ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ

УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

«НИЖЕГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

им. Р.Е. Алексеева»

Институт Радиоэлектроники и Информационных Технологий

Кафедра "Вычислительные системы и технологии"

ОТЧЁТ

По курсовой работе

«Утилита дефрагментации файлов FAT»

по дисциплине «Принципы и методы организации системных программных средств»

Выполнил:

Проверил:

Кочешков А.А.

Нижний Новгород

Содержание

1. Введение
   1. Постановка задачи
   2. Условия выполнения задачи
2. Теория
   1. FAT (File Allocation Table)
   2. Специальный файл устройства
3. Ход работы
   1. Описание концепции работы программы
   2. Описание структуры программы
   3. Подготовка к демонстрации программы
4. Итог
   1. Результат работы программы
   2. Выводы
5. Листинг

Введение

Постановка задачи

Задание 2.7

"Утилита дефрагментации файлов FAT"

Разработать программу дефрагментации заданного файла или всех файлов заданного каталога файловой системы FAT. При запуске контролировать тип и свойства файловой системы на томе. Выводить отчет.

Условия выполнения задачи

Тип файловой системы: FAT16 (16-битная);

Расширение VFAT: нет;

Фрагментированным будет считаться любой файл, расположение кластеров которого не является строго последовательным и однонаправленным;

Это необходимо для корректного восстановления файлов в случае их удаления.

Тип программы: консольное приложение;

Платформа: Linux;

Язык программирования: C++, стандарт: C++20;

Компилятор: clang++;

Устройство: съёмный накопитель Flash Drive;

Размер используемого раздела: 16Мб.

Раздел искусственно ограничивается с помощью встроенной утилиты Диски и дополнительно установленной программы GParted. Благодаря второй, удобно создавать разделы заданного размера и контролировать тип форматируемой системы. С помощью первой можно легко очистить память накопителя, задать имя разделу, монтировать и размонтировать раздел.

Файловая система FAT16 была выбрана, как методом исключения: поскольку FAT12, несмотря на формальную поддержку, может некорректно определяться системой и приложениями, и форматировать раздел в эту морально устаревшую систему не позволяет подавляющее число приложений, а реализация адресной арифметики для таблиц FAT представляется излишне затруднительной; так и из практических и академических соображений: FAT32 практически не отличается по своей общей структуре от FAT16, с той разницей, что в FAT32 были введены дополнительные поля данных, которые, на сегодняшний день, стали по большому счёту неактуальны, а также в FAT32 корневая директория больше не привязана к началу области данных. Изменения в FAT32 по сравнению с FAT16 больше количественные: размер блоков таблиц FAT увеличен до 4 байт, а макимальный объём раздела с 4Гб до 2Тб. По этой причине, при изучении файловой системы FAT, необходимость брать расширенный стандарт отсутствует.

Теория

Данный блок предоставляет собранную информацию из открытых источников для осведомления в специфических темах и терминах курсовой работы.

FAT

FAT (англ. File Allocation Table «таблица размещения файлов») — классическая архитектура файловой системы, которая из-за своей простоты всё ещё широко применяется для флеш-накопителей. Используется в дискетах, картах памяти и некоторых других носителях информации. Ранее находила применение и на жёстких дисках.

Разработана Биллом Гейтсом и Марком МакДональдом в 1976—1977 годах. Использовалась в качестве основной файловой системы в операционных системах семейств MS-DOS и Windows 9x.

Существует четыре версии FAT — FAT12, FAT16, FAT32 и exFAT (FAT64). Они различаются разрядностью записей в дисковой структуре, то есть количеством бит, отведённых для хранения номера кластера. FAT12 применяется в основном для дискет, FAT16 — для дисков малого объёма. На основе FAT была разработана новая файловая система exFAT (extended FAT), используемая преимущественно для флеш-накопителей.

Изначально FAT не поддерживала иерархическую систему каталогов — все файлы располагались в корне диска. Это было сделано для упрощения, так как на односторонних дискетах ёмкостью всего 160–180 Кбайт сортировать немногочисленные файлы по каталогам попросту не было смысла. С распространением дискет на 320 и более килобайт хранение всех файлов в корне оказалось неудобным, к тому же малый размер корневого каталога ограничивал количество файлов на диске. Каталоги были введены с выходом MS-DOS 2.0.

В Windows 95 и Linux появилось расширение VFAT - поддержка длинных имён файлов (LFN) в формате Unicode (Virtual FAT — VFAT).

В FAT имена файлов имеют формат 8.3 (8 байт на имя, 3 на расширение) и состоят только из символов кодировки ASCII. В VFAT была добавлена поддержка длинных (до 255 символов) имён файлов (англ. Long File Name, LFN) в кодировке UTF-16LE, при этом LFN хранятся одновременно с именами в формате 8.3, ретроспективно называемыми SFN (англ. Short File Name).

В файловой системе FAT смежные секторы диска объединяются в единицы, называемые кластерами. Количество секторов в кластере равно степени двойки. Для хранения данных файла отводится целое число кластеров (минимум один), так что, например, если размер файла составляет 40 байт, а размер кластера 4 Кбайт, реально занят информацией файла будет лишь 1 % отведённого для него места.

Во избежание подобных ситуаций целесообразно уменьшать размер кластеров, а для сокращения объёма адресной информации и повышения скорости файловых операций — наоборот. На практике выбирают некоторый компромисс. Так как ёмкость диска вполне может и не выражаться целым числом кластеров, обычно в конце тома присутствуют так называемые surplus sectors — «остаток» размером менее кластера, который не может отводиться ОС для хранения информации.

Пространство тома FAT логически разделено на три смежные области:

* Зарезервированная область. Содержит служебные структуры, которые принадлежат загрузочной записи раздела (Partition Boot Record — PBR) и используются при инициализации тома.
* Область таблицы FAT, содержащая массив индексных указателей («ячеек»), соответствующих кластерам области данных. Для повышения надёжности на диске обычно представлено две копии таблицы FAT.
* Область данных, где записано собственно содержимое файлов, а также каталоги.

В FAT12 и FAT16 также специально выделяется область корневого каталога. Она имеет фиксированное положение (непосредственно после последнего элемента таблицы FAT) и фиксированный размер в 32-байтных элементах, то есть при описании в Partition Boot Record указывается именно количество 32-байтных элементов, каждый из которых описывает какой-либо элемент корневого каталога (будь то файл или другой вложенный каталог).

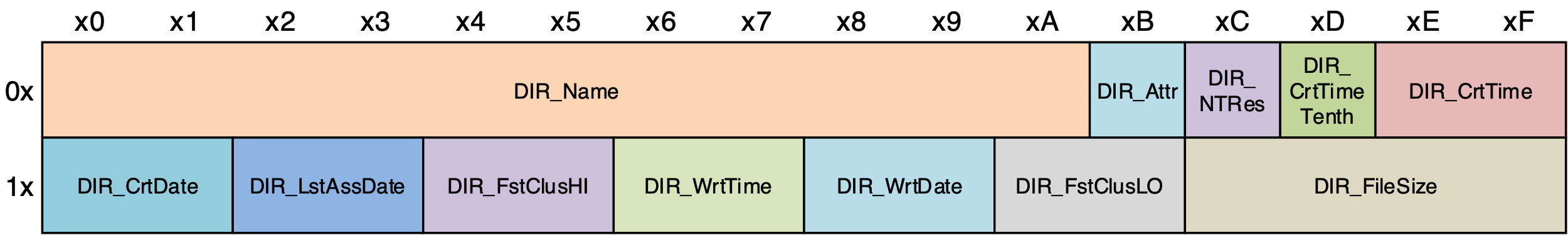
Если кластер принадлежит файлу, то соответствующая ему ячейка в таблице FAT содержит номер следующего кластера этого же файла. Если ячейка соответствует последнему кластеру файла, то она содержит специальное значение (0xFFFF для FAT16). Таким образом выстраивается цепочка кластеров файла. Неиспользуемым кластерам в таблице соответствуют нули. «Плохим» кластерам (которые исключаются из обработки, например, по причине нечитаемости соответствующей области устройства) также соответствует специальный код (0xFFF7 для FAT16).

При удалении файла первый знак имени заменяется специальным кодом 0xE5, и цепочка кластеров файла в таблице размещения обнуляется. Поскольку информация о размере файла (которая располагается в каталоге рядом с именем файла) при этом остаётся нетронутой, в случае, если кластеры файла располагались на диске последовательно и не были перезаписаны новой информацией, удалённый файл можно восстановить.

Структура файловой записи FAT

Файловая запись FAT32 состоит из следующих структур:

* 11-байтовое поле по относительному адресу 0, содержит короткое имя файла (в рамках стандарта 8.3).
* Байт по адресу 0x0B, отвечающий за атрибуты файла.
* Байт по адресу 0x0C, используется в Windows NT.
* Байт по адресу 0x0D. Счётчик десятков миллисекунд времени создания файла, допустимы значения 0–199. Поле часто неоправданно игнорируется.
* 2 байта по адресу 0x0E. Время создания файла с точностью до 2 секунд.
* 2 байта по адресу 0x10. Дата создания файла.
* 2 байта по адресу 0x12. Дата последнего доступа к файлу.
* 2 байта по адресу 0x14. Номер первого кластера файла (старшее слово, на томе FAT12/FAT16 равен нулю).
* 2 байта по адресу 0x16. Время последней записи (модификации) файла, например его создания.
* 2 байта по адресу 0x18. Дата последней записи (модификации) файла, в том числе создания.
* 2 байта по адресу 0x1A. Номер первого кластера файла (младшее слово).
* 4 последних байта, содержащие значение размера файла в байтах. Фундаментальное ограничение FAT(FAT32) — максимально допустимое значение размера файла составляет 0xFFFFFFFF (то есть 4 Гбайт минус 1 байт).



Специальный файл устройства

(или просто файл устройства) - это один из типов файлов в UNIX-подобной операционной системе. Специальные файлы устройств содержат данные, необходимые операционной системе для взаимодействия с физическими устройствами, такими как диски и дисководы, принтеры и факсы и т. п. Фактически, специальные файлы устройств являются указателями на драйверы устройств, и когда процесс обращается к файлу устройств, он по сути работает с драйвером этого устройства.

Файлы устройств размещаются в каталоге /dev или в его подкаталогах.

Узнать информацию о файле устройства можно с помощью команд file и ls.

