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 свободной памяти на диске FAT»

по дисциплине «Принципы и методы организации системных программных средств»

Выполнил:

Проверил:

Кочешков А.А.

Нижний Новгород

Содержание:

1. Вводная часть
2. Теоретическая часть
   1. Общие сведения
   2. Зарезервированная область
   3. Область таблиц FAT
   4. Область данных
   5. Корневая директория
3. Практическая часть
   1. Описание реализованного алгоритма дефрагментации свободной памяти FAT16
   2. Описание структуры программы
   3. Описание работы программы
   4. Результат работы программы
   5. Вывод
4. Листинг

Вводная часть

Задание: разработать программу дефрагментации свободной памяти на диске файловой системы FAT. При запуске контролировать тип и свойства файловой системы на томе. Выводить отчёт.

Данная работа выполнялась при определённых вводных данных.

Операционная система: Fedora Linux 36 (Workstation Edition)

Ядро: Linux 5.17.9-300.fc36.x86\_64 x86\_64

Архитектура: x86\_64

Система запускалась под виртуальной машиной.

Была выбрана файловая система FAT16.

Решено пренебречь современными решениями по взаимодействию с данной файловой системой и отбросить возможность создания каталогов, а также поддержку длинных имён. На момент появления системы, накопители могли предоставить около нескольких сотен килобайт, в рамках которых не имело смысла создавать иерархии файлов, а поддержка длинных имён появилась только в модификации VFAT, хотя в дальнейшем была реализована и на базе FAT16.

Для выполнения задания, использован флеш накопитель объёмом 2 гигабайта.

Накопитель был форматирован под FAT16 с помощью программы GParted.

Средство разработки: язык программирования C.

Программа написана для запуска в терминале.

При запуске, необходимо указать путь к специальному файлу накопителя, который по умолчанию располагается в линукс по пути /dev/, и имеет обозначение sdAN, где A – буква, обозначающая физическое устройство, а N – цифра, обозначающая раздел (например, раздел флеш накопителя с файловой системой FAT16).

Для корректной работы программы, устройство необходимо размонтировать.

Теоретическая часть

Общие сведения

FAT (таблица размещения файлов) - классическая архитектура файловой системы, разработанная Биллом Гейтсом и Марком МакДональдом, и применявшаяся в MSDOS и Windows 95.

Существует четыре версии FAT — FAT12, FAT16, FAT32 и exFAT (FAT64). Их различие состоит в разрядности записей в дисковой структуре, то есть количеством бит, отведённых для хранения номера кластера.

В своём первозданном виде FAT не поддерживала иерархическую структуру каталогов – все файлы располагались в корневой директории накопителя.

Имелось ограничение по размеру имён и их форме записи – существовал стандарт 8.3, позволяющий хранить имена длиной до 8 символов в верхнем регистре, и 3 символа отводилось на расширение файла.

В дальнейшем, в разных операционных системах были внедрены модификации файловой системы, расширяющие атрибуты файлов, добавляющие поддержку длинных имён и так далее.

Файловая система FAT16 разделена на три условные части:

1. Зарезервированная область;
2. Область таблиц FAT;
3. Область данных

Информация в файловой системе FAT интерпретируется, как набор шестнадцатеричных символов, по два на байт. Данные можно представить в виде таблицы по 16 байт в строке. У каждого байта имеется смещение или адрес – количество байт от начала тома FAT.

Зарезервированная область

В зарезервированной области находится BIOS Parameter Block (BPB), или загрузочная запись. Она хранит сведения о томе, такие, как:

- Количество байтов в секторе

- Количество секторов в кластере

- Количество таблиц FAT (обычно, их две)

- Общее число секторов в разделе (объём накопителя в секторах)

Область таблиц FAT

Таблица FAT представляет собой массив двухбайтовых значений, соответствующих кластерам в области данных. Если файл занимает определённый кластер, эта информация будет зафиксирована в таблице FAT. Когда файл занимает один кластер – в таблице FAT, в ячейке, соответствующей этому кластеру, будет код FFFF в шестнадцатеричной системе счисления или 65535. Если файл больше размера одного кластера, то каждая ячейка в таблице будет хранить номер следующего кластера, кроме последнего кластера – в нём будет записан код FFFF. Нулевой элемент таблицы обозначает начало таблицы FAT, первый элемент связан с первым кластером, а в случае с FAT16, это корневая директория.

Область данных

В области данных располагаются непосредственно данные. Каждый файл занимает кратное количество кластеров. Если размер файла превышает размер кластера – система выделит ему следующий свободных кластер. В одном кластере может находиться только один файл. Таким образом, для хранения множества маленьких файлов рациональнее использовать меньший размер кластера, однако от размера кластера зависит размер корневой директории, которая определяет допустимое количество записываемых файлов.

Корневая директория

Информация о файлах – имя, тип, размер, расположение (а именно, начальный кластер) – записывается в корневой директории в виде 32 байтовой записи на файл. Ниже представлена схема такой записи:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| h | 00 | 01 | 02 | 03 | 04 | 05 | 06 | 07 | 08 | 09 | 0A | 0B | 0C | 0D | 0E | 0F |
| 00 | 46 | 49 | 4C | 45 | 20 | 20 | 20 | 20 | 54 | 58 | 54 | 20 |  |  |  |  |
| 10 |  |  |  |  |  |  |  |  |  |  | 03 | 00 | 05 | 00 | 00 | 00 |

Байты с нулевого по седьмой отводятся под имя, а с восьмого по десятый (A) - под расширение. Буквы записываются в верхнем регистре, окончание имени обозначается шестнадцатеричным номером 20 (означает пробел). Для примера, в верхней таблице написано имя «FILE.TXT». Один байт по смещению «0B» содержит в себе атрибут файла. Номер 20 – метка простого файла (как текстовый документ, изображение и тд). Два байта по смещению 1A содержат номер первого кластера, занимаемого файлом. Если файл занимает больше одного кластера, цепочка кластеров выстраивается по таблице FAT. Числа в файловой системе FAT записываются слева направо, то есть первый байт является младшим. Таким образом, запись «0300» означает, что первым кластером файла является третий кластер. Четыре байта по смещению 1C содержат размер файла. Для примера, здесь указано 5 байт.

Практическая часть

Описание реализованного алгоритма дефрагментации свободной памяти в FAT16

Всякий неповреждённый файл занимает определённое количество кластеров. Занятые кластеры отмечены в таблице FAT. В корневой директории, в записи каждого файла указан номер первого кластера.

Это данные, которые присутствуют в томе по умолчанию. И при дефрагментации, эти данные могут быть изменены. Чтобы контролировать изменение этих данных, было решено реализовать реестр, хранящий такие данные по кластерам, как:

принадлежность определённым файлам; привязка к другим кластерам; является ли кластер первым кластером файла.

Эти сведения помогут определить действия при необходимости перемещения кластеров.

Каждый элемент в списке реестра соотносится по порядку с кластерами, то есть записей в реестре должно быть столько же, сколько кластеров в файловой системе.

После создания реестра и заполнения его данными, можно начать процесс дефрагментации.

Нулевую, первую и вторую позиции требуется пропустить. В таблице FAT нулевая позиция обозначает начало таблицы FAT. Первая позиция относится к корневой директории, которая должна оставаться нетронутой. Вторая позиция по неизвестной причине зарезервирована системой и не используется при записи данных (информация актуальна для системы, упомянутой в разделе вводных данных).

При сканировании реестра, начиная с третьей записи, проверяется занятость соответствующего кластера. Если кластер свободен, увеличивается счётчик свободных кластеров. Как только появляется запись о заполненном кластере, если счётчик свободных кластеров ненулевой, первая заполненная запись смещается на количество зафиксированных свободных кластеров. Если кластер является первым, номер нового первого кластера записывается в запись файла в корневой директории. Если кластер не первый, перезаписывается номер следующего кластера в предыдущем кластере. Затем, происходит обновление данных в реестре. Сканирование продолжается уже с позиции, следующей после кластера, куда был перезаписан занятый кластер. Предыдущая позиция кластера освобождается.

Таким образом, сканируется весь реестр до конца. По окончании сканирования, дефрагментация будет завершена.

Описание структуры программы

my\_header.h

– заголовочный файл. В нём описаны структуры и все используемые функции. Ввиду количества функций и объёма кода, было решено разбить проект по файлам, придав специфическое обозначение каждому. Заголовочный файл позволяет связать, при сборке проекта, реализованные в разных файлах функции, структуры и другие решения.

Внутри заголовочного файла описаны используемые структуры:

* BPBSector – структура, хранящая базовые данные о системе из загрузочной записи. Используется для идентификации тома FAT16;
* Drive – структура, хранящая такие сведения, как смещения областей данных, размер кластера, снимок таблицы FAT, её размер и количество этих таблиц;
* Array – структура, служащая обёрткой для указателя на список элементов и размера списка. Удобна для передачи в функции и получения из них результатов. Универсальна для любого типа данных.
* DriveFile – структура, хранящая смещение записи файла в корневой директории и номера первого кластера. Используется для учёта записей файлов.
* Table – структура, хранящая информацию о кластере. Используется в виде списка (массива), созданного в соответствии с таблицей FAT.

main.c

– главный файл проекта. В нём собираются воедино и используются в требуемом порядке почти все средства, описанные в других файлах. По нему можно увидеть шаги, которые предпринимает программа в ходе своей работы.

readers.c

– здесь описаны функции, считывающие данные с накопителя и предоставляющие их в удобном для работы виде.

*readBytesAsInt()* – функция, считывающая заданное количество байт и возвращающая их, как целочисленное значение.

*readBytesAsString()* – функция, считывающая заданное количество байт и возвращающая их, как строку.

*readBytesAsUShort()* – функция, считывающая заданное количество байт и возвращающая их, как массив целочисленных без-знаковых двухбайтовых значений. Используется для сканирования таблицы FAT.

getters.c

– здесь описаны функции, скрывающие заполнение структур данными, и возвращающие их экземпляры или указатели.

