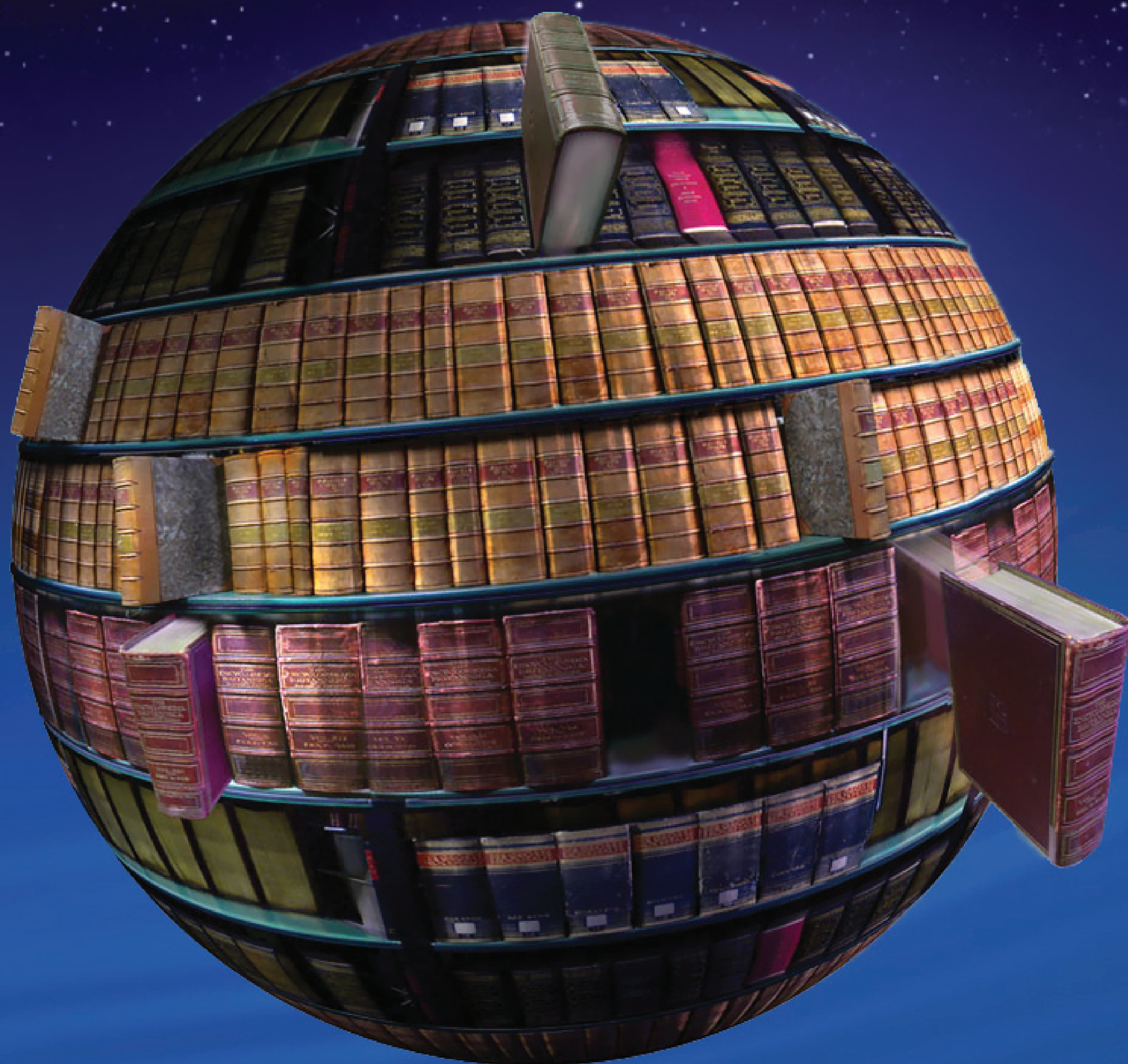
АРХИТЕКТУРА ФАЙЛОВОЙ СИСТЕМЫ FAT



ВЛАДИМИР МЕШКОВ

Общая характеристика файловой системы FAT. Структура раздела с файловой системой FAT

Файловая система FAT (File Allocation Table) была разработана Биллом Гейтсом и Марком Макдональдом в 1977 году и первоначально использовалась в операционной системе 86-DOS. Чтобы добиться переносимости программ из операционной системы CP/M в 86-DOS, в ней были сохранены ранее принятые ограничения на имена файлов. В дальнейшем 86-DOS была приобретена Microsoft и стала основой для ОС MS-DOS 1.0, выпущенной в августе 1981 года. FAT была предназначена для работы с гибкими дисками размером менее 1 Мб и вначале не предусматривала поддержки жёстких дисков. Структура раздела FAT изображена на рисунке.

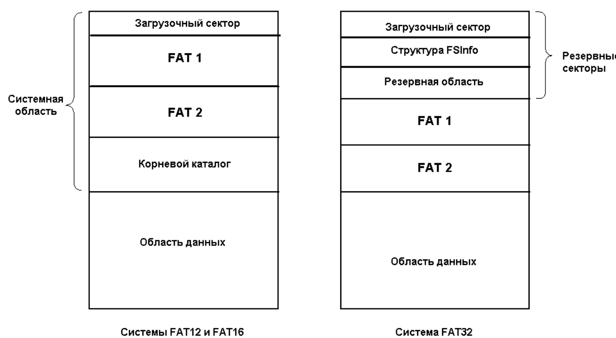


Рисунок 1. Структура раздела с файловой системой FAT

В файловой системе FAT дисковое пространство логического раздела делится на две области – системную и область данных (см. рис. 1). Системная область создается и инициализируется при форматировании, а впоследствии обновляется при манипулировании файловой структурой. Системная область файловых систем FAT состоит из следующих компонентов:

- загрузочная запись (boot record, BR);
- резервная область;
- таблицы размещения файлов;
- область корневого каталога (не существует в FAT32).

Область данных логического диска содержит файлы и каталоги, подчиненные корневому, и разделена на участки одинакового размера – кластеры. Кластер может состоять из одного или нескольких последовательно расположенных на диске секторов. Число секторов в кластере должно быть кратно 2^N и может принимать значения от 1 до 64. Размер кластера зависит от типа используемой файловой системы и объема логического диска.

Назначение, структура и типы таблицы размещения файлов

Своё название FAT получила от одноимённой таблицы размещения файлов – File Allocation Table, FAT. В таблице размещения файлов хранится информация о кластерах логического диска. Каждому кластеру соответствует элемент таблицы FAT, содержащий информацию о том, свободен данный кластер или занят данными файла. Если кластер занят под файл, то в соответствующем элементе таблицы размещения файлов указывается адрес кластера, содержащего следующую часть файла. Номер начального кластера, занятого файлом, хранится в элементе каталога, содержащего запись об этом файле. Последний элемент списка кластеров содержит признак конца файла (EOF – End Of File). Первые два элемента FAT являются резервными.

Файловая система FAT всегда заполняет свободное место на диске последовательно от начала к концу. При создании нового файла или увеличении уже существующего она ищет самый первый свободный кластер в таблице размещения файлов. Если в процессе работы одни файлы были удалены, а другие изменились в размере, то появляющиеся в результате пустые кластеры будут рассеяны по диску. Если кластеры, содержащие данные файла, расположены не подряд, то файл оказывается фрагментированным.

Существуют следующие типы FAT – FAT12, FAT16, FAT32. Названия типов FAT ведут свое происхождение от размера элемента: элемент FAT12 имеет размер 12 бит (1,5 байт), FAT16 – 16 бит (2 байта), FAT32 – 32 бита (4 байта). В FAT32 четыре старших двоичных разряда зарезервированы и игнорируются в процессе работы операционной системы.

