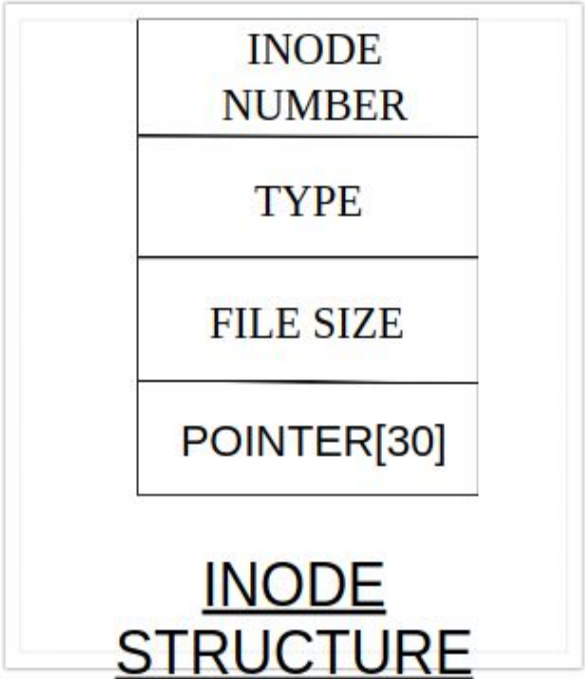
- **Free_fd_list**
  vector<int> - it will give all the free file descriptor available in the system

- **fileDescriptor_map**
  Map <int, pair<int,int> > - it will store key as file descriptor and value as pair of inode number and file pointer

- Struct super_block
  {
    firstDbOfDir - it will give first disk block after superblock.noOfDbUsedinDir - it will give number of disk block needed to store dir structure array.
    startingInodesDb - it will give disk block number from where inodes are starts.
    total_Usable_DB_Used_for_inodes - it will gives number of disk blocks used to maintain.
    starting_DB_Index- it will give disk block number from where DB are stored.
    total_Usable_DB - it will give total number of disk block which can be used as data block.
    inode_Bitmap[NO_OF_INODES] - it will store whether inodes is free or not.
    datablock_Bitmap[DISK_BLOCKS] - it will store whether data block is free or not.
  }

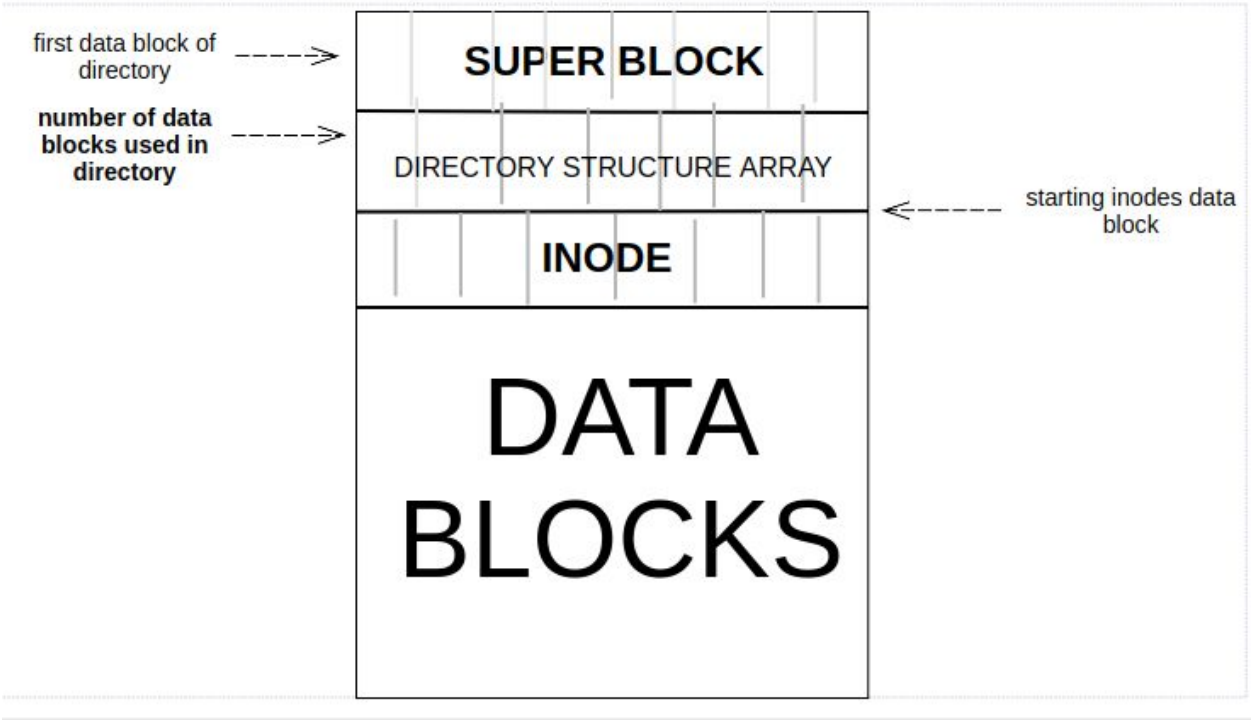

META DATA
INODE BITMAP
DATA BLOCK BIT MAP
DATA INFORMATION


DIRECTORY INFORMATION
DATA INFORMATION
SUPER BLOCK



Creation of Disk

- It will create a disk of size NUM_BLOCK * SIZE_OF_BLOCK
- As per specification we have taken number of blocks = 1024 bytes and size of block = 32KB.
-We are creating a binary disk file of above size.
-If the disk is already present in the system then it will load the disk.




```
Terminal  File  Edit  View  Search  Terminal  Help
rajat@rajat-SVE15117FNB:~/Dropbox/slower_file_system/unintialised/code$ make
g++ -std=c++11 -o  main main.cpp LibFS.cpp LibFS.h LibDisk.h LibDisk.cpp
rajat@rajat-SVE15117FNB:~/Dropbox/slower_file_system/unintialised/code$ ./main a
a
disk name:aa
FS_Boot aa
Virtual Disk aa created sucessfully
FS_Sync
Disk is mounted now
Disk Initialized
------------------------------------------------------------
Please enter your choice
0.Exit
1.File Create   2.File Open     3.File Read     4.File Write
5.File Seek     6.File Close    7.File Unlink
8.Dir Create    9.Dir Read      10.Dir Unlink
------------------------------------------------------------
```

Directory create-
We will choose one inode number and set the type of inode as Directory and in the pointer we will store the inode number of sub directories. On creation we set this as NULL(-1).

Some Questions

Q. How to handle internal and external fragmentation?
A.We are dividing the “disk memory” into fixed size blocks so no chance for external fragmentation.
A new block will be allocated to the file when previously allocated block has filled completely. This allows only very small fraction of internal fragmentation only on the last block of the file.

Q. **How to load big files?**
A.We can divide the whole file into small chunks and read as many chunks as possible. Then as needed, we can replace the chunk data from disk into memory.However, we have already given the maximum file size fixed as 30 blocks.
We can also use the concept of indirect pointers. We can take some pointer to store data pointers which itself contain block pointers.

REFERENCES

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

1. http://pages.cs.wisc.edu/~dusseau/Classes/CS537-F07/Projects/P4/index.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

Other than the specific websites which have been mentioned here, a lot other websites were visited in order to answer our various queries.