*getBPB()* – функция, заполняющая структуру BPBSector и возвращающая её экземпляр.

*getDrive()* – функция, заполняющая структуру Drive и возвращающая её экземпляр.

*getFiles()* – функция, сканирующая корневую директорию файловой системы FAT16 и возвращающая список экземпляров структуры DriveFile.

*getTable()* – функция, создающая массив пустых структур Table в соответствии со сканированной таблицей FAT, и возвращающая указатель на список экземпляров.

printers.c

– здесь описаны функции, выводящие какую-либо информацию в стандартный поток вывода (консоль). Используются для отчётности и отладки.

*printBPB()* – функция, используемая для вывода информации из структуры BPBSector. Формирует читаемую выдержку о базовых параметрах тома.

*printDrive()* – функция, выводящая данные структуры Drive.

*printFileList()* – функция печатает список структур DriveFile, соответствующих найденным записям файлов. Служит для отладки.

*printSectors()* – функция печатает в читаемом виде указанное число байт, кратное 512, по указанному смещению. Используется для отладки.

*printTable()* – функция печатает данные указанного количества экземпляров списка структур Table. Используется для отладки.

checkers.c

– здесь описаны функции проверки каких-либо данных.

*checkBPB()* – функция, используемая для проверки данных структуры BPBSector, считанных из загрузочной записи по стандартным смещениям.

*checkFreeSpace()* – функция проверяет количество фрагментов свободной памяти.

markTable.c

– внутри реализована функция *markTable()*, заполняющая элементы массива структур Table, соответствующие занятым кластерам.

defragFreeSpace.c

– внутри описана функция *defragmentFreeSpace()*, выполняющая дефрагментацию свободного пространства.

movers.c

– здесь описаны функции по перемещению кластеров.

*moveCluster()* – функция перемещает кластер, не являющий первым в цепочке.

*moveFirstCluster()* – функция по перемещению первого кластера файла.

Описание работы программы

// 1

При запуске программы, вторым аргументом (первым является название программы) вводится путь к специальному файлу накопителя.

Программа пытается открыть поток к данному файлу, при неудаче происходит завершение.

// 2

Создаётся экземпляр структуры BPBSector, хранящей требуемые данные для идентификации тома FAT16, как например количество байтов в секторе, количество секторов в кластере и так далее.

Экземпляр передаётся в функцию checkBPB(), которая проверяет данные внутри структуры на совпадение с допустимыми значениями.

По успешной проверке, происходит вывод считанных данных в консоль.

// 3

Далее, создаётся экземпляр структуры Drive, содержащей данные тома, к которым требуется частое обращение.

Эти данные также выводятся в стандартный поток вывода для демонстрации и отчётности.

// 4

Создаётся экземпляр структуры Array, служащей обёрткой для хранения указателя на список элементов любого типа и размера этого списка.

Экземпляру присваивается значение на выходе функции getFiles().

Функция берёт на вход экземпляр структуры Drive и анализирует корневую директорию тома на наличие существующих файлов. При нахождении записи файла, создаётся экземпляр структуры DriveFile. Экземпляры связываются в список. Параллельно ведётся учёт их количества. По завершении, функция возвращает экземпляр структуры Array, внутри которого указатель на список структур DriveFile и размер этого списка.

Затем, создаётся указатель на структуру FileDrive, которому присваивается значение полученного из getFiles() экземпляра структуры Array (ptr).

Также, создаётся переменная, которой присваивается значение размера списка.

// 5

Создаётся список структур Table. Данный список служит реестром таблицы FAT. Каждый экземпляр этой структуры соответствует по порядку ячейкам таблицы FAT. В ней хранится информация о том, каким файлом занята ячейка; является ли кластер, соответствующий ячейке FAT, первым в цепочке кластеров файла; если кластер не первый в цепочке, экземпляр структуры Table также будет хранить номер предыдущего кластера.

С помощью данного списка будет происходить процесс дефрагментации свободной памяти путём перемещения занятых кластеров в левую часть области данных.

Для создания данного списка, используется функция getTable().

После получения списка, функция markTable() заполняет таблицу в соответствии с имеющимися данными, а именно, сверяет список файлов, полученный ранее.

// 6

Когда таблица заполнена, у программы имеется достаточная информация для дефрагментации свободной памяти.

Вызывается функция defragmentFreeSpace(), куда передаётся таблица и сопутствующие данные.

// 6.1

Внутри функции defragmentFreeSpace(), цикл for проходит по ячейкам FAT таблицы, начиная с третьей, поскольку нулевая используется для обозначения начала таблицы FAT, первая соответствует первому кластеру, в котором располагается корневая директория, а во второй кластер система не осуществляет запись файлов и причин такому поведению узнать не удалось, поэтому было решено пропустить данную позицию.

Производится проверка элементов списка структур Table. Пустые ячейки фиксируются счётчиком. Как только попадается занятый кластер, при ненулевом счётчике, запускается проверка, является ли кластер первым в цепочке.

// 6.2

Если да, вызывается функция moveFirstCluster(), которая переносит кластер на количество, зафиксированное в счётчике, и новая позиция перезаписанного кластера вносится в запись файла в корневой директории.

// 6.3

Если кластер не является первым в цепочке, вызывается функция moveCluster(), которая перемещает кластер и вносит новое положение кластера в предшествующую в цепочке ячейку таблицы FAT, указывающую на старый номер кластера.

Далее, изменившиеся значения фиксируются в списке структур Table и списке структур DriveFile. Счётчик цикла for уменьшается на количество зарегистрированных пустых ячеек, а счётчик пустых ячеек сбрасывается.

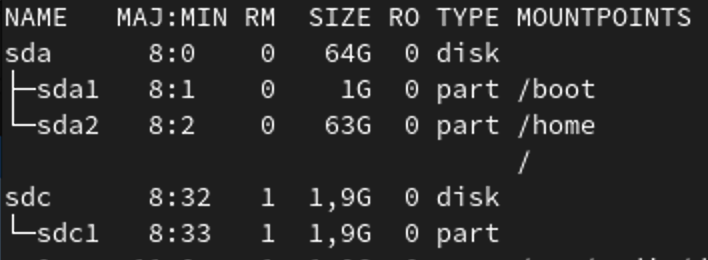
По окончании работы функции defragmentFreeSpace(), свободная память будет дефрагментирована.

// 7

В конце происходит вывод количества фрагментов свободной памяти до дефрагментации и после, а затем происходит очистка памяти, которая была выделена под списки, и закрытие потока файла устройства. Программа завершается.

Результат работы программы

Чтобы программа получила доступ к накопителю, необходимо размонтировать его и указать путь к его специальному файлу. Узнать название файла можно с помощью команды lsblk.



Выполнив команду, находим требуемое обозначение раздела флеш накопителя.

Для сборки программы, можно ввести две простые команды:

gcc -c \*.c

- данная команда создаёт объектные файлы из всех найденных в каталоге исходных кодов.

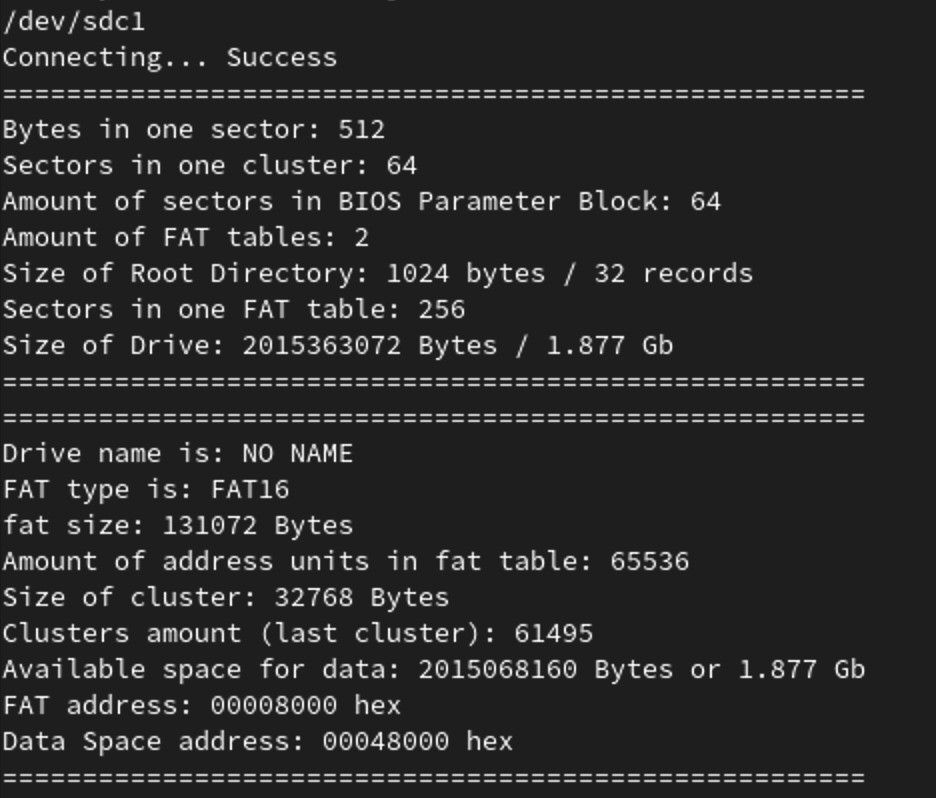
gcc -o main \*.o

- данная команда создаёт исполняемый файл main из всех найденных объектных файлов.

Теперь, запустим программу.

