Министерство образования Республики Беларусь

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет Информационных технологий и управления
Кафедра Интеллектуальных информационных технологий

ОТЧЁТ
по лабораторной работе №5
по дисциплине “Операционные системы”

Выполнили:
Р. В. Липский, гр. 121701
Проверил:
В. А. Цирук

Цель и постановка задачи

Цель: Изучить подходы к организации файловой системы.

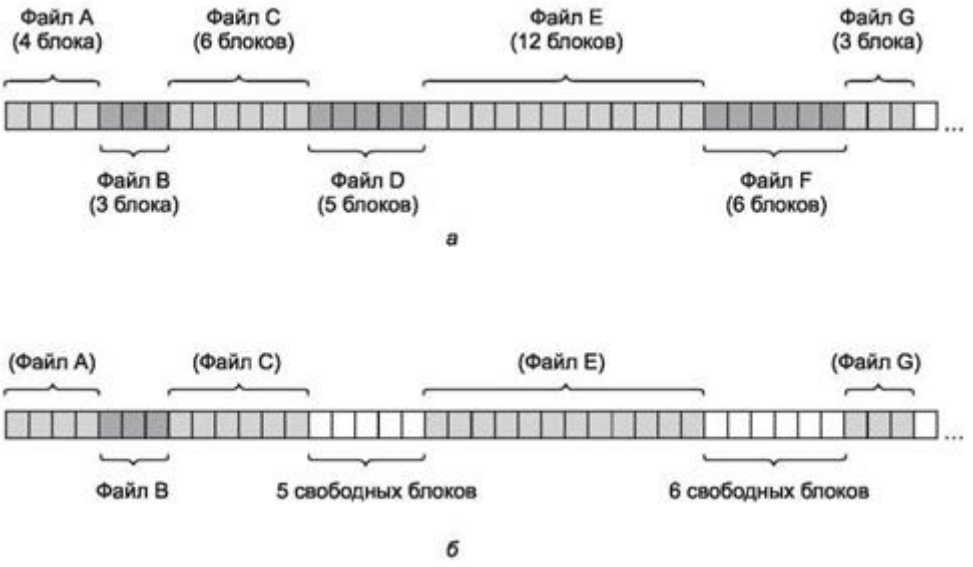
Задание: Реализовать модель файловой системы и реализовать базовые команды просмотра файлов на диске – создание, удаление, копирование, перемещение, запись в файл, чтение файла. Обеспечить создание дампа файловой системы таким образом, чтобы можно было просмотреть структуру файловой системы и содержимое файлов.

Вариант: 1. Одноуровневая файловая система, с физической организацией файла - непрерывное размещение;

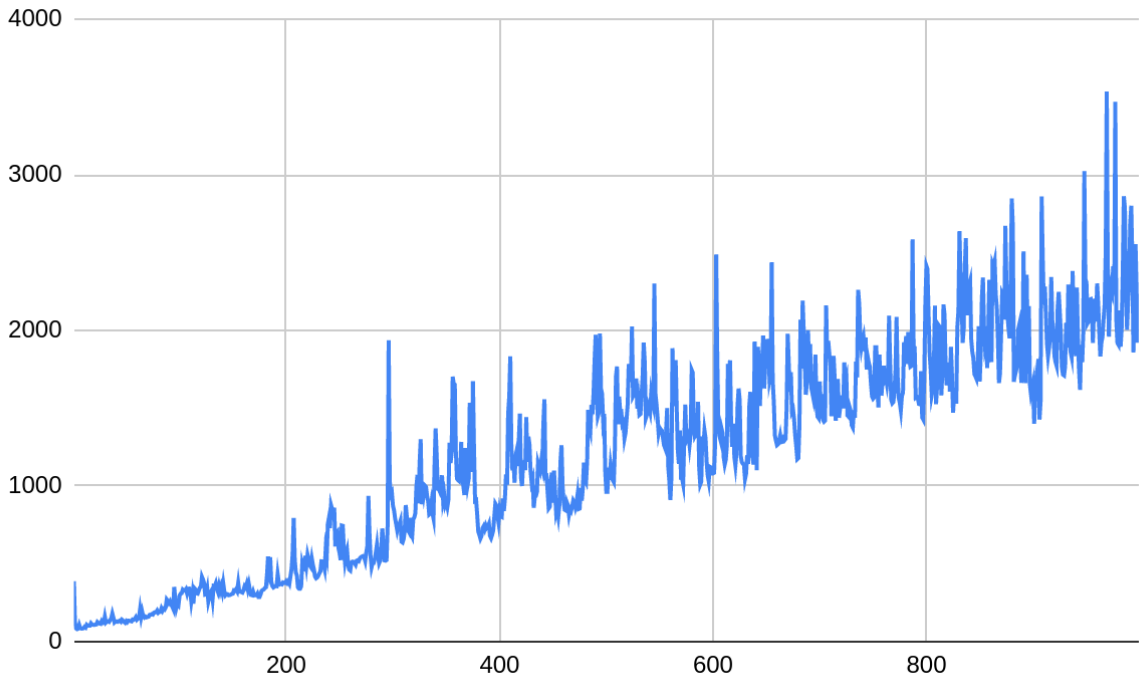
Краткое описание задачи:

Файловая система - это система хранения файлов и организации каталогов. Для дисков с небольшим количеством файлов (до нескольких десятков) удобно применять одноуровневую файловую систему, когда каталог (оглавление диска) представляет собой линейную последовательность имен файлов. Для отыскания файла на диске достаточно указать лишь имя файла.

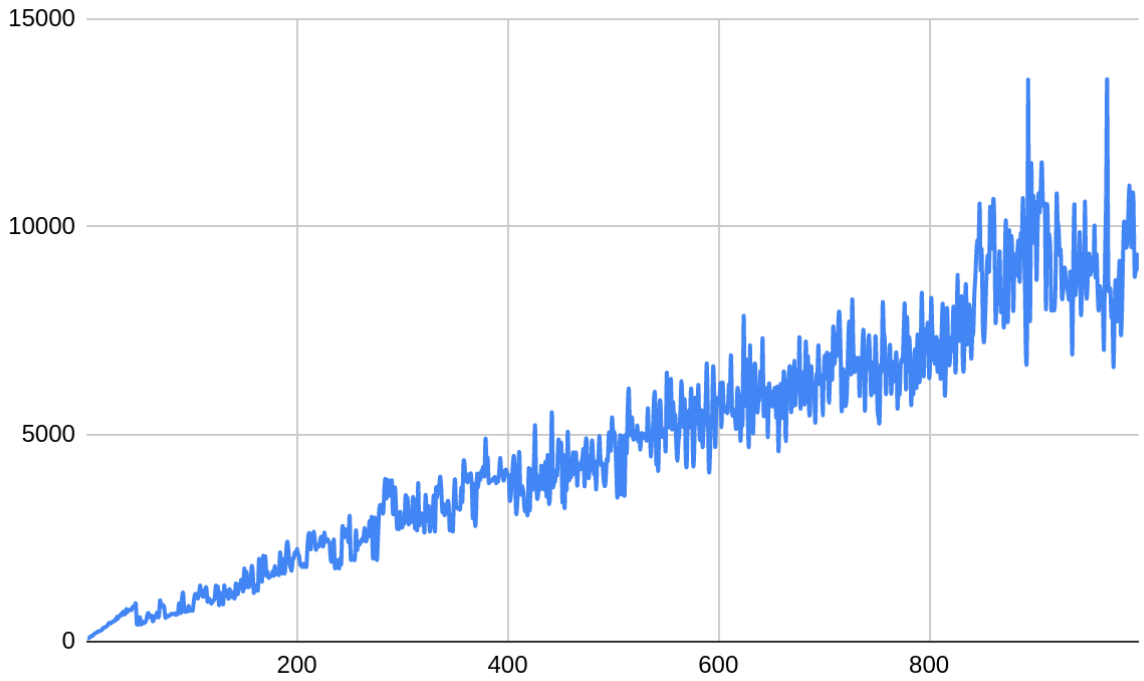
Простейшая схема размещения заключается в хранении каждого файла на диске в виде непрерывной последовательности блоков. Таким образом, на диске с блоками, имеющими размер 1 Кбайт, файл размером 50 Кбайт займет 50 последовательных блоков.



