


ГУАП

КАФЕДРА № 14

ОТЧЕТ
ЗАЩИЩЕН С ОЦЕНКОЙ
ПРЕПОДАВАТЕЛЬ

Старший преподаватель
должность, уч. степень, звание


подпись, дата

Т.Л. Прокофьева
инициалы, фамилия

ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ №3
Создание файловой системы
по курсу: Операционные системы

РАБОТУ ВЫПОЛНИЛ

СТУДЕНТ ГР. №

1842

29.05.2021

подпись, дата

Геращенко А.В.
инициалы, фамилия

Санкт-Петербург 2021

Оглавление

1. Постановка задачи.....	3
2. Схема файловой системы	4
3. Структура дискового раздела	5
4. Суперблок	6
5. Группы блоков.....	7
6. Каталоги	8
7. Система адресации данных	9
8. Структура данных	11
9. Блок схема.....	12
10. Листинг	13
10.1. .../core/bitmap.h	13
10.2. directory.h.....	16
10.3. inode.h	21
10.4. sectors.h	26
10.5. .../commands/base.h.....	30
10.6. dir_inst.h	35
10.7. initial.h	39
10.8. main.h	42
10.9. run.h.....	51
10.10. str_proc.h	57
11. Результат работы	63
12. Список возможных команд.....	64

1. Постановка задачи

На основе предыдущей лабораторной работы написать программу на языке Си эмулирующую работу файловой системы, структурно похожей к ext2.

2. Схема файловой системы

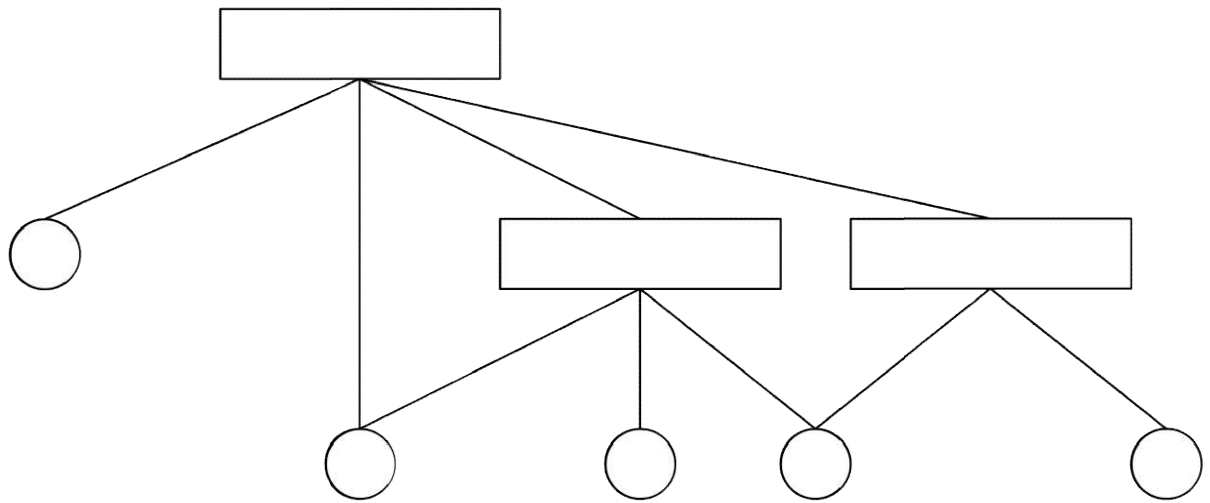


Рис. 1 – файловоедерево

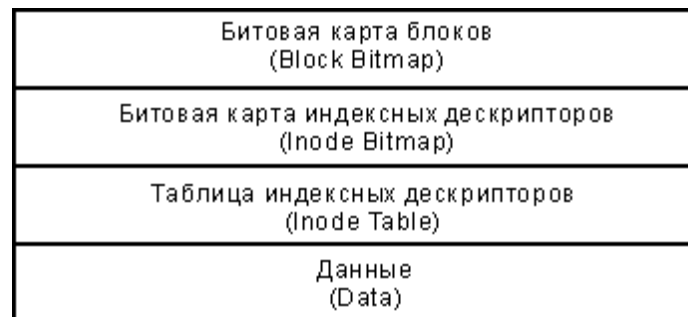


Рис.2 – обобщеннаяструктурнаясхема ФС

3. Структура дискового раздела

Система приближена к ext2.

Выделим следующие составляющие для нашей файловой системы:

- Блоки и группы блоков;
- inode
- суперблок

Всё пространство раздела диска разбивается на блоки фиксированного размера, кратные размеру сектора: 2048 байт. Размер блока указывается при создании файловой системы на разделе диска. Меньший размер блока позволяет сэкономить место на жёстком диске, но также ограничивает максимальный размер файловой системы. Все блоки имеют порядковые номера. С целью уменьшения фрагментации и количества перемещений головок жёсткого диска при чтении больших массивов данных блоки объединяются в группы блоков. (в нашем случае группа блоков – одна)

Базовым понятием файловой системы является индексный дескриптор, или inode (англ. informationnode). Это специальная структура, которая содержит информацию об атрибутах и физическом расположении файла. Индексные дескрипторы объединены в таблицу, которая содержится в начале каждой группы блоков.

4. Суперблок

Суперблок находится в 1024 байтах от начала раздела. В следующем блоке после суперблока располагается глобальная дескрипторная таблица — описание групп блоков, представляющее собой массив, содержащий общую информацию обо всех группах блоков раздела.

От целостности суперблока напрямую зависит работоспособность файловой системы. Операционная система создаёт несколько резервных копий суперблока на случай повреждения раздела. С помощью флага состояния операционная система определяет текущее состояние файловой системы. Если файловая система монтируется на чтение, то флаг состояния будет указывать, что файловая система целостна (состояние «clean»). Если файловая система монтируется на чтение и запись, то в флаг состояния заносится информация о том, что файловая система используется (состояние «notclean»), а после размонтирования файловой системы флаг состояния снова должен указывать на целостность файловой системы. Флаг состояния помогает определять возможные повреждения файловой системы. Например, если питание компьютера было неожиданно отключено, то флаг состояния будет указывать на некорректное завершение работы с файловой системой. При следующей загрузке компьютера операционная система должна будет проверить файловую систему на ошибки, если флаг состояния не указывает на целостность файловой системы.

В нашем случае суперблок отсутствует, но его функциональность будет реализована непосредственно в момент компиляции. Вся информация будет храниться в отдельном файле заголовке.

5. Группы блоков

Все блоки раздела ext2 объединяются в группы блоков. Для каждой группы создаётся отдельная запись в глобальной дескрипторной таблице, в которой хранятся основные параметры:

- номер блока в битовой карте блоков,
- номер блока в битовой карте inode,
- номер блока в таблице inode,
- число свободных блоков в группе,
- число индексных дескрипторов, содержащих каталоги.

Битовая карта блоков — это структура, каждый бит которой показывает, отведён ли соответствующий ему блок какому-либо файлу. Если бит равен 1, то блок занят. Аналогичную функцию выполняет битовая карта индексных дескрипторов, которая показывает, какие именно индексные дескрипторы заняты, а какие нет. Ядро Linux, используя число индексных дескрипторов, содержащих каталоги, пытается равномерно распределить inode каталогов по группам, а inode файлов старается по возможности переместить в группу с родительским каталогом. Все оставшееся место, обозначенное в таблице как данные, отводится для хранения файлов.