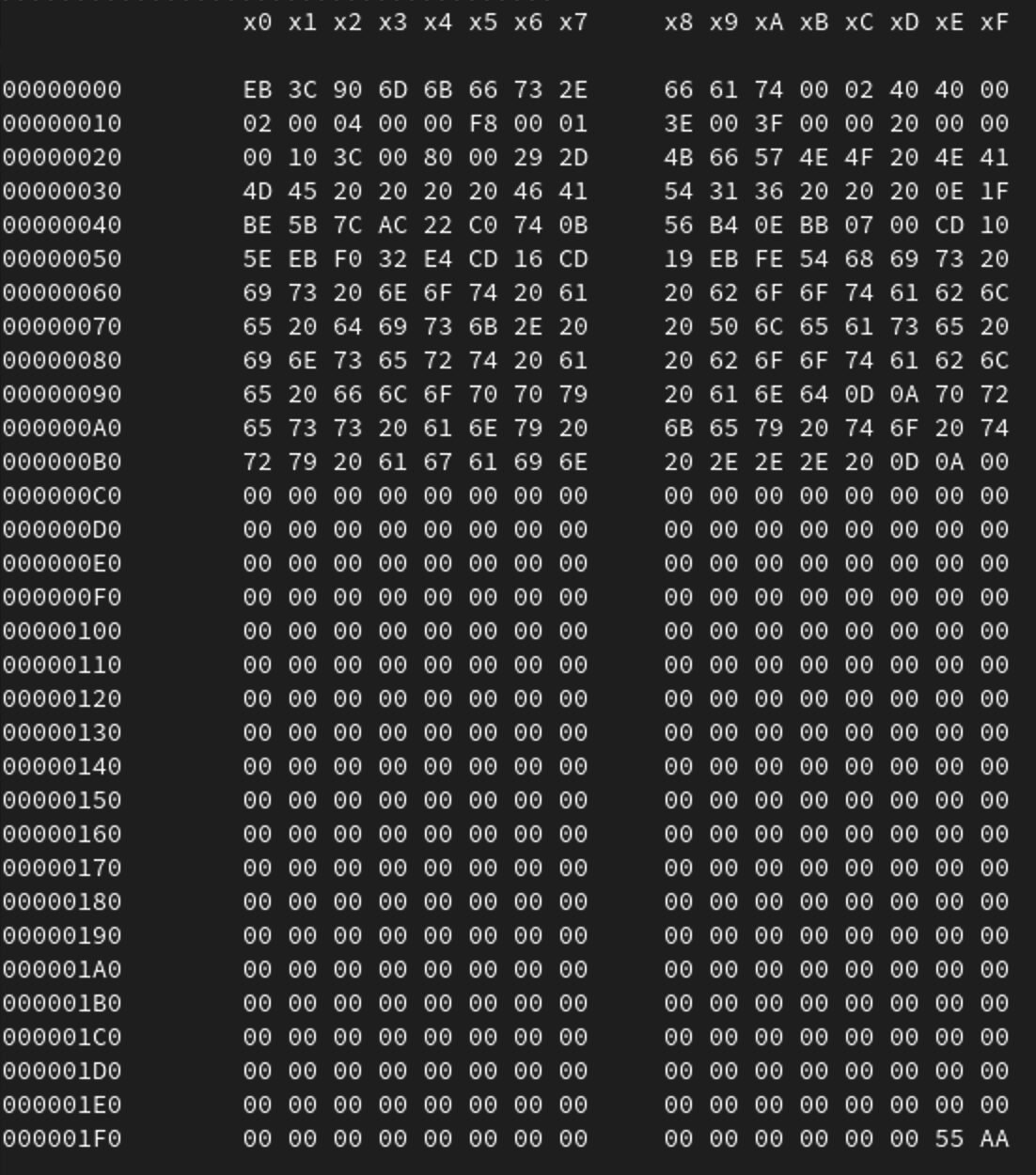
Вводим команду:

sudo ./main /dev/sdc1

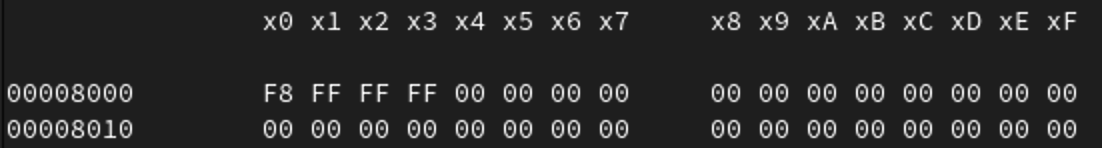


Программа сообщает об успешном подключении и выводит на экран отчёт о сведениях файловой системы.

Сделаем скан некоторых разделов.



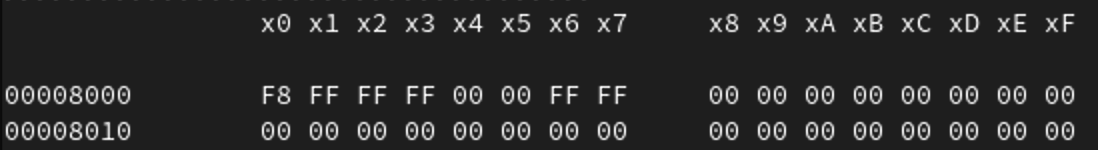
На скриншоте выше показана загрузочная запись используемого накопителя. Зная смещения нужных данных, можно получить полезные сведения о файловой системе. Такие сведения отображены на предыдущем скриншоте.



Выше представлен скан первых байтов таблицы FAT. Здесь можно увидеть, что на устройстве отсутствуют записи.

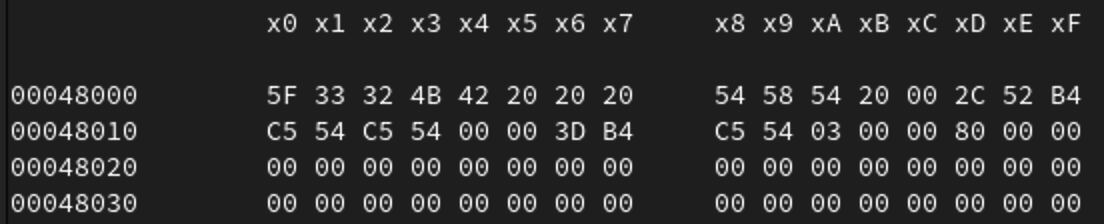
Эксперимент 1.

Создам файл со случайными данными, размером с один кластер, и помещу его на накопитель.



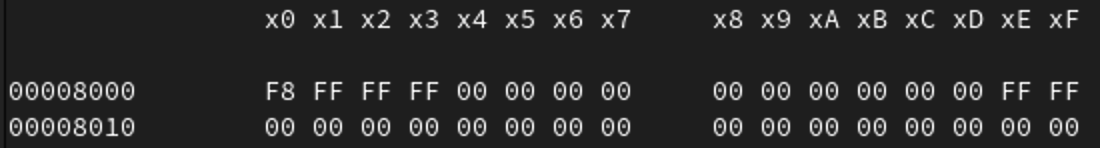
Как можно увидеть из снимка таблицы FAT, ячейка третьего кластера занята кодом FFFF, что говорит о том, что там записаны данные и что этот кластер является последним.

Сделаем скан корневой директории.

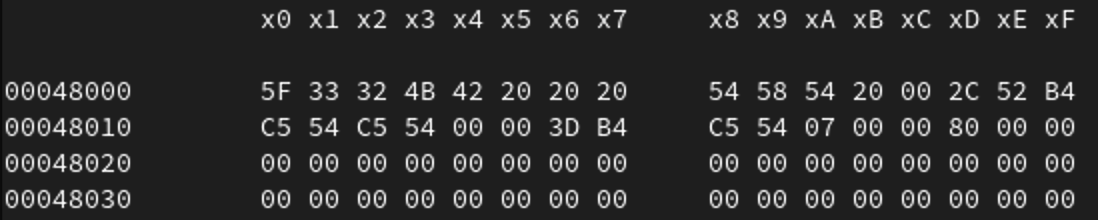


В корневой директории появилась запись. Из неё можно извлечь имя файла, его тип, размер и номер первого кластера, который совпадает с занятой ячейкой в таблице FAT.

Теперь, с помощью функции moveFirstCluster(), передвину вручную кластер данного файла вправо на 7 ячейку (смещение E).



Как можно заметить, ячейка третьего кластера теперь свободна, однако ячейка 7 кластера по смещению 800E занята кодом FFFF. Посмотрим запись в корневой директории.



Здесь видно, что номер первого кластера изменился с 3 на 7.

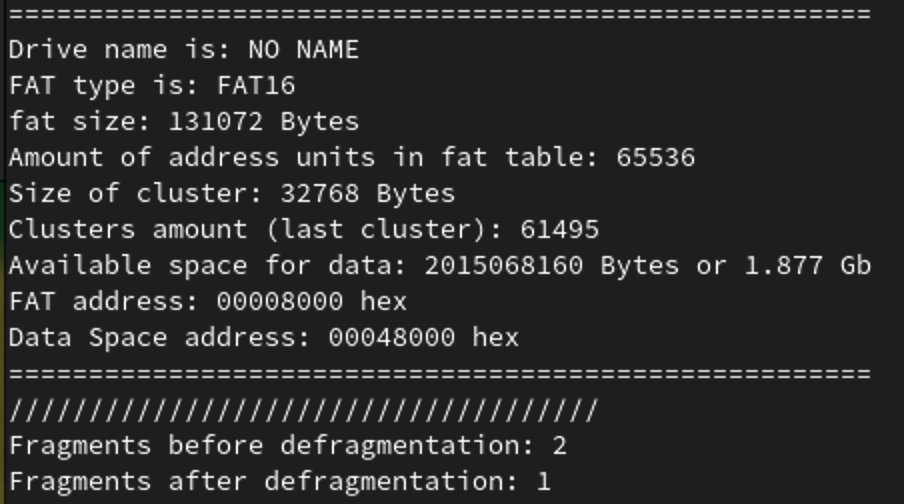
Чтобы убедиться, что функция сработала корректно, попробуем открыть файл и проверим на целостность данных.



Данные отображаются корректно, что говорит об успешном применении функции.

Размонтируем накопитель и продолжим испытания.

