Habib University
Operating Systems - CS232

Assignment 01 - Report

Instructor: Munzir Zafar

Ali Muhammad Asad - aa07190

Contents

1 Introduction

The assignment is to simulate a simple file system in C language. The file system is simulated using a single file **"myfs"** which acts as the hard disk. The whole disk is 128KB in size, has the root directory **"/"**, and can have a maximum of 16 files/directories (therefore 16 inodes). A file can have a maximum of 8 blocks, each of size 1KB.

2 Disk Layout

The disk is divided into 128 blocks, each of size 1KB. The first block is the super block, and the rest 127 blocks are the data blocks. The super block contains the 128 byte free block list where each byte contains a boolean value indicating whther that particular block is free or not. Just after the free block list, the inodes are stored - in total there are 16 inodes, where each inode corresponds to one file or directory. Each inode is 56 bytes into size and contains the metadata about the stored files/directories. The contents of the directory are a series of directory entries comprising of dirent structures (will be further explained later on). The disk layout can be visually represented as follows:

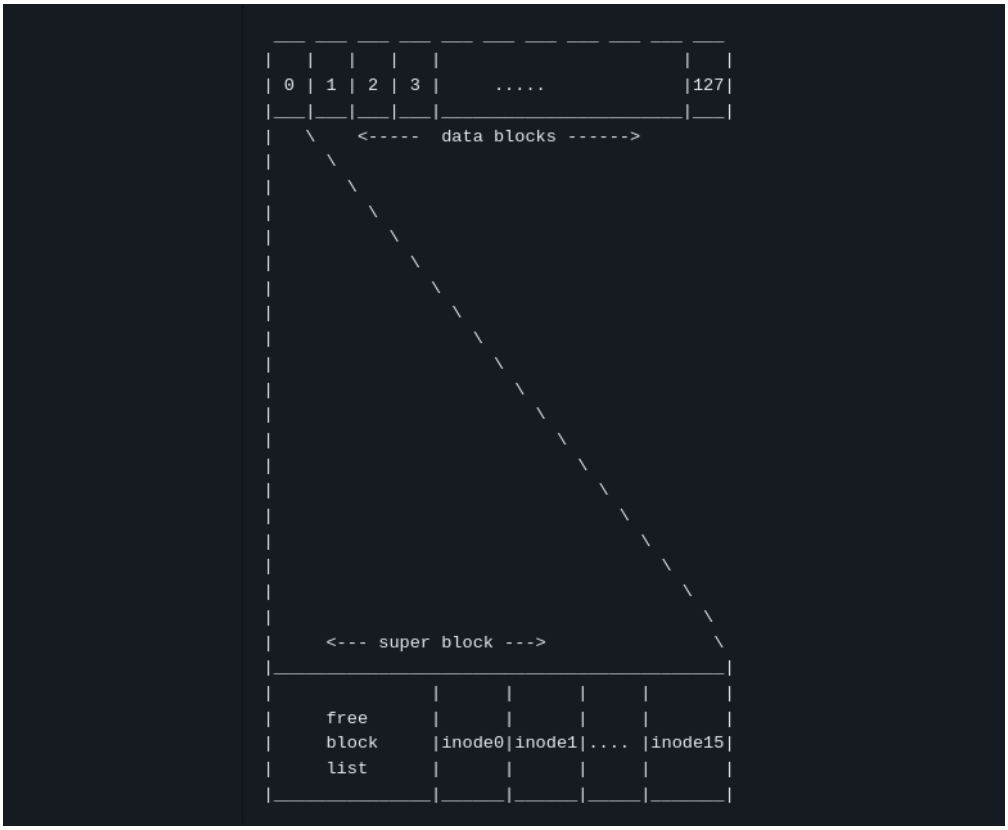

Figure 1: Disk Layout

3 Input and Makefile

3.1 Input

The program takes a command line argument which will be a file containing the commands to be executed. The commands are read by the program and executed one by one, and each command should be on a separate line. The input should always be in the correct format, i.e., `Command Arg1 Arg2` or `Command Arg`, otherwise the program will not work as expected. The input file should be in the same directory as the program.