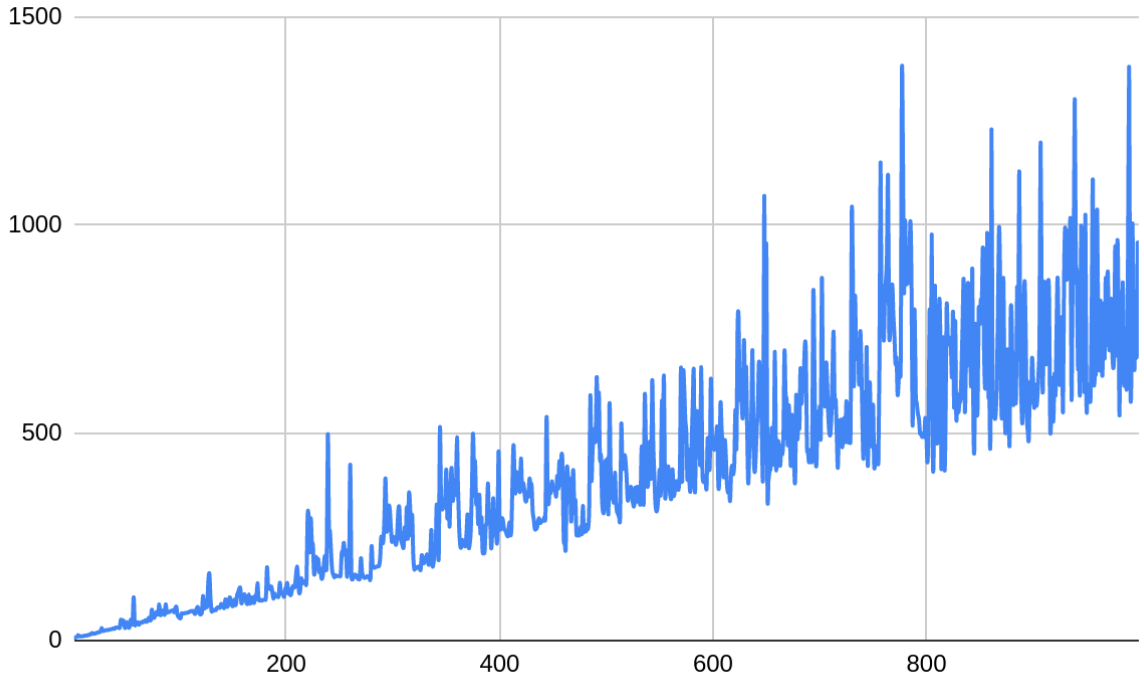
Нагрузочные тесты



Этот график показывает зависимость времени необходимого на дамп состояния памяти в зависимости от количества файлов в ней. В целом, график растёт линейно (спайки появляются, скорее всего, от нагрузки на файловую систему, несвязанную с выполнением программы). Алгоритм вполне эффективный, поскольку имеет временную сложность O(n).



Этот график показывает зависимость времени чтения от количества файлов в образе. график также растёт линейно, следовательно алгоритм имеет сложность O(n), что говорит о его эффективности.



Этот график показывает время поиска файла по файловой системе. График также имеет линейную сложность, а, поскольку это единственный времязатратный механизм,

используемый в системе, другие методы (копирование, перемещение и т.д.) будут иметь такую же сложность.

Именно в этом случае, возможно более эффективно организовать структуру файловой системы, чтобы уменьшить время доступа к файлам - например, использовать деревья поиска, таким образом можно было бы добиться логарифмической сложности.