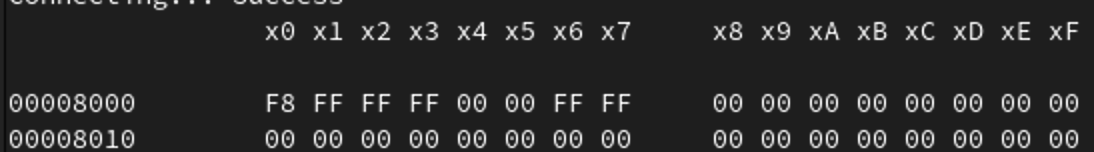
Теперь, попробуем использовать функцию дефрагментации и посмотрим на результат.

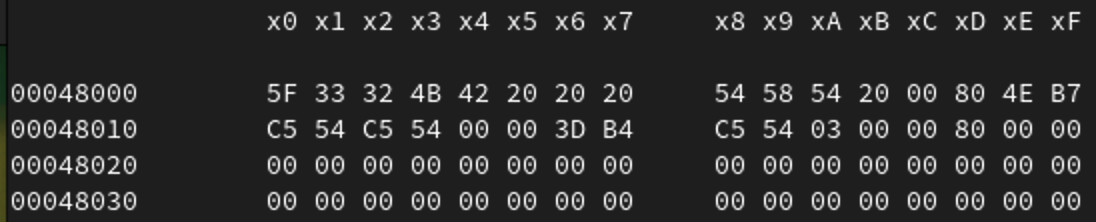


По выполнении программы, в терминал выводится сообщение и количестве фрагментов свободной памяти до дефрагментации и после.

Как можно видеть, фрагментация была устранена.

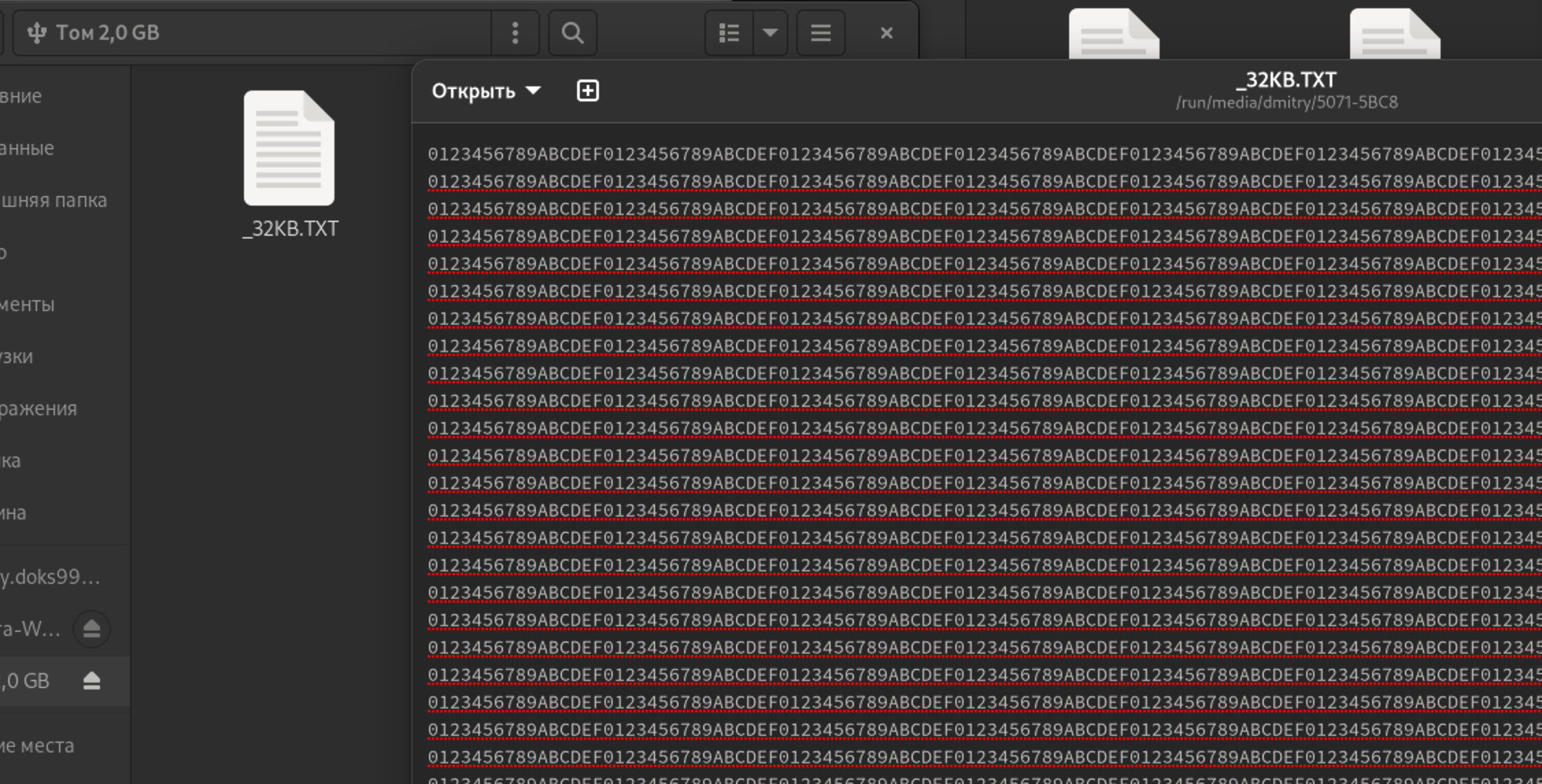
Посмотрим на снимки таблицы FAT и корневой директории.





Файл вернулся на своё место, данные скорректированы как в таблице, так и в корневой директории.

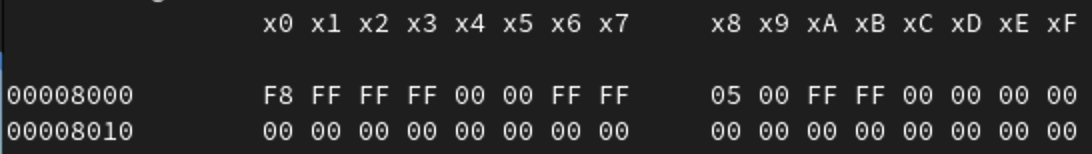
Проверим, откроется ли файл теперь.



Файл так же открывается, данные отображаются. Выполнение основной задачи успешно.

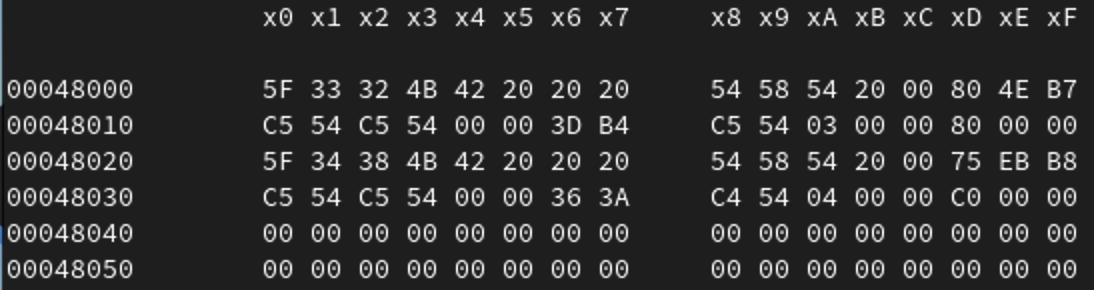
Эксперимент 2.

Создам файл, размером более одного кластера и помещу его на накопитель.



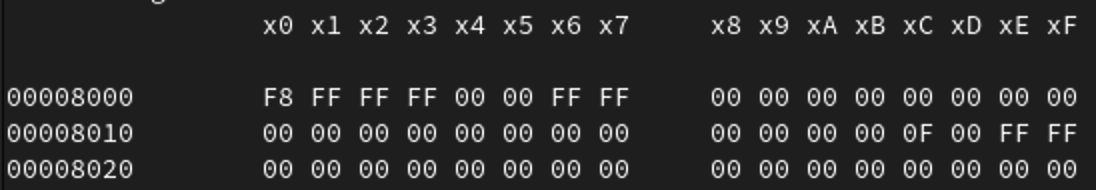
В таблице FAT видно, что в ячейках 4 и 5 появились данные. Значение в ячейке 4 указывает на номер следующего кластера в цепочке кластеров файла.

Взглянем на корневую директорию.

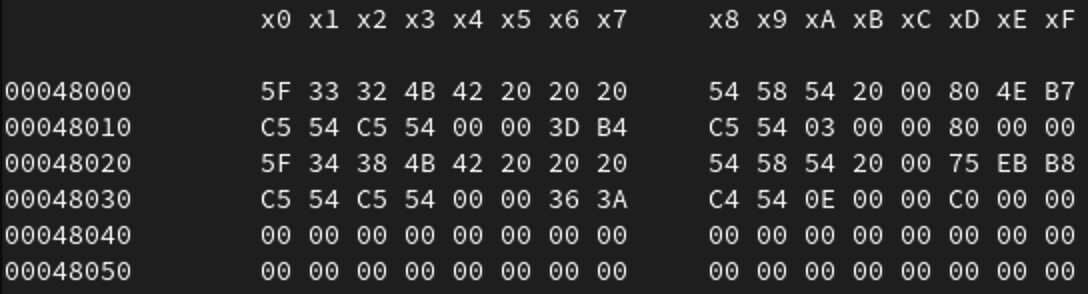


Здесь так же появилась новая запись, в которой номер первого кластера указан 4, а размер составляет C000=49152 байт=48килобайт и превышает размер кластера в 32 килобайт, что совпадает с данными из таблицы FAT.

Вручную, с помощью функций moveFirstCluster() и moveCluster(), передвину кластеры, относящиеся к данному файлу.