/dev (от англ. devices — устройства) — каталог в системах типа UNIX, содержащий так называемые специальные файлы — интерфейсы работы с драйверами ядра. Как правило (хотя и не всегда), /dev является обычным каталогом в корневой файловой системе, куда можно (но не нужно) помещать и обычные файлы. Доступ на запись к /dev (то есть право добавлять и перемещать специальные файлы) имеет только суперпользователь. Сами «специальные файлы» могут быть как доступны простому пользователю (терминал, псевдоустройства), так и недоступны (жёсткие диски).

Примеры содержимого:

/dev/sd буква — жёсткий диск (в системах на ядре Linux).

/dev/sd буква номер — раздел диска (в системах на ядре Linux).

/dev/sr номер (/dev/scd номер) — CD-ROM.

/dev/eth номер — сетевые интерфейсы Ethernet.

/dev/wlan номер — сетевые интерфейсы Wireless.

/dev/lp номер — принтеры.

/dev/tty номер — текстовый терминал.

/dev/zero — нулевые байты (псевдоустройство).

Согласно данным из официальных источников IBM, имя жесткого диска и соответствующих разделов имеет следующий формат:

sd[a-z][a-z][1-15]

– вторая буква опциональна, для жестких дисков исключается номер.

Ход работы

Описание концепции работы программы

Для осуществления дефрагментации файла, необходимо получить доступ к сырым данным файла, то есть, к байтам раздела. Поскольку в Линукс всё представлено файлами, под данной системой удобнее всего решать поставленную задачу. Специальные файлы устройств расположены по фиксированному пути /dev/, откуда подключенные файловые системы уже монтируются к привычным папкам пользователей. Именно такой файл требуется, чтобы получить доступ с байтам раздела.

Язык Си++ имеет в стандартной библиотеке методы для работы с вводом/выводом в файлы, также поддерживающие опции бинарных чтения и записи. Благодаря нему, специальные файлы можно открывать и просматривать.

Разделы FAT поделены на области, и первой из них является Загрузочная Запись Раздела (Partition Boot Record). В ней хранятся данные о разделе, его характеристики, а также он хранит сигнатуры, свидетельствующие о том, что это именно файловая система FAT. Область, хранящая эти сведения, фиксирована по размеру и расположению. Считывая её можно быстро найти все доступные разделы, соответствующие поиску.

После того, как раздел будет найдет, требуется обнаружить в нём файл. Чтобы это сделать, необходимо последовательно пройти по пути файла, открыв вначале корневую директорию. Таблицу FAT следует закэшировать, чтобы иметь к ней быстрый доступ, поскольку она является необходимым звеном в адресации данных файлов.

В директориях FAT записи о файлах хранятся в особом виде – в 32 байтовых SFN (Short File Name) записях, в которых хранятся, из полезного, имя файла в формате 8.3, где 8 байт отведено на имя файла (заглавными буквами), и 3 байта на расширение файла (у каталогов последние 3 байта не используются), тип, размер файла, номер первого кластера. Отыскав в корневой директории каталог из цепочки пути, из SFN записи каталога нужно взять номер первого кластера, вычислить сдвиг от начала раздела до этого кластера и перейти к нему. Так нужно сделать, пока не будет найден файл.

По нахождении файла, из его SFN записи извлекается номер первого кластера и в таблице FAT проверяется последовательность его кластеров на фрагментацию. При обнаружении фрагментации данных, программа размещает данные последовательно в памяти раздела, записывает изменения в таблицу FAT, и, при необходимости, вносит изменения в SFN запись файла. Если файл открывается и не повреждён – дефрагментация прошла успешно.

Описание структуры программы

В программе представлено 4 основных класса:

1. Класс-контейнер Bytes, используемый для хранения байт данных. С его помощью данные можно закэшировать в оперативной памяти компьютера, благодаря чему, к информации можно обращаться быстрее, чем при непосредственном считывании с памяти устройства. Данный класс также позволяет вносить изменения в кэшированные байты, чтобы затем записать их обратно на накопитель для фиксации изменений.
2. Класс PBR. Данный класс используется для определения раздела FAT и получения его характеристик. С его помощью можно в любой момент быстро получить необходимую информацию для манипуляции данными раздела.
3. Класс Partition. Отвечает за взаимодействие с разделом. На него возложены функции поиска и дефрагментации файлов и каталогов. Данный класс максимально использует возможности предыдущих двух классов для реализации своих функций.
4. Класс Program. Простой, но не менее важный – он отвечает за диалог с пользователем, предоставляет поиск раздела FAT с помощью класса PBR, либо оставляет возможность указать путь самостоятельно, а также, посредством класса Partition, открывает раздел и запускает его функцию поиска файла по запрошенному у пользователя пути.

Подбробнее о каждом из них:

Класс Bytes

Данные:

char\* m\_bytes - Символьный динамический массив для хранения байт.

size\_t m\_size - Переменная, хранящая размер массива.

Операторы:

operator char\*()/ operator const char\*() - Перегрузка данных операторов позволяет, при передаче класса, передавать указатель на символьный массив, когда это ожидается.

operator= - Перегруженный оператор= для присваивания экземпляров класса для реализации глубокого копирования (выделяется новая память и элементы копируются по значению).

operator[]- Перегруженный оператор[] для получения элементов массива по индексу.

Методы:

copy() - Скрытый метод копирования для реализации глубокого (чтобы избежать копирования адресов).

length() - Метод возвращает длину символьного массива.

get\_pointer() - Возвращает указатель на символьный массив.

clear() - Очищение (освобождение) выделенной памяти. Обнуление размера.

resize() - Изменяет размер выделенной памяти для буфера.

get\_string() - Возвращает строку из байт указанного размера, по указанному смещению.

print\_bytes() - Функция вывода байт в контейнере. В работе программы не используется. Задействуется при проверке работы кода.

get\_value() – два шаблона метода, возвращают значения из указанных байт. Они могут возвращать результат разных типов и размеров. Из-за особенности хранения байт данных в файловой системе FAT (обратная последовательность байт), данные методы также решают эту проблему, возвращая корректный результат.

insert() - два шаблона методов, вставляют указанные значения, которые могут быть разных размеров и разных типов, в символьный (байтовый) буфер. Впоследствие, буфер может быть использован для записи его в файл.

Вложенные типы:

Перечисление TypeSize - Используется для указания байт, которое необходимо считать или записать в буфер.

Класс PBR (Partition Boot Record – загрузочная запись раздела).

Вложенные типы:

Перечисление FAT\_Type - Перечисление для определения и сверки типа раздела.

Структура Parameters - Структура для хранения исчерпывающей информации о разделе.

Данные:

Bytes m\_buff - Контейнер для хранения байт записи раздела.

Parameters m\_parameters - Экземпляр структуры.

Методы:

init() - Методы для инициализации данных экземпляра класса. Они открывают специальный файл устройства (раздела) и переносят загрузочную запись в контейнер.

set\_pbr() - Главный метод класса - он инициализирует структуру данными, извлекаемыми из байт загрузочной записи раздела.

is\_pbr() - Проверка сигнатуры загрузочной записи. Проверяется сигнатура загрузочной записи, а также сигнатура записи FAT раздела.

is\_fat() - Если экземпляр класса считал истинный раздел FAT, метод возвращает true, иначе false.

set() - Два метода для считывания указанного раздела и перезаписи информации в экземпляре.

clear() - Метод для очистки (обнуления) данных экземпляра. Используется перед переинициализацией методами set().

get\_parameters() - Самый часто используемый метод класса - возвращает ссылку на структуру с данными по разделу. Позволяет обращаться к данным напрямую, а изменение данных извне запрещено.

get\_bytes() - Метод возвращает ссылку на контейнер. Не востребован.

print\_parameters() - Вывод значений структуры. Используется при отладке и тестах.

print() - Используется для оповещения пользователя об обнаруженном разделе. Выводит основную, полезную пользователю информацию.

Класс Partition

Вложенные типы:

Перечисление FileType - Перечисление для определения и сравнения типа файла.

Класс FileInfo - Внутренний класс, экземпляры которого используются

для хранения сведений о файлах. В отличие от структур, позволяет скрыть поля и сделать внешние вмешательства недопустимыми, однако публичность класса позволяет получать из класса Partition результат в виде экземпляра класса FileInfo, который можно сохранить и впоследствии передать обратно в класс Partition для обработки.

Данные:

PBR m\_pbr - Экземпляр класса PBR (Partition Boot Record). Используется для хранения сведений о разделе. К нему регулярно приходится обращаться для работы с разделом.

Bytes m\_FAT - Контейнер для байт таблицы FAT. Из него считываются данные о занимаемых файлами кластерах. При необходимости, в таблицу вносятся изменения, после чего, она может быть записана обратно в файл устройства для фиксации изменений.

std::fstream m\_drive - Экземпляр, реализующий доступ к файлу устройства. Через него осуществляется доступ к файлам в разделе, его данным. С помощью этого экземпляра также осуществляется запись всех изменений на раздел.

Методы:

init() - Метод для инициализации экземпляра класса:

- Открывается поток к файлу устройства для чтения и записи;

- Считывается загрузочная запись раздела, и если запись подлинная,

- Считывается таблица FAT в контейнер.

extract\_name() - Вспомогательные методы для поиска файла. Метод извлекает имя файла/директории до первого слэша ("/").

cut\_string() - Метод обрезает строку, в которой указан путь к файлу, от начала до первого слэша ("/").

search\_file() - Метод ищет файл по указанному имени в кластере буферизованной директории. При нахождении, возвращает заполненный экземпляр класса FileInfo с данными файла.

get\_entry\_name() - Вспомогательный метод для извлечения имени файла из байт в буфферизованной директории. Имя затем сравнивается с искомым.

get\_file\_type() - Вспомогательный метод, извлекающий из байт буфферизованной директории информацию о типе файла (файл, директория, неизвестное).

get\_file\_from\_entry() - Метод для получения экземпляра класса из буфферизованной директории по указанному смещению байт. Номер кластера директории указывается, чтобы заполнить информацию о смещении записи файла в разделе, для возможного внесения изменений.

get\_root\_dir() - Метод возвращает экземпляр, заполненный данными по корневой директории.

defragment\_file() - Главная функция дефрагментации - осуществляет проверку на фрагментацию и дефрагментацию указанного файла или каталога, поскольку, каталоги также, в теории, могут быть фрагментированы.

defragment\_dir() - Вспомогательный метод при обработке директорий. Проходится по всем кластерам директории, вызывая для каждого кластера метод defragment\_dir\_cluster().

defragment\_dir\_cluster() - Метод, извлекающий из буферизованных кластеров директорий файлы, которые затем передаёт в метод defragment\_file().

copy\_cluster() - Метод, копирующий указанный кластер по указанному адресу.

find\_empty\_space() - Метод, используемый для поиска требуемого свободного пространства для дефрагментации файла.

count\_file\_clusters() - Метод для подсчёта занимаемых файлом кластеров. Универсален для любых типов файлов, поскольку высчитывает кластеры по таблице FAT из контейнера.

is\_open() - Метод для проверки, был ли инициализирован экземпляр корректно.

get\_file() - Метод для получения экземпляра класса FileInfo. Используется для поиска файла по заданному пути для дальнейших манипуляций, а именно, дефрагментации.

is\_file\_fragmented() - Метод, используемый для проверки файла на фрагментацию.

defragment() - Открытый метод, запускающий процесс дефрагментации файла. Возвращает количество дефрагментированных файлов.

