МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
**«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ
УНИВЕРСИТЕТ им. В. Г. ШУХОВА»**
(БГТУ им. В.Г. Шухова)

ИНСТИТУТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И УПРАВЛЯЮЩИХ СИСТЕМ

Лабораторная работа №3

по дисциплине: Операционные системы
тема: «Файловые системы в ОС Linux (Ubuntu): сравнение, области эффективности. Виртуальная файловая система. Пользовательская файловая система.»

Выполнил: ст. группы ПВ-223
Пахомов Владислав Андреевич

Проверили:
доц. Островский Алексей Мичеславович
асс. Четвертухин Виктор Романович

Белгород 2024 г.

Цель работы: Изучить популярные файловые системы в ОС Linux (ext4, Btrfs, ReiserFS, NTFS, FAT32), определить область эффективности каждой их них, разобраться как осуществляется работа с виртуальной файловой системой (VFS) ОС Linux и выполнить разработку пользовательской файловой системы в соответствии с индивидуальным заданием.

Условие индивидуального задания: Поиск подстроки в файле (моделирование нагрузки на чтение файлов). Алгоритм Бойера- Мура.

Реализовать файловую систему для временного хранения данных. Файлы автоматически удаляются через заданный промежуток времени после создания.

Ход выполнения работы

stats_prepare.py

```
import os
import time
import matplotlib.pyplot as plt
from pathlib import Path
from subprocess import Popen, PIPE, STDOUT

MOUNT_POINTS = {
    "ext4": "/mnt/ext4",
    "btrfs": "/mnt/btrfs",
    "reiserfs": "/mnt/reiserfs",
    "ntfs": "/mnt/ntfs",
    "fat32": "/mnt/fat32",
}

# matplotlib.use('TkAgg')

LARGE_FILE_SIZE_MB = 100
SMALL_FILES_COUNT = 1000
SMALL_FILE_SIZE_KB = 10

def write_large_file(directory, size_mb):
    filepath = os.path.join(directory, "large_file.test")
    start_time = time.time()
    with open(filepath, 'wb') as f:
        f.write(b'\0' * (size_mb * 1024 * 1024))
    duration = time.time() - start_time
    os.remove(filepath)
    return duration

def write_small_files(directory, count, size_kb):
    start_time = time.time()
    for i in range(count):
        filepath = os.path.join(directory, f"small_file_{i}.test")
        with open(filepath, 'wb') as f:
            f.write(b'\0' * (size_kb * 1024))
        os.remove(filepath)
    return time.time() - start_time
```

```
def search_large_file(directory, size_mb):
    filepath = os.path.join(directory, "search_large_file.test")
    with open(filepath, 'wb') as f:
        f.write(b'r' * (size_mb * 1024 * 1024))
        f.write(b'b')
        f.write(b'\0')
    p = Popen(["./lab3_task1"], stdout=PIPE, stdin=PIPE, stderr=PIPE, text=True)
    start_time = time.time()
    stdout_data = p.communicate(input=f"{filepath}\nb\n")
    status = p.wait()
    duration = time.time() - start_time
    os.remove(filepath)
    return duration

def search_large_file_not_found(directory, size_mb):
    filepath = os.path.join(directory, "search_large_file_not_found.test")
    with open(filepath, 'wb') as f:
        f.write(b'r' * (size_mb * 1024 * 1024))
        f.write(b'b')
        f.write(b'\0')
    p = Popen(["./lab3_task1"], stdout=PIPE, stdin=PIPE, stderr=PIPE, text=True)
    start_time = time.time()
    stdout_data = p.communicate(input=f"{filepath}\na\n")
    status = p.wait()
    duration = time.time() - start_time
    os.remove(filepath)
    return duration



def search_small_file(directory, size_kb):
    filepath = os.path.join(directory, "search_small_file.test")