В нашем примере используется только 1 группа блоков.

6. Каталоги

Каталоги могут содержать внутри себя другие каталоги или файлы. Физически каталог представляет из себя специальный файл, содержащий записи произвольной длины. Каждая запись хранит в себе следующие данные:

- номер индексного дескриптора файла,
- размер записи,
- длину имени файла,
- имя файла.

Подобная организация каталога позволяет хранить в нём длинные имена файлов без потери места на диске.

Когда операционная система пытается найти расположение файла (или каталога) на диске, она по очереди загружает в память содержимое каждого каталога, указанного в пути к файлу (или каталогу), чтобы по имени найти индексный дескриптор следующего каталога, указанного в пути. Обход каталогов продолжается, пока необходимый файл или каталог не будет найден.

В нашем случае запись будет состоять из:

- номера индексного дескриптора файла
- имени файла

7. Система адресации данных

Система адресации данных — это одна из самых важных составляющих файловой системы. Именно она позволяет находить нужный файл среди множества как пустых, так и занятых блоков на диске.

Файловая система ext2 использует следующую схему адресации блоков файла. Для хранения адреса файла выделено 15 полей, каждое из которых состоит из 4 байт. Если файл умещается в 12 блоков, то номера соответствующих кластеров непосредственно перечисляются в первых двенадцати полях адреса. Если размер файла превышает 12 блоков, то следующее поле содержит адрес кластера, в котором могут быть расположены номера следующих блоков файла. Таким образом, 13-е поле используется для косвенной адресации.

При максимальном размере блока в 4096 байт кластер, соответствующий 13-му полю, может содержать до 1024 номеров следующих блоков файла. Если размер файла превышает 12+1024 блоков, то используется 14-е поле, в котором находится адрес кластера, содержащего 1024 номеров кластеров, каждый из которых ссылается на 1024 блока файла. Здесь применяется уже двойная косвенная адресация. И наконец, если файл включает более 12+1024+1048576 блоков, то используется последнее 15-е поле для тройной косвенной адресации.

Данная система адресации позволяет при максимальном размере блока в 4096 байт иметь файлы, размер которых превышает 2 ТБ.

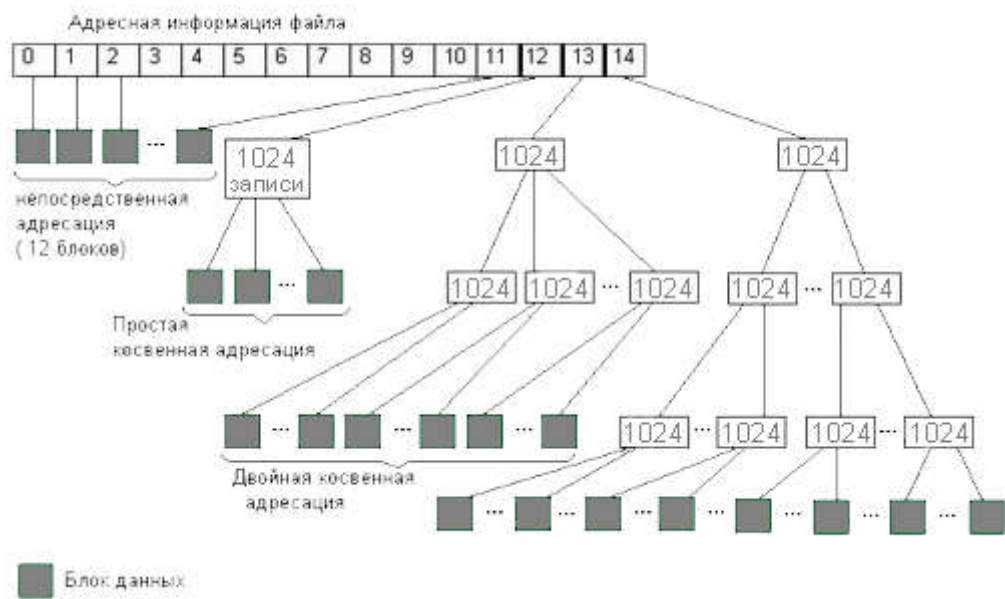


Рис.3 – физическая организация и адресация файла

8. Структура данных

inode:

- type: char – символьный тип - 1 байт
- len_iblock – целочисленный тип – 4 байта
- iblock[14] – целочисленный массив – 56 байт

В сумме: 64 байта

inode_table:

- $1024 \text{ элемента} \cdot 64 \text{ байта (размер inode)} = 65536 \text{ байт} = 64 \text{ Кбайт.}$

directory_element:

- inodeID: int – 4 байта
- name[10]: char – 10 байт

В сумме: 16 байт

bitmap[1024]:

- $\text{int} - 4 * 1024 = 4096 \text{ байт} = 4 \text{ Кбайт}$

9. Блок схема

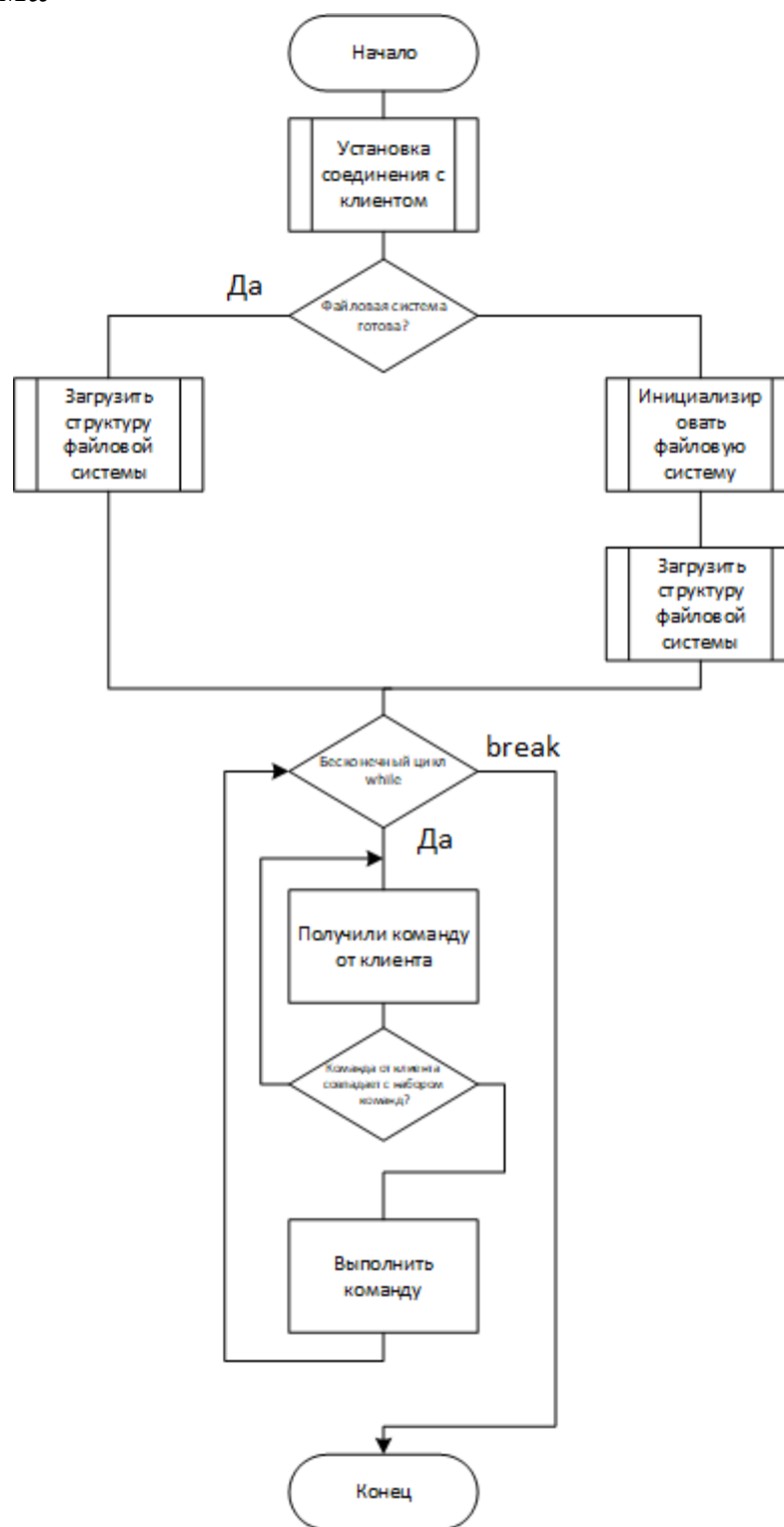


Рис.4 – примитивная блок схема алгоритма

10. Листинг

10.1. .../core/bitmap.h

```
#ifndef FILESYSTEM_SECTORS_BITMAP_H
#define FILESYSTEM_SECTORS_BITMAP_H

#include "../settings.h"
#include "sectors.h"

/**
 * 1 - занят, 0 - свободен
 */
int bitmap[1024];

/**
 * Первичное заполнение bitmap блока BITMAP_BLOCK
 */
void filling_blocks_bitmap(){
    char buf[BLOCK_SIZE];

    for(int i = 0; i < BLOCK_SIZE; i++){
        buf[i] = '|';
    }

    for(int i = 0; i < BITMAP_SIZE; i++){
        buf[i] = '0';
    }
}
```

```

    }

for(int i = 0; i< INODE_TABLE_START_BLOCK +
INODE_TABLE_BLOCK_COUNT; i++) {
    buf[i] = '1';
    }
buf[ROOT_DIRECTORY_BLOCK] = '1';

buf[BLOCK_SIZE - 1] = '\0';

set_sector(buf, BITMAP_BLOCK);
}

/**
 * Чтение bitmap блока
 */
void read_bitmap(){
    char buf[BLOCK_SIZE];
    get_sector(buf, BITMAP_BLOCK);

    for(int i = 0; i< BITMAP_SIZE; i++){
        char c = buf[i];
        if(c == '1'){
            bitmap[i] = 1;
        } else if ( c == '0'){
            bitmap[i] = 0;
        } else {
            perror("Read bitmap error: unknown symbol");
        }
    }
}

```

```

    }
}

```

```

/**

```

```

 * Запись bitmap в блок

```

```

 */

```

```

void write_bitmap(){
    char buf[BLOCK_SIZE];
    for(int i = 0; i< BLOCK_SIZE; i++){
        buf[i] = '|';
    }
    for(int i = 0; i< BITMAP_SIZE; i++){
        buf[i] = bitmap[i] + '0';
    }
    buf[BLOCK_SIZE - 1] = '\0';
    set_sector(buf, BITMAP_BLOCK);
}

```

```

/**

```

```

 * @return первый свободный блок данных, -1 - если свободных нет

```

```

 */

```

```

int get_free_block(){
    for(int i = 0; i< BITMAP_SIZE; i++){
        if(!bitmap[i]){
            return i;
        }
    }
}

```

```

#endif //FILESYSTEM_SECTORS_BITMAP_H

```

10.2. directory.h

```
#ifndef FILESYSTEM_DIRECTORY_H
#define FILESYSTEM_DIRECTORY_H
```

```
#include <stdlib.h>
#include "../settings.h"
#include "sectors.h"
#include "../commands/str_proc.h"
```

```
struct directory_element{
    int inodeID;
    charname[10];
};
```

```
/**
```

```
 * Первичная запись в блок пустой папки
```

```
 * @param block номер блока
```

```
 */
```

```
void filling_new_directory(int block){
    char buf[BLOCK_SIZE];
    for(int i = 0; i < BLOCK_SIZE; i++){
        buf[i] = EMPTY_SYMBOL;
    }
```

```
buf[0] = '|';
```



```

buf[BLOCK_SIZE - 1] = '\0';
set_sector(buf, block);
}

/**
 * Чтение папки из блока
 * @param directory массив directory_element длиной
MAX_FILE_IN_DIRECTORY
 * @param file_count ссылка int*, в которой будет размер массива
 * @param block номер блока
 */
void read_directory(struct directory_element
directory[MAX_FILE_IN_DIRECTORY],
int* file_count, int block){
    char buf[BLOCK_SIZE];
    get_sector(buf, block);

    if(buf[0] == '|'){
        *file_count = 0;
        return;
    }

    for(int i = 0; i < MAX_FILE_IN_DIRECTORY; i++){
        int s = DIRECTORY_ELEMENT_SIZE * i; // позиция в строке

        int num1 = buf[s++] - '0';
        int num2 = buf[s++] - '0';
        int num3 = buf[s++] - '0';
    }
}

```

```

    int num4 = buf[s++] - '0';
    directory[i].inodeID = num1 * 1000 + num2 * 100 + num3 * 10 + num4;
    s += 1;

for(int t = 0; t < FILE_NAME_SIZE; t++){
    directory[i].name[t] = buf[s++];
}

valid_name(directory[i].name);

    if(buf[s] == '|'){
        *file_count = i + 1;
        return;
    }

}

}

/**
 * Запись папки в блок
 * @param directory массив directory_element длиной file_count
 * @param file_count размер массива
 * @param block номер блока
 */
void write_directory(struct directory_element
directory[MAX_FILE_IN_DIRECTORY],
                    int file_count, int block){
    char buf[BLOCK_SIZE];

```

```

for(int i = 0; i< BLOCK_SIZE; i++){
buf[i] = EMPTY_SYMBOL;
}

int s = 0;

for(int i = 0; i<file_count; i++){
    int num = directory[i].inodeID;
    int num1 = num / 1000;
    int num2 = num / 100 % 10;
    int num3 = num / 10 % 10;
    int num4 = num % 10;

    buf[s++] = num1 + '0';
    buf[s++] = num2 + '0';
    buf[s++] = num3 + '0';
    buf[s++] = num4 + '0';

    buf[s++] = '.';

    char name[10];
    strcpy(name, directory[i].name);
    stored_name(name);

    for(int t = 0; t < FILE_NAME_SIZE; t++){
        buf[s++] = name[t];
    }

    buf[s++] = ';';