Класс Program

Методы:

get\_files\_from\_dir() - Функция поиска файла в указанной директории по заданному условию, которое передаётся в виде функции, обрабатывающей строковые значения. Передаваемая функция проверяет элементы (файлы) директории на предмет соответствия реализованных условий. Ранее, программа могла определять доступные разделы из сектора Master Boot Record, но данная реализация оказалась неактуальной.

find\_fat\_partitions() - Метод получает список файлов, проверяет, какие относятся к FAT разделам и возвращает список подходящих файлов.

is\_partition() – статический метод. Проверяет строку на соответствие шаблону специального файла раздела файловой системы (sd[a-z][a-z][1-15]).

Нестатические методы класса нельзя передать в качестве аргументов, поскольку они привязаны к экземплярам класса (имеют скрытый аргумент this). Однако, статические методы принадлежат классу и не имеют связи с экземплярами класса. Их можно передавать в кач-ве аргументов.

start() - Метод запуска программы. Начинает диалог с пользователем.

fat\_search() - Открытый метод программы, позволяющий начать поиск указанного раздела. При автоматическом поиске, спрашивает у пользователя, какой из предложенных разделов необходим. Либо предлагает ручной ввод пути к специальному файлу раздела.

open\_partition() - Метод, открывающий раздел и спрашивающий у пользователя путь до файла или директории, к которым необходимо применить алгоритм дефрагментации. Путь задаётся в формате "/ПУТЬ.../ИМЯ". Для дефрагментации файлов из корневой директории, достаточно написать слэш "/".

fragmentation\_check() - Метод, предоставляющий пользователю возможность запустить дефрагментацию, если это возможно, и выводящая сообщение о проделанной процедуре в случае её выполнения.

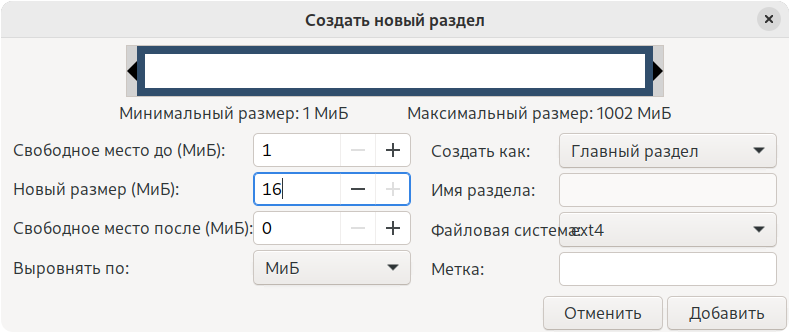
Также, дополнительно, мной были реализованы два класса для компиляции тестовой сборки, которая поможет мне создать условия для испытания работы программы:

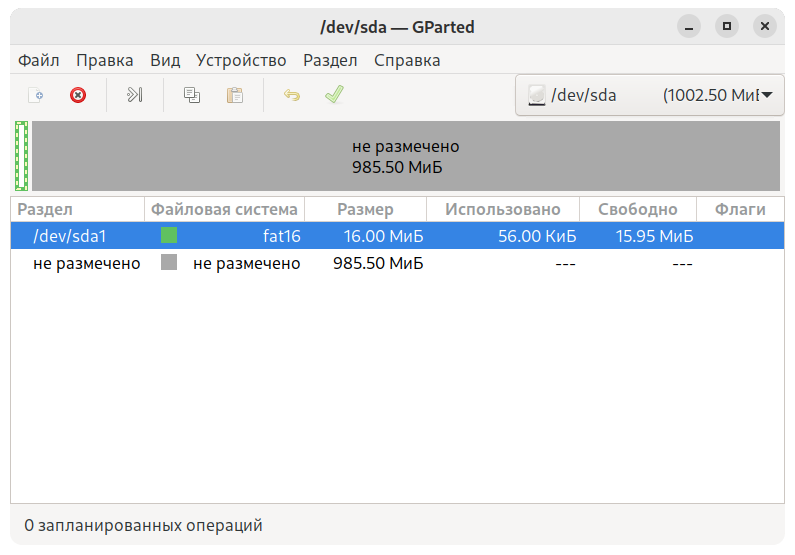
Класс TPart - Наследник класса Partition, созданный с целью использования методов предшествующего класса в альтернативных сценариях. А именно, можно быстро вывести основную информацию о разделе, и скопировать выбранный кластер по указанному адресу.

Класс Test - Класс Test аналогичен по назначению классу Program, с той разницей, что предоставляет опции для тестирования и создания фрагментации, вместо её обнаружения и устранения.

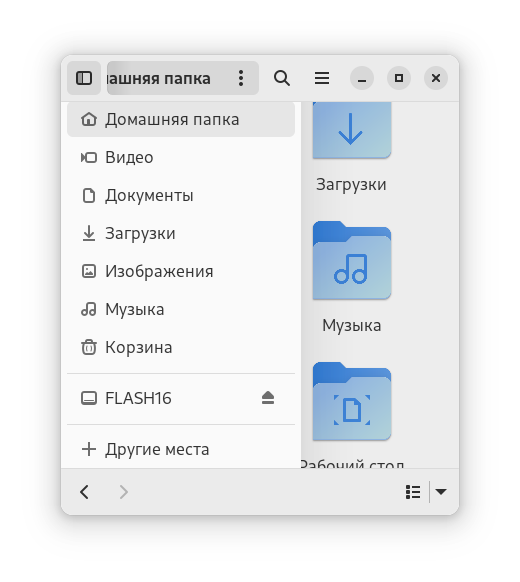
Подготовка к демонстрации программы

Вначале, создам раздел размером 16Мб с помощью утилиты GParted.



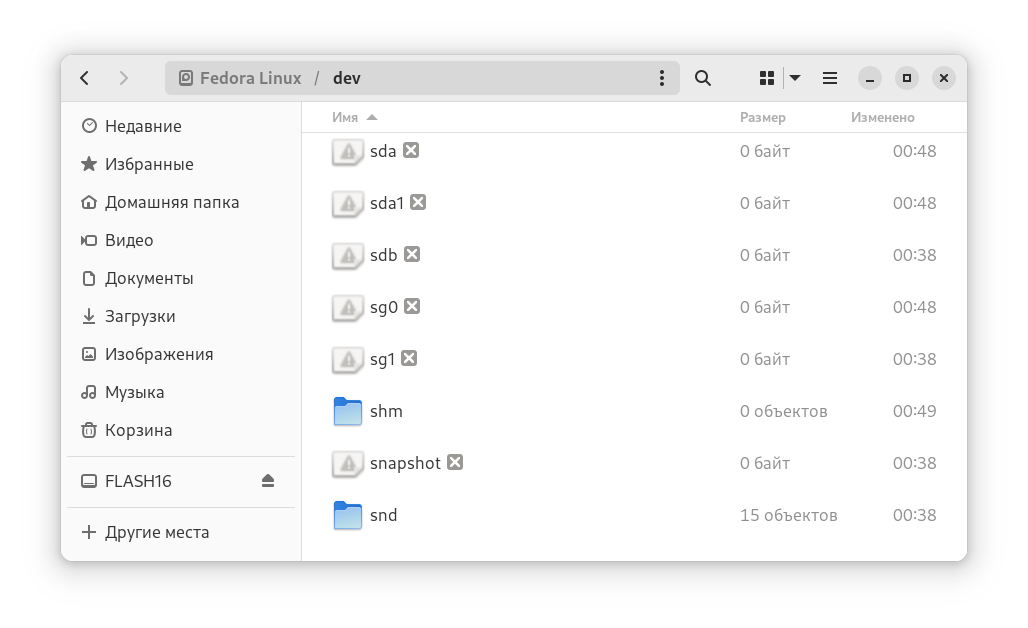


С помощью встроенной утилиты Диски, окружения gnome, установлю название раздела «FLASH16».



На скриншоте выше можно увидеть, что в системе присутствует накопитель с таким именем. Установка прошла успешно.

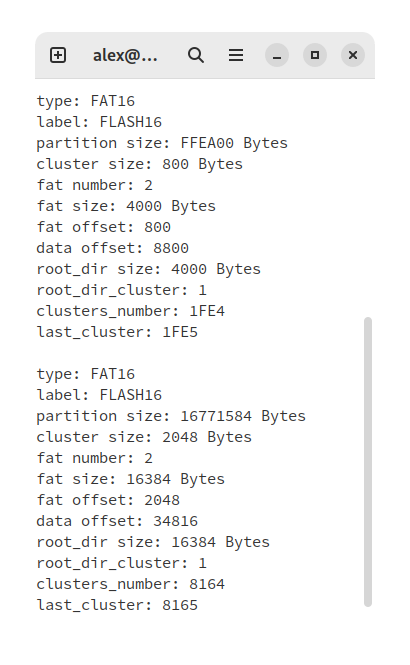
Узнаю, что название специального файла раздела – sda1.



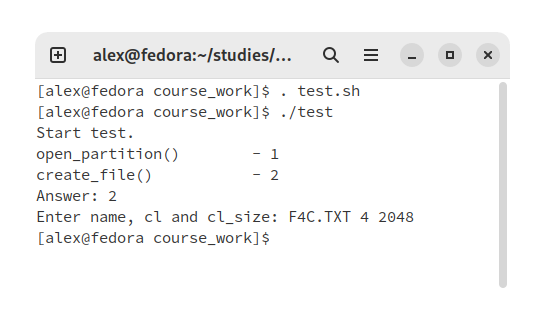
На скриншоте выше его видно среди файлов в директории /dev/.

Позже, из-за необходимости отсоединять устройство, название несколько раз поменяется.

С помощью специальной тестовой сборки, узнаю дополнительные параметры раздела.