3.2 Makefile

The accompanying makefile has also been provided with the program. The makefile supports the following commands:

- `make build:` builds the program and creates the executable file titled **"myfs.out"**.
- `make run:` runs the program with the default input file **"sampleinput.txt"**. The input file can be changed to any other file as needed. Running the program will also create the **"myfs"** file which acts as the hard disk.
- `make clean:` cleans the directory by removing the executable file ("myfs.out") and the disk file ("myfs").

4 Implementation

4.1 Data Structures

There are several data structures that are being used in this implementation to manage and store the "myfs" filesystem's data and key components.

1. **Super Block:** The super block, although not explicitly defined as a data structure, can be conceptually understood as a data structure storing the first few bytes of the **"myfs"** filesystem. The super block is stored in the first block of the disk and contains critical information about the file system such as the identification marker "A" (corresponding to the initial of my name as I implemented this file system) and the directory bitmap "1".

   It contains the free block list and the inodes. The free block list is an array of 128 bytes, where each byte represents a block in the disk. If the value of the byte is 0, then the block is free, otherwise it is not. The inodes are an array of 16 inodes, where each inode corresponds to a file or directory.

2. **Inode (struct inode):** The inode is a data structure - implemented using a struct - that stores the metadata of a file or directory. In my implementation, there are 16 inodes, stored consecutively in the disk just after the free block list in the file system. Inividual inodes are accessed using the `lseek` function to the appropriate position, followed by a subsequent read or write operation. Each inode is 56 bytes in size, and is defined as follows containing the following information:

```
typedef struct inode {
    int dir;                    // 1: directory, 0: file
    char name[FILENAME_MAXLEN]; // File / directory name
    int size;                   // File / directory size in bytes
    int blockptrs[8];           // Direct pointers to data blocks
    int used;                   // 1 if the entry is in use
    int rsvd;                   // Reserved for future use
} inode;

```

Listing 1: Inode Structure

3. **Directory Entry (struct dirent:)** The directory entry or *dirent* for short, is a data structure - implemented using a struct - that stores the information about a file or directory within a directory. These entries are stored within data blocks. Similar to the inodes, individual directory entries are accessed using the `lseek` function to the appropriate position, followed by a subsequent read or write operation. Each directory entry is 32 bytes in size, and is defined as follows containing the following information:

```
typedef struct dirent {
    char name[FILENAME_MAXLEN]; // Name of the entry
    int namelen;                // Length of entry name
    int inode;                  // Index of the corresponding inode
} dirent;

```

Listing 2: Dirent Structure

4. **Data Blocks:** Data blocks store the actual content of files and directories. Again, the data blocks are accessed using the `lseek` function to the appropriate position, followed by a subsequent read or write operation. Each data block is 1KB in size, made as 127 blocks in the disk. The data blocks are accessed using the blockpointers in the inode. The blockpointers are an array of 8 integers. The data blocks can be conceptually visualized as shown in Figure 1.

5. **MYFS - My File System:** The "myfs", although a file, is the fundamental data structure in the implementation, serving as the underlying storage medium that imitates the hard disk. "myfs" is where all the file system data is stored. A function "`init()`" initializes the file system which is shown below in Listing 3. Then the function is called in the main function, which returns the file descriptor of the file system. The file descriptor is then used to access the file system. The file system is then accessed using the `lseek` function to the appropriate position, followed by a subsequent read or write operations.

4.2 Algorithms and Code

There is no specific algorithm that was followed to implement the file system. The implementation was done in a very simple fashion, where the file system was first initialized, then the subsequent functions, and data structures were created, followed by the main function where the initialization was done, and the input file was parsed line by line. In general, the algorithm followed the following pattern:

- use `lseek` to seek over to the required inode, or the directory entry, or the data block
- use `read` or `write` to read or write the required data