```

```

    }

    if(file_count == 0){
    buf[s] = '|';
        }
        else {
    buf[s - 1] = '|';
        }
    buf[BLOCK_SIZE - 1] = '\0';
    set_sector(buf, block);
}

void print_dir_by_structure(struct directory_element *directory, int len){
for(int i = 0; i<len; i++){
printf("%d -> ", i);
printf("inodeID:%d name:%s \n", directory[i].inodeID, directory[i].name);
    }
}

void print_dir_by_inode(int inode_ID){
    int dir_block = get_block(inode_ID);

    struct directory_element directory[MAX_FILE_IN_DIRECTORY];
    int* file_count = malloc(sizeof(int));
    read_directory(directory, file_count, dir_block);

    print_dir_by_structure(directory, *file_count);
}

#endif //FILESYSTEM_DIRECTORY_H

```

10.3. inode.h

```
#ifndef FILESYSTEM_INODE_H
#define FILESYSTEM_INODE_H

#include "../settings.h"
#include "sectors.h"

/**
 * type: e - empty, f - file, d - directory
 */
struct inode {
    char type;
    int len_iblock;
    int iblock[14];
};

struct inode inode_table[1024];

/**
 * Первичное заполнение блоков inode таблицы
 */
void filling_inode_table(){
    char inode_str[INODE_SIZE];

    inode_str[0] = 'e';
    inode_str[1] = ';';
```

```

for(int i = 2; i < 6; i++){
inode_str[i] = '0';
}
inode_str[6] = ';';

```

```

int s = 7;
for(int i = 0; i < 14; i++){
inode_str[s++] = '0';
inode_str[s++] = '0';
inode_str[s++] = '0';
inode_str[s++] = '0';
}

```

```

inode_str[INODE_SIZE - 1] = '|';

```

```

char table_buf[BLOCK_SIZE];
int n = BLOCK_SIZE / INODE_SIZE;
for(int i = 0; i < n; i++){
for(int j = 0; j < INODE_SIZE; j++){
table_buf[i * INODE_SIZE + j] = inode_str[j];
}
}
table_buf[BLOCK_SIZE - 1] = '\0';

```

```

for(int i = INODE_TABLE_START_BLOCK; i <
INODE_TABLE_BLOCK_COUNT + INODE_TABLE_START_BLOCK; i++){
set_sector(table_buf, i);
}
}

```

```

/**
 * Чтение inode таблицы
 */
void read_inode_table() {
    char buf[BLOCK_SIZE];

    for(int table_block = 0; table_block < INODE_TABLE_BLOCK_COUNT;
        table_block++) {
        get_sector(buf, table_block + INODE_TABLE_START_BLOCK);

        for (int i = 0; i < INODE_TABLE_IN_BLOCK; i++) {
            int j = i * INODE_SIZE; //номер в буфере
            int t = INODE_TABLE_IN_BLOCK * table_block + i; //номер в таблице

            inode_table[t].type = buf[j];
            j += 2;

            int num1, num2, num3, num4;
            num1 = buf[j++] - '0';
            num2 = buf[j++] - '0';
            num3 = buf[j++] - '0';
            num4 = buf[j++] - '0';
            inode_table[t].len_iblock = 1000 * num1 + 100 * num2 + 10 * num3 + num4;
            j += 1;

            for (int s = 0; s < IBLOCK_ARRAY_SIZE; s++) {
                num1 = buf[j++] - '0';
                num2 = buf[j++] - '0';
                num3 = buf[j++] - '0';
            }
        }
    }
}

```

```

        num4 = buf[j++] - '0';
inode_table[t].iblock[s] = 1000 * num1 + 100 * num2 + 10 * num3 + num4;
    }

    }

}

}

/**
 * Запись inode таблицы в блоки
 */
void write_inode_table(){
    char buf[BLOCK_SIZE];
    for(int table_block = 0; table_block < INODE_TABLE_BLOCK_COUNT;
        table_block++) {
        for (int i = 0; i < INODE_TABLE_IN_BLOCK; i++) {
            int j = i * INODE_SIZE; //номер в буфере
            int t = INODE_TABLE_IN_BLOCK * table_block + i; //номер в таблице

buf[j++] = inode_table[t].type;
buf[j++] = ';';

            int num = inode_table[t].len_iblock;
            int num1 = num / 1000;
            int num2 = num / 100 % 10;
            int num3 = num / 10 % 10;
            int num4 = num % 10;

```



```

buf[j++] = num1 + '0';
buf[j++] = num2 + '0';
buf[j++] = num3 + '0';
buf[j++] = num4 + '0';
buf[j++] = ';';

```

```

    for (int s = 0; s < IBLOCK_ARRAY_SIZE; s++) {
        int num = inode_table[t].iblock[s];
        int num1 = num / 1000;
        int num2 = num / 100 % 10;
        int num3 = num / 10 % 10;
        int num4 = num % 10;
        buf[j++] = num1 + '0';
        buf[j++] = num2 + '0';
        buf[j++] = num3 + '0';
        buf[j++] = num4 + '0';
    }
}

```

```

set_sector(buf, table_block + INODE_TABLE_START_BLOCK);
}
}

```

```

void print_inode_table(){
for(int i = 0; i < INODE_TABLE_SIZE; i++){
printf("%c;%d;", inode_table[i].type, inode_table[i].len_iblock);
for(int j = 0; j < IBLOCK_ARRAY_SIZE; j++){
printf("%d:", inode_table[i].iblock[j]);

```

```

}
printf("\n");
}
}
/**
 * Получение первого блока файла по inode
 * @param inodeID inode файла
 * @return первый блок
 */
int get_block(int inodeID){
    return inode_table[inodeID].iblock[0];
}
/**
 * @return первый свободный inode, -1 - если свободных нет
 */
int get_free_inode(){
    for(int i = 0; i < INODE_TABLE_SIZE; i++){
        if(inode_table[i].type == 'e'){
            return i;
        }
    }
}

```

```
#endif //FILESYSTEM_INODE_H
```

10.4. sectors.h

```

#ifndef FILESYSTEM_SECTORS_H
#define FILESYSTEM_SECTORS_H

```

```

#include <stdio.h>
#include <memory.h>
#include "../settings.h"

/**
 * Первичное заполнение всего файла
 */
void filling_main_file() {
    FILE* file = fopen(MAIN_FILE, "w");

    for(int i = 0; i < BLOCK_COUNT; i++) {
        for(int j = 0; j < BLOCK_SIZE; j++) {
            if(j == BLOCK_SIZE - 1) {
                fprintf(file, "\0");
            } else {
                fprintf(file, "%c", EMPTY_SYMBOL);
            }
        }
    }

    fclose(file);
}

/**
 * Считывание данных из сектора
 * @param buf - строка, в которую будут считаны данные длиной
    SECTOR_SIZE с завершающим нулем
 * @param sector_num - номер сектора с нуля
 */
void get_sector(char* buf, int sector_num) {
    FILE* file = fopen(MAIN_FILE, "r");