    with open(filepath, 'wb') as f:
        f.write(b'r' * (size_kb * 1024))
        f.write(b'b')
        f.write(b'\0')
    p = Popen(["./lab3_task1"], stdout=PIPE, stdin=PIPE, stderr=PIPE, text=True)
    start_time = time.time()
    stdout_data = p.communicate(input=f"{filepath}\nb\n")
    status = p.wait()
    duration = time.time() - start_time
    os.remove(filepath)
    return duration

def search_small_file_not_found(directory, size_kb):
    filepath = os.path.join(directory, "search_small_file_not_found.test")
    with open(filepath, 'wb') as f:
        f.write(b'r' * (size_kb * 1024))
        f.write(b'b')
        f.write(b'\0')
    p = Popen(["./lab3_task1"], stdout=PIPE, stdin=PIPE, stderr=PIPE, text=True)
    start_time = time.time()
    stdout_data = p.communicate(input=f"{filepath}\na\n")
    status = p.wait()
    duration = time.time() - start_time
```

```
    os.remove(filepath)
    return duration

def read_large_file(directory, size_mb):
    filepath = os.path.join(directory, "large_file.test")
    with open(filepath, 'wb') as f:
        f.write(b'\0' * (size_mb * 1024 * 1024))
    start_time = time.time()
    with open(filepath, 'rb') as f:
        f.read()
    duration = time.time() - start_time
    os.remove(filepath)
    return duration

def read_small_files(directory, count, size_kb):
    filepaths = []
    for i in range(count):
        filepath = os.path.join(directory, f"small_file_{i}.test")
        with open(filepath, 'wb') as f:
            f.write(b'\0' * (size_kb * 1024))
        filepaths.append(filepath)
    start_time = time.time()
    for filepath in filepaths:
        with open(filepath, 'rb') as f:
            f.read()
        os.remove(filepath)
    return time.time() - start_time

def delete_files(directory, count):
    filepaths = [os.path.join(directory, f"delete_file_{i}.test") for i in range(count)]
    for filepath in filepaths:
        Path(filepath).touch()
    start_time = time.time()
    for filepath in filepaths:
        os.remove(filepath)
    return time.time() - start_time



results = {fs: {} for fs in MOUNT_POINTS.keys()}
for fs, path in MOUNT_POINTS.items():
    os.makedirs(path, exist_ok=True)
    results[fs]['Поиск (успешный) в маленьком файле'] = search_small_file(path, 10)
    results[fs]['Поиск (неуспешный) в маленьком файле'] = search_small_file_not_found(path, 10)
    results[fs]['Поиск (успешный) в большом файле'] = search_large_file(path, 1)
    results[fs]['Поиск (неуспешный) в большом файле'] = search_large_file_not_found(path, 1)
    results[fs]['Запись большого файла'] = write_large_file(path, LARGE_FILE_SIZE_MB)
    results[fs]['Запись маленьких файлов'] = write_small_files(path, SMALL_FILES_COUNT, SMALL_FILE_SIZE_KB)
    results[fs]['Чтение большого файла'] = read_large_file(path, LARGE_FILE_SIZE_MB)
    results[fs]['Чтение маленьких файлов'] = read_small_files(path, SMALL_FILES_COUNT, SMALL_FILE_SIZE_KB)
    results[fs]['Удаление файлов'] = delete_files(path, SMALL_FILES_COUNT)

for operation in list(results.values())[0].keys():
    plt.figure(figsize=(10, 6))
    times = [results[fs][operation] for fs in MOUNT_POINTS.keys()]
```

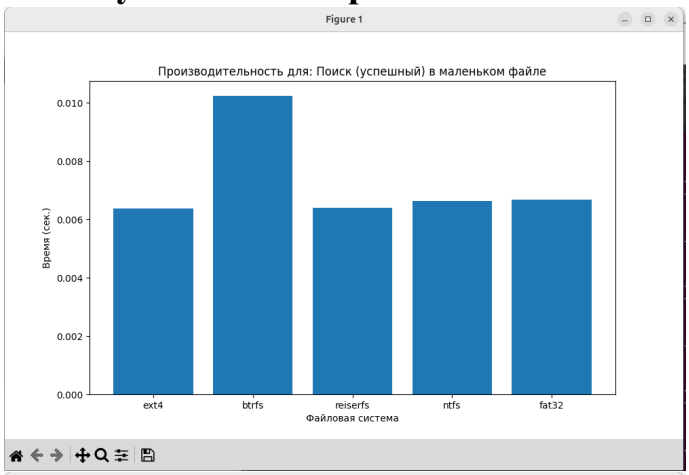
```
    plt.bar(MOUNT_POINTS.keys(), times)
    plt.title(f"Производительность для: {operation}")
    plt.xlabel("Файловая система")
    plt.ylabel("Время (сек.)")
    plt.show()
```