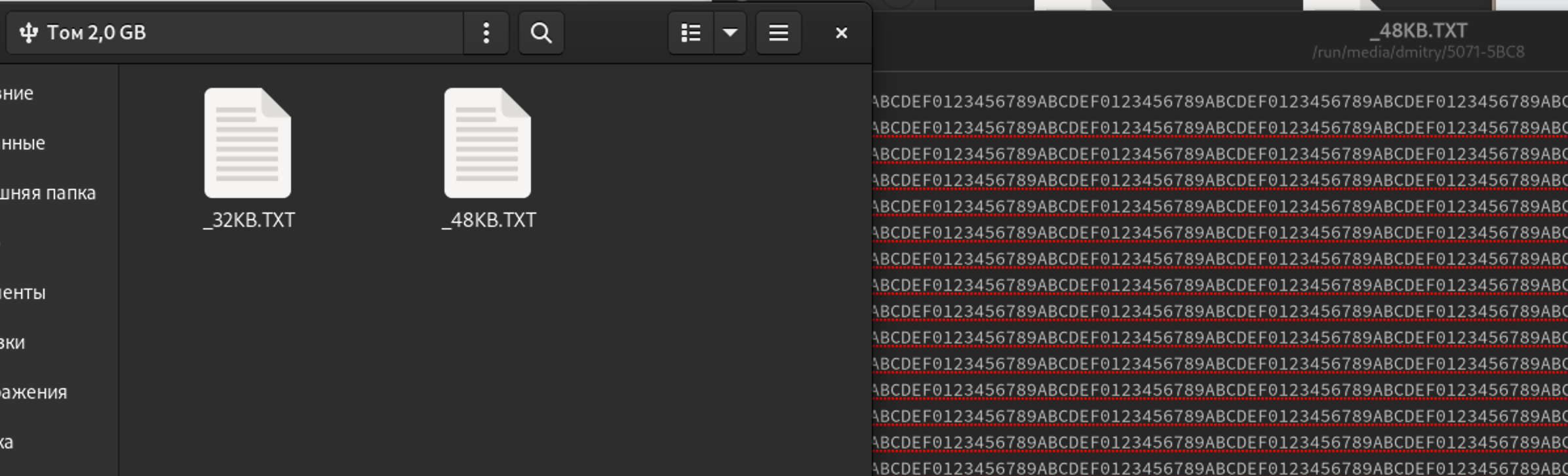


По таблице видно, что кластеры смещены на 10 ячеек. Посмотрим, как изменились данные в корневой директории.



В корневой директории изменения также отражены.

Проверим файл на запуск.

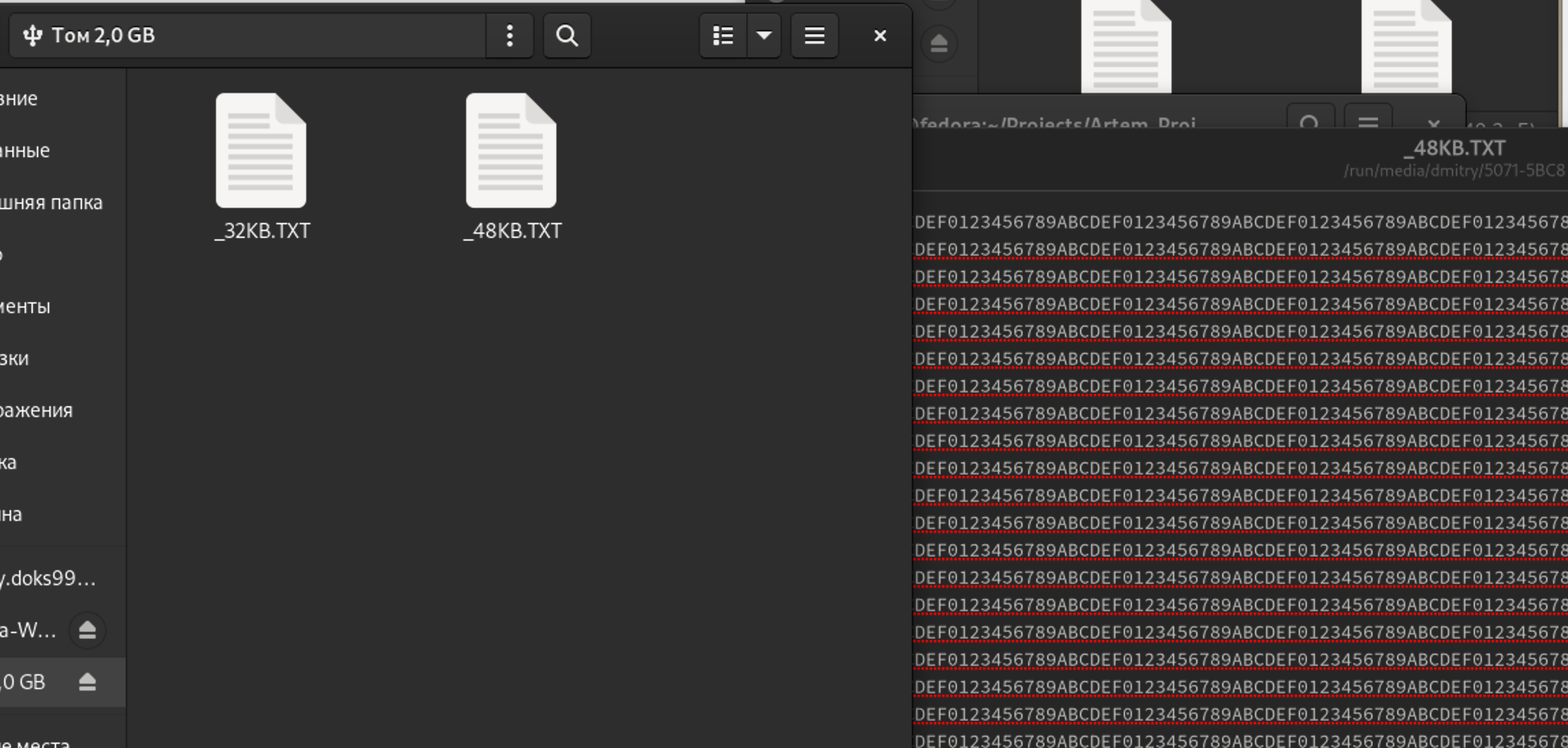


Данные отображаются корректно.

Запустим процедуру дефрагментации и посмотрим результат.



Программа сообщает, что до дефрагментации было 2 фрагмента свободной памяти, а после остался один. Проверим файл на открытие.

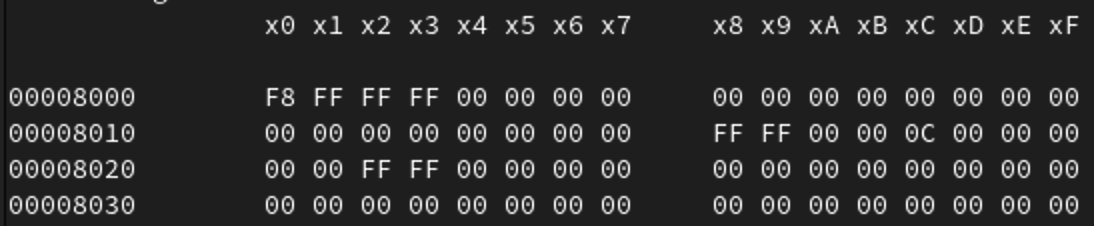


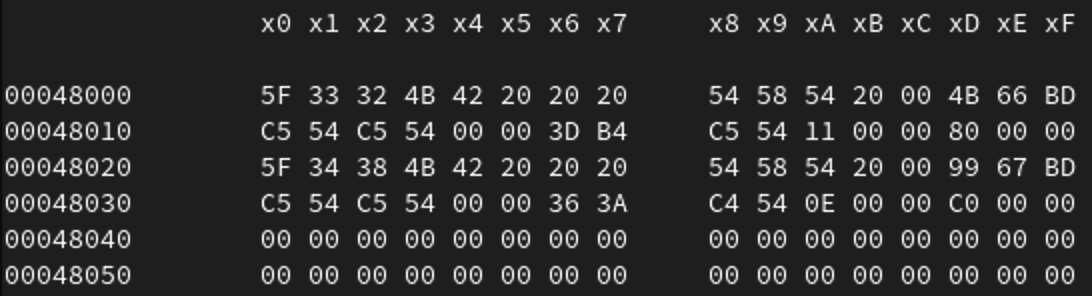
Файл открывается корректно.

Выполнение программы успешно.

Эксперимент 3.

Возьмём имеющиеся файлы и разнесём их кластеры по некоторой области данных.

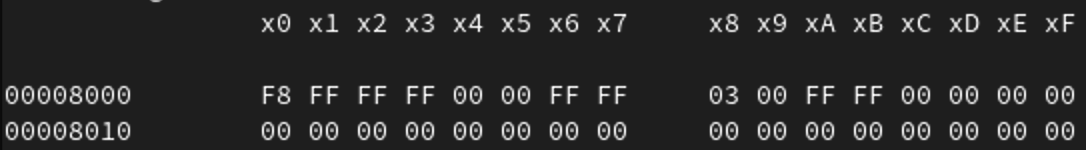


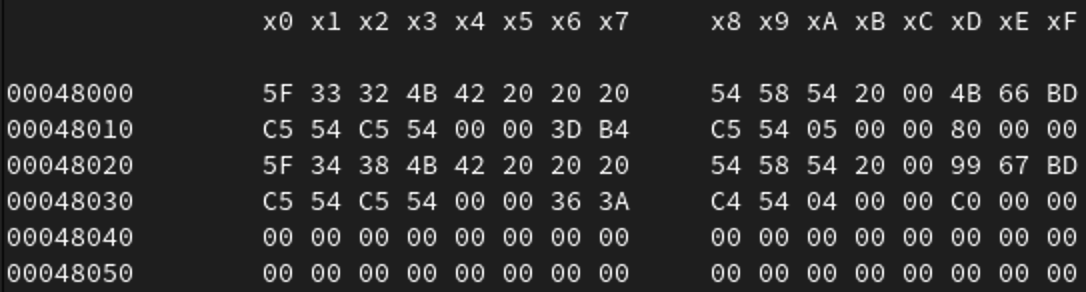


Кластеры случайным образом были перезаписаны. Теперь запустим дефрагментацию и посмотрим на результат.