```

```

fseek(file, sector_num * BLOCK_SIZE, SEEK_SET);
fgets(buf, BLOCK_SIZE, file);

fclose(file);
}

/**
 * Запись данных в сектор
 * @paramstr - строка для записи длиной SECTOR_SIZE с завершающим
нулем
 * @paramsector_num - номер сектора с нуля
 */
void set_sector(const char* str, int sector_num){
    if(strlen(str) != BLOCK_SIZE - 1){
        fprintf(stderr, "Error argument len in set_sector\n");
        printf("%d", strlen(str));
    }

    FILE* file;

    char full[BLOCK_SIZE * BLOCK_COUNT];
    char buf[BLOCK_SIZE];

    file = fopen(MAIN_FILE, "r");

    for(int i = 0; i < BLOCK_COUNT; i++){
        get_sector(buf, i);
        for(int j = 0; j < BLOCK_SIZE; j++){
            full[i * BLOCK_SIZE + j] = buf[j];
        }
    }

```

```

    }

fclose(file);

    file = fopen(MAIN_FILE, "w");

    for(int i = 0; i < BLOCK_COUNT; i++){
        if(i == sector_num){
            for(int j = 0; j < BLOCK_SIZE; j++){
                fprintf(file, "%c", str[j]);
            }
        } else {
            for(int j = 0; j < BLOCK_SIZE; j++){
                fprintf(file, "%c", full[i * BLOCK_SIZE + j]);
            }
        }
    }

fclose(file);
}

/**
 * Затирает сектор пустыми символами
 * @param sector_num
 */
void clear_sector(int sector_num){
    char buf[BLOCK_SIZE];
    for(int i = 0; i < BLOCK_SIZE; i++){
        buf[i] = EMPTY_SYMBOL;
    }
}

```

```

    }
    buf[BLOCK_SIZE - 1] = '\0';

    set_sector(buf, sector_num);
}

#endif //FILESYSTEM_SECTORS_H

```

10.5. .../commands/base.h

```

#ifndef FILESYSTEM_COMMAND_H
#define FILESYSTEM_COMMAND_H

#include <stdlib.h>
#include "../settings.h"
#include "../core/inode.h"
#include "../core/directory.h"
#include "../core/bitmap.h"
#include "dir_inst.h"

```

```

/**
 * Добавляет файл/папку в структуру директории блока
 * @paramdir_inodeinode папки
 * @paramname имя файла с \0
 * @paramtype тип файла/папки
 */
int create_file_or_dir_in_directory(int dir_inode, const char *name, char type){
    if(inode_table[dir_inode].type != 'd'){
        fprintf(stderr, "It's not a directory\n");
        return -1;
    }

    char name_copy[FILE_NAME_SIZE];
    strcpy(name_copy, name);
    valid_name(name_copy);

    if(find_inode_in_directory(name_copy, dir_inode) != -1){
        fprintf(stderr, "This file has been created in the directory\n");
        return -1;
    }

    int dir_block = inode_table[dir_inode].iblock[0];

    struct directory_elementdirectory[MAX_FILE_IN_DIRECTORY];
    int* file_count = malloc(sizeof(int));
    read_directory(directory, file_count, dir_block);

    if(*file_count == MAX_FILE_IN_DIRECTORY){
        fprintf(stderr, "Max file in the directory\n");

```

```

        return -1;
    }

    int new_inode = get_free_inode();
    if(new_inode == -1){
        fprintf(stderr, "No free inode yet\n");
        return -1;
    }

    int new_block = get_free_block();
    if(new_block == -1){
        fprintf(stderr, "No free block yet\n");
    }

    bitmap[new_block] = 1;

    inode_table[new_inode].type = type;
    inode_table[new_inode].len_iblock = 1;
    inode_table[new_inode].iblock[0] = new_block;

    if(type == 'd'){
        filling_new_directory(new_block);
    }

    directory[*file_count].inodeID = new_inode;
    strcpy(directory[*file_count].name, name_copy);

    write_bitmap();
    write_inode_table();
    write_directory(directory, *file_count + 1, dir_block);

```



```
}
```

```
/**
```

```
* Удаляет файл/папку из структуры директории блока
```

```
* @paramdir_inode папки
```

```
* @paramname имя файла \0
```

```
*/
```

```
int delete_file_or_dir_in_directory(int dir_inode,  
                                   const char *name){
```

```
    if(inode_table[dir_inode].type != 'd'){
```

```
        fprintf(stderr, "It's not a directory\n");
```

```
        return -1;
```

```
    }
```

```
    char name_copy[FILE_NAME_SIZE];
```

```
    strcpy(name_copy, name);
```

```
    valid_name(name_copy);
```

```
    int file_inode = find_inode_in_directory(name_copy, dir_inode);
```

```
    if(file_inode == -1){
```

```
        fprintf(stderr, "This file was not found in the directory\n");
```

```
        return -1;
```

```
    }
```

```
    int file_block = inode_table[file_inode].iblock[0];
```

```
    if(inode_table[file_inode].type == 'd'){
```

```
        struct directory_elementdirectory[MAX_FILE_IN_DIRECTORY];
```

```
        int* file_count = malloc(sizeof(int));
```

```
        read_directory(directory, file_count, file_block);
```

```

if(*file_count != 0){
    fprintf(stderr, "This directory is not empty\n");
        return -1;
    }
}

int dir_block = inode_table[dir_inode].iblock[0];

struct directory_element directory[MAX_FILE_IN_DIRECTORY];
int* file_count = malloc(sizeof(int));
read_directory(directory, file_count, dir_block);

delete_inode_in_directory_by_structure(directory, *file_count, file_inode);
inode_table[file_inode].type = 'e';
inode_table[file_inode].len_iblock = 0;
    bitmap[file_block] = 0;
clear_sector(file_block);

write_bitmap();
write_inode_table();
write_directory(directory, *file_count - 1, dir_block);
}

#endif //FILESYSTEM_COMMAND_H

```

10.6. dir_inst.h

```
#ifndef FILESYSTEM_DIR_INST_H
#define FILESYSTEM_DIR_INST_H

#include <stdlib.h>
#include "../settings.h"
#include "../core/inode.h"
#include "../core/directory.h"

/**
 * Ищет inode по имени в папке
 * @param name имя файла с \0
 * @param dir_inode inode папки
 * @return inodeID - если файл в папке есть, -1 - если нет
 */
int find_inode_in_directory(const char name[FILE_NAME_SIZE],
                           int dir_inode){
    if(inode_table[dir_inode].type != 'd'){
        fprintf(stderr, "It's not a directory\n");
        return -1;
    }
    int dir_block = inode_table[dir_inode].iblock[0];

    struct directory_element directory[MAX_FILE_IN_DIRECTORY];
    int* file_count = malloc(sizeof(int));
    read_directory(directory, file_count, dir_block);

    for(int i = 0; i < *file_count; i++){
        if(!strcmp(directory[i].name, name)){
```

```

        return directory[i].inodeID;
    }
}
return -1;
}

/**
 * Находит inode последней папки в пути
 * ex. /dir1/dir2 ->inode dir2
 * "\0" -> ROOT_INODE_ID
 * @parampath полный путь к папке
 * @returninode или -1 в случае ошибки
 */
int find_inode_directory(const char path[MAX_PATH_LEN]){
    if(path[0] == '\0'){
        return ROOT_INODE_ID;
    }
    if(path[0] != '/'){
        fprintf(stderr, "Incorrect path\n");
        return -1;
    }

    int i_path = 1;
    int i_name = 0;
    char name[FILE_NAME_SIZE];
    int current_inodeID = ROOT_INODE_ID;
    while(1){
        char c = path[i_path++];
        if(c == '/'){
            name[i_name] = '\0';

```

```

current_inodeID = find_inode_in_directory(name, current_inodeID);
if(current_inodeID == -1){
fprintf(stderr, "This file was not found in the directory\n");

    return -1;

}
i_name = 0;
}
else if(c == '\0'){
    name[i_name] = c;
current_inodeID = find_inode_in_directory(name, current_inodeID);
if(current_inodeID == -1){
fprintf(stderr, "This file was not found in the directory\n");

    return -1;

}
    return current_inodeID;
}
else {
    name[i_name++] = c;
}
}
}

```

/**

* Удаляет элемент из переданной директории

* Warning: функция только удаляет файл из блока директории - она не удаляет файл

* из inode и bitmap.