Корневой каталог

За таблицами размещения файлов следует корневой каталог. Каждому файлу и подкаталогу в корневом каталоге соответствует 32-байтный элемент каталога (directory entry), содержащий имя файла, его атрибуты (архивный, скрытый, системный и «только для чтения»), дату и время создания (или внесения в него последних изменений), а также прочую информацию. Для файловых систем FAT12 и FAT16 положение корневого каталога на разделе и его размер жестко зафиксированы. В FAT32 корневой каталог может быть расположен в любом месте области данных раздела и иметь произвольный размер.

Форматы имен файлов

Одной из характеристик ранних версий FAT (FAT12 и FAT16) является использование коротких имен файлов. Короткое имя состоит из двух полей – 8-байтного поля, содержащего собственно имя файла, и 3-байтного поля, содержащего расширение (формат «8.3»). Если введенное пользователем имя файла короче 8 символов, то оно дополняется пробелами (код 0x20); если введенное расширение короче трёх байтов, то оно также дополняется пробелами.

Структура элемента каталога для короткого имени файла представлена в таблице 1.

Первый байт короткого имени выполняет функции признака занятости каталога:

- если первый байт равен 0xE5, то элемент каталога свободен и его можно использовать при создании нового файла;
- если первый байт равен 0x00, то элемент каталога свободен и является началом чистой области каталога (после него нет ни одного задействованного элемента).

Таблица 1. Структура элемента каталога для короткого имени файла

Смещение	Размер (байт)	Содержание
0x00	11	Короткое имя файла.
0x0B	1	Атрибуты файла.
0x0C	1	Зарезервировано для Windows NT. Поле обрабатывается только в FAT32.
0x0D	1	Поле, уточняющее время создания файла (содержит десятки миллисекунд). Поле обрабатывается только в FAT32.
0x0E	1	Время создания файла. Поле обрабатывается только в FAT32.
0x10	2	Дата создания файла. Поле обрабатывается только в FAT32.
0x12	2	Дата последнего обращения к файлу для записи или считывания данных. Поле обрабатывается только в FAT32.
0x14	2	Старшее слово номера первого кластера файла. Поле обрабатывается только в FAT32.
0x16	2	Время выполнения последней операции записи в файл.
0x18	2	Дата выполнения последней операции записи в файл.
0x1A	2	Младшее слово номера первого кластера файла.
0x1C	4	Размер файла в байтах.

На использование ASCII-символов в коротком имени накладывается ряд ограничений:

- нельзя использовать символы с кодами меньше 0x20 (за исключением кода 0x05 в первом байте короткого имени);
- нельзя использовать символы с кодами 0x22, 0x2A, 0x2B, 0x2C, 0x2E, 0x2F, 0x3A, 0x3B, 0x3C, 0x3D, 0x3E, 0x3F, 0x5B, 0x5C, 0x5D, 0x7C;
- нельзя использовать символ пробела (0x20) в первом байте имени.

В файловых системах FAT32 и VFAT (виртуальная FAT, расширение FAT16) включена поддержка длинных имен файлов (long file name, LFN). Для хранения длинного имени используются элементы каталога, смежные с основным элементом. Имя файла записывается не ASCII-символами, а в Unicode. В одном элементе каталога можно сохранить фрагмент длиной до 13 символов Unicode. Неиспользованный участок последнего фрагмента заполняется кодами 0xFFFF. Структура элемента каталога для длинного имени файла представлена в таблице 2.

Таблица 2. Структура элемента каталога для длинного имени файла

Смещение	Размер (байт)	Содержание
0x00	1	Номер фрагмента.
0x01	10	Символы 1-5 имени файла в Unicode.
0x0B	1	Атрибуты файла.
0x0C	1	Байт флагов.
0x0D	1	Контрольная сумма короткого имени.
0x0E	12	Символы 6-11 имени файла в Unicode.
0x1A	2	Номер первого кластера (заполняется нулями).
0x1C	4	Символы 12-13 имени файла в Unicode.

Длинное имя записывается в каталог первым, причем фрагменты размещены в обратном порядке, начиная с последнего. Вслед за длинным (полным) именем размещается стандартный описатель файла, содержащий укороченный по специальному алгоритму вариант этого имени. Пример хранения длинного имени файла показан здесь: http://www.ntfs.com/fat-filenames.htm

Загрузочный сектор

В первом секторе логического диска с системой FAT располагается загрузочный сектор и блок параметров BIOS. Начальный участок данного блока для всех типов FAT идентичен (таблица 3). Различия в структуре загрузочных секторов для разных типов FAT начинаются со смещения 0x24. Для FAT12 и FAT16 структура имеет вид, показанный в таблице 4, для FAT32 – в таблице 5.

Таблица 3. Начальный участок загрузочного сектора

Смещение	Размер (байт)	Описание
0x00	3	Безусловный переход (jmp) на загрузочный код.
0x03	8	Идентификатор фирмы-изготовителя.
0x0B	2	Число байт в секторе (512).
0x0D	1	Число секторов в кластере.
0x0E	2	Число резервных секторов в резервной области раздела, начиная с первого сектора раздела.
0x10	1	Число таблиц (копий) FAT.
0x11	2	Для FAT12/FAT16 - количество 32-байтных дескрипторов файлов в корневом каталоге; для FAT32 это поле имеет значение 0.
0x13	2	Общее число секторов в разделе; если данное поле содержит 0, то число секторов задается полем по смещению 0x20.
0x15	1	Тип носителя. Для жесткого диска имеет значение 0xF8; для гибкого диска (2 стороны, 18 секторов на дорожке) - 0xF0.
0x16	2	Для FAT12/FAT16 это поле содержит количество секторов, занимаемых одной копией FAT; для FAT32 это поле имеет значение 0.
0x18	2	Число секторов на дорожке (для прерывания 0x13).
0x1A	2	Число рабочих поверхностей (для прерывания 0x13).
0x1C	4	Число скрытых секторов перед разделом.
0x20	4	Общее число секторов в разделе. Поле используется, если в разделе свыше 65535 секторов, в противном случае поле содержит 0.

Таблица 4. Структура загрузочного сектора FAT12/FAT16

Смещение	Размер (байт)	Описание
0x24	1	Номер дисковода для прерывания 0x13.
0x25	1	Зарезервировано для Windows NT, имеет значение 0.
0x26	1	Признак расширенной загрузочной записи (0x29).
0x27	4	Номер логического диска.
0x2B	11	Метка диска.
0x36	8	Текстовая строка с аббревиатурой типа файловой системы.

Таблица 5. Структура загрузочного сектора FAT32

Смещение	Размер (байт)	Описание
0x24	4	Количество секторов, занимаемых одной копией FAT.
0x28	2	Номер активной FAT.
0x2A	2	Номер версии FAT32: старший байт – номер версии, младший – номер ревизии. В настоящее время используется значение 0:0.
0x2C	4	Номер кластера для первого кластера корневого каталога.
0x30	2	Номер сектора структуры FSINFO в резервной области логического диска.
0x32	2	Номер сектора (в резервной области логического диска), используемого для хранения резервной копии загрузочного сектора.
0x34	12	Зарезервировано (содержит 0).

Кроме перечисленных в таблицах 2-го и 3-го полей, нулевой сектор логического диска должен содержать в байте со смещением 0x1FE код 0x55, а в следующем байте (смещение 0x1FF) – код 0xAA. Указанные два байта являются признаком загрузочного диска.

Таким образом, загрузочный сектор выполняет две важные функции: описывает структуру данных на диске, а также позволяет осуществить загрузку операционной системы.

На логическом диске с организацией FAT32 дополнительно присутствует структура FSInfo, размещаемая в первом секторе резервной области. Эта структура содержит информацию о количестве свободных кластеров на диске и о номере первого свободного кластера в таблице FAT. Формат структуры описан в таблице 6.