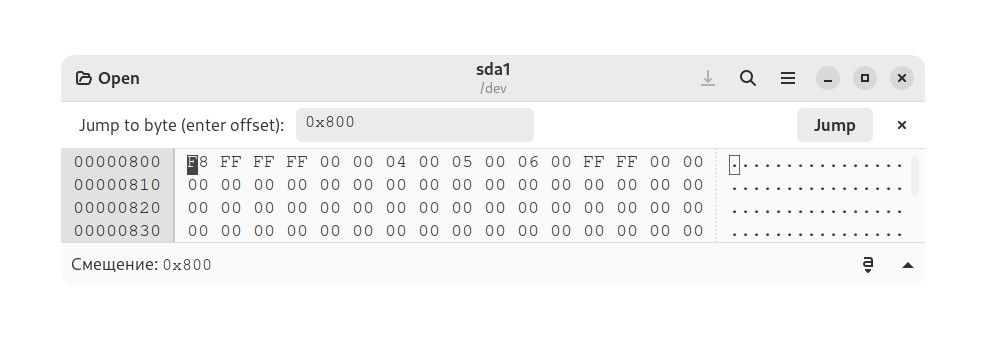


Также, с помощью тестовой сборки создаю файл F4C.TXT размером 4 кластера и помещаю его в корневую директорию раздела.



С помощью программы ghex открываю файл раздела. hex-редакторы или шестнадцатиричные редакторы позволяют просматривать файлы, как набор байт, интерпретируемых в шестнадцатиричном виде.

На данный момент, загрузочная запись раздела не представляет интереса. Перемещаемся к таблицам FAT.

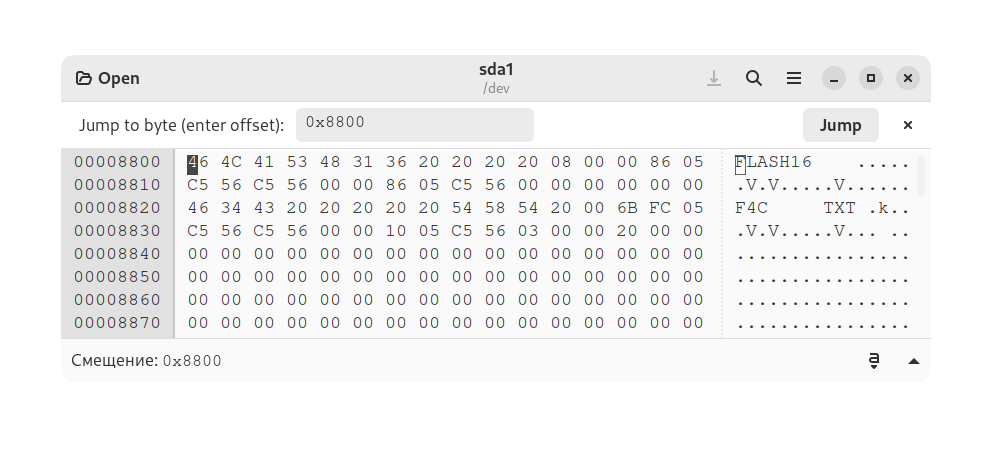


Здесь нужно обратить внимание на байты с 7 по 14. Таблица FAT в файловой системе FAT16 представляет собой двухбайтовые блоки, где первый байт – младший, а второй – старший. По этой причине, запись 0400 на самом деле будет читаться, как 0004. Каждый блок соотносится с кластером того же номерного порядка. Таблица начинается с нулевого блока. Нулевой и первый блоки зарезервированы. Нулевой блок служит для идентификации таблицы. В FAT12 и FAT16 в первом кластере размещается корневая директория. Каждый блок в таблице, занятый файлом, хранит в себе номер следующего блока. Последний блок в цепочке записывается, как FFFF. На скриншоте видно, что блоки с 3-го по 6-й заняты файлом, который я поместил в раздел.

Откроем вторую таблицу, чтобы удостовериться, что они идентичны.

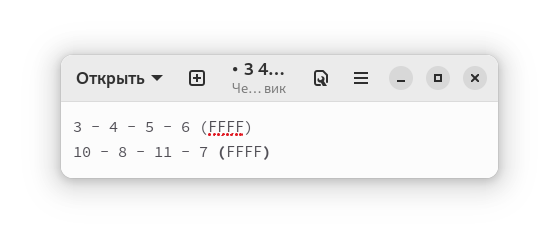


Вторая таблица не отличается от первой. Теперь откроем запись файла в корневой директории.



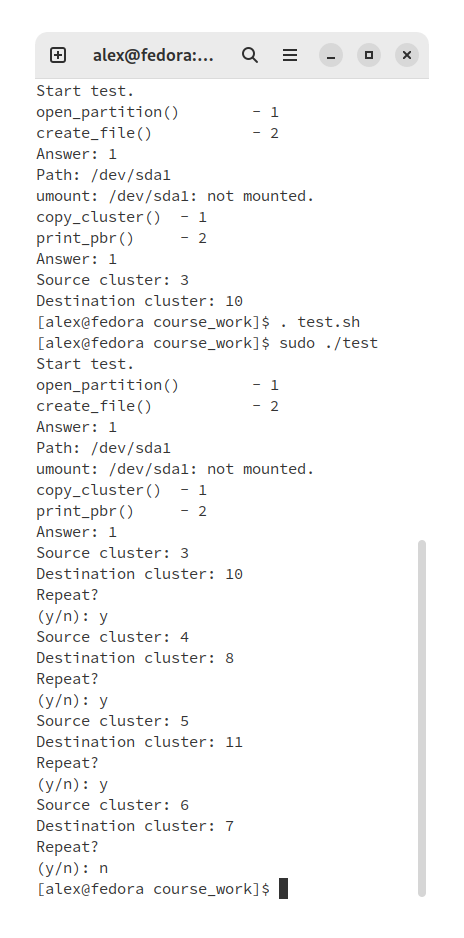
Первые 32 байта занимает запись названия раздела. А во второй записи расположился созданный файл. Здесь, по смещению в 26 байт от начала записи, в обратном виде указан номер первого кластера файла. Таким образом, я подтверждаю, что блоки с 3-го по 6-й заняты данным файлом.

Теперь, искусственным путём создам ситуацию фрагментированного файла.



Запишу в черновик последовательность кластеров файла и новую случайную последовательность, характерную фрагментации.

Далее, с помощью тестовой сборки скопирую кластеры по новым адресам.



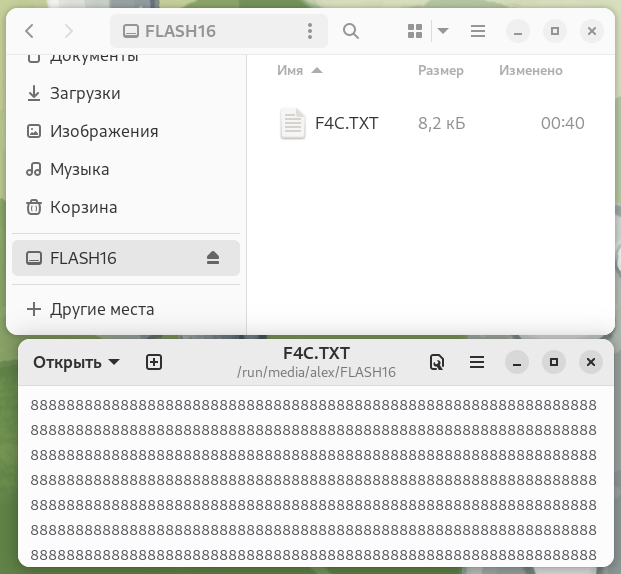
Открою редактор ghex и уже вручную изменю последовательность кластеров в таблицах FAT.



А также, в записи файла в корневой директории (меняю 3 кластер на 10).



Осталось проверить, что файл не повреждён и корректно открывается.

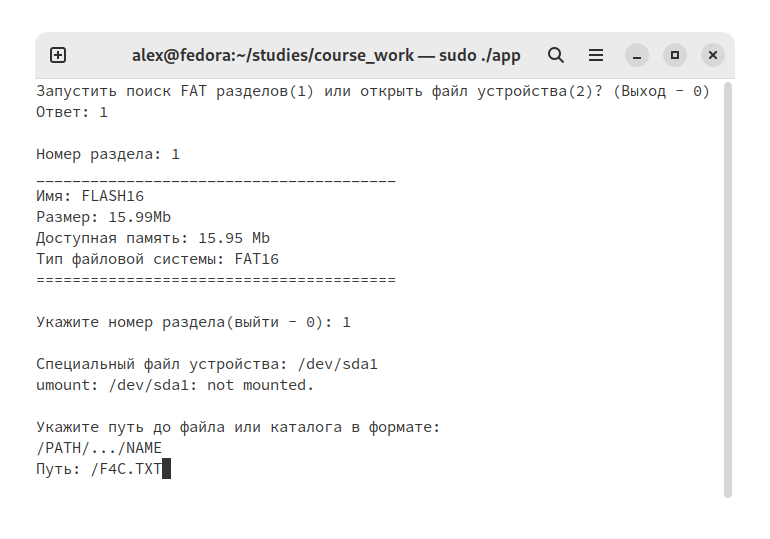


Из скриншота видно, что фрагментированный файл открывается без ошибок.

Итог

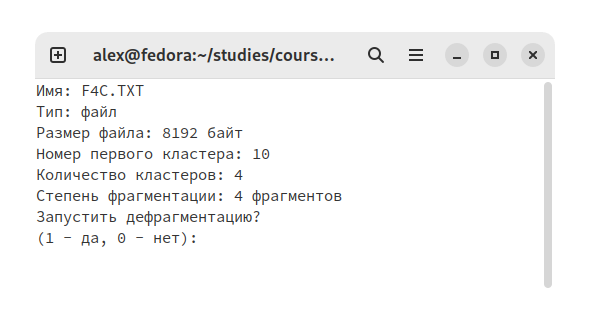
Результат работы программы

Проведу дефрагментацию с помощью созданной мной программы.

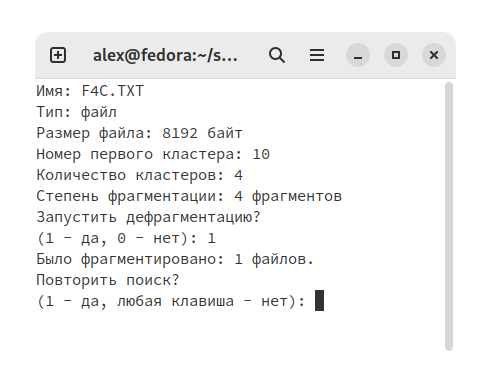


Запускаю поиск FAT разделов.

