

PROJECT OBJECTIVES

Project objective was to design a Virtual File System (VFS), which resides on a file on the operating system, and a program which allows basic operations on the filesystem. The specifications are followed:

- The filesystem should have complete directory structure. With a root directory being the main one
- filesystem should allow deleting and making directories.
- The user can add files from their OS to the VFS and they can also remove the files from the VFS. They can also extract the files from the VFS/
- The Filesystem should be properly made, i.e., managing free blocks, keeping track of them and using the ones which are free.

PROJECT DESCRIPTION

The project contains a program which is capable of making or opening an existing filesystem made by the program, then it goes into an interactive shell which is totally designed inside the program with custom commands, from where you can perform operations such as:

1. Add directory.
2. Remove directory.
3. Change current shell directory.
4. Add file.
5. Remove file.
6. Extract file
7. Get filesystem usage.
8. Get help regarding commands.
9. Rename a file or folder.
10. Print current shell directory.
11. Quit the application.

The filesystem uses I-Node structure to keep track of files and folders. The filesystem is divided into blocks of 4kb. A block is a smallest unit of space a file or folder can take on a filesystem. The filesystem can be broken down into 5 parts:

1. The Super Block: Super block is the first block of a filesystem. It contains information about the filesystem such as name, number of blocks in the filesystem, number of blocks used, number of inodes, number of inodes used and some basic byte offsets to where the specific parts of filesystem are starting from.

2. The Inode Bitmap: A bitmap is an array of bits, each bit in this bitmap defines if the corresponding inode is free or not, for example, if 24th bit is 1 it means that inode number 24 is taken and is not free, if 56th bit is 0 it means inode number 56 is free.
3. The Block Bitmap: This bitmap keeps track of blocks of the filesystem, having each bit corresponding to a block, the value telling us whether the block is taken or not.
4. The Inode Region: Here the inodes resides, each inode contains number which is used to identify the inodes. Inodes are like identifier of a file and folder. When a folder or file is made an inode is assigned to it. Inode contains the name of the folder or file, the size of it, the block offset where the file data is located and the creation and modified time.
5. The Data Region: the rest of the filesystem is used to keep actual data of the files and folder.

CODE

```
#include <stdio.h>
#include <fcntl.h>
#include <time.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
#include <math.h>

#define BLOCK_SIZE 4096
#define BASE_SIZE 134217728 //128 megabytes
#define INODE_SIZE 128

struct superBlock {
    char name[128];
    int nBlocks;
    int blockSize;
    int freeBlocks;
    int nInodes;
    int nInodeBlocks;
    int inodeBitmapSize;
    int inodeBitmapPos;
    int blockBitmapSize;
    int blockBitmapPos;
    int inodeBlockoffset;
    int dataBlockStart;
} super;

int filesystem;
```

```
char *inodeBitmap;
char *blockBitmap;
int currentDirInode;

struct inode {
    char name[88];
    int number;
    int blockOffset;
    int type; //0 for file 1 for dir
    time_t ctime;
    time_t mtime;
    int size;
};

struct fileBlock {
    int nextBlockOffset;
    char data[BLOCK_SIZE - 4];
};

struct dirBlock {
    int nextdirBlockOffset;
    int inodeCurrent;
    int inodePrev;
    int inodesNumbers[1021];
};

struct combinedDir {
    struct inode i;
    struct dirBlock d;
```

```
};
```

```
char *strrev(char *str) {
```

```
    char *p1, *p2;
```

```
    if (!str || !*str)
```

```
        return str;
```

```
    for (p1 = str, p2 = str + strlen(str) - 1; p2 > p1; ++p1, --p2) {
```

```
        *p1 ^= *p2;
```

```
        *p2 ^= *p1;
```

```
        *p1 ^= *p2;
```

```
    }
```

```
    return str;
```

```
}
```

```
bool getBit(char *bitmap, int pos) {
```

```
    char *temp;
```

```
    int actualPos = pos % 8;
```

```
    temp = bitmap + ((pos - actualPos) / 8);
```

```
    return ((*temp) >> actualPos) & 1;
```

```
}
```

```
void setBit(char *bitmap, int pos, int val) {
```

```
    char *temp;
```

```
    int actualPos = pos % 8;
```

```
    temp = bitmap + ((pos - actualPos) / 8);
```

```
    (*temp) ^= (-val ^ (*temp)) & (1 << actualPos);
```

```
}
```

```
void initializeBitmap(char **bitmap, int len) {
```

```

    (*bitmap) = (char *)malloc(len / 8);
    int i;
    for (i = 0; i < len; i++) {
        setBit(*bitmap, i, 0);
    }
}

void wrapperInitNodeandBlockBitmaps() {
    initializeBitmap(&blockBitmap, super.nBlocks);
    initializeBitmap(&inodeBitmap, super.nInodes);
}

int findFreeBlock(char *bitmap, int len) {
    for (int i = 0; i < len; i++) {
        bool val = getBit(bitmap, i);
        if (!val) {
            return i;
        }
    }
    return -1;
}

int ceilToBlockSize(int nbytes) {
    return ceil(nbytes / (double)BLOCK_SIZE) * BLOCK_SIZE;
}

int ceilToBlockSizeForFileBlocks(int nbytes) {
    return ceil(nbytes / (double)(BLOCK_SIZE - 4)) * (BLOCK_SIZE - 4);
}