Таблица 6. Структура сектора FSInfo и резервного загрузочного сектора FAT32

Смещение	Размер (байт)	Описание
0x000	4	Значение 0x41615252 – сигнатура, которая служит признаком того, что данный сектор содержит структуру FSInfo.
0x004	480	Зарезервировано (содержит 0).
0x1E4	4	Значение 0x61417272 (сигнатура).
0x1E8	4	Содержит текущее число свободных кластеров на диске. Если в поле записано значение 0xFFFFFFFF, то число свободных кластеров не известно, и его необходимо вычислять.
0x1EC	4	Содержит номер кластера, с которого дисковый драйвер должен начинать поиск свободных кластеров. Если в поле записано значение 0xFFFFFFFF, то поиск свободных кластеров нужно начинать с кластера номер 2.
0x1F0	12	Зарезервировано (содержит 0).
0x1FC	4	Сигнатура 0xAA550000 – признак конца структуры FSInfo.

Для доступа к содержимому файла, находящемуся на разделе с файловой системой FAT, необходимо получить номер первого кластера файла. Этот номер, как мы уже установили, входит в состав элемента каталога, содержащего запись о файле. Номеру первого кластера соответствует элемент таблицы FAT, в котором хранится адрес кластера, содержащего следующую часть файла. Элемент FAT, соответствующий последнему кластеру в цепочке, содержит сигнатуру конца файла. Для FAT12 это значение составляет 0xFFF, для FAT16 – 0xFFFF, для FAT32 – 0xFFFFFFFF.

Рассмотрим программную реализацию алгоритма чтения для каждого типа FAT, и начнём с FAT16.

Все исходные тексты, рассматриваемые в статье, доступны на сайте журнала.

Программная реализация алгоритма чтения файла с логического раздела с файловой системой FAT16

Разработаем модуль, выполняющий чтение N первых кластеров файла, созданного на разделе с файловой системой FAT16. Параметр N (число кластеров для считывания) является переменной величиной и задается пользователем. Имя файла соответствует формату «8.3», т.е. является коротким. Модуль функционирует под управлением ОС Linux.

Определим необходимые заголовочные файлы:

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <linux/msdos_fs.h>
#include "split.h"
```

Заголовочный файл split.h имеет следующее содержание:

```
#include <linux/types.h>
// максимальная длина короткого имени файла
#define SHORT_NAME 13

struct split_name {
   __u8 name[9];   // имя файла
   __u8 ext[4];    // расширение файла
   int name_len,   // длина имени файла
   ext_len;        // длина расширения файла
};
```

Структура split_name предназначена для хранения составных частей короткого имени файла (имени и расширения) и их длин.

В заголовочном файле <linux/msdos_fs.h> определены структурные типы, описывающие основные компоненты файловой системы FAT – загрузочный сектор, сектор FSInfo, структуры элементов каталога для короткого и длинного имён файлов.

Рассмотрим кратко поля, которые входят в каждую из этих структур.

1. Структура загрузочного сектора struct fat_boot_sector:
- __s8 system_id[8] – системный идентификатор;
- __u8 sector_size[2] – размер сектора в байтах;
- __u8 cluster_size – размер кластера в секторах;
- __u16 reserved – число резервных секторов в резервной области раздела;
- __u8 fats – количество копий FAT;
- __u8 dir_entries[2] – количество 32-байтных дескрипторов файлов в корневом каталоге;
- __u8 sectors[2] – число секторов на разделе; если это поле равно 0, используется поле total_sect;
- __u8 media – тип носителя, на котором создана файловая система;
- __u16 fat_length – размер FAT в секторах;
- __u32 total_sect – размер раздела FAT в секторах (если поле sectors == 0).

Следующие поля данной структуры используются только FAT32:
- __u32 fat32_length – размер FAT32 в секторах;
- __u32 root_cluster – номер первого кластера корневого каталога;
- __u16 info_sector – номер сектора, содержащего структуру FSInfo.

2. Структура сектора FSInfo struct fat_boot_fsinfo:
- __u32 signature1 – сигнатура 0x41615252;
- __u32 signature2 – сигнатура 0x61417272;
- __u32 free_clusters – количество свободных кластеров. Если поле содержит -1, поиск свободных кластеров нужно начинать с кластера номер 2.

3. Структура элемента каталога короткого имени struct msdos_dir_entry:
- __s8 name[8],ext[3] – имя и расширение файла;
- __u8 attr – атрибуты файла;
- __u8 ctime_ms – это поле уточняет время создания файла до мс (используется только FAT32);

- __u16 ctime – время создания файла (используется только FAT32);
- __u16 cdate – дата создания файла (используется только FAT32);
- __u16 adate – дата последнего доступа к файлу (используется только FAT32);
- __u16 starthi – старшие 16 бит номера первого кластера файла (используется только FAT32);
- __u16 time,date,start – время и дата создания файла, номер первого кластер файла;
- __u32 size – размер файла (в байтах).

4. Структура элемента каталога длинного имени:

- __u8 id – номер элемента;
- __u8 name0_4[10] – символы 1 – 5 имени;
- __u8 attr – атрибуты файла;
- __u8 alias_checksum – контрольная сумма короткого имени;
- __u8 name5_10[12] – символы 6 – 11 имени;
- __u8 name11_12[4] – символы 12 – 13 имени.

Продолжим рассмотрение программной реализации алгоритма и определим имя раздела, на котором создана файловая система FAT16:

```
#ifndef FAT16_PART_NAME
#define FAT16_PART_NAME "/dev/hda1"
#endif
```

Глобальные структуры:

```
struct fat_boot_sector fbs; // структура загрузочного сектора
struct msdos_dir_entry dentry; // структура элемента каталога
```

Глобальные переменные:

```
__u16 *fat16;              // сюда копируем таблицу FAT16
__u16 sector_size;         // размер сектора (из FAT16)
__u16 dir_entries;         // число 32-байтных дескрипторов
                           // в root-каталоге (0 для FAT32)
__u16 sectors;             // общее число секторов в разделе
__u32 fat16_size;          // размер FAT16
__u32 root_size;           // размер корневого каталога
__u32 data_start;          // начало области данных
__u16 byte_per_cluster;    // размер кластера в байтах
__u16 next_cluster;        // очередной кластер в цепочке
__u8 *dir_entry = NULL;    // указатель на записи каталога

int hard;                  // дескриптор файла устройства
int fat;
```

Начнём рассмотрение с главной функции:

```
int main()
{
  int num;
```

Задаем полное имя файла, содержимое которого мы хотим прочитать. Напомню, что мы работаем только с короткими именами файлов. Порядок работы с длинными именами в данной статье не рассматривается.