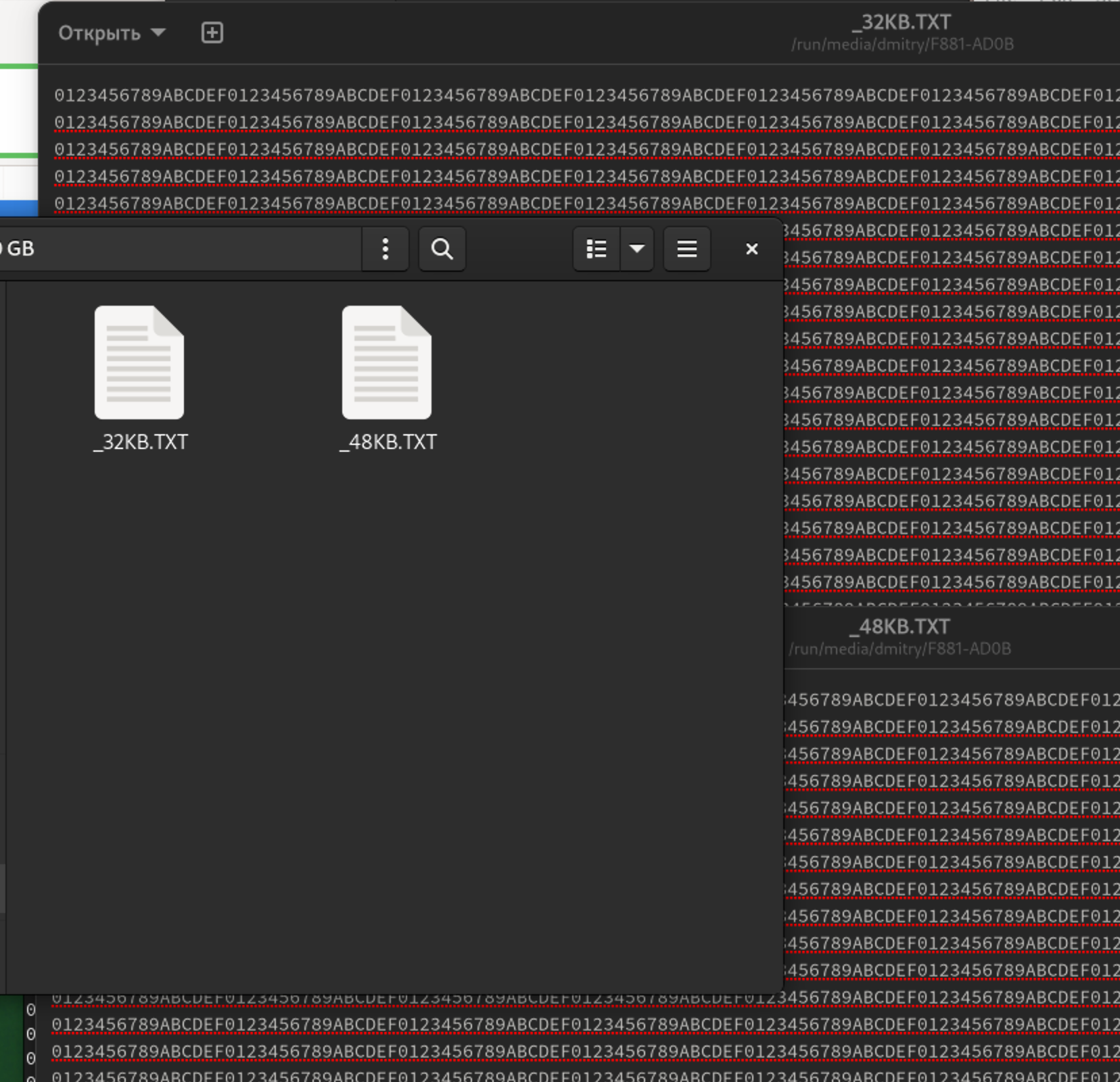
Программа сообщает, что фрагменты были устранены. Оценим результат.





Перемещение кластеров привело к фрагментации не только свободной памяти, но и данных файлов. При работе данного алгоритма фрагментация файлов не устранятся, однако фрагментация свободной памяти устраняется в различных случаях.

Проверим открытие файлов.



И, на удивление, они всё ещё открываются. Даже после фрагментации их данных.

Вывод

В ходе данной работы, я изучил архитектуру файловой системы FAT16; узнал, каким образом данные хранятся на накопителе с данной системой; научился читать сведения о томе из байтов зарезервированной области; увидел, каким образом сведения о файлах содержатся в корневой директории. Также, я разработал алгоритм дефрагментации свободного пространства и реализовал его в своём консольном приложении. По результатам работы этого приложения, я подтвердил успешность работы программы и использованного подхода к решению.

Листинг

my\_header.h

#ifndef MY\_HEADER\_H

#define MY\_HEADER\_H

// Структура с данными из загрузочного сектора ФС ФАТ16

**typedef** **struct** bpbSector {

// Число байт в секторе

// Допустимые значения - 512, 1024, 2048 и 4096

**unsigned** **short** bytesInSector; // address - 0x0B // size - 2 bytes

// Количество секторов на кластер

// допустимо 2, 4, 8, 16, 32 (также 64, поддерживается не во всех системах)

**unsigned** **short** sectorsInCluster; // address - 0x0D // size - 1 byte

// Количество зарезервированных секторов под загрузочный раздел

**unsigned** **short** sectorsInBPB; // address - 0x0E // size - 2 bytes

// количество таблиц ФАТ

**unsigned** **short** amountOfFAT; // address - 0x10 // size - 1 byte

// Размер корневой директории

**unsigned** **short** sizeOfRootDir; // address - 0x11 // size - 2 bytes

// Количество секторов в одной таблице ФАТ

// равно нулю, если это FAT32

**unsigned** **short** sectorsInFAT; // address - 0x16 // size - 2 bytes

// размер накопителя, указывается в секторах

**unsigned** sizeOfDrive; // адрес - 0x20 // size - 4 bytes

} BPBSector;

**typedef** **struct** driveData {

//BPBSector bpb;

**unsigned** **short** \* fat\_table;

**unsigned** **long** fat\_size;

**unsigned** **short** amountOfFAT;

**unsigned** **long** fat\_address;

**unsigned** **long** dataSpace\_address;

**unsigned** sizeOfCluster;

**unsigned** **long** lastCluster;

FILE \*file;

} Drive;

**typedef** **struct** array {

**void** \*ptr;

**unsigned** size;

} Array;

**typedef** **struct** driveFile {

**unsigned** **long** address;

**unsigned** **short** first\_cluster;

} DriveFile;

**typedef** **struct** table {

**unsigned** **long** occupied;

**unsigned** **long** first;

**unsigned** **short** prev;

} Table;

// READERS.C /////////////////////////////////////////////////////////////////////////////

**unsigned** **long** readBytesAsInt(FILE \*file, size\_t size, **unsigned** **long** address);

**char** \* readBytesAsString(FILE \*file, size\_t size, **unsigned** **long** address);

**unsigned** **short** \* readBytesAsUShort(FILE \*file, **unsigned** amount, **unsigned** **long** address);

///////////////////////////////////////////////////////////////////////////// READERS.C //

// GETTERS.C /////////////////////////////////////////////////////////////////////////////

BPBSector getBPB(FILE \*file);

Drive getDrive(FILE \*file, BPBSector bpb);

Array getFiles(Drive drive);

Table \* getTable(Drive drive);

///////////////////////////////////////////////////////////////////////////// GETTERS.C //

// PRINTERS.C ////////////////////////////////////////////////////////////////////////////

**void** printBPB(BPBSector bpb);

**void** printDrive(Drive drive);

**void** printFilesList(DriveFile \*list, **unsigned** size);

**void** printSectors(FILE \*file, **unsigned** **short** sectors, **unsigned** **long** **int** offset);

**void** printTable(Table \*table, **unsigned** amount);

//////////////////////////////////////////////////////////////////////////// PRINTERS.C //

// CHECKERS.C ////////////////////////////////////////////////////////////////////////////

**int** checkBPB(BPBSector bpb);

**unsigned** checkFreeSpace(Drive drive);

//////////////////////////////////////////////////////////////////////////// CHECKERS.C //

// MOVERS.C ////////////////////////////////////////////////////////////////////////////

**void** moveCluster(Drive drive, **unsigned** **short** prev\_cell, **unsigned** **short** nw\_cell);

**void** moveFirstCluster(Drive drive, DriveFile dfile, **unsigned** **short** new\_cell);

//////////////////////////////////////////////////////////////////////////// MOVERS.C //

// MARKTABLE.C ////////////////////////////////////////////////////////////////////////////

Table \* markTable(Drive drive, Table \*table, DriveFile \*list, **unsigned** list\_size);

//////////////////////////////////////////////////////////////////////////// MARKTABLE.C //

// DEFRAGFREESPACE.C /////////////////////////////////////////////////////////////////////////

**void** defragmentFreeSpace(Drive drive, Table \*table, DriveFile \*list, **unsigned** list\_size);

///////////////////////////////////////////////////////////////////////// DEFRAGFREESPACE.C //

#endif /\* MY\_HEADER\_H \*/

main.c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <limits.h>

#include "my\_header.h"

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

// 1

**if** (argc < 2) **return** 0;

**char** \* string;

string = \*(argv + 1);

printf("%s\n", string);

FILE \*file;

printf("Connecting... ");

**if** (!(file = fopen(string, "rb+"))) {

perror("Invalid path\n");

exit(1);

} **else** {

printf("Success\n");

}

// 2

BPBSector bpb = getBPB(file);

**if** (checkBPB(bpb) != 0) {

printf("wrong path\n");

**return** 0;

}

printBPB(bpb);

// 3

Drive drive = getDrive(file, bpb);

printDrive(drive);