Указываю доступный раздел. Ввожу путь до файла.

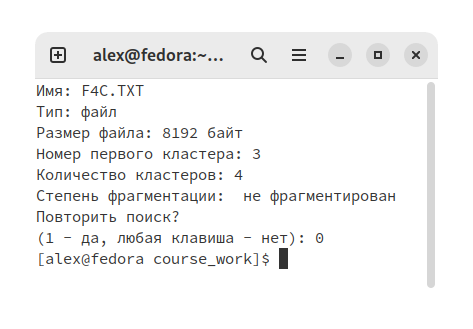


Программа выдаёт сообщение о найденном файле и степени его фрагментации. Следующим сообщением программа предлагает провести дефрагментацию. Соглашаюсь.



Программа сообщает, что 1 фрагментированный файл был исправлен.

Открою информацию о файле ещё раз.



Теперь, программа сообщает, что файл не фрагментирован.

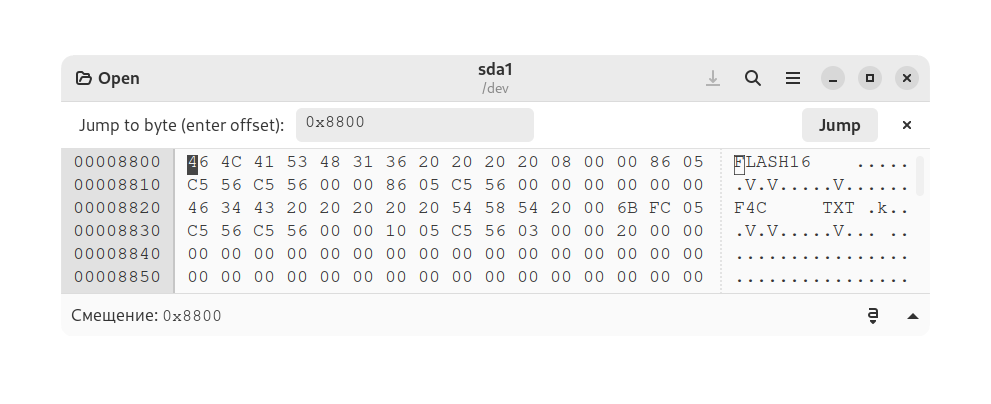
Открою программу ghex для проверки байт в таблицах FAT и в SFN записи файла.





В таблицах FAT видно, что файл действительно был дефрагментирован.

Открою также корневую директорию.



Программа также успешно исправила первый кластер в SFN записи файла.

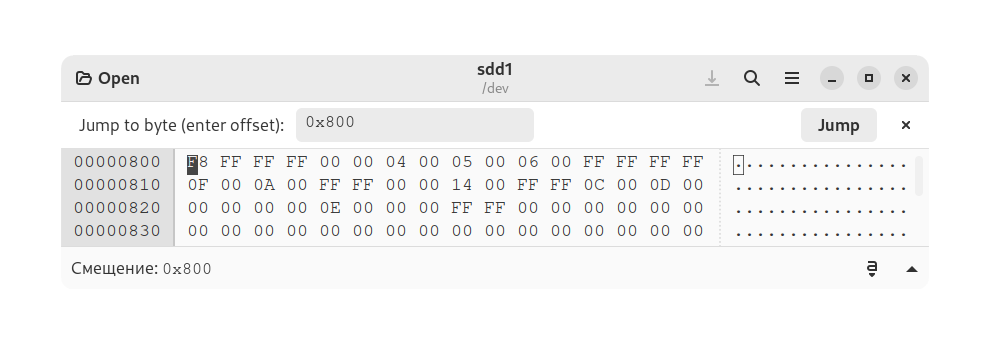
Испытание дефрагментации файла считаю успешным.

Далее, проведу проверку дефрагментации файлов каталога.

В корневой каталог помещу папку FOLDER, а внутрь неё добавлю 3 файла:

F2C.TXT, F4C.TXT и FILE3C.TXT, где номер перед буквой C означает количество занимаемых кластеров.

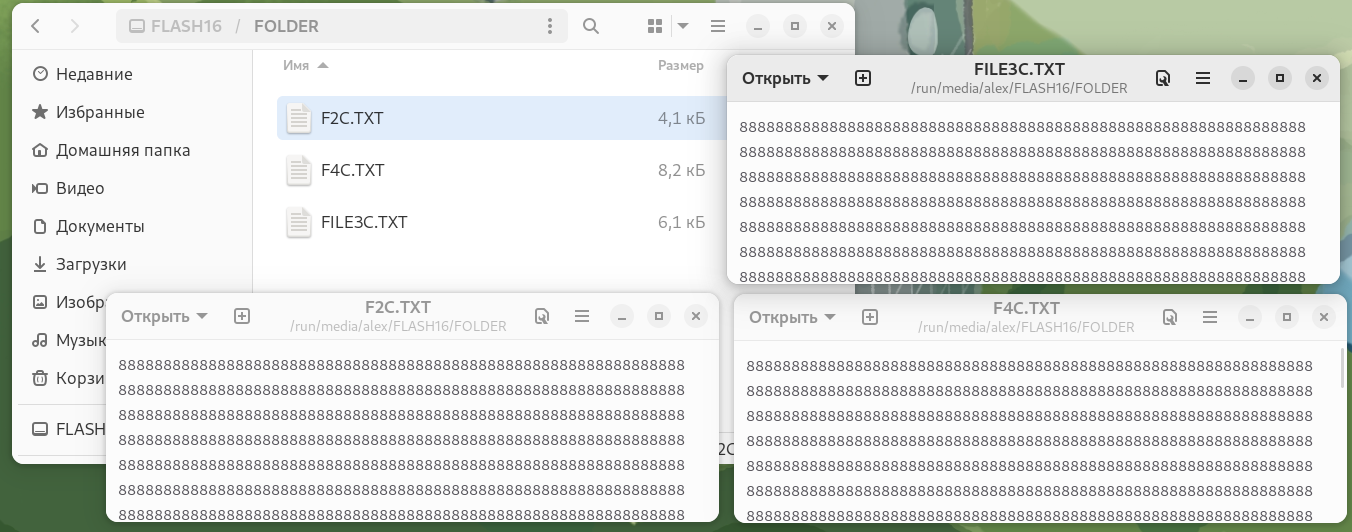
Таким же методом искусственно создам фрагментацию для двух больших файлов, а файл F2C.TXT оставлю не фрагментированным.



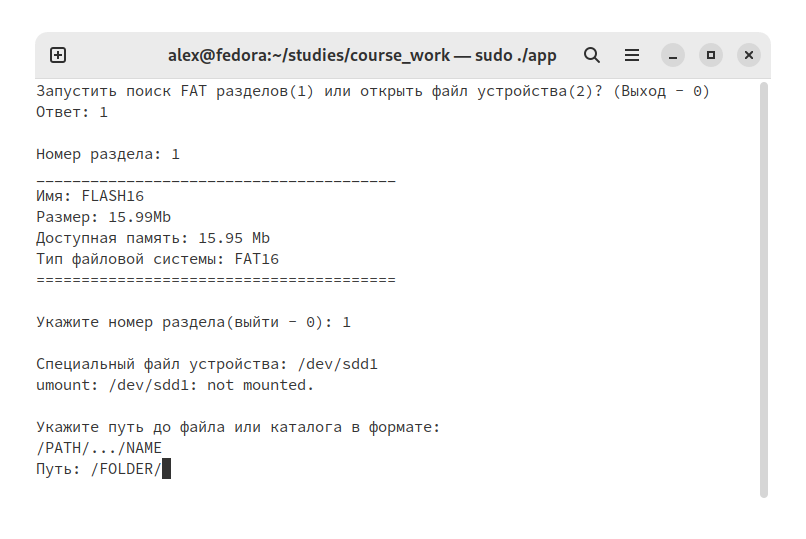


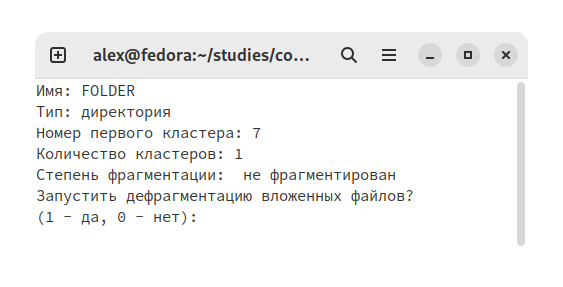
На скриншоте выше виден каталог FOLDER,в котором разместились три файла, два из которых уже фрагментированы.

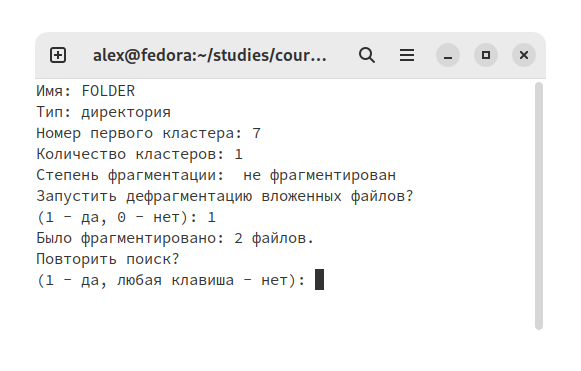
Проверим, что они ещё открываются.



Файлы открываются без ошибок. Запускаю программу дефрагментации.

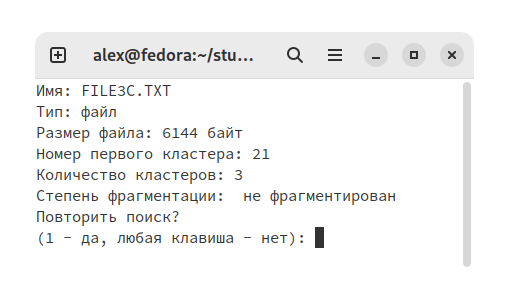


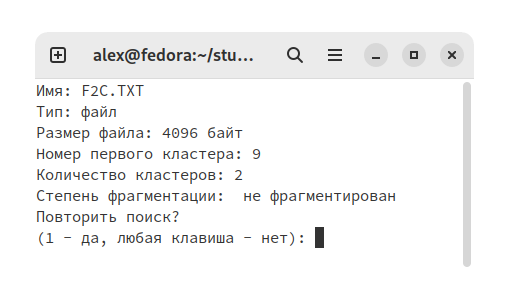


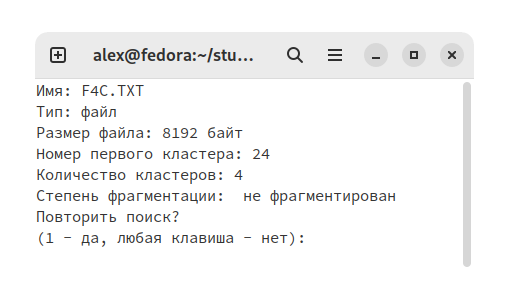


Программа сообщает, что два файла были исправлены.