```

```
int getByteOffset(int dataBlockOffset) {
    return BLOCK_SIZE * (dataBlockOffset + super.dataBlockStart);
}

int getInodeByteOffset(int inodeNum) {
    return (BLOCK_SIZE * super.inodeBlockoffset) + (inodeNum * 128);
}

int seekToInode(int inodeNum) {
    lseek(filesystem, getInodeByteOffset(inodeNum), SEEK_SET);
}

int seekToDataBlock(int dataBlockOffset) {
    lseek(filesystem, getByteOffset(dataBlockOffset), SEEK_SET);
}

void updateInodeBitmap() {
    int cur = lseek(filesystem, 0, SEEK_CUR);
    lseek(filesystem, BLOCK_SIZE * super.inodeBitmapPos, SEEK_SET);
    write(filesystem, inodeBitmap, super.nInodes / 8);
    lseek(filesystem, cur, SEEK_SET);
}

void updateBlockBitmap() {
    int cur = lseek(filesystem, 0, SEEK_CUR);
    lseek(filesystem, BLOCK_SIZE * super.blockBitmapPos, SEEK_SET);
    write(filesystem, blockBitmap, super.nBlocks / 8);
    lseek(filesystem, cur, SEEK_SET);
}
```

```
void wrapperUpdateBitmaps() {
    updateInodeBitmap();
    updateBlockBitmap();
}

void loadInodeBitmap() {
    int cur = lseek(filesystem, 0, SEEK_CUR);
    lseek(filesystem, BLOCK_SIZE * super.inodeBitmapPos, SEEK_SET);
    read(filesystem, inodeBitmap, super.nInodes / 8);
    lseek(filesystem, cur, SEEK_SET);
}

void loadBlockBitmap() {
    int cur = lseek(filesystem, 0, SEEK_CUR);
    lseek(filesystem, BLOCK_SIZE * super.blockBitmapPos, SEEK_SET);
    read(filesystem, blockBitmap, super.nBlocks / 8);
    lseek(filesystem, cur, SEEK_SET);
}

void loadSuper() {
    int cur = lseek(filesystem, 0, SEEK_CUR);
    lseek(filesystem, 0, SEEK_SET);
    read(filesystem, &super, sizeof(super));
    lseek(filesystem, cur, SEEK_SET);
}

void LoadInfo() {
    loadSuper();
    wrapperInitInodeandBlockBitmaps();
    loadInodeBitmap();
}
```

```
    loadBlockBitmap();
}

void updateSuper() {
    int cur = lseek(filesystem, 0, SEEK_CUR);
    lseek(filesystem, 0, SEEK_SET);
    write(filesystem, &super, sizeof(super));
    lseek(filesystem, cur, SEEK_SET);
}

void readInode(struct inode *i) {
    read(filesystem, i, sizeof(struct inode));
}

void readDataBlock(struct fileBlock *f) {
    read(filesystem, f, sizeof(struct fileBlock));
}

void readDirBlock(struct dirBlock *d) {
    read(filesystem, d, sizeof(struct dirBlock));
}

void writeInode(struct inode *i) {
    write(filesystem, i, sizeof(struct inode));
}

void writeDataBlock(struct fileBlock *f) {
    write(filesystem, f, sizeof(struct fileBlock));
}
```

```

void writeDirBlock(struct dirBlock *d) {
    write(filesystem, d, sizeof(struct dirBlock));
}

void readDirFromInode(struct combinedDir *cd, int inodeN) {
    int cur = lseek(filesystem, 0, SEEK_CUR);
    seekToInode(inodeN);
    readInode(&(cd->i));
    seekToDataBlock(cd->i.blockOffset);
    readDirBlock(&(cd->d));
}

void mkdir(char *dirName) {
    int freeinode = findFreeBlock(inodeBitmap, super.nInodes);
    int a = super.dataBlockStart;
    int freedatablock = findFreeBlock(blockBitmap, super.nBlocks) - super.dataBlockStart;
    if (freeinode == -1 || freedatablock == -1) {
        printf("Can't make folder, FileSystem Full\n");
        return;
    }
    struct inode dirinode;
    strcpy(dirinode.name, dirName);
    dirinode.ctime = time(NULL);
    dirinode.mtime = time(NULL);
    dirinode.number = freeinode;
    dirinode.type = 1;
    dirinode.size = 4096;
    dirinode.blockOffset = freedatablock;
    seekToInode(freeinode);
    writeInode(&dirinode);
}