Контрольные вопросы

Что такое файл?

Именованная область данных на носителе информации, используемая как базовый объект взаимодействия с данными в операционных системах.

Назовите типы файлов?

- Обычные файлы (-): Сюда относятся все файлы с данными, играющими роль ценной информации сами по себе.
- Каталоги (d): в Linux каталог представляет собой такой тип файла, данными которого является список имен других файлов и каталогов, вложенных в данный каталог.
- Символьные ссылки (l): файл, в данных которого содержится адрес другого файла по его имени.
- Символьные (c) и блочные устройства (b): файлы устройств предназначены для обращения к аппаратному обеспечению компьютера (дискам, принтерам, терминалам и др.). Когда происходит обращение к файлу устройства, то ядро операционной системы передает запрос драйверу этого устройства. К символьным устройствам обращение происходит последовательно (символ за символом). Примером символьного устройства может служить терминал. Считывать и записывать информацию на блочные устройства можно в произвольном порядке, причем блоками определенного размера. Пример: жесткий диск.
- Сокеты (s) и каналы (p): ключевым отличием канала от сокета является то, что канал однонаправлен

Дайте определение термину файловая система?

Порядок, определяющий способ организации, хранения и именования данных на носителях информации. Файловая система определяет формат содержимого и способ физического хранения информации, которую принято группировать в виде файлов. Конкретная файловая система определяет размер имен файлов (и каталогов), максимальный возможный размер файла и раздела, набор атрибутов файла. Некоторые файловые системы предоставляют сервисные возможности, например, разграничение доступа или шифрование файлов.

Что такое атрибут файла? Приведите примеры.

Атрибут файла — метаданные, которые описывают файл. Атрибут может находиться в двух состояниях: либо установленный, либо снятый. Атрибуты рассматриваются

отдельно от других метаданных, таких как даты, расширения имени файла или права доступа. Каталоги и другие объекты файловой системы также могут иметь определенные атрибуты. Также существуют расширенные атрибуты файлов, хранящие данные другого типа. Примеры: readonly, system, hidden.

Назовите и расскажите про подходы к организации доступа к файлам?

Различают два основных подхода к определению прав доступа:

- избирательный доступ, когда для каждого файла и каждого пользователя сам владелец может определить допустимые операции;
- мандатный подход, когда система наделяет пользователя определенными правами по отношению к каждому разделяемому ресурсу (в данном случае файлу) в зависимости от того, к какой группе пользователь отнесен.

Что такое логическая структура файла. Что такое логическая запись?

В общем случае данные, содержащиеся в файле, имеют некую логическую структуру. Эта структура является базой при разработке программы, предназначенной для обработки этих данных. Например, чтобы текст мог быть правильно выведен на экран, программа должна иметь возможность выделить отдельные слова, строки, абзацы и т. д. Признаками, отделяющими один структурный элемент от другого, могут служить определенные кодовые последовательности или просто известные программе значения смещений этих структурных элементов относительно начала файла. Поддержание структуры данных может быть либо целиком возложено на приложение, либо в той или иной степени эту работу может взять на себя файловая система.

Логическая запись – это поименованная совокупность элементарных данных, имеющая смысловую завершенность.

Примером записи может служить строка из списка студентов:

Фамилия	*Год рождения*	*Год поступления в ВУЗ*	*Курс*	*Номер зачетной книжки*

Логическая запись объединяет не разрозненные по смыслу данные, а только те, что характеризуют некоторую систему или объект – именно в этом смысле следует понимать слова «смысловая завершенность» в определении. Запись отражает совокупность свойств (атрибутов) системы или объекта.

Что такое физическая структура файла. Что такое физическая запись?

Физическая организация файла описывает правила расположения файла на устройстве внешней памяти, в частности на диске. Файл состоит из физических записей – блоков.