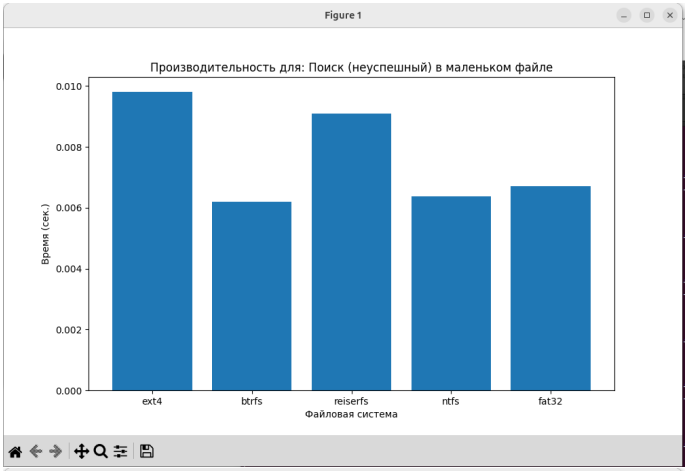
task1.rs

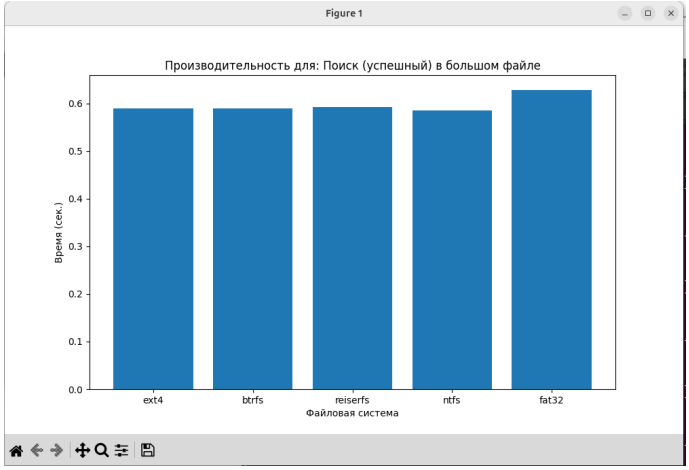
```
use std::io::{Read, Seek};

fn main() {
    println!("Введите путь к файлу: ");
    let mut file_path = String::new();
    std::io::stdin().read_line(&mut file_path).unwrap();
    let file_path = file_path.trim();

    let mut file = match std::fs::File::open(file_path) {
        Ok(res) => res,
        Err(_err) => {
            println!("Файла не существует");
            std::process::exit(1);
        }
    };

    println!("Введите искомую подстроку: ");
    let mut search_string = String::new();
    std::io::stdin().read_line(&mut search_string).unwrap();
    let search_string = search_string.trim();
    let mut search_index = search_string.as_bytes().len() as i64 - 1;
    let mut read_index = search_string.as_bytes().len() as i64 - 1;

    let mut offset_map = std::collections::HashMap::new();

    for i in 0..(search_string.as_bytes().len()) {
        offset_map.insert(search_string.as_bytes()[i], i as i64);
    }
    file.seek(std::io::SeekFrom::Start(read_index as u64)).unwrap();

    let found_index: i64 = loop {
        let mut buf = [0u8];
        match file.read_exact(&mut buf) {
            Ok(res) => {
                res
            },
            Err(_err) => {
                break -1
            }
        };

        if search_string.as_bytes()[search_index as usize] == buf[0] {
            if search_index == 0 {
                break read_index as i64
            }
```

