# Operating Systems Assignment 1 Report

Dua Batool

September 30, 2023

## 1 Introduction

In this report, I will discuss the design and implementation of the file system that we were given to create for our assignment, with a specific focus on the data structure, implementation of all the functions, challenges faced during the assignment, and potential improvements.

## 2 File System Structure

### 2.1 Heap Allocation

I allocated 128KB on the heap for disk storage. The first 1KB is dedicated to the SuperBlock, and the remaining 127KB is for DataBlock.

### 2.2 DataBlock Structure

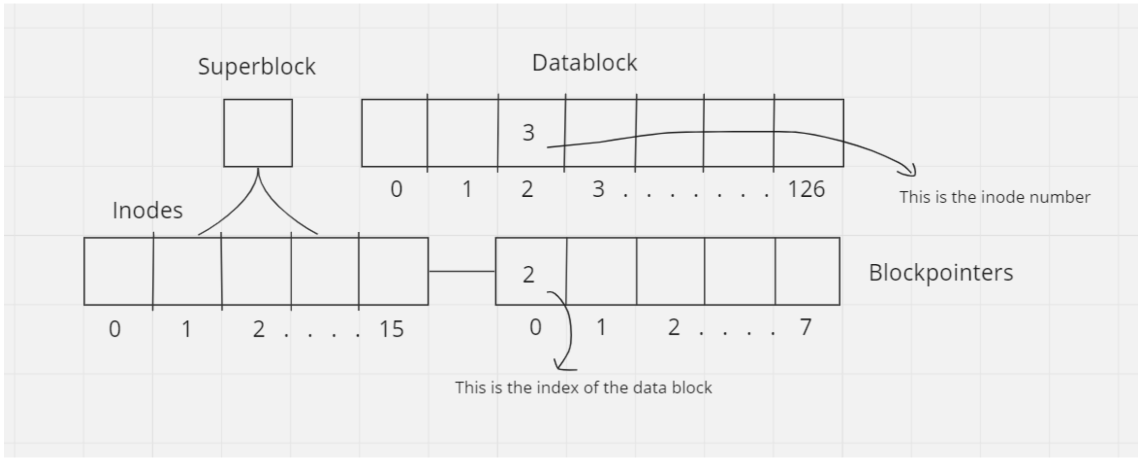
The DataBlock is divided into 127 indexes, each of size 1KB. Each index is represented by the `Datablock` struct, which allows us to access individual bytes within each index.

### 2.3 SuperBlock and Inodes

Throughout the code, the SuperBlock is used to access inodes. The SuperBlock contains 16 inodes, and each inode has 8 block pointers.

### 2.4 Directory Structure

Directories are managed using the block pointers of the inode. Each directory points to the index of the DataBlock where the inode number of the sub-directory is stored. This creates a hierarchical structure resembling a tree. A directory can have a maximum of 8 subdirectories or files, based on the 8 block pointers in each inode. This is also represented by a diagram below:



## 3 Functions

In addition to the main functions of my code, I have introduced helper functions where I found them necessary, considering the frequency of their utilization within the same function or across multiple functions.

### 3.1 Createfile

I parse the path and then loop over each element. In this function, I store the inode number of the file being created in a datablock index and then create the link between this file and its parent by finding a blockpointer of the parent which would point towards a datablock index. This datablock indes is where the inode number of the file being created is stored. In addition to that, characters a-z are filled repeatedly (according to the filesize) in a datablock index. A blockpointer of the file would be pointing towards this datablock index. Whenever a file is being created, the size of its parent also changes.

### 3.2 Removefile

I parse the path and then find the inode of the file to be deleted. I then break the link between it and its parent by finding the blockpointer of the parent inode which contains the datablock index. This datablock index contains the inode of the file to be deleted. I set the data at this index to null. After that I ensure that the blockpointers of the file do not contain any data index (initally they would have the datablock index with characters corresponding to that filesize). I set the data at these indexes to null. Whenever a file is being deleted, the size of its parent also changes.

### 3.3 Copyfile

I parse the path to find the inode number of the file to be copied. From the inode number, I get the size of the file. Then I just call the createfile function which contains the path where the file is to be copied and the filesize.

### 3.4 Movefile

I parse both the paths, original path and new path. I find the original parent of the file to be moved so that I could get the datablock index where the inode number of the file is stored through its blockpointer. Then I just put that datablock index to a blockpointer of the new parent. I also ensure that the original parent is no longer pointing towards that datablock index.

### 3.5 Createdirectory

I initially call this function in the main function before any command of `sampleinput` to create the root directory. There is a seperate condition for the root directory in this function as it would not create because of parsing. For other directories, I parse the path and then loop over each element. In this function, I store the inode number of the directory being created in a datablock index and then create the link between this file and its parent by finding a blockpointer of the parent which would point towards a datablock index. This datablock indes is where the inode number of the file being created is stored.

### 3.6 Removedirectory

I parse the path and then find the inode of the directory to be deleted. I then break the link between it and its parent by finding the blockpointer of the parent inode which contains the datablock index. This datablock index contains the inode of the file to be deleted. I set the data at this index to null. After that I ensure that the blockpointers of the file do not contain any data index (the blockpointers could contain the datablock index with inode number of a subdirectory or file). I set the data at these indexes to null.

### 3.7 List

This functions prints the names and the sizes of all the files and directories in the filesystem. The sizes of the directories are based on the sizes of the files that they contain.

## 4 Error Handling

In addition to the error handling that we were supposed to do as asked in the assignment, I am also handling errors when allocating heap memory and when entering data into `my_fs`.

## 5 Initializing and Saving

After executing all the commands from `sampleinput.txt`, I am saving the file system's state to the `my_fs.txt`. This file has a size of 128KB, with a specific structure for data storage. To ensure the integrity of the file system, I follow a structured approach:

1. I write the Superblock struct at the beginning of the `my_fs.txt`. This initial KB of data is reserved to store the Superblock.

2. The remaining 127KB of the file is designated for the DataBlock. To manage the precise positioning of data within the file, I utilize the `fseek` function. This allows me to navigate to specific locations within the file, ensuring that the indexes of the DataBlock data are correctly written withing the file.

3. I adopt an approach where I do not write data into the file after every individual command of `sampleinput.txt`. Instead, I accumulate the changes and write them collectively once all commands have been processed. This strategy leverages the heap memory used for the file system, minimizing redundant file write operations.

Despite successfully writing data to the file, a significant challenge arises during the subsequent phase: reading and recovering the states of the Superblock and DataBlock. This issue arises because of a heap-based file system and necessitates further investigation and refinement.

## 6 Challenges Faced

- **Debugging:** Debugging was a significant challenge, consuming 80% of the development time. Complex data structures and interactions can make debugging difficult.
- **Managing Memory Using Heap:** The decision to use heap memory instead of stack memory introduced complexity in memory management. Frequent allocation and deallocation of memory using pointers led to challenges, including segmentation faults, which required significant effort to identify and resolve.
- **Writing Data to `my_fs.txt`:** To persist the file system data, adjustments in data types were necessary to correctly write the SuperBlock and DataBlock structures to the `my_fs.txt` file. Additionally, the absence of a structured representation for DataBlocks initially caused problems during data writing, prompting the creation of a DataBlock struct and necessitating changes in the existing codebase.

## 7 A Better Approach

One potential improvement to the file system design is to introduce a `dirent` object. Instead of having the block pointer of an inode point directly to the index of the DataBlock, it could point to a `dirent` object, indicating whether it is a subdirectory or a file. This would enhance the clarity of the file system structure.