```

```

setBit(inodeBitmap, freeinode, 1);
struct dirBlock dirblock;
dirblock.nextdirBlockOffset = -1;
dirblock.inodePrev = currentDirInode;
dirblock.inodeCurrent = freeinode;
for (int i = 0; i < 1021; i++) {
    dirblock.inodesNumbers[i] = -1;
}
seekToDataBlock(freedatablock);
writeDirBlock(&dirblock);
super.freeBlocks--;
updateSuper();
setBit(blockBitmap, freedatablock + super.dataBlockStart, 1);
wrapperUpdateBitmaps();
struct combinedDir prevDir;
readDirFromInode(&prevDir, currentDirInode);
int flag = 1;
for (int i = 0; i < 1021; i++) {
    if (prevDir.d.inodesNumbers[i] == -1) {
        prevDir.d.inodesNumbers[i] = freeinode;
        flag = 0;
        break;
    }
}
seekToDataBlock(prevDir.i.blockOffset);
writeDirBlock(&prevDir.d);
// if(flag) {
//     int nextfreeDataBlock = findFreeBlock(blockBitmap, super.nBlocks) - super.dataBlockSt
art;

```

```

    //}
}

void listFiles() {
    struct combinedDir dir;
    struct inode temp;
    readDirFromInode(&dir, currentDirInode);
    printf("\n%-6.6s%-20.20s%-30.30sType \tSize\n\n", "Inode", "Name", "Date Created");
    char buf[256];
    strftime(buf, 256, "%H:%M:%S %d %h %Y", localtime(&(dir.i.ctime)));
    printf("%-6d%-20.20s%-30.30sfolder\t%d\n", currentDirInode, ".", buf, 4096);
    if (dir.d.inodePrev != -1) {
        seekToInode(dir.d.inodePrev);
        readInode(&temp);
        char buf[256];
        strftime(buf, 256, "%H:%M:%S %d %h %Y", localtime(&(temp.ctime)));
        printf("%-6d%-20.20s%-30.30sfolder\t%d\n", dir.d.inodePrev, "..", buf, 4096);
    }
    for (int i = 0; i < 1021; i++) {
        if (dir.d.inodesNumbers[i] != -1) {
            seekToInode(dir.d.inodesNumbers[i]);
            readInode(&temp);
            char buf[256];
            strftime(buf, 256, "%H:%M:%S %d %h %Y", localtime(&(temp.ctime)));
            if (temp.type == 1) {
                printf("%-6d%-20.20s%-30.30sfolder\t%d\n", temp.number, temp.name, buf, temp.size);
            }
        }
    }
}

```

```

for (int i = 0; i < 1021; i++) {
    if (dir.d.inodesNumbers[i] != -1) {
        seekTolnode(dir.d.inodesNumbers[i]);
        readlnode(&temp);
        char buf[256];
        strftime(buf, 256, "%H:%M:%S %d %h %Y", localtime(&(temp.ctime)));
        if (temp.type == 0) {
            printf("%-6d%-20.20s%-30.30sfile \t%d\n", temp.number, temp.name, buf, temp.size);
        }
    }
}
}