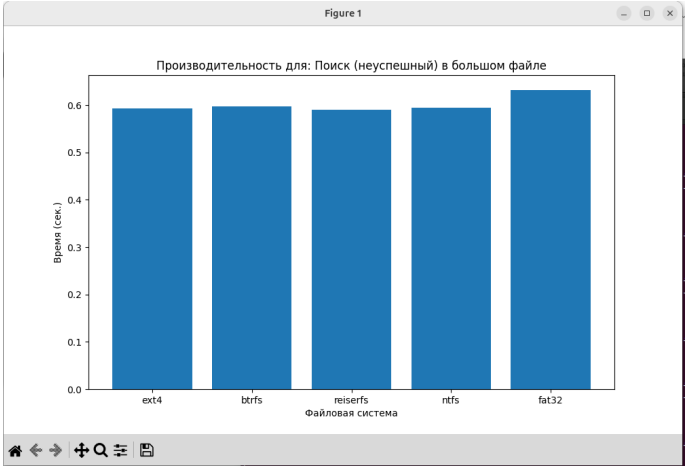
```
            search_index = search_index - 1;
            read_index -= 1;
            file.seek(std::io::SeekFrom::Current(-2)).unwrap();
        } else {
            if offset_map.contains_key(&buf[0]) {
                search_index = search_string.as_bytes().len() as i64 - 1;
                read_index += search_string.as_bytes().len() as i64 - offset_map.get(&buf[0]).unwrap() - 1;
                file.seek(std::io::SeekFrom::Current(search_string.as_bytes().len() as i64 - offset_map.get(&buf[0]).unwrap() -
↪  2)).unwrap();
            } else {
                search_index = search_string.as_bytes().len() as i64 - 1;
                read_index += search_string.as_bytes().len() as i64;
                file.seek(std::io::SeekFrom::Current(search_string.as_bytes().len() as i64 - 1)).unwrap();
            }
        }
    };

    if found_index == -1 {
        println!("Подстрока не найдена");
    } else {
        println!("Подстрока найдена на позиции {found_index}")
    }
}
```

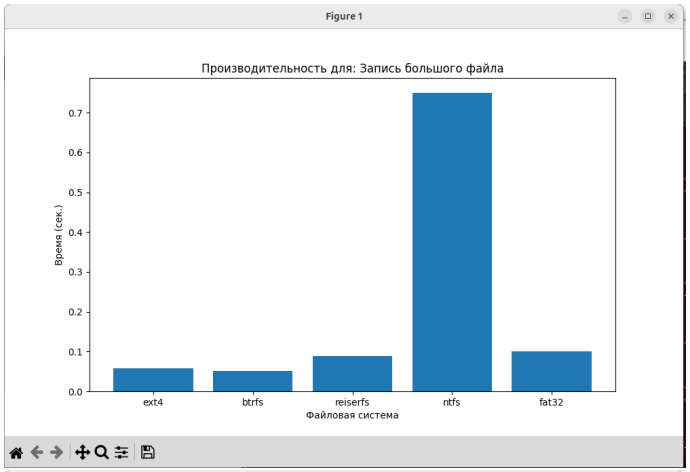
Результаты замеров:

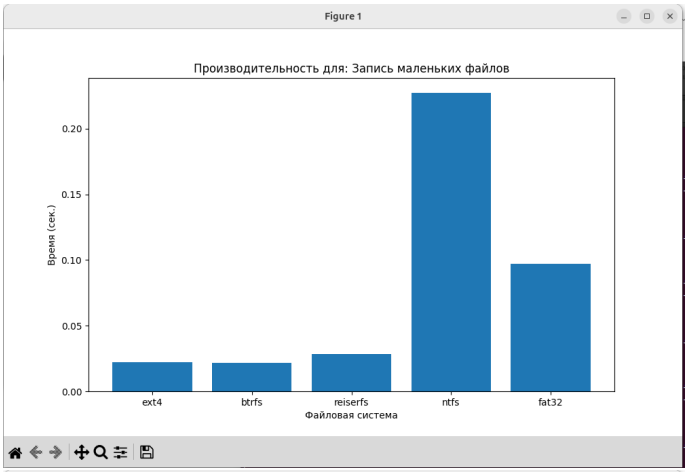


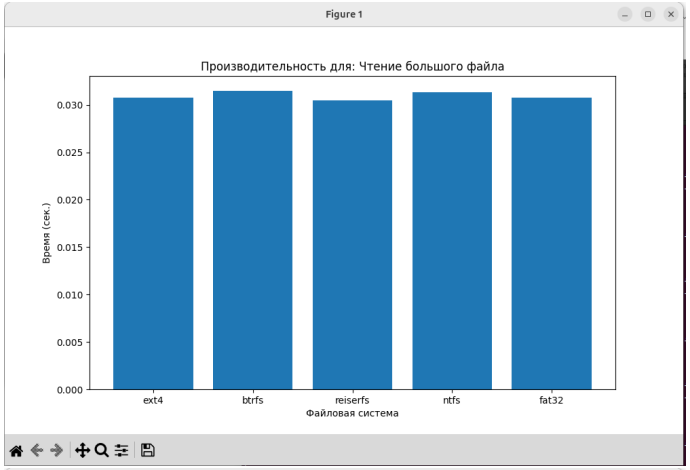


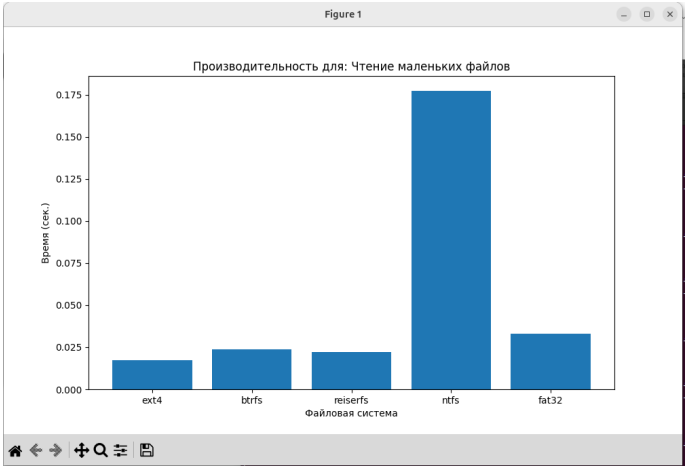


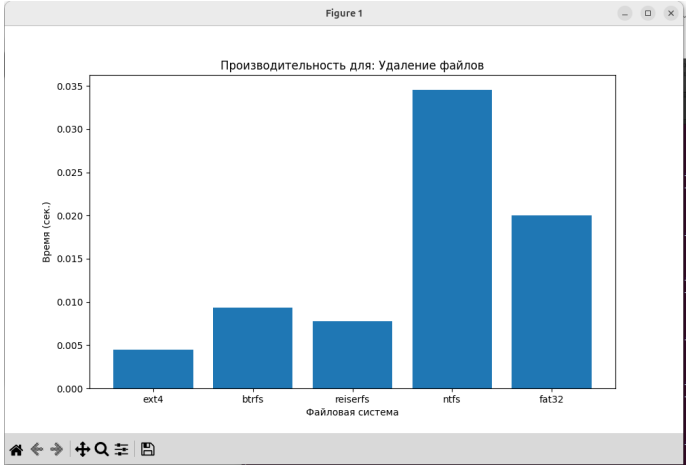












task2.c

```c
#define _POSIX_C_SOURCE 200809L // Стандарт POSIX версии 2008 года с некоторыми дополнениями
#define FUSE_USE_VERSION 30

#include <time.h>
#include <fuse3/fuse.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/stat.h> // Для S_IFDIR и S_IFREG
#include <time.h>

static const char *LOG_FILE = "/tmp/os_lab3_tmp.txt";
```

```c
// Возвращает 1 и сохраняет дату в parsed_time,
// если имя файла name начинается с валидной даты, иначе - 0
int parse_time(struct tm *parsed_time, const char *name)
{
    struct tm local;
    if (!strptime(name, "%d.%m.%Y %H:%M:%S", &local))
    {
        return 0;
    }

    *parsed_time = local;
    return 1;
}

// Возвращает 1, если tm1 > tm2, иначе - 0.
int cmp_time(struct tm *tm1, struct tm *tm2)
{
    if (tm1->tm_year != tm2->tm_year)
    {
        return tm1->tm_year > tm2->tm_year;
    }
    else if (tm1->tm_mon != tm2->tm_mon)
    {
        return tm1->tm_mon > tm2->tm_mon;
    }
    else if (tm1->tm_mday != tm2->tm_mday)
    {
        return tm1->tm_mday > tm2->tm_mday;
    }
    else if (tm1->tm_hour != tm2->tm_hour)
    {
        return tm1->tm_hour > tm2->tm_hour;
    }
    else if (tm1->tm_min != tm2->tm_min)
    {
        return tm1->tm_min > tm2->tm_min;
    }
    else if (tm1->tm_sec != tm2->tm_sec)
    {
        return tm1->tm_sec > tm2->tm_sec;
    }
    return 1;
}

typedef struct file_node
{
    char *name;
    char *contents;
    struct file_node *next;
    struct tm expires;
} file_node;

static file_node *file_list = NULL;
```

```c
void log_operation(const char *operation, const char *path)
{
    FILE *fp = fopen(LOG_FILE, "a");
    if (!fp)
    {
        perror("Ошибка открытия лог-файла");
        return;
    }

    time_t now = time(NULL);
    struct tm *t = localtime(&now);

    fprintf(fp, "[%04d-%02d-%02d %02d:%02d:%02d] PID: %d | Операция: %s | Путь: %s\n",
            t->tm_year + 1900,
            t->tm_mon + 1,
            t->tm_mday,
            t->tm_hour,
            t->tm_min,
            t->tm_sec,
            getpid(),
            operation,
            path);

    fclose(fp);
}

file_node *find_file(const char *name)
{
    if (file_list == NULL)
        return NULL;

    time_t now_time_t;
    struct tm *now;
    time(&now_time_t);
    now = localtime(&now_time_t);

    while (file_list && cmp_time(now, &file_list->expires))
    {
        file_node *old_file_list = file_list;
        file_list = file_list->next;
        free(old_file_list);
    }