Физическая запись, с которой работает файловая система– совокупность данных, размещаемых в файле обычно на внешнем носителе, которые могут быть считаны или записаны как единое целое одной командой ввода-вывода.

Что такое символьное имя файла?

Строка символов, однозначно определяющая файл в некотором пространстве имён файловой системы (ФС), обычно называемом каталогом, директорией или папкой. Имена файлов строятся по правилам, принятым в той или иной файловой и операционной системах (ОС). Многие системы позволяют назначать имена как обычным файлам, так и каталогам и специальным объектам (символическим ссылкам, блочным устройствам и т. п.).

Что такое файлы проецируемые в память?

Проецируемые файлы позволяют резервировать регион адресного пространства и передавать ему физическую память, которая не выделяется из страничного файла, а берется из файла, уже находящегося на диске.

Описание методов, использованных для решения задачи

Структура файла

```
struct file {
    static attrs_t const ATTR_READONLY = 0x1;
    static attrs_t const ATTR_HIDDEN = 0x2;
    static attrs_t const ATTR_SYSTEM = 0x4;

    std::string name;
    attrs_t attrs;
    std::vector<char> content;
};
```

Поле **name** - стандартная строка, содержащая в себе полное название файла.
Поле **attrs** - число, представляющее атрибуты файла типа **attrs_t**:

- 1-5 бит - не используются
- 6 бит - **ATTR_SYSTEM** - является ли файл системным
- 7 бит - **ATTR_HIDDEN** - является ли файл скрытым
- 8 бит - **ATTR_READONLY** - является ли файл доступным только для чтения

Поле **content** - содержимое файла, представленное вектором байт.

При сериализации файл имеет следующий формат:

```
File {
      u8 file_name_length;
      u1 file_name[file_name_length];
      attrs_t attrs;
      u8 file_content_length;
      u1 file_content[file_content_length];
}
```

Реализация дампа памяти

Метод **save_state** реализуют сериализацию настоящего состояния памяти. Структура образа файловой системы может быть описана следующим образом:

```
FileSystem {
      u4    magic; // магическое число, идентифицирующее двоичный
формат
      u8    files_count; // количество файлов в памяти
      File  files[files_count]; // сами файлы
}
```

Метод **save_state** записывает дамп памяти, путем открытия файла на хост-машине в бинарном режиме и побайтной записи необходимой информации в соответствии с указанным выше двоичным форматом.

```
void save_state() {
    std::ofstream out(image_path, std::ios::binary);
    // write magic
    out.write("MYFS", 4);

    write_prim(out, files.size(), sizeof(size_t));

    for (const auto& file : files) {
        write_prim(out, file.name.length(), sizeof(long long));
        out.write(file.name.c_str(), file.name.size());
        write_prim(out, file.attrs, sizeof(attrs_t));
        write_prim(out, file.content.size(), sizeof(long long));
        auto content = vect_to_str(file.content);
        out.write(content.c_str(), content.size());
    }

    out.close();
}
```

Метод **load_state**, действует аналогично, только для чтения дампа из файла на хост-машине.

```
void load_state() {
    std::ifstream in(image_path, std::ios::binary);
    check_magic(read_bytes(in, 4));
    auto pages_count = read_prim<size_t>(in);
    for (int i = 0; i < pages_count; i++) {
        auto filename_length = read_prim<size_t>(in);

        std::string filename;
        for (int j = 0; j < filename_length; j++) {
            filename += read_prim<char>(in);
        }

        auto attrs = read_prim<attrs_t>(in);
        auto content_length = read_prim<size_t>(in);
        std::vector<char> content;
        for (int j = 0; j < content_length; j++) {
            content.emplace_back(read_prim<char>(in));
        }
```

```
        files.emplace_back(file{filename, attrs, content});
    }

    in.close();
}
```

API

Метод **add_file** добавляет структуру файла в конец массива файлов, если файла с таким названием не существует. В ином случае, бросается исключения.

```
void add_file(file f) {
    if (file_exists(f.name)) {
        throw fs_error("file exists");
    }
    this->files.emplace_back(f);
}
```