```

```

void changeDir(char *dirName) {
    struct combinedDir dir;
    struct inode temp;
    readDirFromlnode(&dir, currentDirlnode);
    if (strncmp(dirName, "..", 2) == 0 && dir.d.inodePrev == -1) {
        printf("No such file or directory\n");
        return;
    }
    if (strncmp(dirName, ".\\0", 2) == 0) {
        return;
    }
    if (strncmp(dirName, "..", 2) == 0) {
        currentDirlnode = dir.d.inodePrev;
        return;
    }
    for (int i = 0; i < 1021; i++) {
        if (dir.d.inodesNumbers[i] == -1) {

```

```

        continue;
    }

    seekToInode(dir.d.inodesNumbers[i]);
    readInode(&temp);
    if (strcmp(dirName, temp.name) == 0) {
        if (temp.type == 0) {
            printf("%s is a file\n", temp.name);
        }

        currentDirInode = temp.number;
        return;
    }
}

printf("No such file or directory\n");
}

```

```

void removeFile(int Inode) {
    struct inode toDestroy;
    seekToInode(Inode);
    readInode(&toDestroy);
    int block = toDestroy.blockOffset;
    setBit(inodeBitmap, Inode, 0);
    while (block != -1) {
        struct fileBlock fblock;
        seekToDataBlock(block);
        readDataBlock(&fblock);
        setBit(blockBitmap, block + super.dataBlockStart, 0);
        super.freeBlocks++;
        block = fblock.nextBlockOffset;
    }

    wrapperUpdateBitmaps();
}

```

```

    updateSuper();
}

void wrapperRemoveFile(char *name) {
    struct combinedDir dir;
    struct inode temp;
    readDirFromInode(&dir, currentDirInode);
    if (strncmp(name, "..", 2) == 0) {
        printf("Not a File\n");
        return;
    }
    if (strncmp(name, ".\\0", 2) == 0) {
        printf("Not a File\n");
        return;
    }
    for (int i = 0; i < 1021; i++) {
        if (dir.d.inodesNumbers[i] == -1) {
            continue;
        }
        seekToInode(dir.d.inodesNumbers[i]);
        readInode(&temp);
        if (strcmp(name, temp.name) == 0) {
            if (temp.type == 1) {
                printf("%s is a directory", temp.name);
                return;
            }
            removeFile(dir.d.inodesNumbers[i]);
            dir.d.inodesNumbers[i] = -1;
            seekToDataBlock(dir.i.blockOffset);
            writeDirBlock(&(dir.d));
        }
    }
}

```

```

        return;
    }
}

printf("no such file\n");
}

void removeDirRecurse(int Inode) {
    struct inode temp;
    struct combinedDir dir;
    seekToInode(Inode);
    readInode(&temp);
    if (temp.type == 0) {
        removeFile(temp.number);
    } else if (temp.type == 1) {
        readDirFromInode(&dir, Inode);
        for (int i = 0; i < 1021; i++) {
            if (dir.d.inodesNumbers[i] == -1) {
                continue;
            }
            removeDirRecurse(dir.d.inodesNumbers[i]);
        }
        setBit(inodeBitmap, dir.i.number, 0);
        setBit(blockBitmap, dir.i.blockOffset + super.dataBlockStart, 0);
        wrapperUpdateBitmaps();
        super.freeBlocks++;
        updateSuper();
    }
}

void removeDir(char *dirName) {

```

```

struct combinedDir dir;
struct inode temp;

readDirFromInode(&dir, currentDirInode);
if (strncmp(dirName, "..", 2) == 0) {
    printf("Cannot delete previous folder\n");
    return;
}
if (strncmp(dirName, ".\\0", 2) == 0) {
    printf("Cannot delete current folder\n");
    return;
}
for (int i = 0; i < 1021; i++) {
    if (dir.d.inodesNumbers[i] == -1) {
        continue;
    }
    seekToInode(dir.d.inodesNumbers[i]);
    readInode(&temp);
    if (strcmp(dirName, temp.name) == 0) {
        if (temp.type == 0) {
            printf("%s is a file", temp.name);
            return;
        }
        removeDirRecurse(temp.number);
        dir.d.inodesNumbers[i] = -1;
        seekToDataBlock(dir.i.blockOffset);
        writeDirBlock(&(dir.d));
        return;
    }
}
}