* @paramdirectory ссылка на папку

* @paramfile_count количество файлов в папке

```

* @param inodeID inodeID удаляемого элемента
*/
void delete_inode_in_directory_by_structure(struct directory_element *directory,
                                           int file_count,
                                           int inodeID){
    int index = -1;
    for(int i = 0; i < file_count; i++){
        if(directory[i].inodeID == inodeID){
            index = i;
            break;
        }
    }
    if(index == -1){
        fprintf(stderr, "This file was not found in the directory\n");
        return;
    }
    for(int i = index + 1; i < file_count; i++){
        directory[i - 1].inodeID = directory[i].inodeID;
        strcpy(directory[i - 1].name, directory[i].name);
    }
}

#endif //FILESYSTEM_DIR_INST_H

```

10.7. initial.h

```
#ifndef FILESYSTEM_INITIAL_H
#define FILESYSTEM_INITIAL_H

#include "../core/sectors.h"
#include "../core/bitmap.h"
#include "../core/inode.h"
#include "../core/directory.h"

/**
 * Устанавливает блок статуса системы на уже создана
 */
void set_fs_creation_status(){
    char buf[BLOCK_SIZE];
    buf[0] = '1';
    for(int i = 1; i < BLOCK_SIZE - 1; i++){
        buf[i] = EMPTY_SYMBOL;
    }
    buf[BLOCK_SIZE - 1] = '\0';
    set_sector(buf, STATUS_BLOCK);
}

/**
 * Загружает блок статуса системы
 * @return 1 - создана, 0 - нет
 */
int get_fs_creation_status(){
    char buf[BLOCK_SIZE];
```

```

get_sector(buf, STATUS_BLOCK);
    returnbuf[0] = '0';
}

/**
 * Заполняет главный файл и блоки inode таблицы и bitmap и создает root
 каталог
 * при первом запуске системы.
 */
voidinit_file_system(){
    filling_main_file();
    filling_blocks_bitmap();
    filling_inode_table();

    filling_new_directory(ROOT_DIRECTORY_BLOCK);

    read_inode_table();
    inode_table[ROOT_INODE_ID].type = 'd';
    inode_table[ROOT_INODE_ID].len_iblock = 1;
    inode_table[ROOT_INODE_ID].iblock[0] = ROOT_DIRECTORY_BLOCK;
    write_inode_table();

    read_bitmap();
    bitmap[STATUS_BLOCK] = 1;
    write_bitmap();

    set_fs_creation_status();
}

```



```
/**
 * Загрузка в память всех нужных структур.
 * Должна запускаться перед каждой работой ФС.
 */
void load_file_system_structure(){
    read_bitmap();
    read_inode_table();
}

#endif //FILESYSTEM_INITIAL_H
```

10.8. main.h

```
#ifndef FILESYSTEM_COMMAND_MAIN_H
#define FILESYSTEM_COMMAND_MAIN_H

#include "../settings.h"
#include "str_proc.h"
#include "base.h"

int create_directory(char path[MAX_PATH_LEN]){
    if(path[0] != '/' && path[0] != '\0'){
        added_slash(path);
    }

    char dir[MAX_PATH_LEN];
    char name[FILE_NAME_SIZE];
    if(get_dir_and_name_in_path(path, dir, name) == -1){
        fprintf(stderr, "Incorrect path\n");
        return -1;
    }

    int inodeDir = find_inode_directory(dir);
    if(create_file_or_dir_in_directory(inodeDir, name, 'd') == -1){
        return -1;
    }
    return 0;
}

int delete_directory(char path[MAX_PATH_LEN]){
    if(path[0] != '/' && path[0] != '\0'){
```

```
added_slash(path);
```

```
}
```

```
char dir[MAX_PATH_LEN];
```

```
char name[FILE_NAME_SIZE];
```

```
if(get_dir_and_name_in_path(path, dir, name) == -1){
```

```
fprintf(stderr, "Incorrect path\n");
```

```
return -1;
```

```
}
```

```
int inodeDir = find_inode_directory(dir);
```

```
if(delete_file_or_dir_in_directory(inodeDir, name) == -1){
```

```
return -1;
```

```
}
```

```
return 0;
```

```
}
```

```
int create_file(char path[MAX_PATH_LEN]){
```

```
if(path[0] != '/' && path[0] != '\0'){
```

```
added_slash(path);
```

```
}
```

```
char dir[MAX_PATH_LEN];
```

```
char name[FILE_NAME_SIZE];
```

```
if(get_dir_and_name_in_path(path, dir, name) == -1){
```

```
fprintf(stderr, "Incorrect path\n");
```

```
return -1;
```

```
}
```

```
int inodeDir = find_inode_directory(dir);
```

```

if(create_file_or_dir_in_directory(inodeDir, name, 'f') == -1){
    return -1;
}
return 0;
}

```

```

int delete_file(char path[MAX_PATH_LEN]){
    if(path[0] != '/' && path[0] != '\0'){
added_slash(path);
    }
}

```

```

char dir[MAX_PATH_LEN];
char name[FILE_NAME_SIZE];
if(get_dir_and_name_in_path(path, dir, name) == -1){
fprintf(stderr, "Incorrect path\n");
    return -1;
}
}

```

```

int inodeDir = find_inode_directory(dir);
if(delete_file_or_dir_in_directory(inodeDir, name) == -1){
    return -1;
}
return 0;
}

```

```

int print_directory(char path[MAX_PATH_LEN]){
    if(path[0] != '/' && path[0] != '\0'){
added_slash(path);
    }
}

```

```

    int inodeDir = find_inode_directory(path);
    if(inodeDir == -1){
        fprintf(stderr, "Incorrect path\n");
        return -1;
    }

    int blockDir = get_block(inodeDir);

    struct directory_element directory[MAX_FILE_IN_DIRECTORY];
    int* file_count = malloc(sizeof(int));
    read_directory(directory, file_count, blockDir);

    if(*file_count == 0){
        printf("Directory is empty\n");
        return 0;
    }

    for(int i = 0; i < *file_count; i++){
        printf("%c %s\n",
            inode_table[directory[i].inodeID].type,
            directory[i].name);
    }
    return 0;
}

char* get_ls_directory(char path[MAX_PATH_LEN]){
    char* response = (char*) malloc(MAX_RESPONSE_LEN);

    if(path[0] != '/' && path[0] != '\0'){
        added_slash(path);
    }
}