Посмотрю информацию по каждому из них.

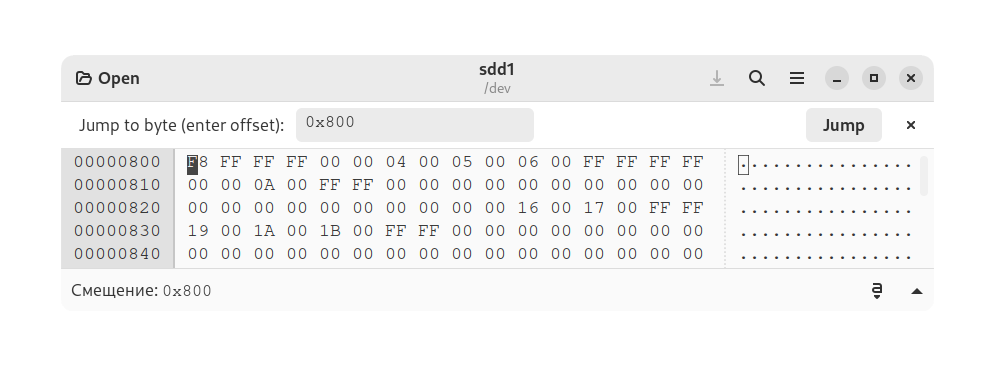






Как можно видеть, файлы больше не фрагментированы.

Открою редактор ghex для проверки.



Из таблиц FAT видно, что файлы были дефрагментированы.

Также открою каталог.



В записи последнего файла, первый кластер которого был смещён, внесена корректива.

Дефрагментация файлов каталога прошла успешно.

Вывод

В этой работе мне удалось изучить архитектуру одной из самых популярных файловых систем, которая до сих пор встречается на используемых устройствах в том или ином виде. Я воспользовался многими ресурсами языка си++ в процессе написания программы и реализовал алгоритм дефрагментации, близкий к тем, что используются в реальных системах. Вывод программы показал 100% результат работы без ошибок, за исключением незначительных лексических недочётов.

Листинг

Main.cpp

#include "Program.h"

**int** main(**int** argc, **char**\* argv[])

{

Program program;

program.start();

**return** 0;

}

Program.h

#pragma once

#include <vector>

#include <string>

#include <functional>

#include "Partition.h"

// Класс программы. Реализует диалог с пользователем и инициирует

// выполнение процедур для выполнения поставленных задач.

**class** Program

{

**private**:

/\* Функция поиска файла в указанной директории по заданному

\* условию, которое передаётся в виде функции,

\* обрабатывающей строковые значения.

\* Передаваемая функция проверяет элементы (файлы)

\* директории на предмет соответствия реализованных условий.

\* Ранее, программа могла определять доступные разделы

\* из сектора Master Boot Record, но данная реализация

\* оказалась неактуальной. \*/

std::vector<std::string> get\_files\_from\_dir

( **const** std::string& directory\_path,

std::function<**bool**(**const** std::string&)> condition );

// Метод получает список файлов, проверяет,

// какие относятся к FAT разделам

// и возвращает список подходящих файлов.

std::vector<std::string> find\_fat\_partitions

( **const** std::vector<std::string>& list );

/\* Нестатические методы класса нельзя передать

\* в качестве аргументов, поскольку они привязаны

\* к экземплярам класса (имеют скрытый аргумент this).

\* Однако, статические методы принадлежат классу

\* и не имеют связи с экземплярами класса.

\* Их можно передавать в кач-ве аргументов. \*/

// Метод проверяет строку на соответствие шаблону

// специального файла раздела файловой системы (sd[a-z][a-z][1-15])

**static** **bool** is\_partition(**const** std::string& str);

**public**:

// Метод запуска программы. Начинает диалог с пользователем.

**void** start();

// Открытый метод программы, позволяющий начать поиск указанного

// раздела. При автоматическом поиске, спрашивает у пользователя,

// какой из предложенных разделов необходим. Либо предлагает

// ручной ввод пути к специальному файлу раздела.

**auto** fat\_search(**const** std::string& directory) -> std::string;

// Метод, открывающий раздел и спрашивающий у пользователя

// путь до файла или директории, к которым необходимо

// применить алгоритм дефрагментации.

// Путь задаётся в формате "/ПУТЬ.../ИМЯ". Для дефрагментации

// файлов из корневой директории, достаточно написать слэш "/".

**auto** open\_partition(**const** std::string& sd\_filename) -> **void**;

// Метод, предоставляющий пользователю возможность запустить

// дефрагментацию, если это возможно, и выводящая сообщение

// о проделанной процедуре в случае её выполнения.

**auto** fragmentation\_check(Partition& p,

Partition::FileInfo& f) -> **void**;

};

Program.cpp

#include "Program.h"

#include <iostream> // std::cout, std::cin, std::streamsize

#include <cstdlib> // system()

#include <limits> // std::numeric\_limits<>::max()

#include <filesystem> // directory\_iterator()

#include <cstring> // strcat, strcpy

#include "PBR.h"

#include "Partition.h"

**void** Program::start()

{

**int** ch = -1;

**do**

{

system("clear");

std::cout << "Запустить поиск FAT разделов(1) "

<< "или открыть файл устройства(2)? (Выход - 0)\nОтвет: ";

std::cin >> ch;

**if** (std::cin.fail())

{

std::cin.clear();

std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');

ch = -1;

}

} **while** (ch != 2 && ch != 1 && ch != 0);

std::cout << '\n';

std::string path;

**switch**(ch)

{

**case** 1:

{

std::string dev = "/dev/";

path = fat\_search(dev);

**break**;

}

**case** 2:

{

std::cout << "Укажите путь до файла устройства.\n"

<< "Путь: ";

std::cin >> path;

**break**;

}

**case** 0:

**return**;

}

**if** (path.length())

{

**if** (ch == 1)

std::cout << "Специальный файл устройства: " << path << '\n';

**do**

{

open\_partition(path);

std:: cout << "Повторить поиск?\n(1 - да, любая клавиша - нет): ";

std::cin >> ch;

// Проверка ввода

**if** (std::cin.fail())

{

std::cin.clear();

std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');

ch = -1;

**continue**;

}

} **while** (ch == 1);

}

**else**

{

std::cout << "Разделы с файловой системой FAT не найдены.\n";

}

}

std::string Program::fat\_search(**const** std::string& directory)

{

// sd[a-z][optional: a-z][1-15]

std::vector<std::string> sd\_list = get\_files\_from\_dir

( directory, is\_partition );

std::vector<std::string> fat\_list = find\_fat\_partitions(sd\_list);

**if** (fat\_list.size())

{

**int** num = -1;

**do**

{

std::cout << "Укажите номер раздела(выйти - 0): ";

std::cin >> num;

// Проверка ввода

**if** (std::cin.fail())

{

std::cin.clear();

std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');

num = -1;

**continue**;

}

} **while** (!(num >= 0 && num <= fat\_list.size()));

std::cout << '\n';

**if** (num > 0)

**return** fat\_list[num - 1];

}

**return** std::string();

}

**void** Program::open\_partition(**const** std::string& sd\_filename)

{

Partition partition(sd\_filename);

**if** (!partition.is\_open())

{

std::cout << "Некорректный путь или файл устройства.\n";

**return**;

}

**int** num = -1;

std::string path;

// Запрашиваем имя файла

std::cout << "\nУкажите путь до файла или каталога в формате:\n"

<< "/PATH/.../NAME\nПуть: ";

std::cin >> path;

std::cout << '\n';

Partition::FileInfo file;

file = partition.get\_file(path);

**if** (file.get\_type() == Partition::NONE)

std::cout << "Файл по указанному пути не найден.\n";

**else**

{

system("clear");

partition.print\_file\_info(file);

fragmentation\_check(partition, file);

}

}

**void** Program::fragmentation\_check(Partition& p, Partition::FileInfo& file)

{

**if** ( (file.get\_type() == Partition::FILE && p.is\_file\_fragmented(file))

|| file.get\_type() == Partition::DIR

|| file.get\_type() == Partition::ROOT\_DIR)

{

**int** num = -1;

**if** (file.get\_type() == Partition::FILE)

std::cout << "Запустить дефрагментацию?\n";

**else**

std::cout << "Запустить дефрагментацию вложенных файлов?\n";

**do**

{

std::cout << "(1 - да, 0 - нет): ";

std::cin >> num;

// Проверка ввода

**if** (std::cin.fail())

{

std::cin.clear();

std::cin.ignore(std::numeric\_limits<std::streamsize>::max(), '\n');

num = -1;

**continue**;

}

} **while** (num != 0 && num != 1);

**if** (num == 1)

{

**int** amount = p.defragment(file);

std::cout << "Было фрагментировано: " << amount << " файлов.\n";

}

}

}

**bool** Program::is\_partition(**const** std::string& str)

{

**if** (str[0] != 's' && str[1] != 'd')

**return** **false**;

**if** (str.size() == 4)

**if** ( str[2] >= 'a' && str[2] <= 'z'

&& str[3] >= '1' && str[3] <= '9') // sd[a-z][1-9]

**return** **true**;

**if** (str.size() == 5)

{

**if** ( str[2] >= 'a' && str[2] <= 'z'

&& str[3] >= 'a' && str[3] <= 'z'

&& str[4] >= '1' && str[4] <= '9') // sd[a-z][a-z][1-9]

**return** **true**;

**else** **if** ( str[2] >= 'a' && str[2] <= 'z'

&& str[3] == '1'

&& str[4] >= '0' && str[4] <= '5') // sd[a-z][10-15]

**return** **true**;

}

**if** (str.size() == 6)

**if** ( str[2] >= 'a' && str[2] <= 'z'

&& str[3] >= 'a' && str[3] <= 'z'

&& str[3] == '1'

&& str[4] >= '0' && str[4] <= '5') // sd[a-z][a-z][10-15]

**return** **true**;

**return** **false**;

}

/\* Описание:

\* - Функция, проверящая в директории имена файлов и возвращающая список подходящих.