```

```

    printf("no such Directory\n");
}

void addFile(char *path) {
    int file = open(path, O_RDWR);
    if (file < 0) {
        printf("Could not open file, check path\n");
        return;
    }
    char *filename;
    (filename = strrchr(path, '/')) ? ++filename : (filename = path);
    int filesize = lseek(file, 0, SEEK_END);
    lseek(file, 0, SEEK_SET);
    int blocksneeded = ceilToBlockSizeForFileBlocks(filesize) / (BLOCK_SIZE - 4);
    //printf("%d", blocksneeded);
    if (blocksneeded > super.freeBlocks) {
        printf("Not Enough Space\n");
        return;
    }
    int freenode = findFreeBlock(inodeBitmap, super.nInodes);
    int freeBlock = findFreeBlock(blockBitmap, super.nBlocks) - super.dataBlockStart;
    struct inode fileinode;
    strcpy(fileinode.name, filename);
    fileinode.ctime = time(NULL);
    fileinode.mtime = time(NULL);
    fileinode.number = freenode;
    fileinode.blockOffset = freeBlock;
    fileinode.size = filesize;
    fileinode.type = 0;
    seekToNode(freenode);
}

```

```

writelNode(&fileInode);
setBit(inodeBitmap, freelNode, 1);
updateInodeBitmap();
char buf[4092];
int inlen;
int datablockoffset = freeBlock;
int blocksFilled = 0;
while ((inlen = read(file, buf, sizeof(buf))) > 0) {
    struct fileBlock datablock;
    for (int i = 0; i < inlen; i++) {
        datablock.data[i] = buf[i];
    }
    // memcpy(datablock.data, buf, sizeof(inlen));
    if (++blocksFilled == blocksneeded) {
        datablock.nextBlockOffset = -1;
        seekToDataBlock(datablockoffset);
        writeDataBlock(&datablock);
        setBit(blockBitmap, datablockoffset + super.dataBlockStart, 1);
        updateBlockBitmap();
    } else {
        datablock.nextBlockOffset = findFreeBlock(blockBitmap, super.nBlocks);
        seekToDataBlock(datablockoffset);
        writeDataBlock(&datablock);
        setBit(blockBitmap, datablockoffset + super.dataBlockStart, 1);
        updateBlockBitmap();
        datablockoffset = datablock.nextBlockOffset;
    }
}
super.freeBlocks -= blocksFilled;
updateSuper();

```

```

    struct combinedDir dir;
    readDirFromInode(&dir, currentDirInode);
    int flag = 1;
    for (int i = 0; i < 1021; i++) {
        if (dir.d.inodesNumbers[i] == -1) {
            dir.d.inodesNumbers[i] = freenode;
            flag = 0;
            break;
        }
    }
    seekToDataBlock(dir.i.blockOffset);
    writeDirBlock(&dir.d);
    close(file);
}

void freeSpace() {
    printf("Total Blocks: %d\nUsed Blocks: %d\n", super.nBlocks, super.nBlocks - super.freeBlocks);
    printf("Size in Bytes: %d / %d\n", (super.nBlocks - super.freeBlocks) * BLOCK_SIZE, super.nBlocks * BLOCK_SIZE);
    printf("Percentage usage: %.2f%%\n", ((double)(super.nBlocks - super.freeBlocks)/(double)(super.nBlocks)) * 100);
}

void renamee(char *args) {
    char *oldname = args;
    char *newname;
    int err = 0;
    (newname = strrchr(args, '.')) ? (++newname) : (err = 1);
    if (err) {
        printf("Arguments not correct\n");
    }
}

```

```

    return;
}

*(newname - 1) = '\0';

struct combinedDir dir;
struct inode temp;

readDirFromInode(&dir, currentDirInode);

if (strncmp(oldname, "..", 2) == 0) {
    printf("Cannot rename that\n");
    return;
}

if (strncmp(oldname, ".\0", 2) == 0) {
    printf("Cannot rename that\n");
    return;
}

for (int i = 0; i < 1021; i++) {
    if (dir.d.inodesNumbers[i] == -1) {
        continue;
    }

    seekToInode(dir.d.inodesNumbers[i]);
    readInode(&temp);
    if (strcmp(oldname, temp.name) == 0) {
        strcpy(temp.name, newname);
        seekToInode(temp.number);
        writeInode(&temp);
        return;
    }
}

printf("no such file\n");
}