```

```

    }

    int inodeDir = find_inode_directory(path);
    if(inodeDir == -1){
    strcpy(response, "Incorrect path\0");
        return response;
    }

    int blockDir = get_block(inodeDir);

    struct directory_element directory[MAX_FILE_IN_DIRECTORY];
    int* file_count = malloc(sizeof(int));
    read_directory(directory, file_count, blockDir);

    if(*file_count == 0){
    strcpy(response, "Directory is empty\0");
        return response;
    }

    response[0] = '\0';
    for(int i = 0; i < *file_count; i++){
        char buf[MAX_RESPONSE_LEN];

        sprintf(buf, "%c %s\n\0",
            inode_table[directory[i].inodeID].type,
                directory[i].name);
        strcat(response, buf);
    }
    response[strlen(response) - 1] = '\0';
    return response;

```

```

}

int write_file(char path[MAX_PATH_LEN],
               char str[BLOCK_SIZE]){
    if(path[0] != '/' && path[0] != '\0'){
added_slash(path);
    }

    char dir[MAX_PATH_LEN];
    char name[FILE_NAME_SIZE];
    if(get_dir_and_name_in_path(path, dir, name) == -1){
        fprintf(stderr, "Incorrect path\n");
        return -1;
    }

    int dir_inode = find_inode_directory(dir);
    int file_inode = find_inode_in_directory(name, dir_inode);
    if(inode_table[file_inode].type != 'f'){
        printf("It's not a file\n");
        return -1;
    }
    int file_block = get_block(file_inode);

    int s = 0;
    for(int i = 0; i < BLOCK_SIZE; i++){
        if(str[i] == '\0'){
            s = i;
            break;
        }
    }
}

```

```

for(int i = s; i< BLOCK_SIZE; i++){
    str[i] = EMPTY_SYMBOL;
}
str[BLOCK_SIZE - 1] = '\0';

set_sector(str, file_block);
return 0;
}

int read_file(char path[MAX_PATH_LEN]){
    if(path[0] != '/' && path[0] != '\0'){
added_slash(path);
    }

    char dir[MAX_PATH_LEN];
    char name[FILE_NAME_SIZE];
    if(get_dir_and_name_in_path(path, dir, name) == -1){
        fprintf(stderr, "Incorrect path\n");
        return -1;
    }

    int dir_inode = find_inode_directory(dir);
    int file_inode = find_inode_in_directory(name, dir_inode);
    if(inode_table[file_inode].type != 'f'){
        printf("It's not a file\n");
        return -1;
    }

    int file_block = get_block(file_inode);

```



```

    char buf[BLOCK_SIZE];
    get_sector(buf, file_block);
    for(int i = 0; i < BLOCK_SIZE; i++){
        if(buf[i] == EMPTY_SYMBOL){
            buf[i] = '\0';
        }
    }
    printf("%s", buf);
    return 0;
}

```

```

char* get_file(char path[MAX_PATH_LEN]){
    if(path[0] != '/' && path[0] != '\0'){
        added_slash(path);
    }
}

```

```

char* response = (char*) malloc(MAX_RESPONSE_LEN);

```

```

char dir[MAX_PATH_LEN];
char name[FILE_NAME_SIZE];
if(get_dir_and_name_in_path(path, dir, name) == -1){
    strcpy(response, "Incorrect path\0");
    return response;
}

```

```

int dir_inode = find_inode_directory(dir);
int file_inode = find_inode_in_directory(name, dir_inode);
if(inode_table[file_inode].type != 'f'){
    strcpy(response, "It's not a file\0");
    return response;
}

```

```

    }
    int file_block = get_block(file_inode);

    char buf[BLOCK_SIZE];
    get_sector(buf, file_block);
    for(int i = 0; i < BLOCK_SIZE; i++){
        if(buf[i] == EMPTY_SYMBOL){
            buf[i] = '\0';
        }
    }
    strcpy(response, buf);
    return response;
}

#endif //FILESYSTEM_COMMAND_MAIN_H

```

10.9. run.h

```
#ifndef FILESYSTEM_RUN_H
#define FILESYSTEM_RUN_H

#include "../settings.h"
#include "initial.h"
#include "main.h"

/**
 * Функция, циклично обрабатывающая команды пользователя
 */
void run() {
    char response[MAX_RESPONSE_LEN];

    if(get_fs_creation_status() == 1){
load_file_system_structure();

strcpy(response, "File system is ready!\n"
            "Enter command:\n");
printf(response);

    }
    else{
init_file_system();
load_file_system_structure();

strcpy(response, "File system created!\n");
printf(response);
    }
}
```

```

    char full_command[FULL_COMMAND_SIZE];
while(1){
fgets(full_command, FULL_COMMAND_SIZE, stdin);
    if(full_command[0] == '\n'){
        continue;
    }
for(int i = 0; i< FULL_COMMAND_SIZE; i++){
    if(full_command[i] == '\n'){
full_command[i] = '\0';
        break;
    }
}
if(!strcmp(full_command, INIT)){
    if(get_fs_creation_status() == 1){
strcpy(response, "File system has already been created.\n"
        "Want to re-create? (All data will be lost) [yes/no]:\n");
printf(response);

        char answer[4];
fgets(answer, 4, stdin);
for(int i = 0; i< 4; i++){
    if(answer[i] == '\n'){
        answer[i] = '\0';
        break;
    }
}
if(strcmp(answer, "yes\0")){
    continue;
}

```

```

        }

fprintf(stdin, "Processing...\n");

init_file_system();
load_file_system_structure();

strcpy(response, "File system created!\n");
printf(response);
    }
    else if(!strcmp(full_command, EXIT)){
        break;
    }
    else if(!strcmp(full_command, HELP)){
sprintf(response, "%s\n"
                "%s\n"
                "%s <path>\n"
                "%s <path>\n"
                "%s <path>\n"
                "%s <path>\n"
                "%s <path>\n",
                INIT, EXIT, MKDIR, RMDIR, TOUCH, RM, LS);
printf(response);
    }
    else if(!strcmp(full_command, LS)){
strcpy(response, get_ls_directory("\0"));
printf(response);
    }
    else {
        char command[COMMAND_SIZE];
        char path[MAX_PATH_LEN];

```

```

command[0] = '\0';
path[0] = '\0';

get_command_and_path(full_command, command, path);
if(!strcmp(command, MKDIR)){
fprintf(stdin, "Processing...\n");
    if(create_directory(path) != 0){
strcpy(response, "Error\n");
printf(response);
    }
    else {
strcpy(response, "Directory created!\n");
printf(response);
    }
}
else if(!strcmp(command, RMDIR)){
fprintf(stdin, "Processing...\n");
    if(delete_directory(path) != 0){
strcpy(response, "Error\n");
printf(response);
    }
    else {
strcpy(response, "Directory deleted!\n");
printf(response);
    }
}
else if(!strcmp(command, TOUCH)){
fprintf(stdin, "Processing...\n");
    if(create_file(path) != 0){

```

```

strcpy(response, "Error\n");
printf(response);
    }
    else {
strcpy(response, "File created!\n");
printf(response);
    };
}
    else if(!strcmp(command, RM)){
fprintf(stdin, "Processing...\n");
        if(delete_file(path) != 0){
strcpy(response, "Error\n");
printf(response);
        }
        else {
strcpy(response, "File deleted!\n");
printf(response);
        };
    }
    else if(!strcmp(command, LS)){
        if(print_directory(path) != 0){
strcpy(response, "Error\n");
printf(response);
        }
    }
    else if(!strcmp(command, CAT)){
        if(read_file(path) != 0){
strcpy(response, "Error\n");
printf(response);
        }
    }