    file_node *current = file_list;
    while (current)
    {
        if (strcmp(current->name, name) == 0)
        {
            return current;
        }

        while (current->next && cmp_time(now, &current->next->expires))
        {
            file_node *old_next = current->next;
```

```
            current->next = current->next->next;
            free(old_next);
        }

        current = current->next;
    }
    return NULL;
}

file_node *create_file(const char *name)
{
    struct tm file_expires;
    if (!parse_time(&file_expires, name))
        return NULL;

    time_t now_time_t;
    struct tm *now;
    time(&now_time_t);
    now = localtime(&now_time_t);
    if (cmp_time(now, &file_expires))
    {
        return NULL;
    }

    file_node *new_file = malloc(sizeof(file_node));
    if (!new_file)
    {
        return NULL;
    }
    new_file->name = strdup(name);
    new_file->contents = strdup("");
    new_file->next = file_list;
    new_file->expires = file_expires;
    file_list = new_file;
    return new_file;
}

void create_backup(const char *name)
{
    file_node *file = find_file(name);
    if (!file)
        return;

    size_t bak_name_len = strlen(name) + 5; // Длина имени + ".bak"
    char *bak_name = malloc(bak_name_len);
    if (!bak_name)
        return;

    snprintf(bak_name, bak_name_len, "%s.bak", name);

    // Проверяем, есть ли уже .bak файл, и удаляем его
    file_node *bak_file = find_file(bak_name);
    if (bak_file)
    {
```

```
        free(bak_file->contents);
        bak_file->contents = NULL;
    }
    else
    {
        bak_file = create_file(bak_name);
    }

    if (bak_file)
    {
        bak_file->contents = strdup(file->contents);
    }

    free(bak_name);
}

static int fs_getattr(const char *path, struct stat *stbuf, struct fuse_file_info *fi)
{
    (void)fi; // Параметр не используется
    memset(stbuf, 0, sizeof(struct stat));

    if (strcmp(path, "/") == 0)
    {
        stbuf->st_mode = S_IFDIR | 0755;
        stbuf->st_nlink = 2;
    }
    else
    {
        file_node *file = find_file(path + 1);
        if (!file)
        {
            return -ENOENT;
        }
        stbuf->st_mode = S_IFREG | 0644;
        stbuf->st_nlink = 1;
        stbuf->st_size = strlen(file->contents);
    }

    return 0;
}

static int fs_readdir(const char *path, void *buf, fuse_fill_dir_t filler, off_t offset,
                      struct fuse_file_info *fi, enum fuse_readdir_flags flags)
{
    (void)offset;
    (void)fi;
    (void)flags; // Не используются

    if (strcmp(path, "/") != 0)
    {
        return -ENOENT;
    }

    filler(buf, ".", NULL, 0, 0);
```