// 4

Array array = getFiles(drive);

DriveFile \*list = array.ptr;

**unsigned** list\_size = array.size;

// 5

Table \*table = getTable(drive);

table = markTable(drive, table, list, list\_size);

// 6

defragmentFreeSpace(drive, table, list, list\_size);

printf("/////////////////////////////////////\n");

// 7

printf("Fragments before defragmentation: %d\n", checkFreeSpace(drive));

free(drive.fat\_table);

drive.fat\_table = readBytesAsUShort(file, drive.fat\_size, drive.fat\_address);

printf("Fragments after defragmentation: %d\n", checkFreeSpace(drive));

free(drive.fat\_table);

free(table);

free(list);

fclose(file);

**return** 0;

}

defragFreeSpace.c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <limits.h>

#include "my\_header.h"

**void** defragmentFreeSpace(Drive drive, Table \*table, DriveFile \*list, **unsigned** list\_size) {

**unsigned** i = 0;

**unsigned** **short** new\_cell = 0;

**unsigned** zeros = 0;

// 6.1

**for** (i = 3; i < drive.fat\_size / 2; i++) {

**if** (i < 3) {

perror("Unpredicted behavior. i < 3\n");

exit(3);

}

**if** ((table + i)->occupied == 0) {

zeros++;

**continue**;

}

**if** (((table + i)->occupied != 0) && (zeros != 0)) {

DriveFile dfile = \*(list + (table + i)->occupied - 1);

/////////////////////////////////////////////////////////

new\_cell = i - zeros;

/////////////////////////////////////////////////////////

**if** ((table + i)->first != 0) {

// 6.2

**unsigned** **short** next = 0;

fseek(drive.file, drive.fat\_address + dfile.first\_cluster \* 2, SEEK\_SET);

fread(&next, 2, 1, drive.file);

/////////////////////////////////////////////////////////////////////////

moveFirstCluster(drive, \*(list + (table + i)->occupied - 1), new\_cell);

(list + (table + i)->occupied - 1)->first\_cluster = new\_cell;

/////////////////////////////////////////////////////////////////////////

**if** ((next != 0xFFFF) || (next != 0)) {

(table + next)->prev = new\_cell;

}

} **else** {

// 6.3

moveCluster(drive, (table + i)->prev, new\_cell);

(table + new\_cell)->prev = (table + i)->prev;

(table + i)->prev = 0;

}

(table + new\_cell)->occupied = (table + i)->occupied;

(table + i)->occupied = 0;

(table + new\_cell)->first = (table + i)->first;

(table + i)->first = 0;

i -= zeros;

zeros = 0;

} // if()

} // for()

} //defragmentFreeSpace()

movers.c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <limits.h>

#include "my\_header.h"

**void** moveCluster(Drive drive, **unsigned** **short** prev\_cell, **unsigned** **short** nw\_cell) {

**unsigned** **char** \*cluster = (**unsigned** **char**\*) calloc(drive.sizeOfCluster, **sizeof**(**unsigned** **char**));

**unsigned** **long** offset = 0;

**unsigned** **short** new\_cell = 0, current\_cell = 0, previous\_cell = 0, next\_cell = 0, zero = 0;

**unsigned** i = 0;

previous\_cell = prev\_cell; // должно быть указано

fseek(drive.file, drive.fat\_address + previous\_cell \* 2, SEEK\_SET);

fread(&current\_cell, 2, 1, drive.file);

printf("MC currect cluster = %04X\n", current\_cell);

new\_cell = nw\_cell; // должно быть указано

fseek(drive.file, drive.fat\_address + current\_cell \* 2, SEEK\_SET);

fread(&next\_cell, 2, 1, drive.file);

printf("previous\_cell = %d\n", previous\_cell);

printf("current\_cell = %d\n", current\_cell);

printf("new\_cell = %d\n", new\_cell);

printf("next\_cell = %d\n", next\_cell);

// перемещение кластера в области данных

// по смещению читается кластер, который будет перемещён

offset = drive.dataSpace\_address + ((current\_cell - 1) \* drive.sizeOfCluster);

fseek(drive.file, offset, SEEK\_SET);

fread(cluster, 1, drive.sizeOfCluster, drive.file);

// данные записываются в новый указанный кластер

offset = drive.dataSpace\_address + ((new\_cell - 1) \* drive.sizeOfCluster);

fseek(drive.file, offset, SEEK\_SET);

fwrite(cluster, 1, drive.sizeOfCluster, drive.file);

fflush(drive.file);

free(cluster);

// перезапись таблиц FAT

**for** (i = 0; i < drive.amountOfFAT; i++) {

// ячейка в фат таблице, указывающая на перемещённый кластер, перезаписывается

offset = drive.fat\_address + (previous\_cell \* 2) + (drive.fat\_size \* i);

fseek(drive.file, offset, SEEK\_SET);

fwrite(&new\_cell, **sizeof**(**unsigned** **short**), 1, drive.file);

// номер следующего в цепочке кластера записывается в новую ячейку

offset = drive.fat\_address + (new\_cell \* 2) + (drive.fat\_size \* i);

fseek(drive.file, offset, SEEK\_SET);

fwrite(&next\_cell, **sizeof**(**unsigned** **short**), 1, drive.file);

// зануление старой ячейки в таблице FAT

offset = drive.fat\_address + (current\_cell \* 2) + (drive.fat\_size \* i);

//printf("offset: %08X\n", offset);

fseek(drive.file, offset, SEEK\_SET);

fwrite(&zero, **sizeof**(**unsigned** **short**), 1, drive.file);

}

}

**void** moveFirstCluster(Drive drive, DriveFile dfile, **unsigned** **short** new\_cell) {

**if** (dfile.first\_cluster == new\_cell) **return**;

**unsigned** **char** \*cluster = (**unsigned** **char**\*) calloc(drive.sizeOfCluster, **sizeof**(**unsigned** **char**));

**unsigned** **long** address = 0;

**unsigned** **short** next\_cell = 0, zero = 0;

**unsigned** i = 0;

fseek(drive.file, drive.fat\_address + dfile.first\_cluster \* 2, SEEK\_SET);

fread(&next\_cell, 2, 1, drive.file);

// перемещение кластера в области данных

// по смещению читается кластер, который будет перемещён

address = drive.dataSpace\_address + (dfile.first\_cluster - 1) \* drive.sizeOfCluster;

fseek(drive.file, address, SEEK\_SET);

fread(cluster, 1, drive.sizeOfCluster, drive.file);

// данные записываются в новый указанный кластер

address = drive.dataSpace\_address + (new\_cell - 1) \* drive.sizeOfCluster;

fseek(drive.file, address, SEEK\_SET);

fwrite(cluster, 1, drive.sizeOfCluster, drive.file);

fflush(drive.file);

// перезапись первого кластера в 32 байтовой записи файла в корневой директории

fseek(drive.file, dfile.address + 0x1A, SEEK\_SET);

fwrite(&new\_cell, 2, 1, drive.file);

fflush(drive.file);

free(cluster);

// перезапись таблиц FAT

**for** (i = 0; i < drive.amountOfFAT; i++) {

// номер следующего в цепочке кластера (или 0xFFFF) записывается в новую ячейку

address = drive.fat\_address + (new\_cell \* 2) + (drive.fat\_size \* i);

fseek(drive.file, address, SEEK\_SET);

fwrite(&next\_cell, **sizeof**(**unsigned** **short**), 1, drive.file);

fflush(drive.file);

// зануление старой ячейки в таблице FAT

address = drive.fat\_address + (dfile.first\_cluster \* 2) + (drive.fat\_size \* i);

fseek(drive.file, address, SEEK\_SET);

fwrite(&zero, **sizeof**(**unsigned** **short**), 1, drive.file);

fflush(drive.file);

}

}

readers.c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <limits.h>

#include "my\_header.h"

**unsigned** **long** readBytesAsInt(FILE \*file, size\_t size, **unsigned** **long** address) {

**unsigned** **long** variable = 0;

fseek(file, address, SEEK\_SET);

fread(&variable, size, 1, file);

**return** variable;

}

**char** \* readBytesAsString(FILE \*file, size\_t size, **unsigned** **long** address) {

**char** \*string = (**char**\*) calloc(size + 1, **sizeof**(**char**));

**char** l;

**int** i = 0;

fseek(file, address, SEEK\_SET);

**for** (i = 0; i < size; i++) {

fread(&l, 1, 1, file);

\*(string + i) = ((l >= 0x20) && (l <= 0x7E)) ? l : '\_';

}

\*(string + i) = '\0';

**return** string;

}

**unsigned** **short** \* readBytesAsUShort(FILE \*file, **unsigned** amount, **unsigned** **long** address) {

**unsigned** **short** \* array = (**unsigned** **short** \*) calloc(amount / 2, **sizeof**(**unsigned** **short**));

fseek(file, address, SEEK\_SET);

fread(array, **sizeof**(**unsigned** **short**), amount / 2, file);

**return** array;

}

getters.c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <limits.h>

#include "my\_header.h"

BPBSector getBPB(FILE \*file) {

BPBSector bpb;

bpb.bytesInSector = readBytesAsInt(file, 2, 0x0B);

bpb.sectorsInCluster = readBytesAsInt(file, 1, 0x0D);

bpb.sectorsInBPB = readBytesAsInt(file, 2, 0x0E);

bpb.amountOfFAT = readBytesAsInt(file, 1, 0x10);

bpb.sizeOfRootDir = readBytesAsInt(file, 2, 0x11);

bpb.sectorsInFAT = readBytesAsInt(file, 2, 0x16);

bpb.sizeOfDrive = readBytesAsInt(file, 4, 0x20) \* bpb.bytesInSector;