```

```

    }
    else if(!strcmp(command, ECHO)){
strcpy(response, "Enter text:\n");
printf(response);

        char buf[BLOCK_SIZE];
fgets(buf, BLOCK_SIZE, stdin);
for(int i = 0; i< FULL_COMMAND_SIZE; i++){
        if(full_command[i] == '\n'){
full_command[i] = '\0';
                break;
        }
    }
if(write_file(path, buf) != 0){
strcpy(response, "Error\n");
printf(response);
        } else {
strcpy(response, "Text saved\n");
printf(response);
        }
    }
else{
strcpy(response, "Unknown command\n");
printf(response);
    }
}
}
}

#endif //FILESYSTEM_RUN_H

```


10.10. str_proc.h

```
#ifndef FILESYSTEM_STRING_PROC_H
#define FILESYSTEM_STRING_PROC_H

#include <memory.h>
#include <stdio.h>
#include "../settings.h"

/**
 * Преобразует строку для хранения в структуре
 * ex. "name++++++" -> "name\0"
 * @param name
 */
void valid_name(char name[FILE_NAME_SIZE]){
    for(int i = 0; i< FILE_NAME_SIZE; i++){
        if(name[i] == EMPTY_SYMBOL){
            name[i] = '\0';
            break;
        }
    }
}

/**
 * Преобразует строку для хранения в блоке
 * ex. "name\0" -> "name++++++"
 * @param name
 */
void stored_name(char name[FILE_NAME_SIZE]){
    int s = 0;
```

```

for(int i = 0; i< FILE_NAME_SIZE; i++){
    if(name[i] == '\0'){
        s = i;
        break;
    }
}
for(int i = s; i< FILE_NAME_SIZE; i++){
    name[i] = EMPTY_SYMBOL;
}
}

/**
 * Разделяет входную строку на команду и путь
 * ex. mkdir /dir/dir1 ->mkdir and /dir/dir1
 * @param full_command
 * @param command
 * @param path
 */
void get_command_and_path(const char
full_command[FULL_COMMAND_SIZE],
                        char command[COMMAND_SIZE],
                        char path[MAX_PATH_LEN]){
    int s = 0;
    int is_command = 1;
    for(int i = 0; i<= strlen(full_command); i++){
        if(i == strlen(full_command)){
            path[i - s] = '\0';
            break;
        }
        char c = full_command[i];

```

```

if(c == ' '){
    s = i + 1;
    command[i] = '\0';
is_command = 0;
}
else {
    if(is_command){
        command[i] = c;
    }
    else {
path[i - s] = c;
}
}
}
}

/**
 * Получение каталоговой и файловой части пути
 * ex. /dir1/dir2/file -> /dir1/dir2 and file
 * ex. /file -> "\0" and file
 * @param path полный путь с \0
 * @param dir каталоговая часть
 */
int get_dir_and_name_in_path(const char path[MAX_PATH_LEN],
                             char dir[MAX_PATH_LEN],
                             char name[FILE_NAME_SIZE]){
    if(path[0] != '/'){
fprintf(stderr, "Incorrect path\n");
        return -1;
    }
}

```

```

dir[0] = '\0';
name[0] = '\0';

int i_path = 1;
int i_dir = 0;
int i_name = 0;
int i_cur = 0;
char cur_name[FILE_NAME_SIZE];
cur_name[i_cur++] = '/';
int is_first = 1;
while(1){
    char c = path[i_path++];
    if(c == '/'){
        cur_name[i_cur++] = c;
        for(int i = 0; i < i_cur; i++){
            dir[i + i_dir] = cur_name[i];
        }
        i_dir += i_cur;
        i_cur = 0;
        is_first = 0;
    }
    else if(c == '\0'){
        int def = 0;
        for(int i = 0; i < i_cur; i++){
            if(is_first){
                def = 1;
            }
            is_first = 0;
            continue;
        }
        name[i - def] = cur_name[i];
    }
}

```

```

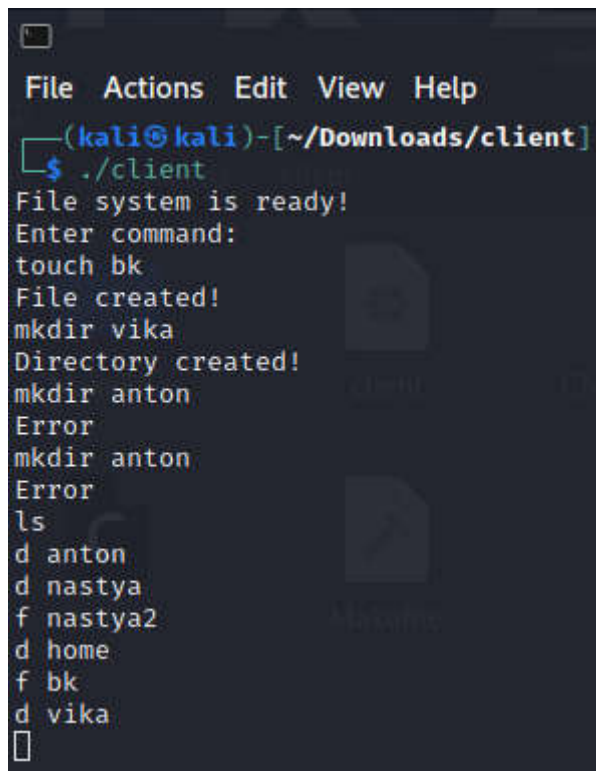
    i_name += i_cur;
    }
    name[i_cur - def] = '\0';
    if(is_first) {
    dir[i_dir] = '\0';
    }
    else{
    dir[i_dir - 1] = '\0';
    }
    break;
    }
    else {
    cur_name[i_cur++] = c;
    }
    }
    return 0;
}

/**
 * Добавляет / в начало переданной строки
 * @param str
 */
void added_slash(char str[MAX_PATH_LEN]){
for(int i = strlen(str) + 1; i >= 1; i--){
    str[i] = str[i - 1];
    }
    str[strlen(str) + 1] = '\0';
    str[0] = '/';
}

```

```
#endif //FILESYSTEM_STRING_PROC_H
```

11. Результат работы



```
File Actions Edit View Help
(kali㉿kali)-[~/Downloads/client]
$ ./client
File system is ready!
Enter command:
touch bk
File created!
mkdir vika
Directory created!
mkdir anton
Error
mkdir anton
Error
ls
d anton
d nastya
f nastya2
d home
f bk
d vika

```

Рис. 5 –пример работы клиента-механизма

12. Список возможных команд

Для того, чтобы посмотреть какие функции в проекте реализованы необходимо ввести команду `help`.

- `init` - Заполняет главный файл и блоки `inode` таблицы и `bitmap` и создает `root` каталог при первом запуске системы.
- `mkdir` – создание директории.
- `rmdir` – удаление директории.
- `touch` – создание файла.
- `rm` – удаление файла.
- `ls` – вывод содержимого директории.
- `cat` – просмотреть содержимое файла.
- `echo` – запись строки в файл.