```

```

void extractFile(char *name) {
    struct combinedDir dir;
    struct inode temp;
    readDirFromInode(&dir, currentDirInode);
    if (strncmp(name, "..", 2) == 0) {
        printf("Not a File\n");
        return;
    }
    if (strncmp(name, ".\\0", 2) == 0) {
        printf("Not a File\n");
        return;
    }
    for (int i = 0; i < 1021; i++) {
        if (dir.d.inodesNumbers[i] == -1) {
            continue;
        }
        seekToInode(dir.d.inodesNumbers[i]);
        readInode(&temp);
        if (strcmp(name, temp.name) == 0) {
            if (temp.type == 1) {
                printf("%s is a directory", temp.name);
                return;
            }
            int fd = open(temp.name, O_RDWR | O_CREAT, 0666);
            char *buf[4092];
            struct fileBlock block;
            seekToDataBlock(temp.blockOffset);
            readDataBlock(&block);
            int size = temp.size;
            while (1) {

```

```

        if (size <= 4092) {
            memcpy(buf, block.data, size);
            write(fd, buf, size);
            close(fd);
            break;
        } else {
            memcpy(buf, block.data, 4092);
            write(fd, buf, 4092);
            seekToDataBlock(block.nextBlockOffset);
            readDataBlock(&block);
            size -= 4092;
        }
    }
    return;
}

printf("no such file\n");
}

```

```

void pwd() {
    int temp = currentDirInode;
    char *prompt = malloc(1000);
    struct combinedDir dir;
    readDirFromInode(&dir, temp);
    while (1) {
        strrev(dir.i.name);
        strcat(prompt, dir.i.name);
        if (dir.d.inodePrev == -1) {
            break;
        } else {

```

```

        readDirFromInode(&dir, dir.d.inodePrev);
        if (dir.d.inodePrev == -1) {
            continue;
        }
        strcat(prompt, "/");
    }
}
strrev(prompt);
printf("%s\n", prompt);
free(prompt);
}

```

```

void printprompt() {
    int temp = currentDirInode;
    char *prompt = malloc(1000);
    struct combinedDir dir;
    readDirFromInode(&dir, temp);
    while (1) {
        strrev(dir.i.name);
        strcat(prompt, dir.i.name);
        if (dir.d.inodePrev == -1) {
            break;
        } else {
            readDirFromInode(&dir, dir.d.inodePrev);
            if (dir.d.inodePrev == -1) {
                continue;
            }
            strcat(prompt, "/");
        }
    }
}

```

```

    strrev(prompt);
    printf("%s >>> ", prompt);
    free(prompt);
}

void interactiveShell() {
    setbuf(stdin, NULL);

    printf("\nYou are in the filesystem %s\nType \"help\" to get a list of commands\n", super.name);

    char inputBuf[1000];
    while (1) {
        //listFiles();
        //char c;

        printf("\n");
        printprompt();

        fgets(inputBuf, 1000, stdin);
        inputBuf[strcspn(inputBuf, "\n")] = 0;
        if (strncmp(inputBuf, "listFiles", 9) == 0) {
            listFiles();
        } else if (strncmp(inputBuf, "makeDir ", 8) == 0) {
            mkdir(inputBuf + 8);
        } else if (strncmp(inputBuf, "changeDir ", 10) == 0) {
            changeDir(inputBuf + 10);
        } else if (strncmp(inputBuf, "removeDir ", 10) == 0) {
            removeDir(inputBuf + 10);
        } else if (strncmp(inputBuf, "addFile ", 8) == 0) {
            addFile(inputBuf + 8);
        } else if (strncmp(inputBuf, "removeFile ", 11) == 0) {
            wrapperRemoveFile(inputBuf + 11);
        } else if (strncmp(inputBuf, "extractFile ", 12) == 0) {