**return** bpb;

}

Drive getDrive(FILE \*file, BPBSector bpb) {

Drive drive;

drive.fat\_size = bpb.sectorsInFAT \* bpb.bytesInSector;

drive.amountOfFAT = bpb.amountOfFAT;

drive.fat\_address = bpb.bytesInSector \* bpb.sectorsInBPB;

drive.dataSpace\_address = bpb.bytesInSector \* (bpb.sectorsInBPB

+ bpb.sectorsInFAT \* bpb.amountOfFAT);

drive.sizeOfCluster = bpb.bytesInSector \* bpb.sectorsInCluster;

drive.lastCluster = (bpb.sizeOfDrive - drive.dataSpace\_address)

/ drive.sizeOfCluster;

drive.fat\_table = readBytesAsUShort(file, drive.fat\_size, drive.fat\_address);

drive.file = file;

**return** drive;

}

Array getFiles(Drive drive) {

**unsigned** i = 0, size = 0;

**unsigned** **long** var = 0, address = 0;

**unsigned** **short** attr = 0, frstCl = 0;

**for** (i = 0; i < drive.sizeOfCluster; i += 0x20) {

address = drive.dataSpace\_address + i;

// читаем первый байт 32 байтовой записи файла

fseek(drive.file, address, SEEK\_SET);

fread(&var, 1, 1, drive.file);

// если байт равен нулю - дальше записей нет, завершаем цикл

**if** (var == 0) **break**;

//если у файла метка удалённого файла - пропустить

**if** (var == 0xE5 || var == 0x05) **continue**;

// читаем атрибут файла

fseek(drive.file, address + 0x0B, SEEK\_SET);

fread(&attr, 1, 1, drive.file);

// если атрибут файла не соответствует атрибуту архивного файла - пропустить

**if** ((attr & 0x20) != 0x20) **continue**;

// читаем первый кластер файла

fseek(drive.file, address + 0x1A, SEEK\_SET);

fread(&frstCl, 2, 1, drive.file);

// если номер первого кластера не указан - пропускаем

**if** (frstCl == 0) **continue**;

// main block //////////////////////////////////////////////////////////

size++;

////////////////////////////////////////////////////////// main block //

}

DriveFile \*list = (DriveFile\*)calloc(size, **sizeof**(DriveFile));

**unsigned** **short** flag = 0;

**for** (i = 0; i < drive.sizeOfCluster; i += 0x20) {

address = drive.dataSpace\_address + i;

// читаем первый байт 32 байтовой записи файла

fseek(drive.file, address, SEEK\_SET);

fread(&var, 1, 1, drive.file);

// если байт равен нулю - дальше записей нет, завершаем цикл

**if** (var == 0) **break**;

//если у файла метка удалённого файла - пропустить

**if** (var == 0xE5 || var == 0x05) **continue**;

// читаем атрибут файла

fseek(drive.file, address + 0x0B, SEEK\_SET);

fread(&attr, 1, 1, drive.file);

// если атрибут файла не соответствует атрибуту архивного файла - пропустить

**if** ((attr & 0x20) != 0x20) **continue**;

// читаем первый кластер файла

fseek(drive.file, address + 0x1A, SEEK\_SET);

fread(&frstCl, 2, 1, drive.file);

// если номер первого кластера не указан - пропускаем

**if** (frstCl == 0) **continue**;

// main block //////////////////////////////////////////////////////////

// создаём экзмпляр структуры, хранящей первый кластер файла и адрес

// его записи в корневой директории

DriveFile dfile;

dfile.address = address;

dfile.first\_cluster = frstCl;

// добавляем экземпляр к списку

\*(list + flag++) = dfile;

**if** (flag == size) **break**;

////////////////////////////////////////////////////////// main block //

}

Array array;

array.ptr = list;

array.size = size;

**return** array;

}

Table \* getTable(Drive drive) {

**unsigned** i = 0;

Table \*table = (Table\*) calloc(drive.fat\_size / 2, **sizeof**(Table));

**for** (i = 0; i < drive.fat\_size / 2; i++) {

(table + i)->occupied = 0;

(table + i)->first = 0;

(table + i)->prev = 0;

}

**return** table;

}

checkers.c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <limits.h>

#include "my\_header.h"

**int** checkBPB(BPBSector bpb) {

**if** (bpb.sectorsInFAT != 0

&& bpb.sizeOfRootDir != 0

&& (bpb.bytesInSector == 512

|| bpb.bytesInSector == 1024

|| bpb.bytesInSector == 2048

|| bpb.bytesInSector == 4096)

&& (bpb.sectorsInCluster == 2

|| bpb.sectorsInCluster == 4

|| bpb.sectorsInCluster == 8

|| bpb.sectorsInCluster == 16

|| bpb.sectorsInCluster == 32

|| bpb.sectorsInCluster == 64))

{

**return** 0;

} **else** {

**return** -1;

}

}

**unsigned** checkFreeSpace(Drive drive) {

**unsigned** i = 0;

**unsigned** fragments = 0;

**short** flag = 0;

**for** (i = 3; i < drive.fat\_size / 2; i++) {

**if** ((\*(drive.fat\_table + i) == 0)) {

flag = 1;

**continue**;

}

**if** ((\*(drive.fat\_table + i) != 0) && (flag == 1)) {

fragments += 1;

flag = 0;

}

}

**if** (flag == 1) fragments += 1;

**return** fragments;

}

printers.c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <limits.h>

#include "my\_header.h"

**void** printBPB(BPBSector bpb) {

printf("======================================================\n");

printf("Bytes in one sector: %d\n", bpb.bytesInSector);

printf("Sectors in one cluster: %d\n", bpb.sectorsInCluster);

printf("Amount of sectors in BIOS Parameter Block: %d\n", bpb.sectorsInBPB);

printf("Amount of FAT tables: %d\n", bpb.amountOfFAT);

printf("Size of Root Directory: %d bytes / %d records\n", bpb.sizeOfRootDir, bpb.sizeOfRootDir/32);

printf("Sectors in one FAT table: %d\n", bpb.sectorsInFAT);

printf("Size of Drive: %d Bytes / %.3lf Gb\n", bpb.sizeOfDrive, (**double**) bpb.sizeOfDrive / (1024\*1024\*1024));

printf("======================================================\n");

}

**void** printDrive(Drive drive) {

printf("======================================================\n");

**char** \* fat\_type = readBytesAsString(drive.file, 8, 0x36);

**char** \*drive\_name = readBytesAsString(drive.file, 11, 0x2B);

printf("Drive name is: %s\n", drive\_name);

printf("FAT type is: %s\n", fat\_type);

printf("fat size: %d Bytes\n", drive.fat\_size);

printf("Amount of address units in fat table: %d\n", drive.fat\_size / 2);

printf("Size of cluster: %d Bytes\n", drive.sizeOfCluster);

printf("Clusters amount (last cluster): %d\n", drive.lastCluster);

**unsigned** **long** av\_space = drive.sizeOfCluster \* drive.lastCluster;

printf("Available space for data: %d Bytes or %.3lf Gb\n", av\_space, (**double**) av\_space / (1024 \* 1024 \* 1024));

printf("FAT address: %08X hex\n", drive.fat\_address);

printf("Data Space address: %08X hex\n", drive.dataSpace\_address);

printf("======================================================\n");

free(fat\_type);

free(drive\_name);

}

**void** printFilesList(DriveFile \*list, **unsigned** size) {

**unsigned** i = 0;

**for** (i = 0; i < size; i++) {

printf("File %d. Record address: %08X / first cluster: %04X\n",

i + 1, (list + i)->address, (list + i)->first\_cluster);

}

}

// функция отображения данных секторами, по смещению - (file, amount of sectors, offset)

**void** printSectors(FILE \*file, **unsigned** **short** sectors, **unsigned** **long** **int** offset) {

**unsigned** **char** a = 0; // буфер для байта

**int** i = 0, j = 0, k = 0; // счётчики

fseek(file, offset, SEEK\_SET); // возвращаем указатель в файле на начало

// здесь реализован красивый вывод

**for** (i = 0; i < sectors \* 0x20; i++) { // смещение по оси Y

**if** ((i % 512) == 0) {

**for** (j = 0; j < 16; j++) {

**if** (j == 0) printf("\t\t");

printf("x%01X ", j);

**if** (j == 7) printf(" ");

**if** (j == 0xF) printf("\n\n");

}

}

//printf("%d.\t", ++k);

printf("%08X\t", offset + 0x10 \* i);

**for** (j = 0; j < 0x10; j++) { // смещение по оси X

fread(&a, 1, 1, file); // Читаем байты из файла

**if** (a <= 0xF) { // Рисуем нолик для красоты, если байт меньше 16

printf ("0%X ", a);

} **else** {

printf("%X ", a);

}

**if** (j == 7) printf("%s", " "); // пропуск после каждого восьмого байта

}

printf("%s", "\n"); // переход на новую строку каждые 16 байт

//if (k % 2 == 0) printf("\n");

**if** ((i+1) % 0x20 == 0) printf("%s", "\n"); // делаем пропуск после конца каждого сектора

}

fseek(file, 0, SEEK\_SET); // возвращаем указатель в файле на начало

}

**void** printTable(Table \*table, **unsigned** amount) {

**unsigned** i = 0;

**for** (i = 0; i < amount; i++) {

printf("FAT\_N %d. occupied: %d / first: %d / prev: %04X\n", i, (table + i)->occupied, (table + i)->first, (table + i)->prev);

}

}

markTable.c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <limits.h>

#include "my\_header.h"

Table \* markTable(Drive drive, Table \*table, DriveFile \*list, **unsigned** list\_size) {

**unsigned** i = 0;

**unsigned** **short** current = 0, next = 0;

**for** (i = 0; i < list\_size; i++) {

(table + (list + i)->first\_cluster)->occupied = i + 1;

(table + (list + i)->first\_cluster)->first = i + 1;

current = (list + i)->first\_cluster;

next = \*(drive.fat\_table + current);

**while**(next != 0xFFFF) {

**if** (next == 0) {

perror("markTable: unpredicted behavior\n");

exit(2);

}

(table + next)->occupied = i + 1;

(table + next)->prev = current;

current = next;

next = \*(drive.fat\_table + current);

}

}

**return** table;

}