```
  __u8 *full_path = "/Folder1/Folder2/text.txt";
```

Открываем файл устройства:

```
  hard = open(FAT16_PART_NAME, O_RDONLY);
  if(hard < 0) {
    perror(FAT16_PART_NAME);
    exit(-1);
  }
```

Считываем первые 10 кластеров файла. Считывание выполняет функция fat16_read_file(). Параметры функции – полное имя файла и число кластеров для чтения. Функция возвращает число прочитанных кластеров или -1, если при чтении произошла ошибка:

```
  num = fat16_read_file(full_path, 10);
  if(num < 0) perror("fat16_read_file");
  else printf("Read %d clusters\n", num);
```

Закрываем файл устройства и выходим:

```
  close(hard);
  return 0;
}
```

Функция чтения кластеров файла имеет следующий вид:

```
int fat16_read_file(__u8 *full_path, int num)
{
  // структура для хранения составных частей файла
  struct split_name sn;
  // буфер для временного хранения составных элементов
  // полного пути файла
  __u8 tmp_name_buff[SHORT_NAME];
  static int i = 1;
  int n;

  __u8 *tmp_buff;
  __u16 start_cluster, next_cluster;
```

Параметры функции мы перечислили при рассмотрении функции main.

Подготовительные операции – обнуляем буфер tmp_name_buff и структуру struct split_name sn:

```
  memset(tmp_name_buff, 0, SHORT_NAME);
  memset((void *)&sn, 0, sizeof(struct split_name));
```

Первым символом в абсолютном путевом имени файла должен быть прямой слэш (/). Проверяем это:

```
  if(full_path[0] != '/') return -1;
```

Считываем с раздела загрузочный сектор:

```
  if(read_fbs() < 0) return -1;
```

Считанный загрузочный сектор находится сейчас в глобальной структуре struct fat_boot_sector fbs. Скопируем из этой структуры размер сектора, число записей в корневом каталоге и общее число секторов на разделе:

```
  memcpy((void *)&sector_size, (void *)fbs.sector_size, 2);
  memcpy((void *)&dir_entries, (void *)fbs.dir_entries, 2);
  memcpy((void *)&sectors, (void *)fbs.sectors, 2);
```

Определим размер кластера в байтах:

```
  byte_per_cluster = fbs.cluster_size * 512;
```

Отобразим информацию, находящуюся в загрузочном секторе:

```
printf("System id - %s\n", fbs.system_id);
printf("Sector size - %d\n", sector_size);
printf("Cluster size - %d\n", fbs.cluster_size);
printf("Reserved - %d\n", fbs.reserved);
printf("FATs number - %d\n",fbs.fats);
printf("Dir entries - %d\n", dir_entries);
printf("Sectors - %d\n", sectors);
printf("Media - 0x%X\n", fbs.media);
printf("FAT16 length - %u\n", fbs.fat_length);
printf("Total sect - %u\n", fbs.total_sect);
printf("Byte per cluster - %d\n", byte_per_cluster);
```

Вычисляем размер FAT16 в байтах и считываем её:

```
fat16_size = fbs.fat_length * 512;
if(read_fat16() < 0) return -1;
```

Считываем корневой каталог:

```
if(read_root_dentry() < 0) return -1;
```

Сейчас указатель dir_entry позиционирован на область памяти, содержащую записи корневого каталога. Размер этой области памяти равен размеру корневого каталога (root_size).

Сохраним (для контроля) содержимое корневого каталога в отдельном файле:

```
#ifdef DEBUG
  fat = open("dir16", O_CREAT|O_WRONLY, 0600);
  write(fat, dir_entry, root_size);
  close(fat);
#endif
```

Вычисляем начало области данных:

```
data_start = 512 * fbs.reserved + fat16_size * fbs.fats + ↵
  root_size;
```

Имея все записи корневого каталога, мы можем добраться до содержимого файла test.txt. С этой целью организуем цикл. В теле цикла проведем разбор полного имени файла, выделяя его элементы – подкаталоги (их у нас два, Folder1 и Folder2) и имя искомого файла (test.txt).

```
while(1) {

  memset(tmp_name_buff, 0, SHORT_NAME);
  memset((void *)&sn, 0, sizeof(struct split_name));

  for(n = 0 ; n < SHORT_NAME; n++, i++) {
    tmp_name_buff[n] = full_path[i];
    if((tmp_name_buff[n] == '/') || (tmp_name_buff[n] == ↵
      '\0')) {
      i++;
      break;
    }
  }

  tmp_name_buff[n] = '\0';
```

Заполняем структуру struct split_name sn соответствующей информацией. Заполнение выполняет функция split_name, при этом выполняется проверка имени файла на соответствие формату «8.3»:

```
  if(split_name(tmp_name_buff, &sn) < 0) {
    printf("not valid name\n");
    return -1;
  }
```

Для каждого элемента полного имени файла определяем начальный кластер. Для этого ищем в элементах каталога (начиная с корневого) запись, соответствующую элементу полного имени, и считываем эту запись. Процедуру поиска выполняет функция get_dentry():

```
  if(get_dentry(&sn) < 0) {
    printf("No such file!\n");
    return -1;
  }
```

Проверяем атрибуты файла. Если это каталог, считываем его содержимое и продолжаем цикл:

```
  if(dentry.attr & 0x10) {
    if(read_directory(dentry.start) < 0) return -1;
    continue;
  }
```

Если это файл – считываем первые num кластеров. Для контроля считанную информацию сохраним в отдельном файле:

```
  if(dentry.attr & 0x20) {

    start_cluster = dentry.start;
    // сюда будет считываться содержимое кластера
    tmp_buff = (__u8 *)malloc(byte_per_cluster);
    // в этом файле сохраним считанную информацию
    n = open("clust", O_CREAT|O_RDWR, 0600);
    if(n < 0) {
      perror("open");
      return -1;
    }

    printf("file`s first cluster - 0x%X .. ", start_cluster);
```

Для считывания кластеров файла организуем цикл:

```
    for(i = 0; i < num; i++) {
      memset(tmp_buff, 0, byte_per_cluster);
```

Считываем содержимое кластера в буфер tmp_buff и сохраняем его в отдельном файле:

```
      if(read_cluster(start_cluster, tmp_buff) < 0) ↵
        return -1;

      if(write(n, tmp_buff, ↵
        byte_per_cluster) < 0) {
        perror("write");
        close(n);
        return -1;
      }
```

Считываем из FAT16 номер следующего кластера, занятого под данный файл. Если это последний кластер – прерываем цикл и возвращаемся в главную функцию:

```
      next_cluster = fat16[start_cluster];

#ifdef DEBUG
      printf("OK. Readed\n");
      printf("file`s next cluster - 0x%X .. ", next_cluster);
#endif

      if(next_cluster == EOF_FAT16) {

#ifdef DEBUG
        printf("last cluster.\n");
#endif

        free(tmp_buff);
        close(n);
        return ++i;
      }
```

```
            start_cluster = next_cluster;
        }

#ifdef DEBUG
        printf("stop reading\n");
#endif
        return i;
      }
    }
}
```

Чтение загрузочного сектора FAT16 выполняет функция read_fbs(). Результат помещается в глобальную структуру fbs:

```
int read_fbs()
{
  if(read(hard,(__u8 *)&fbs, sizeof(fbs)) < 0) return -1;
  return 0;
}
```

Чтение таблицы размещения файлов файловой системы FAT16 выполняет функция read_fat16():

```
int read_fat16()
{
  // смещение к FAT16 от начала раздела
  __u64 seek = (__u64)(fbs.reserved) * 512;
  fat16 = (void *)malloc(fat16_size);
  if(pread64(hard, (__u8 *)fat16, fat16_size, ↲
    seek) < 0) return -1;
  return 0;
}
```

Чтение корневого каталога выполняет функция read_root_dentry():

```
int read_root_dentry()
{
  // смещение к корневому каталогу от начала раздела
  __u64 seek = (__u64)fbs.reserved * 512 + fat16_size * fbs.fats;
  // вычисляем размер корневого каталога
  root_size = 32 * dir_entries;
  dir_entry = (__u8 *)malloc(root_size);
  if(!dir_entry) return -1;
  memset(dir_entry, 0, root_size);
  if(pread64(hard, dir_entry, root_size, seek) < 0) return -1;
  return 0;
}
```

Чтение кластера, принадлежащего файлу, выполняет функция read_cluster(). Входные параметры функции – номер кластера cluster_num и указатель на буфер __u8 *tmp_buff, куда нужно поместить результат чтения. Смещение к кластеру на разделе вычисляется по формуле (см. [1]):

```
    SEEK = DATA_START + (CLUSTER_NUM - 2) * BYTE_PER_CLUSTER,

где
    SEEK - смещение к кластеру на разделе;
    DATA_START - начало области данных;
    CLUSTER_NUM - порядковый номер кластера;
    BYTE_PER_CLUSTER - размер кластера в байтах.