A function "`init()`" initializes the file system:

```
int init(){
    int myfs = open("./myfs", O_CREAT | O_RDWR, 0666); // create a file
    named myfs with read write enabled
    if(myfs == -1){
        printf("Error: Cannot create file system myfs\n"); return -1;
    }

    ftruncate(myfs, BLOCK_SIZE * NUM_BLOCKS); // 128 * 1024 = 128KB
    allocated to myfs

    // identification and directory bitmap
    char fs = 'A'; write(myfs, (char*)&fs, 1);
    char dbm = (char)1; write(myfs, (char*)&dbm, 1);

    // Initializing the Root Inode
    struct inode root_inode;
    root_inode.dir = 1; // root inode is a directory
    strcpy(root_inode.name, "/"); // first root directory named "/"
    root_inode.size = sizeof(struct dirent);
    root_inode.blockptrs[0] = 1;
    root_inode.used = 1; // yes it is in use
    root_inode.rsvd = 0; // no it is not reserved for future use

    // write the root inode into myfs
    lseek(myfs, NUM_BLOCKS, SEEK_SET);
    write(myfs, (char*)&root_inode, sizeof(struct inode));

    // Initializing the Root Directory Entry
    struct dirent root_dirent;
    strcpy(root_dirent.name, ".");
    root_dirent.namelen = 1;
    root_dirent.inode = 0;

    // write the root directory entry into myfs
    lseek(myfs, BLOCK_SIZE, SEEK_SET);
    write(myfs, (char*)&root_dirent, sizeof(struct dirent));

    return myfs;
}
```

Listing 3: Initialization of MYFS

In the above listing, "myfs" is first opened as a file, with read and write permissions enabled. If

there's any issue in creating the file system, an error message is thrown and the function exits. If the file is created successfully, the file is truncated to 128KB in size. The first two bytes of the file are then written with the identification marker "A" and the directory bitmap "1". The root inode is then initialized and written into the file system. The root inode is a directory, and has the name "/". The size of the root inode is the size of a single directory entry. The root inode has a single block pointer, which points to the first data block. The root inode is then written into the file system.
The root directory entry is then initialized and written into the file system. The root directory entry has the name ".", and the inode number of the root inode. The root directory entry is then written into the file system. The function then returns the file descriptor of the file system, which is then used to access the file system.

The `main` function is responsible for opening the "myfs" file system, and parsing over the input file. This `main` serves as the control center of the file system implementation, by first creating the file system if it does not exist, and then parsing over the input file line by line. The command is parsed using the `sscanf` function, and then the appropriate function is called. The input file is parsed line by line using the `getline` function. The `getline` function returns -1 when the end of the file is reached. The `getline` function also allocates memory to the line variable, which is freed at the end of the program. The input file is opened using the `fopen` function, and is closed at the end of the program. The file system is opened using the `open` function, and is closed at the end of the program.

5 Takeaway and Reflection

While doing this assignment, the sheer dedication, will power, sleepless nights and the amount of time and effort that goes into making a file system was realized. The assignment was a great learning experience, and I learned a lot about how file systems work (in a very simple fashion though, not too complex). I also learned that I have much more patience than I realized to not completely give up while understanding the implementation, not to mention the fact that twice I had to completely start from scratch due to lack of understanding and wrong implementation, and once more because my whole system crashed due to an error I still don't know about. This assignment seemed nothing less than a project to me, and I am glad I was able to complete it (in whatever state it is currently in). I would not want to indulge in this ever again. The image attached below sums up my emotions, reflections and feelings during and after the assignment.

6 References

References

[1] *GitHub*. [Online]. Available: https://github.com/bartooo/very-simple-file-system/blob/main/main.c

[2] *GitHub*. [Online]. Available: https://github.com/Luminoid/Simple-File-System

[3] *GitHub*. [Online]. Available: https://github.com/jcbbeiter/simple-file-system

[4] *GitHub*. [Online]. Available: https://github.com/paleumm/vsfs/blob/main/src/block.c

[5] *GitHub*. [Online]. Available: https://github.com/SrLozano/Simple-File-System/blob/master/filesystem/filesystem.c

[6] OpenAI. *ChatGPT* [Online]. Available: https://chat.openai.com/ mainly for understanding, error resolving, and commentation.

A Appendix

The file system has been uploaded on to my repository on GitHub. The repository can be accessed through the following link: https://github.com/AliMuhammadAsad/A-Simple-File-System/