\* Параметры:

\* - путь к директории;

\* - булевая функция, проверяющая строку.

\* Возвращаемое значение:

\* - вектор значений типа std::string (возврат по значению).

\* Дополнительно:

\* - Возврат по значению необходим, поскольку при возврате по адресу/ссылке

\* будет возвращён висячий указатель, что является ошибкой. Это происходит,

\* поскольку при выходе точки выполнения программы из области видимости вектора,

\* вектор автоматически освобождает выделенную ему память.

\* Чтобы это избежать, используется механизм элизии копирования - при возврате

\* по значению, в действительности, копирования не происходит. Локальный вектор

\* функции становится вектором caller-а (функции, вызвавшей данную функцию).

\* Однако, механизм должен поддерживать компилятор.

\* clang++ и g++ поддерживают copy elision. \*/

std::vector<std::string> Program::get\_files\_from\_dir

(

**const** std::string& directory\_path,

std::function<**bool**(**const** std::string&)> condition

)

{

std::vector<std::string> list;

// Проверяем каждый файл директории:

// для этого служит итератор directory\_iterator() - в кач-ве аргумента передаётся

// путь к директории, после чего, итератор можно использовать с помощью цикла foreach

// для перебора всех значений (файлов директории).

**for** (**const** **auto**& entry : std::filesystem::directory\_iterator(directory\_path))

{

// Если переданная функция, при получении имени файла из директории, возвращает

// значение true - добавляем полный путь к файлу в вектор, как строку типа std::string.

**if** (condition( entry.path().filename().string() ))

list.push\_back(entry.path().string());

}

**return** list;

}

std::vector<std::string> Program::find\_fat\_partitions

( **const** std::vector<std::string>& list )

{

std::vector<std::string> fp\_list;

PBR pbr;

**int** counter = 0;

**for** (**auto**& el : list)

{

pbr.set(el);

**if** (pbr.is\_fat())

{

std::cout << "Номер раздела: " << ++counter << '\n';

pbr.print();

fp\_list.push\_back(el);

}

}

**return** fp\_list;

}

Partition.h

#ifndef PARTITION\_H

#define PARTITION\_H

#include <cstdint> // uint\_t

#include <fstream> // std::fstream

#include "PBR.h"

#include "Bytes.h"

// Класс, отвечающий за взаимодействие с разделом.

// Функции поиска файла и дефрагментации лежат в его реализации.

**class** Partition

{

**public**:

// Перечисление для определения и сравнения типа файла.

**enum** FileType

{

NONE = 0,

DIR,

FILE,

ROOT\_DIR

};

// Внутренний класс, экземпляры которого используются

// для хранения сведений о файлах. В отличие от структур,

// позволяет скрыть поля и сделать внешние вмешательства

// недопустимыми, однако публичность класса позволяет

// получать из класса Partition результат в виде экземпляра

// класса FileInfo, который можно сохранить и впоследствии

// передать обратно в класс Partition для обработки.

**class** FileInfo

{

**friend** **class** Partition;

**private**:

uint32\_t partition\_sn = 0;

FileType type = NONE;

uint32\_t first\_cluster = 0;

uint32\_t size = 0;

uint64\_t entry\_offset = 0;

std::string name;

**public**: FileInfo() {}

**const** std::string& get\_name() **const**

{ **return** name; }

**const** FileType get\_type() **const**

{ **return** type; }

};

// Функция вывода информации об обнаруженном файле.

**void** print\_file\_info(**const** FileInfo&);

**protected**:

// Экземпляр класса PBR(Partition Boot Record).

// Используется для хранения сведений о разделе.

// К нему регулярно приходится обращаться для работы с разделом.

PBR m\_pbr;

// Контейнер для байт таблицы FAT. Из него считываются данные

// о занимаемых файлами кластерах. При необходимости, в таблицу

// вносятся изменения, после чего, она может быть записана

// обратно в файл устройства для фиксации изменений.

Bytes m\_FAT;

// Экземпляр, реализующий доступ к файлу устройства.

// Через него осуществляется доступ к файлам в разделе, его данным.

// С помощью этого экземпляра также осуществляется запись

// всех изменений на раздел.

std::fstream m\_drive;

// Метод для инициализации экземпляра класса:

// - Открывается поток к файлу устройства для чтения и записи;

// - Считывается загрузочная запись раздела, и если запись подлинная,

// - Считывается таблица FAT в контейнер.

**auto** init(**const** std::string& path) -> **void**;

// Поиск файла:

// Вспомогательные методы для поиска файла.

// Метод извлекает имя файла/директории до первого слэша ("/").

**auto** extract\_name(**const** std::string& path) -> std::string;

// Метод обрезает строку, в которой указан путь к файлу, от начала

// до первого слэша ("/").

**auto** cut\_string(std::string& path, **char** ch) -> **void**;

// Метод ищет файл по указанному имени в кластере буферизованной

// директории. При нахождении, возвращает заполненный экземпляр

// класса FileInfo с данными файла.

**auto** search\_file(std::string& filename,

Bytes& dir\_cluster) -> Partition::FileInfo;

// Вспомогательный метод для извлечения имени файла из байт

// в буфферизованной директории. Имя затем сравнивается с искомым.

**auto** get\_entry\_name(Bytes& dir, size\_t offset) -> std::string;

// Вспомогательный метод, извлекающий из байт буфферизованной

// директории информацию о типе файла (файл, директория, неизвестное).

**auto** get\_file\_type(Bytes& dir, size\_t offset) -> FileType;

// Метод для получения экземпляра класса из буфферизованной

// директории по указанному смещению байт. Номер кластера

// директории указывается, чтобы заполнить информацию

// о смещении записи файла в разделе, для возможного внесения

// изменений.

**auto** get\_file\_from\_entry(Bytes& dir,

uint32\_t dir\_cluster\_number, size\_t offset) -> FileInfo;

// Метод возвращает экземпляр, заполненный данными по корневой

// директории.

**auto** get\_root\_dir() -> FileInfo;

// Обнаружение/устранение фрагментации

// Главная функция дефрагментации - осуществляет проверку

// на фрагментацию и дефрагментацию указанного файла или каталога,

// поскольку, каталоги также, в теории, могут быть фрагментированы.

**auto** defragment\_file(FileInfo& file) -> uint32\_t;

// Вспомогательный метод при обработке директорий.

// Проходится по всем кластерам директории, вызывая

// для каждого кластера метод defragment\_dir\_cluster().

**auto** defragment\_dir(FileInfo& file) -> uint32\_t;

// Метод, извлекающий из буферизованных кластеров директорий

// файлы, которые затем передаёт в метод defragment\_file().

**auto** defragment\_dir\_cluster(Bytes& dir\_cluster,

uint32\_t cluster\_number) -> uint32\_t;

// Метод, копирующий указанный кластер по указанному адресу.

**auto** copy\_cluster(uint32\_t source, uint32\_t destination) -> **void**;

// Метод, используемый для поиска требуемого свободного пространства

// для дефрагментации файла.

**auto** find\_empty\_space(uint32\_t clusters\_number) -> uint32\_t;

// Метод для подсчёта занимаемых файлом кластеров.

// Универсален для любых типов файлов, поскольку высчитывает

// кластеры по таблице FAT из контейнера.

**auto** count\_file\_clusters(**const** FileInfo& file) -> uint32\_t;

**public**:

// Конструктор класса. Позволяет создать экземпляр только

// при указании пути. При некорректном пути, потребуется

// создавать новый экземпляр. Это сказывается на гибкости,

// но несколько сокращает возможные ошибки.

Partition(**const** std::string& path) { init(path); }

// Метод для проверки, был ли инициализирован экземпляр корректно.

**auto** is\_open() **const** -> **bool**;

// Метод для получения экземпляра класса FileInfo.

// Используется для поиска файла по заданному пути для дальнейших

// манипуляций, а именно, дефрагментации.

**auto** get\_file(std::string& path) -> FileInfo;

// Метод, используемый для проверки файла на фрагментацию.

**auto** is\_file\_fragmented(**const** FileInfo& file) -> uint32\_t;

// Открытый метод, запускающий процесс дефрагментации файла.

// Возвращает количество дефрагментированных файлов.

**auto** defragment(FileInfo& file) -> uint32\_t;

~Partition()

{

**if** (m\_drive.is\_open())

m\_drive.close();

}

};

#endif // PARTITION\_H

PBR.h

#ifndef PBR\_H

#define PBR\_H

#include <fstream>

#include <cstdint>

#include "Bytes.h"

/\* Класс, используемый для получения и обработки данных о разделах.

\* Считывается первый сектор раздела и определяется, относится ли

\* раздел к искомому. При положительном результате, извлекаются

\* полезные данные. \*/

**class** PBR // Partition Boot Record (Загрузочная запись раздела).

{

**public**:

// Перечисление для определения и сверки типа раздела.

**enum** FAT\_Type

{

FAT12,

FAT16, // Данная программа реализует поддержку только FAT16

FAT32,

NONE

};

// Структура для хранения исчерпывающей информации о разделе.

**struct** Parameters

{

uint32\_t cluster\_size = 0;

uint64\_t fat\_offset = 0;

uint8\_t fat\_number = 0;

uint64\_t fat\_size = 0;

uint64\_t data\_offset = 0;

uint64\_t partition\_size = 0;

uint32\_t root\_dir\_size = 0;

uint32\_t clusters\_number = 0;

FAT\_Type fat\_type = NONE;

uint32\_t root\_dir\_cluster = 0;

uint32\_t last\_cluster = 0;

uint32\_t serial\_number = 0;

std::string label;

};

**private**:

// Контейнер для хранения байт записи раздела.

Bytes m\_buff;

// Экземпляр структуры.

Parameters m\_parameters;

// Методы для инициализации данных экземпляра класса.

// Они открывают специальный файл устройства (раздела)

// и переносят загрузочную запись в контейнер.

**void** init(**const** std::string& path, uint64\_t offset = 0);

**void** init(std::fstream& drive);

// Главный метод класса - он инициализирует структуру

// данными, извлекаемыми из байт загрузочной записи раздела.

**void** set\_pbr(uint64\_t offset = 0);

**public**:

// Конструкторы класса.

PBR();

PBR(**const** std::string path, uint64\_t offset = 0);

// Проверка сигнатуры загрузочной записи. Проверяется сигнатура

// загрузочной записи, а также сигнатура записи FAT раздела.