int read_cluster(__u16 cluster_num, __u8 *tmp_buff)
{
  // вычисляем смещение к кластеру
  __u64 seek = (__u64)(byte_per_cluster) * ↲
          (cluster_num - 2) + data_start;

  if(pread64(hard, tmp_buff, byte_per_cluster, ↲
          seek) < 0) return -1;
  return 0;
}
```

Функция read_directory выполняет чтение записей каталога (не корневого) и помещает результат в область памяти, на которую настроен указатель dir_entry:

```
int read_directory(__u16 start_cluster)
{
  int i = 1;
  __u16 next_cluster;

  for(; ;i++) {
```

Выделяем память для хранения содержимого каталога, считываем содержимое стартового кластера и получаем из таблицы FAT16 значение очередного кластера:

```
    dir_entry = (__u8 *)realloc(dir_entry, i * byte_per_cluster);
    if(!dir_entry) return -1;

    if(read_cluster(start_cluster, (dir_entry + (i - 1) * ↲
          byte_per_cluster)) < 0) return -1;
    next_cluster = fat16[start_cluster];
```

Сохраним содержимое каталога в отдельном файле (для контроля):

```
#ifdef DEBUG
    printf("Next cluster - 0x%X\n", next_cluster);
    fat = open("dir16", O_CREAT|O_WRONLY, 0600);
    write(fat, dir_entry, root_size);
    close(fat);
#endif
```

Если достигнут последний кластер, выходим из цикла, иначе продолжаем чтение каталога, увеличив размер буфера dir_entry ещё на один кластер:

```
    if(next_cluster & EOF_FAT16) break;
    start_cluster = next_cluster;
  }
  return 0;
}
```

Поиск в содержимом каталога элемента, соответствующего искомому файлу, выполняет функция get_dentry(). Входные параметры этой функции – указатель на структуру struct split_name *sn, содержащую элементы короткого имени файла:

```
int get_dentry(struct split_name *sn)
{
  int i = 0;
```

В глобальном буфере dir_entry находится массив элементов каталога, в котором мы собираемся искать запись файла (или каталога). Для поиска организуем цикл. В теле цикла производим копирование элементов каталога в глобальную структуру dentry и сравниваем значение полей name и ext этой структуры с соответствующими полями структуры struct split_name *sn. Совпадение этих полей означает, что мы нашли в массиве элементов каталога запись искомого файла:

```
  for(; ; i++) {

    memcpy((void *)&dentry, dir_entry + i * sizeof(dentry), ↲
          sizeof(dentry));

    if(!(memcmp(dentry.name, sn->name, sn->name_len)) && ↲
          !(memcmp(dentry.ext, sn->ext, n->ext_len)))
      break;

    if(!dentry.name[0]) return -1;
  }

#ifdef DEBUG
  printf("name - %s\n", dentry.name);
  printf("start cluster - 0x%X\n", dentry.start);
```

```
  printf("file size - %u\n", dentry.size);
  printf("file attrib - 0x%X\n", dentry.attr);
#endif

  return 0;
}
```

Весь вышеприведенный код находится в каталоге FAT16, файл fat16.c. Для получения исполняемого модуля создадим Makefile следующего содержания:

```
INCDIR  = /usr/src/linux/include
.PHONY = clean

fat16: fat16.o split.o
  gcc -I$(INCDIR) $^ -g -o $@

%.o: %.c
  gcc -I$(INCDIR) -DDEBUG -c $^

clean:
  rm -f *.o
  rm -f ./fat16
```

Программная реализация алгоритма чтения файла с логического раздела с файловой системой FAT12

В целом алгоритм чтения файла с раздела FAT12 идентичен алгоритму чтения файла с раздела FAT16. Отличие заключается в процедуре чтения элементов из таблицы FAT12. Таблица FAT16 рассматривалась нами как простой массив 16-разрядных элементов. Для чтения элементов таблицы FAT12 в [1] предложен следующий алгоритм:

- умножить номер элемента на 1.5;
- извлечь из FAT 16-разрядное слово, используя в качестве смещения результат предыдущей операции;
- если номер элемента четный, выполнить операцию AND над считанным словом и маской 0x0FFF. Если номер нечетный, сдвинуть считанное из таблицы слово на 4 бита в сторону младших разрядов.

Базируясь на этом алгоритме, реализуем функцию чтения элементов из таблицы FAT12:

```
int get_cluster(__u16 cluster_num)
{
  __u16 seek;
  __u16 clust;
```

Вычисляем смещение в таблице FAT12 и считываем из таблицы 16-разрядное слово:

```
  seek = (cluster_num * 3) / 2;
  memcpy((__u8 *)&clust, (__u8 *)(fat12 + seek), 2);
```

Если стартовый номер кластера – четное число, сдвигаем считанное из таблицы значение на 4 бита в сторону младших разрядов, если нечетное – суммируем его с 0x0FFF:

```
  if(cluster_num % 2) clust >>= 4;
  else clust &= 0x0FFF;
```

Этот фрагмент можно также реализовать на ассемблере:

```
  asm(
    "      xorw %%ax, %%ax \n\t"
    "      btw $0, %%cx \n\t"
    "      jnc 1f \n\t"
    "      shrw $4, %%dx \n\t"
    "      jmp 2f \n\t"
    "1:    andw $0x0FFF, %%dx \n\t"
    "2:    movw %%dx, %%ax \n\t"
    :"=a" (next)
    :"d" (clust), "c" (cluster_num));
```

Возвращаем результат:

```
  return clust;
}
```

Остановимся чуть подробнее на самом алгоритме. Предположим, что на разделе с FAT12 создан файл, который занимает 9-й и 10-й кластеры. Каждый элемент FAT12 занимает 12 бит. Т.к. из таблицы мы считываем 16-разрядные элементы, то смещение к 9-му элементу будет равно 13 байт (9 * 1.5 = 13, остаток отбрасываем), при этом младшие 4 разряда будут принадлежать 8-му элементу FAT. Их необходимо отбросить, а для этого достаточно сдвинуть считанный элемент на 4 бита в сторону младших разрядов, что и предусмотрено алгоритмом. Смещение к 10-му элементу будет равно 15 байт, и старшие 4 бита будут принадлежать 11-му элементу FAT. Чтобы их отбросить, необходимо выполнить операцию AND над 10-м элементом и маской 0x0FFF, что так же соответствует вышеприведенному алгоритму.

Исходные тексты модуля чтения файла с раздела FAT12 находятся в каталоге FAT12, файл fat12.c.

Программная реализация алгоритма чтения файла с логического раздела с файловой системой FAT32

Алгоритм чтения файла с раздела с файловой системой FAT32 практически не отличается от алгоритма для FAT16, за исключением того, что в FAT32 корневой каталог может располагаться в любом месте раздела и иметь произвольный размер. Поэтому, чтобы было интереснее, усложним задачу – предположим, что нам известен только номер раздела с файловой системой FAT32. Чтобы считать с этого раздела информацию, необходимо вначале определить его координаты – смещение к разделу от начала диска. А для этого надо иметь представление о логической структуре жесткого диска.

Логическая структура жесткого диска

Рассмотрим логическую структуру жесткого диска, соответствующую стандарту Microsoft – «основной раздел – расширенный раздел – разделы non-DOS».

Пространство на жестком диске может быть организовано в виде одного или нескольких разделов, а разделы могут содержать один или несколько логических дисков.

На жестком диске по физическому адресу 0-0-1 располагается главная загрузочная запись (Master Boot Record, MBR). В структуре MBR находятся следующие элементы:

- внесистемный загрузчик (non-system bootstrap – NSB);
- таблица описания разделов диска (таблица разделов, partition table, PT). Располагается в MBR по смещению 0x1BE и занимает 64 байта;
- сигнатура MBR. Последние два байта MBR должны содержать число 0xAA55.

Таблица разделов описывает размещение и характеристики имеющихся на винчестере разделов. Разделы диска могут быть двух типов – primary (первичный, основной) и extended (расширенный). Максимальное число primary-разделов равно четырем. Наличие на диске хотя бы одного primary-раздела является обязательным. Extended-раздел может быть разделен на большое количество подразделов – логических дисков. Упрощенно структура MBR представлена в таблице 7. Таблица разделов располагается в конце MBR, для описания раздела в таблице отводится 16 байт.

Таблица 7. Структура MBR

Смещение	Размер (байт)	Содержимое (contents)
0	446	Программа анализа таблицы разделов и загрузки с активного раздела.
0x1BE	16	Partition 1 entry (элемент таблицы разделов).
0x1CE	16	Partition 2 entry.
0x1DE	16	Partition 3 entry.
0x1EE	16	Partition 4 entry.
0x1FE	2	Сигнатура 0xAA55.

Структура записи элемента таблицы разделов показана в таблице 8.

Таблица 8. Структура записи элемента таблицы разделов

Смещение	Размер (байт)	Содержание
0x00	1	Признак активности (0 – раздел не активный, 0x80 - раздел активный).
0x01	1	Номер головки диска, с которой начинается раздел.
0x02	2	Номер цилиндра и номер сектора, с которых начинается раздел.
0x04	1	Код типа раздела System ID.
0x05	1	Номер головки диска, на которой заканчивается раздел.
0x06	2	Номер цилиндра и номер сектора, которыми заканчивается раздел.
0x08	4	Абсолютный (логический) номер начального сектора раздела.
0x0C	4	Размер раздела (число секторов).

Первым байтом в элементе раздела идет флаг активности раздела (0 – неактивен, 0x80 – активен). Он служит для определения, является ли раздел системным загрузочным и есть ли необходимость производить загрузку операционной системы с него при старте компьютера. Активным может быть только один раздел. За флагом активности раздела следуют координаты начала раздела – три байта, означающие номер головки, номер сектора и номер цилиндра. Номера цилиндра и сектора задаются в формате прерывания Int 0x13, т.е. биты 0-5 содержат номер сектора, биты 6-7 – старшие два бита 10-разрядного номера цилиндра, биты 8-15 – младшие восемь бит номера цилиндра. Затем следует кодовый идентификатор System ID, указывающий на принадлежность данного раздела к той или иной операционной системе. Идентификатор занимает один байт. За системным идентификатором расположены координаты конца раздела – три байта, содержащие номера головки, сектора и цилиндра соответственно. Следующие четыре байта – это число секторов перед разделом, и последние четыре байта – размер раздела в секторах.

Таким образом, элемент таблицы раздела можно описать при помощи следующей структуры:

```
struct pt_struct {

    // флаг активности раздела
    u8 bootable;
    // координаты начала раздела
    u8 start_part[3];
    // системный идентификатор
    u8 type_part;
    // координаты конца раздела
    u8 end_part[3];
    // число секторов перед разделом
    u32 sect_before;
    // размер раздела в секторах (число секторов
    // в разделе)
    u32 sect_total;
};
```

Элемент первичного раздела указывает сразу на загрузочный сектор логического диска (в первичном разделе всегда имеется только один логический диск), а элемент расширенного раздела – на список логических дисков, составленный из структур, которые именуются вторичными MBR (Secondary MBR, SMBR).

Свой блок SMBR имеется у каждого диска расширенного раздела. SMBR имеет структуру, аналогичную MBR, но загрузочная запись у него отсутствует (заполнена нулями), а из четырех полей описателей разделов используются только два. Первый элемент раздела при этом указывает на логический диск, второй элемент указывает на следующую структуру SMBR в списке. Последний SMBR списка содержит во втором элементе нулевой код раздела.

Вернемся к рассмотрению модуля чтения файла с раздела FAT32.

Заголовочные файлы:

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <linux/msdos_fs.h>
```

Сигнатура MBR:

```
#define SIGNATURE 0xAA55
```

Файл устройства, с которого будет считываться информация о разделах:

```
#define DEVICE "/dev/hda"
```

Размер элемента таблицы разделов (16 байт):

```
#define PT_SIZE 0x10
```

Следующий массив структур устанавливает соответствие между кодом типа раздела и его символьным отображением:

```
struct systypes {
    __u8 part_type;
    __u8 *part_name;
};
struct systypes i386_sys_types[] = {
    {0x00, "Empty"},
    {0x01, "FAT12"},
    {0x04, "FAT16 <32M"},
    {0x05, "Extended"},
    {0x06, "FAT16"},
    {0x0b, "Win95 FAT32"},
    {0x0c, "Win95 FAT32 (LBA)"},
    {0x0e, "Win95 FAT16 (LBA)"},
    {0x0f, "Win95 Ext'd (LBA)"},
    {0x82, "Linux swap"},
    {0x83, "Linux"},
```

```
      {0x85, "Linux extended"},
      {0x07, "HPFS/NTFS"}
};
```

Определим число элементов в массиве i386_sys_types при помощи макроса PART_NUM:

```
#define PART_NUM (sizeof(i386_sys_types) / sizeof(i386_sys_types[0]))
```

Установим ограничение на количество логических дисков:

```
#define MAX_PART 20
```

Следующий массив структуры будет содержать информацию о логических дисках на устройстве (жестком диске):

```
struct pt_struct {
   __u8 bootable;
   __u8 start_part[3];
   __u8 type_part;
   __u8 end_part[3];
   __u32 sect_before;
   __u32 sect_total;
} pt_t[MAX_PART];

int hard;          // дескриптор файла устройства
__u8 mbr[512];     // сюда считаем MBR
```

Номер раздела, на котором создана файловая система FAT32:

```
#define FAT32_PART_NUM   5
```

Структуры загрузочного сектора, сектора FSInfo и элемента каталога (определены в файле <linux/msdos>):

```
struct fat_boot_sector fbs;
struct fat_boot_fsinfo fsinfo;
struct msdos_dir_entry dentry;
__u32 *fat32 = NULL;      // сюда копируем таблицу FAT32
__u16 sector_size;        // размер сектора (из FAT32)
__u16 dir_entries;        // 0 для FAT32
__u16 sectors;            // число секторов на разделе
__u32 fat32_size;         // размер FAT32
__u32 data_start;         // начало области данных
__u16 byte_per_cluster;   // сколько байт в кластере
                          // (размер кластера в байтах)
__u32 next_cluster;       // очередной кластер в цепочке
__u32 root_cluster;       // ROOT cluster - начальный кластер
                          // корневого каталога
__u8 *dir_entry = NULL;   // указатель на записи каталога
__u64 start_seek = 0;     // стартовое смещение к разделу
                          // (в байтах)
```

Главная функция:

```
int main()
{
   int num = 0;
   // сколько кластеров считывать из файла
   int cluster_num = 5;
   // файл для считывания
   __u8 *full_path = "/Folder1/Folder2/readme";
```

Открываем устройство, получаем информацию о таблице разделов на устройстве и отображаем информацию о разделах:

```
   hard = open(DEV_NAME, O_RDONLY);
   if(hard < 0) {
      perror(DEV_NAME);
      exit(-1);

   }
```

```
   if(get_pt_info(hard) < 0) {
      perror("get_pt_info");
      exit(-1);
   }

   show_pt_info();
```

Вычисляем стартовое смещение к разделу:

```
   start_seek = (__u64)(pt_t[FAT32_PART_NUM - ↵
    1].sect_before) * 512;
```

Считываем кластеры, принадлежащие файлу:

```
   num = fat32_read_file(full_path, cluster_num);
   if(num < 0) perror("fat32_read_file");
   else printf("Read %d clusters\n", num);

   close(hard);
   return 0;
}
```