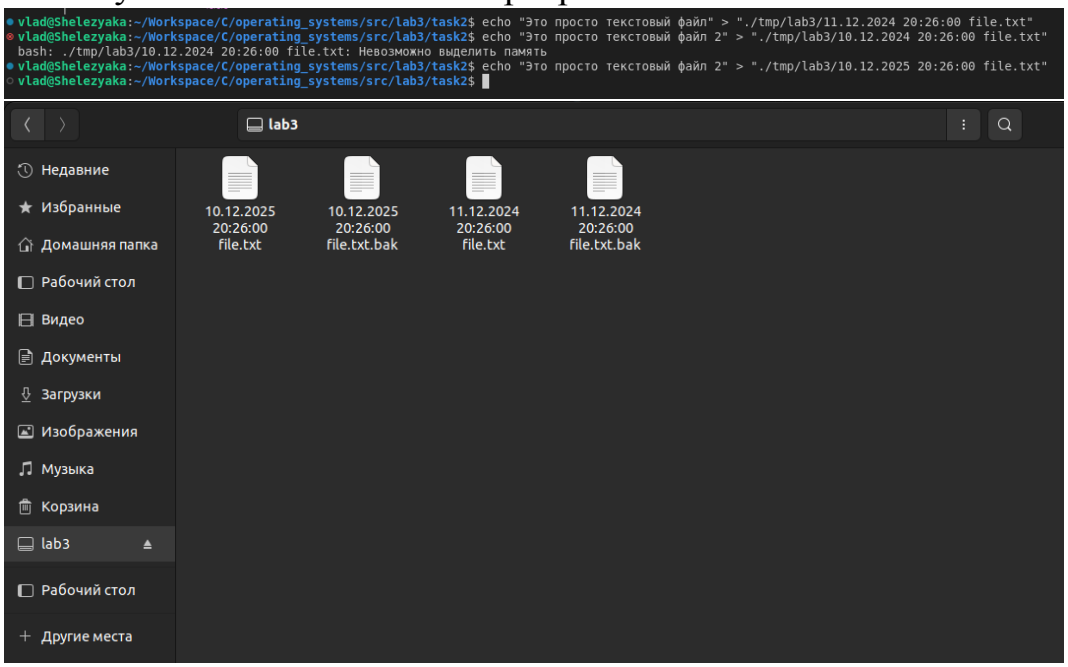
```
    filler(buf, "..", NULL, 0, 0);

    if (file_list == NULL)
        return 0;

    time_t now_time_t;
    struct tm *now;
    time(&now_time_t);
    now = localtime(&now_time_t);

    while (file_list && cmp_time(now, &file_list->expires))
    {
        file_node *old_file_list = file_list;
        file_list = file_list->next;
        free(old_file_list);
    }
    file_node *current = file_list;

    while (current)
    {
        filler(buf, current->name, NULL, 0, 0);

        while (current->next && cmp_time(now, &current->next->expires))
        {
            file_node *old_next = current->next;
            current->next = current->next->next;
            free(old_next);
        }

        current = current->next;
    }

    return 0;
}

static int fs_open(const char *path, struct fuse_file_info *fi)
{
    (void)fi; // Не используются

    if (!find_file(path + 1))
    {
        return -ENOENT;
    }

    return 0;
}

static int fs_read(const char *path, char *buf, size_t size, off_t offset, struct fuse_file_info *fi)
{
    (void)fi;

    file_node *file = find_file(path + 1);
    if (!file)
    {
```

```
        return -ENOENT;
    }

    log_operation("read", path);

    size_t len = strlen(file->contents);
    if ((size_t)offset >= len)
    {
        return 0;
    }
    if ((size_t)offset + size > len)
    {
        size = len - offset;
    }
    memcpy(buf, file->contents + offset, size);
    return size;
}

static int fs_write(const char *path, const char *buf, size_t size, off_t offset, struct fuse_file_info *fi)
{
    (void)fi;

    file_node *file = find_file(path + 1);
    if (!file)
    {
        return -ENOENT;
    }

    create_backup(path + 1);

    log_operation("write", path);

    size_t len = strlen(file->contents);
    if ((size_t)offset + size > len)
    {
        char *new_contents = realloc(file->contents, offset + size + 1);
        if (!new_contents)
        {
            return -ENOMEM;
        }
        file->contents = new_contents;
        memset(file->contents + len, 0, offset + size - len);
    }
    memcpy(file->contents + offset, buf, size);
    file->contents[offset + size] = '\0';
    return size;
}

static int fs_create(const char *path, mode_t mode, struct fuse_file_info *fi)
{
    (void)mode;
    (void)fi;

    if (find_file(path + 1))
```

```
    {
        return -EEXIST;
    }

    log_operation("create", path);
    if (!create_file(path + 1))
    {
        return -ENOMEM;
    }
    return 0;
}

static int fs_unlink(const char *path) {
    char* name = path + 1;
    if (file_list == NULL)
        return -ENOENT;

    log_operation("delete", path);

    time_t now_time_t;
    struct tm *now;
    time(&now_time_t);
    now = localtime(&now_time_t);

    while (file_list && cmp_time(now, &file_list->expires))
    {
        file_node *old_file_list = file_list;
        file_list = file_list->next;
        free(old_file_list);
    }

    if (strcmp(file_list->name, name) == 0)
    {
        file_node *old_file_list = file_list;
        file_list = file_list->next;
        free(old_file_list);
        return 0;
    }

    file_node *current = file_list;

    while (current)
    {
        file_node* next = current->next;

        if (!next)
            return -ENOENT;

        if (strcmp(next->name, name) == 0)
        {
            file_node *old_file = next;
            current->next = next->next;
            free(old_file);
            return 0;
```

```
        }

        while (current->next && cmp_time(now, &current->next->expires))
        {
            file_node *old_next = current->next;
            current->next = current->next->next;
            free(old_next);
        }

        current = current->next;
    }
    return -ENOENT;
}

static struct fuse_operations fs_oper = {
    .getattr = fs_getattr,
    .readdir = fs_readdir,
    .open = fs_open,
    .read = fs_read,
    .write = fs_write,
    .create = fs_create,
    .unlink = fs_unlink
};

int main(int argc, char *argv[])
{
    return fuse_main(argc, argv, &fs_oper, NULL);
}
```