**bool** is\_pbr() **const**;

// Если экземпляр класса считал истинный раздел FAT, метод

// возвращает true, иначе false.

**bool** is\_fat() **const**;

// Два метода для считывания указанного раздела и перезаписи

// информации в экземпляре.

**auto** set(**const** std::string& path, uint64\_t offset = 0) -> **void**;

**auto** set(std::fstream& drive) -> **void**;

// Метод для очистки (обнуления) данных экземпляра. Используется

// перед переинициализацией методами set().

**auto** clear() -> PBR&;

// Самый часто используемый метод - возвращает ссылку на структуру

// с данными по разделу. Позволяет обращаться к данным напрямую,

// а изменение данных извне запрещено.

**auto** get\_parameters() **const** -> **const** Parameters&

{ **return** m\_parameters; }

// Метод возвращает ссылку на контейнер. Не востребован.

Bytes& get\_bytes() { **return** m\_buff; }

// Вывод значений структуры. Используется при отладке и тестах.

**void** print\_parameters() **const**;

// Используется для оповещения пользователя об обнаруженном разделе.

// Выводит основную, полезную пользователю информацию.

**void** print() **const**;

};

#endif // PBR\_H

Bytes.h

#pragma once

#include <cstdint>

#include <cassert>

#include <type\_traits> // std::is\_integral<T>::value

// Контейнерный класс для хранения байт. Используется, в основном,

// в качестве буфера. Позволяет быстро обработать данные,

// при необходимости, изменить и записать обратно в файл.

**class** Bytes

{

**protected**:

/// Символьный динамический массив для хранения байт.

**char**\* m\_bytes;

// Переменная, хранящая размер массива.

size\_t m\_size;

// Скрытый метод копирования для реализации глубокого

// копирования (чтобы избежать копирования адресов).

Bytes& copy(**const** Bytes& b);

**public**:

Bytes() : m\_size(0)

{ m\_bytes = **nullptr**; }

Bytes(size\_t size) : m\_size(size)

{ m\_bytes = **new** **char**[size]; }

Bytes(**const** Bytes& b)

{ copy(b); }

// Перегрузка данных операторов позволяет при передаче

// экземпляра класса передавать указатель на символьный массив,

// когда это ожидается.

**operator** **char**\*() { **return** m\_bytes; }

**operator** **const** **char**\*() **const** { **return** m\_bytes; }

// Перегруженный оператор= для присваивания экземпляров класса

// для реализации глубокого копирования (выделяется новая память

// и элементы копируются по значению).

Bytes& **operator**=(**const** Bytes& b) { **return** copy(b); }

// Перегруженный оператор[] для получения элементов массива по индексу.

**char**& **operator**[](size\_t index)

{

assert(index < m\_size && "index is out of range.");

**return** m\_bytes[index];

}

// Метод возвращает длину символьного массива.

**auto** length() **const** -> size\_t { **return** m\_size; }

// Возвращает указатель на символьный массив.

**auto** get\_pointer() -> **char**\* { **return** m\_bytes; }

// Очищение (освобождение) выделенной памяти. Обнуление размера.

**auto** clear() -> Bytes&;

// Изменяет размер выделенной памяти для буфера.

**auto** resize(size\_t length) -> Bytes&;

// Возвращает строку из байт указанного размера, по указанному смещению.

**auto** get\_string(size\_t offset, size\_t size) **const** -> std::string;

// Функции вывода байт в контейнере. В работе программы не используются.

// Нужны при проверке работы кода.

**auto** print\_bytes() **const** -> **void**;

**static** **void** print\_bytes(**char**\* buff, size\_t amount);

// Вложенный тип (перечисление). Используется для указания

// размера байт, которое необходимо считать или записать в буфер.

**enum** TypeSize

{

BYTE = 1,

WORD = 2,

DOUBLE\_WORD = 4

};

// Следующие два шаблона метода возвращают значения из указанных байт.

// Они могут возвращать результат разных типов и размеров.

// Из-за особенности хранения байт данных в файловой системе FAT

// (обратная последовательность байт), данные функции также решают

// и эту проблему, возвращая корректный результат.

**template** <**typename** T>

T get\_value(size\_t offset) **const**;

**template** <**typename** T>

T get\_value(size\_t offset, TypeSize size) **const**;

// Следующие два шаблона методов вставляют указанные значения,

// которые могут быть разных размеров и разных типов,

// в символьный (байтовый) буфер. Впоследствие, буфер может быть

// использован для записи его в файл.

**template** <**typename** T>

**void** insert(T value, size\_t offset, TypeSize size);

**template** <**typename** T>

**void** insert(T value, size\_t offset);

// Деструктор класса. При выходе экземпляра из области видимости,

// выделенная память под символьный массив освобождается.

~Bytes()

{

**if** (m\_bytes != **nullptr**)

**delete**[] m\_bytes;

}

};

// Реализация шаблонов обязана быть в одном файле с их объявлением.

**template** <**typename** T>

T Bytes::get\_value(size\_t offset) **const**

{

**static\_assert**(std::is\_integral<T>::value, "Expected integral type.");

assert(offset + **sizeof**(T) <= m\_size && "Invalid offset.");

T value = **static\_cast**<T>(0);

**for** (**int** i = **sizeof**(T) - 1; i >= 0; --i)

value |= **static\_cast**<T>(**static\_cast**<**unsigned** **char**>(m\_bytes[offset + i])) << 8 \* i;

**return** value;

}

**template** <**typename** T>

T Bytes::get\_value(size\_t offset, TypeSize size) **const**

{

**static\_assert**(std::is\_integral<T>::value, "Expected integral type.");

assert(size <= **sizeof**(T) && "Size is bigger than type size.");

assert(offset + size <= m\_size && "Invalid offset and size.");

T value = **static\_cast**<T>(0);

**for** (**int** i = size - 1; i >= 0; --i)

value |= **static\_cast**<T>(**static\_cast**<**unsigned** **char**>(m\_bytes[offset + i])) << 8 \* i;

**return** value;

}

**template** <**typename** T>

**void** Bytes::insert(T value, size\_t offset, TypeSize size)

{

**static\_assert**(std::is\_integral<T>::value, "Expected integral type.");

assert(size <= **sizeof**(T) && "Size is bigger than type size.");

assert(offset + size <= m\_size && "Invalid offset and size.");

**for** (size\_t i = 0; i < size; ++i)

{

m\_bytes[offset + i] = **static\_cast**<**char**>(value >> 8 \* i);

}

}

**template** <**typename** T>

**void** Bytes::insert(T value, size\_t offset)

{

**static\_assert**(std::is\_integral<T>::value, "Expected integral type.");

//assert(size <= sizeof(T) && "Size is bigger than type size.");

assert(offset + **sizeof**(T) <= m\_size && "Invalid offset and size.");

**for** (size\_t i = 0; i < **sizeof**(T); ++i)

{

m\_bytes[offset + i] = **static\_cast**<**char**>(value >> 8 \* i);

}

}

Test.cpp

#include <iostream>

#include "Partition.h"

// Наследник класса Partition, созданный с целью использования

// методов предшествующего класса в альтернативных сценариях.

// А именно, можно быстро вывести основную информацию о разделе,

// и скопировать указанный кластер по указанному адресу.

**class** TPart: **public** Partition

{

**public**:

TPart(**const** std::string& path) : Partition(path) {}

**void** copy\_cluster(uint32\_t from, uint32\_t to)

{

Partition::copy\_cluster(from, to);

}

**void** print\_pbr()

{

std::cout << std::hex << std::uppercase << '\n';

m\_pbr.print\_parameters();

std::cout << std::dec << '\n';

m\_pbr.print\_parameters();

std::cout << std::endl;

}

uint32\_t get\_cluster\_size()

{

**return** m\_pbr.get\_parameters().cluster\_size;

}

};

// Класс Test аналогичен по назначению классу Program,

// с той разницей, что предоставляет опции для тестирования

// и создания фрагментации, вместо её обнаружения и устранения.

**class** Test

{

**public**:

Test() {}

// Метод создаёт файл указанного размера в кластерах,

// размер которых также указывается.

// Файл заполняется однобайтовым символом.

**void** create\_file(std::string name, uint32\_t size\_in\_clusters, uint32\_t cl\_s)

{

std::ofstream outfile(name);

uint32\_t limit = size\_in\_clusters \* cl\_s;

**for** (uint32\_t i = 0U; i < limit; ++i)

{

**if** ((i & 0x3FU) == 0x3FU)

outfile << '\n';

**else**

outfile << '8';

}

outfile.close();

}

// Метод начала работы с экземпляром.

// Начинает диалог с пользователем.

// Предлагает открыть раздел или создать файл.

**void** start()

{

**int** num = -1;

std::cout << "open\_partition()\t- 1\n"

<< "create\_file()\t\t- 2\n";

std::cout << "Answer: ";

std::cin >> num;

**switch**(num)

{

**case** 1:

{

open\_partition();

**break**;

}

**case** 2:

{

std::string name;

uint32\_t cl;

uint32\_t cl\_size;

std::cout << "Enter name, cl and cl\_size: ";

std::cin >> name;

std::cin >> cl;

std::cin >> cl\_size;

create\_file(name, cl, cl\_size);

**break**;

}

}

}

// Промежуточный метод, создающий экземпляр тестового класса.

// Передаёт его в следующий метод.

**void** open\_partition()

{

std::cout << "Path: ";

std::string path;

std::cin >> path;

TPart p(path);

**if** (p.is\_open())

{

p\_options(p);

}

}

// Данный метод продолжает диалог с пользователем, предлагая

// скопировать выбранный кластер в указанную позицию,

// либо напечатать подробную информацию о разделе.

**void** p\_options(TPart& p)

{

**int** num = -1;

std::cout << "copy\_cluster()\t- 1\n"

<< "print\_pbr()\t- 2\n";

std::cout << "Answer: ";

std::cin >> num;

**switch**(num)

{

**case** 1:

{

uint32\_t src;

uint32\_t dest;

**char** ch;

**do**

{

std::cout << "Source cluster: ";

std::cin >> src;

std::cout << "Destination cluster: ";

std::cin >> dest;

p.copy\_cluster(src, dest);

std::cout << "Repeat?\n(y/n): ";

std::cin >> ch;

} **while** (ch == 'y');

**break**;

}

**case** 2:

{

p.print\_pbr();

**break**;

}

}

}

};

**int** main(**int** argc, **char**\* argv[])

{

std::cout << "Start test.\n";

Test test;

test.start();

**return** 0;

}