Информацию о таблице разделов считывает функция get_pt_info():

```
int get_pt_info(int hard)
{
   int i = 0;
   __u64 seek;
```

Считываем таблицу разделов из MBR и проверяем сигнатуру:

```
   read_main_ptable(hard);

   if(check_sign() < 0) {
      printf("Not valid signature!\n");
      return -1;
   }
```

Ищем идентификатор расширенного раздела. Если таковой имеется, вычисляем смещение к расширенному разделу и считываем информацию о логических дисках:

```
   for(; i < 4; i++) {
      if((pt_t[i].type_part == 0xF) || ↵
            (pt_t[i].type_part == 0x5) || ↵
            (pt_t[i].type_part == 0x0C)) {
         seek = (__u64)pt_t[i].sect_before * 512;
         read_ext_ptable(hard, seek);
         break;
      }
   }
   return 0;
}
```

Функция чтения таблицы разделов read_main_ptable():

```
void read_main_ptable(int hard)
{
   if(read(hard, mbr, 512) < 0) {
      perror("read");
      close(hard);
      exit(-1);
   }
   memset((void *)pt_t, 0, (PT_SIZE * 4));
   memcpy((void *)pt_t, mbr + 0x1BE, (PT_SIZE * 4));
   return;
}
```

Функция проверки сигнатуры check_sign():

```
int check_sign()
{
   __u16 sign = 0;
   memcpy((void *)&sign, (void *)(mbr + 0x1FE), 2);
```

```
#ifdef DEBUG
  printf("Signature - 0x%X\n", sign);
#endif

  if(sign != SIGNATURE) return -1;
  return 0;
}
```

Функция чтения расширенной таблицы разделов:

```
void read_ext_ptable(int hard, __u64 seek)
{
  // начиная с этой позиции, массив структур pt_t будет
  // заполняться информацией о логических дисках
  int num = 4;
  __u8 smbr[512];
```

Входные данные:
- hard – дескриптор файла устройства;
- seek – смещение к расширенному разделу от начала диска (в байтах).

Для получения информации о логических дисках организуем цикл:

```
  for(;;num++) {
```

Считываем SMBR, находящуюся по смещению seek от начала диска:

```
    memset((void *)smbr, 0, 512);
    pread64(hard, smbr, 512, seek);
```

Заполняем два элемента таблицы pt_t, начиная с позиции num. Первый элемент будет указывать на логический диск, а второй – на следующую структуру SMBR:

```
    memset((void *)&pt_t[num], 0, PT_SIZE * 2);
    memcpy((void *)&pt_t[num], smbr + 0x1BE, PT_SIZE * 2);
```

Вносим поправку в поле «Номер начального сектора» – отсчет ведется от начала диска:

```
    pt_t[num].sect_before += (seek / 512);
```

Если код типа раздела равен нулю, то больше логических дисков нет:

```
    if(!(pt_t[num + 1].type_part)) break;
```

Вычисляем смещение к следующему SMBR:

```
    seek = ((__u64)(pt_t[num].sect_before + ↵
           pt_t[num].sect_total)) * 512;
  }
  return;
}
```

Функция show_pt_info() отображает информацию о найденных логических дисках на устройстве:

```
void show_pt_info()
{
  int i = 0, n;

#ifdef DEBUG
  printf("Число разделов на диске - %d\n", PART_NUM);
#endif

  for(; i < MAX_PART; i++) {
    if(!pt_t[i].type_part) break;
    printf("\nТип раздела %d - ", i);

      for(n = 0; n < PART_NUM; n++) {
        if(pt_t[i].type_part == i386_sys_types[n].part_type) {
          printf("%s\n", i386_sys_types[n].part_name);
          break;
        }
    }
    if(n == PART_NUM) printf("unknown type\n");

    printf("Признак загрузки - 0x%X\n", pt_t[i].bootable);
    printf("Секторов в разделе %d - %d\n", i, ↵
           pt_t[i].sect_total);
    printf("Секторов перед разделом %d - %d\n\n", i, ↵
           pt_t[i].sect_before);
  }
  return;
}
```

Чтение кластеров файла с раздела FAT32 выполняет функция fat32_read_file(). Эта функция имеет много общего с функцией fat16_read_file(), поэтому за подробными комментариями обратитесь к п. 6:

```
int fat32_read_file(__u8 *full_path, int num)
{
  struct split_name sn;
  __u8 tmp_name_buff[SHORT_NAME];
  int i = 1, n;

  __u32 start_cluster, next_cluster;
  __u8 *tmp_buff;
```

Подготовительные операции – чистим буфер, структуру и проверяем первый слэш:

```
  memset(tmp_name_buff, 0, SHORT_NAME);
  memset((void *)&sn, 0, sizeof(struct split_name));

  if(full_path[0] != '/') return -1
```

Считываем загрузочный сектор:

```
  if(read_fbs() < 0) return -1;

  memcpy((void *)&sector_size, (void *)fbs.sector_size, 2);
  memcpy((void *)&dir_entries, (void *)fbs.dir_entries, 2);
  memcpy((void *)&sectors, (void *)fbs.sectors, 2);
```

Считываем структуру FSInfo и отобразим сигнатуру, находящуюся в ней:

```
  if(read_fs_info() < 0) return -1;

  printf("Signature1 - 0x%X\n", fsinfo.signature1);
  printf("Signature2 - 0x%X\n", fsinfo.signature2);
  // размер FAT32 в байтах
  fat32_size = fbs.fat32_length * 512;
  // начало поля данных
  data_start = 512 * fbs.reserved + fat32_size * 2;
  // размер кластера в байтах
  byte_per_cluster = fbs.cluster_size * 512;
  // номер кластера корневого каталога
  root_cluster = fbs.root_cluster;
```

Считываем FAT32:

```
  if(read_fat32() < 0) return -1;
```

Выделяем память для записей каталога:

```
  dir_entry = (__u8 *)malloc(byte_per_cluster);
  if(!dir_entry) return -1;
```

Считываем корневой каталог:

```
    if(read_directory(root_cluster) < 0) return -1;
```

Проводим разбор полного пути файла и разделение каждого элемента на составляющие:

```
    while(1) {
      memset(tmp_name_buff, 0, SHORT_NAME);
      memset((void *)&sn, 0, sizeof(struct split_name));
      for(n = 0 ; n < SHORT_NAME; n++, i++) {
        tmp_name_buff[n] = full_path[i];
        if((tmp_name_buff[n] == '/') || ↲
            (tmp_name_buff[n] == '\0')) {
          i++;
          break;
        }
      }
      tmp_name_buff[n] = '\0';
      if(split_name(tmp_name_buff, &sn) < 0) {
        printf("not valid name\n");
        eturn -1;
      }
      if(get_dentry(&sn) < 0) {
        printf("No such file!\n");
        return -1;
      }
```

Для получения стартового номера кластера в файловой системе FAT32 необходимо задействовать старшее слово номера первого кластера файла – поле starthi структуры dentry:

```
      start_cluster = (((__u32)dentry.starthi << 16) | ↲
            dentry.start);
```

Проверяем байт атрибутов:

```
      if(dentry.attr & 0x10) {    // это каталог
        if(read_directory(start_cluster) < 0) return -1;
        continue;
      }
      if(dentry.attr & 0x20) {    // а это - файл
        tmp_buff = (__u8 *)malloc(byte_per_cluster);
        n = open("clust", O_CREAT|O_RDWR, 0600);
        if(n < 0) {
           perror("open");
           return -1;
        }
        printf("file`s first cluster - 0x%X .. ", start_cluster);
        for(i = 0; i < num; i++) {
           memset(tmp_buff, 0, byte_per_cluster);
           if(read_cluster(start_cluster, tmp_buff) < 0) return -1;
           if(write(n, tmp_buff, byte_per_cluster) < 0) {
             perror("write");
             return -1;
           }
           next_cluster = fat32[start_cluster];
           if(next_cluster == EOF_FAT32) {
             free(tmp_buff);
             close(n);
             return ++i;
           }
           start_cluster = next_cluster;
         }
        return i;
      }
    }
  }
```