Метод **rm_file** использует стандартный метод *erase_if* из библиотеки *algorithm* для итерации по массиву файлов и удалении файла с указанным именем из памяти.

```
void rm_file(const std::string& filename) {
    erase_if(files, [filename](const file& f) {
        return f.name == filename;
    });
}
```

Метод **get_file_by_name** действует аналогично, однако вместо удаления, возвращает итератор на искомом файле. В случае, если итератор достиг конца вектора файлов бросается исключение.

```
file get_file_by_name(const std::string& name) {
    auto it = std::find_if(files.begin(), files.end(), [name](file f) {
        return f.name == name;
    });
    if (it == files.end()) {
        throw fs_error("file not found");
    }
    return *it;
}
```

Метод **cp_file** использует вышеописанный метод *get_file_by_name* для нахождения исходного файла, создает его точную копию при помощи конструктора копирования, и использует метод *add_file* для добавления его в вектор файлов.

```
void cp_file(const std::string& src_name, const std::string& trg_name) {
    auto f = file{get_file_by_name(src_name)};
    f.name = trg_name;
```

```
    add_file(f);
}
```

Метод **mv_file** использует метод *cp_file* для копирования файла и *rm_file* для удаления старого файла.

```
void mv_file(const std::string& old_name, const std::string& new_name) {
    cp_file(old_name, new_name);
    rm_file(old_name);
}
```

Метод **file_exists** итерируется по вектору файлов и ищет файл с указанным названием при помощи стандартного метода *std::any_of*.

```
bool file_exists(const std::string& filename) {
    return std::ranges::any_of(files.begin(), files.end(),
    [filename](const file& f) {
        return f.name == filename;
    });
}
```

Метод **set_content** находит необходимый файл при помощи метода *get_file_by_name* и изменяет его поле *content*.

Утилиты

Метод **convert_to_bytes** производит преобразование любого типа данных (примитивный, класс, структура) в набор байтов путём копирования участка памяти, соответствующей переменной в массив char. Это необходимо для корректной сериализации дампа файловой системы.

```
template<class T>
static void convert_to_bytes(T anything, char* array) {
   memcpy(array, &anything, sizeof(T));
}
```

Метод **convert_to_prim** производит преобразование набора байт в необходимый тип данных путем копирования участка памяти, соответствующего массиву char в участок памяти, отведенной для переменной указанного типа.

```
template<class T>
static T convert_to_prim(char* str) {
   T trg;
   memcpy(&trg, str, sizeof(T));
```

```
    return trg;
}
```

Метод **str_to_vect** производит преобразования стандартного типа std::string в std::vector. Это необходимо преобразования пользовательского ввода (имеет тип std::string, поскольку всегда является читаемым текстом) в вектор байт.

```
static std::vector<char> str_to_vect(std::string str) {
    return { str.begin(), str.end() };
}
```

Обратный ему метод **vect_to_str** работает по аналогичному принципу:

```
static std::string vect_to_str(std::vector<char> vect) {
    return { vect.begin(), vect.end() };
}
```

Метод **write_prim** осуществляет сериализацию входных данных при помощи ранее описанного метода *convert_to_bytes* и записывает полученный набор байт в указанный поток вывода.

```
template<class T>
static void write_prim(std::ostream& out, const T& value,
std::streamsize length) {
    char* bytes = new char[length];
    convert_to_bytes(value, bytes);
    out.write(bytes, length);
}
```

Метод **read_bytes** осуществляет чтение указанного количества байт из потока ввода.

```
static char* read_bytes(std::istream& in, int t) {
    char* bytes = new char[t + 1] { 0 };
    in.read(bytes, t);
    return bytes;
}
```

Метод **read_prim** является композицией методов *read_bytes* и *convert_to_prim*.

```
template<class T>
static T read_prim(std::istream& in) {
    return convert_to_prim<T>(read_bytes(in, sizeof(T)));
}
```