```

```

        extractFile(inputBuf + 12);
    } else if (strncmp(inputBuf, "rename ", 7) == 0) {
        renamee(inputBuf + 7);
    } else if (strncmp(inputBuf, "pwd", 3) == 0) {
        pwd();
    } else if (strncmp(inputBuf, "filesystemUsage", 15) == 0) {
        freeSpace();
    } else if (strncmp(inputBuf, "quit", 4) == 0) {
        break;
    } else if (strncmp(inputBuf, "help", 4) == 0) {
        printf("This is in interactive shell of this filesystem, below is a list of commands\n\nlistF
iles - lists file in the current directory\nmakeDir name - makes a dir with \"name\" in current d
irectory\nchangeDir name - changes current working directory to the given one\npwd - prints
the current working directory\nremoveDir name - removes a dir with \"name\" in currenct dire
ctory\naddFile path - adds a file from \"path\" to the current directory in vfs\nremoveFile nam
e - removes a file \"name\" from the current directory of filesystem\nextractFile name - extra
cts a file from vfs to the actual filesystem of your os\nrename old new - renames a file or fold
er with old name to a new one\nfilesystemUsage - Shows size information\nhelp - brings this
help menu\nquit - quit the vfs manager\n");
    } else {
        printf("Undefined Command\n");
    }
}

}

void makeFileSystem(char *name) {
    strcpy(super.name, name);
    super.blockSize = BLOCK_SIZE;
    super.nBlocks = BASE_SIZE / BLOCK_SIZE;
    super.nInodes = super.nBlocks / 4;
    wrapperInitInodeandBlockBitmaps();
    super.inodeBitmapPos = 1;
}

```

```

super.inodeBitmapSize = ceilToBlockSize(super.nInodes / 8) / BLOCK_SIZE;
super.blockBitmapPos = super.inodeBitmapPos + super.inodeBitmapSize;
super.blockBitmapSize = ceilToBlockSize(super.nBlocks / 8) / BLOCK_SIZE;
super.inodeBlockoffset = super.blockBitmapPos + super.blockBitmapSize;
super.nInodeBlocks = ceilToBlockSize(INODE_SIZE * super.nInodes) / BLOCK_SIZE;
super.dataBlockStart = super.inodeBlockoffset + super.nInodeBlocks;
super.freeBlocks = super.nBlocks - super.dataBlockStart;
for (int i = 0; i < super.dataBlockStart; i++) {
    setBit(blockBitmap, i, 1);
}
filesystem = open(name, O_RDWR | O_CREAT, 0666);
posix_fallocate(filesystem, 0, BASE_SIZE);
write(filesystem, &super, sizeof(struct superBlock));
lseek(filesystem, BLOCK_SIZE * super.inodeBitmapPos, SEEK_SET);
write(filesystem, inodeBitmap, super.nInodes / 8);
lseek(filesystem, BLOCK_SIZE * super.blockBitmapPos, SEEK_SET);
write(filesystem, blockBitmap, super.nBlocks / 8);
seekToInode(0);
struct inode fill;
for (int i = 0; i < super.nInodes; i++) {
    fill.number = i;
    fill.blockOffset = -1;
    write(filesystem, &fill, sizeof(fill));
}
seekToInode(0);
struct inode rootinode;
strcpy(rootinode.name, "/");
rootinode.blockOffset = 0;
rootinode.ctime = time(NULL);
rootinode.mtime = time(NULL);

```

```

    rootinode.type = 1;
    rootinode.size = 4096;
    setBit(inodeBitmap, rootinode.number, 1);
    setBit(blockBitmap, super.dataBlockStart, 1);
    wrapperUpdateBitmaps();
    write(filesystem, &rootinode, sizeof(rootinode));
    struct dirBlock rootblock;
    rootblock.nextdirBlockOffset = -1;
    rootblock.inodePrev = -1;
    rootblock.inodeCurrent = 0;
    for (int i = 0; i < 1021; i++) {
        rootblock.inodesNumbers[i] = -1;
    }
    seekToDataBlock(0);
    write(filesystem, &rootblock, sizeof(rootblock));
    super.freeBlocks--;
    updateSuper();
    close(filesystem);
}

void openFileSystem(char *name) {
    filesystem = open(name, O_RDWR, 0666);
    if (filesystem < 0) {
        printf("FileSystem doesnt exist with this name\n");
        exit(1);
    }
    LoadInfo();
    currentDirInode = 0;
    interactiveShell();
}

```

```
int main() {  
    printf("1) Make a filesystem\n2) Open an existing filesystem\nEnter Choice: ");  
    int choice;  
    scanf("%d", &choice);  
    char name[128];  
  
    if (choice == 1) {  
        printf("Enter name of filesystem to create: ");  
        scanf("%s", name);  
        makeFileSystem(name);  
        openFileSystem(name);  
    } else if (choice == 2) {  
        printf("Enter name of filesystem to open: ");  
        scanf("%s", name);  
        openFileSystem(name);  
    }  
  
    return 0;  
}
```