Назначение следующих трёх функций – получить содержимое системной области, т.е. загрузочного сектора, структуры FSInfo и таблицы FAT32:

1) функция read_fbs() выполняет чтение загрузочного сектора:

```
int read_fbs()
{
if(pread64(hard, (__u8 *)&fbs, sizeof(fbs), ↲
         start_seek) < 0) return -1;
return 0;
}
```

2) функция read_fs_info() считывает структуру FSInfo:

```
int read_fs_info()
{
__u64 seek = (__u64)fbs.info_sector * 512 + start_seek;
if(pread64(hard, (__u8 *)&fsinfo, sizeof(fsinfo), ↲
       seek) < 0) return -1;

     return 0;
}
```

3) функция read_fat32() считывает таблицу FAT32:

```
int read_fat32()
{
__u64 seek = (__u64)fbs.reserved * 512 + start_seek;

fat32 = (void *)malloc(fat32_size);
if(!fat32) return -1;
if(pread64(hard, (__u8 *)fat32, fat32_size, ↲
       seek) < 0) return -1;

return 0;
}
```

Функция read_cluster() выполняет чтения кластера с указанным номером:

```
int read_cluster(__u32 cluster_num, __u8 *tmp_buff)
{

  __u64 seek = (__u64)(byte_per_cluster) * ↲
   (cluster_num - 2) + data_start + start_seek;
  if(pread64(hard, tmp_buff, byte_per_cluster, ↲
    seek) < 0) return -1;

  return 0;
}
```

Чтением каталогов (в том числе и корневого) занимается функция read_directory():

```
int read_directory(__u32 start_cluster)
{
  int i = 2;
  __u32 next_cluster;
```

Параметры функции – стартовый кластер каталога. Считываем содержимое каталога в глобальный буфер dir_entry:

```
  if(read_cluster(start_cluster, dir_entry) < 0) return -1;
  next_cluster = fat32[start_cluster];
```

Если каталог занимает один кластер – выходим, если нет – увеличиваем размер памяти и продолжаем чтение:

```
  if((next_cluster == EOF_FAT32) || (next_cluster == ↲
    0xFFFFFF8)) return 0;

  for(; ;i++) {
    start_cluster = next_cluster;
    dir_entry = (__u8 *)realloc(dir_entry, i * byte_per_cluster);
    if(!dir_entry) return -1;

    if(read_cluster(start_cluster, (dir_entry + (i - 1) * ↲
          byte_per_cluster)) < 0) return -1;
    next_cluster = fat32[start_cluster];
    if((next_cluster == EOF_FAT32) || (next_cluster == ↲
          0xFFFFFF8)) return 0;
  }
  return 0;
}
```

Последняя функция, которую мы рассмотрим, ищет в

содержимом каталога элемент, соответствующий искомому файлу:

```
int get_dentry(struct split_name *sn)
{
  int i = 0;
```

Указатель dir_entry настроен на область памяти, содержащую массив записей каталога, в котором мы собираемся искать файл (или каталог). Для поиска организуем цикл и найденную запись поместим в глобальную структуру dentry:

```
  for(;;i++) {

    memcpy((void *)&dentry, dir_entry + i * sizeof(dentry),
  sizeof(dentry));

    if(!(memcmp(dentry.name, sn->name, sn->name_len)) && ↲
          !(memcmp(dentry.ext, sn->ext, sn->ext_len)))
      break;

    if(!dentry.name[0]) return -1;
  }
  return 0;
}
```

На этом рассмотрение модуля чтения файла с раздела FAT32 завершим.

Исходные тексты модуля находятся в каталоге FAT32, файл fat32.c.

Отличия в организации хранения записей о файлах в каталогах для файловых систем FAT и EXT2

Несколько слов об отличиях в организации хранения записей о файлах в каталогах для файловых систем FAT и EXT2. Структура файловой системы EXT2 была рассмотрена в [3].

С FAT мы только что ознакомились – в ней все элементы каталога имеют фиксированную величину. При создании файла драйвер файловой системы ищет первую незанятую позицию и заполняет её информацией о файле. Если длина каталога не умещается в одном кластере, то под него отводится ещё один кластер и т. д.

Рассмотрим, как обстоят дела в EXT2.

Предположим, у нас есть раздел с файловой системой EXT2, размер блока равен 4096 байт. На этом разделе мы создаем каталог. Размер каталога будет равен размеру блока – 4096 байт. В каталоге операционная система сразу создаёт две записи – запись текущего и запись родительского каталогов. Запись текущего каталога займет 12 байт, в то время как длина записи родительского будет равна 4084 байта. Создадим в этом каталоге какой-нибудь файл. После этого в каталоге будут присутствовать три записи – запись текущего каталога длиной 12 байт, запись родительского каталога длиной уже 12 байт, и запись созданного файла длиной, как вы наверно догадались, 4072 байт. Если мы удалим созданный файл, длина записи родительского каталога опять возрастёт до 4084 байт.

Таким образом, при создании файла драйвер файловой системы EXT2 ищет в каталоге запись максимальной длины и расщепляет её, выделяя место для новой записи. Ну, а если всё-таки места не хватает, под каталог отводится ещё один блок, и длина каталога становится равной 8192 байт.

И в заключение – небольшая правка к статье «Архитектура файловой системы EXT2» ([3]).

Эта правка касается функции определения номера inode по имени файла get_i_num(). Старый вариант этой функции выглядел так:

```
int get_i_num(char *name)
{
  int i = 0, rec_len = 0;
  struct ext2_dir_entry_2 dent;

  for(; i < 700; i++) {
    memcpy((void *)&dent, (buff + rec_len), sizeof(dent));
    if(!memcmp(dent.name, name, dent.name_len)) break;
    rec_len += dent.rec_len;
  }

  return dent.inode;
}
```

Исправленный вариант:

```
int get_i_num(char *name)
{

/* Параметр функции - имя файла. Возвращаемое значение -
 * номер inode файла.
 */

  int rec_len = 0;
  // эта структура описывает формат записи корневого каталога:
  struct ext2_dir_entry_2 dent;

/* В глобальном буфере buff находится массив записей каталога.
 * Для определения порядкового номера inode файла необходимо
 * найти в этом массиве запись с именем этого файла.
 * Для этого организуем цикл:
 */

  for(;;) {

    /* Копируем в структуру dent записи каталога: */

    memcpy((void *)&dent, (buff + rec_len), sizeof(dent));

    /* Длина имени файла, равная нулю, означает, что мы
     * перебрали все записи каталога и записи с именем
     * нашего файла не нашли. Значит, пора возвращаться:
     */
    if(!dent.name_len) return -1;

    /* Поиск выполняется путем сравнения имен файлов.
     * Если имена совпадают - выходим из цикла:
     */

    if(!memcmp(dent.name, name, strlen(name))) break;

    /* Если имена не совпали - смещаемся к следующей записи:
     */

    rec_len += dent.rec_len;
  }

  /* В случае успеха возвращаем номер inode файла: */

  return dent.inode;
}
```

Литература:

1. В.Кулаков. Программирование на аппаратном уровне: специальный справочник. 2-е изд. / – СПб.: Питер, 2003 г. – 848 с.
2. А.В.Гордеев, А.Ю.Молчанов. Системное программное обеспечение / – СПб.: Питер – 2002 г.
3. В.Мешков. «Архитектура файловой системы ext2», журнал «Системный администратор», № 11(12), ноябрь 2003 г. – 26-32 с.