Результаты выполнениям программы:




```
[2024-12-10 23:37:36] PID: 440128 | Операция: create | Путь: /11.12.2024 20:26:00 file.txt
[2024-12-10 23:37:36] PID: 440128 | Операция: write | Путь: /11.12.2024 20:26:00 file.txt
[2024-12-10 23:37:59] PID: 440128 | Операция: create | Путь: /10.12.2024 20:26:00 file.txt
[2024-12-10 23:38:08] PID: 440128 | Операция: create | Путь: /10.12.2025 20:26:00 file.txt
```

```
[2024-12-10 23:38:08] PID: 440128 | Операция: write | Путь: /10.12.2025 20:26:00 file.txt
```

Вывод: в ходе лабораторной работы изучили популярные файловые системы в ОС Linux (ext4, Btrfs, ReiserFS, NTFS, FAT32). В целом, практически все файловые системы показали себя хорошо. NTFS показала худшие результаты почти везде кроме чтения больших файлов. В этом сценарии все файловые системы показали себя одинаково эффективно. Что удивительно, особенности файловых систем были опровергнуты. Так, BTRFS должен давать хороший прирост скорости при чтении файлов. Однако оказался далеко не самым лучшим. ReiserFS должен хорошо работать с маленькими файлами, однако всё так же уступает другим файловым системам. Fat32, ожидаемо, проигрывает в большом количестве сценариев. Ext4 - стандартная файловая система для Linux, она и показывает себя довольно хорошо во многих сценариях. Плохую производительность NTFS можно объяснить плохой поддержкой драйверов. При использовании программы, часто использующей fseek, хуже всего показала себя fat32. В целом, программа не сильно оптимизирована для современных сценариев. Она подразумевает маленькое количество памяти и минимальное использование кеша. Вследствие этого можно наблюдать маленькую скорость и скромные размеры использования оперативной памяти. При помощи fuselib получилось реализовать собственную виртуальную файловую систему с возможностью создания, удаления файлов а также удаления ”просроченных”файлов. Интерфейс Fuse гораздо шире представленной в лабораторной работе, его можно реализовывать для более серьёзной и тонкой работы с файловой системой.