CprE 308 Laboratory 7: The FAT-12 Filesystem

Department of Electrical and Computer Engineering
Iowa State University

1 Submission

Turn in your lab report, well-formatted and commented source code, and a copy of the output, through Blackboard as a single tar-zip file.

2 Introduction

In this lab, you will gain hands-on experience reading a FAT-12 filesystem. Using information we provide, you will decode the boot sector by hand. You will then write a program to do this automatically.

2.1 Terms

Sector the smallest unit of transfer; It's also called a block. There are 512 bytes / sector on a floppy disk.

Boot sector stores vital information about the filesystem. The boot sector is laid out in the following way, starting at the beginning of the disk (logical sector 0, byte 0):

All values are stored as unsigned little-endian numbers unless otherwise specified.

Offset	Length	Contents	Display Format
0x00	3	Binary offset of boot loader	hex
0x03	8	Volume Label (ASCII, null padded)	ASCII
0x0B	2	Bytes / sector	decimal
0x0D	1	Sectors / cluster	decimal
0x0E	2	Reserved sectors	decimal
0x10	1	Number of FATs (generally 2)	decimal
0x11	2	Number of Root Directory entries	decimal
0x13	2	Number of logical sectors	decimal
0x15	1	Medium descriptor	hex
0x16	2	Sectors per FAT	decimal
0x18	2	Sectors per track	decimal
0x1A	2	Number of heads	decimal
0x1C	2	Number of hidden sectors	decimal

Table 1: The Layout of FAT-12 Boot Sector

2.2 Useful system calls

- `int open(const char* path, int flags)`

  Opens a file at path and returns a file descriptor. For example, `open("/tmp/myfile", O_RDONLY)` opens an existing file called myfile in the tmp directory in read-only mode. It returns a file descriptor that can be used like a handle to the open file.

- `ssize_t read(int fd, void *buf, size_t count)`

  Reads `count` bytes from the file descriptor `fd` into the memory location starting at `buf`. It starts reading from the current offset into the file. The offset is set to zero if the file has just been opened.

- `off_t lseek(int fd, off_t offset, int whence)`

  Sets the offset of the file descriptor `fd` to `offset`, depending on `whence`. If `whence` is `SEEK_SET`, then the offset is figured from the start of the file. If `whence` is `SEEK_CUR`, then the offset is relative to the current offset.

2.3 Inspecting binary files

You should have a floppy disk image from the course website and a program for dumping out the boot sector of the disk. In Unix, devices are treated as files. For example, the floppy drive is normally found at a device file such as `/dev/fd0`. We can use a file image as a substitute for an actual floppy disk. Your floppy image contains a real FAT-12 filesystem. Your floppy image contains a real FAT-12 filesystem; if you know how the bits are laid out, then you can decode the structures that describe the filesystem and work with the files. To help get you started, we're going to decode a few fields by hand.

We have supplied you with a program to read from the floppy disk. Given a filename and an offset (in decimal notation), it will print out 32 bytes of hex and ASCII values. For example, `./bytedump image 1536` will produce the following output with this particular floppy image: **Note that you have to give the offset to `bytedump` in the decimal format, not in Hex**.

```
$./bytedump image 1536
addr value ascii
0x0600 0x31 1
0x0601 0x36 6
0x0602 0x53 S
0x0603 0x45 E
0x0604 0x43 C
...
0x061e 0x00
0x061f 0x00
```

The first column is the address (in hexadecimal), and the second is the data at that address (also in hexadecimal). The third column is the ASCII value of the data, but only if it was fit for printing. We can see that `0x31` is the ASCII character '1', and `0x00` is not printable.

Note that data is stored in the little endian format. This means that the most significant byte comes last. For example, if we had the output

```
$./bytedump image 120
addr value ascii
0x0078 0x20
0x0079 0x50 P
```

We would know that `0x20` is stored at `0x0078`, and `0x50` is stored at `0x0079`. If we interpret this as a short value (2 bytes) starting at address `0x0078`, then the value is stored as `0x2050`, but it represents `0x5020`. Similarly, if we were storing an int (4 bytes), the bytes would be stored in reverse order, from the least significant byte to the most significant byte.

When reading data from the floppy disk into memory, we should be careful to understand the little endian format. For example, in reading a short value into memory, we should read two bytes from the file, and then reverse their order, before storing them into memory as a short value. A character is only a single byte, and hence there is no need for reversing the byte order if we read a single character from the file.

Alternately, there is a utility in unix called `hexdump` that will show the binary contents of a file. This is useful to inspect a file containing binary data. If you use `hexdump -C` on the supplied image, the output will look something like this (use `hexdump -C image | less` to more easily read it)

```
00000000  eb 3c 90 6d 6b 64 6f 73  66 73 00 00 02 10 01 00  |.<.mkdosfs......|
00000010  02 e0 00 40 0b f0 01 00  12 00 02 00 00 00 00 00  |...@............|
00000020  00 00 00 00 00 00 29 79  d6 c9 3d 20 20 20 20 20  |......)y..=     |
...
```

The first column is the hexadecimal offset into the file. Since each line contains 16 bytes, this will always end in a 0. The middle 16 columns are the hexadecimal bytes of the file. The third column contains the ASCII representation of the bytes, or a `.` if the byte is not a printable character. For example, the second byte, having offset `0x01` has value `0x3c` corresponding to the character `<`, while the third byte `0x90` is not printable, so is rendered as a dot.

3 Excercises

3.1 Decoding the boot sector by hand

Using the supplied program and the offsets in the tables above, find and decode the values for each of the following fields in the boot sector (remember that it starts at offset 0):

	Hex	Decimal
Bytes/sector		
Sectors/cluster		
Root directory entries		
Sectors/FAT		

Have the lab instructor check your answers. You will use these values for debugging the next part.

3.2 Decoding the boot sector

Create a program to read and then print information from the boot sector. You can use any programming language for this, although C is recommended as it more easily deals with binary data. The starting offset and size of each field can be found in Table 1. All of the information is (of course) stored as binary on disk. When you print the values, use the format specified in Table 1. Use `printf` to print out values correctly after converting any little endian data.

Call your new program bsdump (see the C program template `bsdump-template.c`). Your program should take the name of the file to read, and then decode and print out all of the values in the boot sector (you may skip the jump instruction).

4 Some tips to help maximize your grade

- Write separate functions to perform the following tasks:
  - A function for converting shorts from little to big endian.
  - A function for decoding the boot sector and storing the values in a struct.
  - A function for printing the boot sector from the struct.
- Use Bitwise operators:

```
<< left shift
>> right shift
& and
| or
```

- Your conversion function should take advantage of bit shifting to reverse the bytes. One suggestion is to take two chars and return a short. Restrict your solution to the bitwise operators (no multiplication or addition).
- Do not write a conversion function for binary to hex or hex to decimal. Let the format string in `printf` handle the dirty work. Format strings are described in "`man 3 printf`".
- Do not attempt to generalize the decoding process – simply write a function to decode a FAT12 boot sector one field after another. You shouldn't use any loops (except possibly for printing the name).
- Printing tips:
  - When you print out the values, make the values line up in a column (it's ok to hardcode spaces). For example, the following would be acceptable:

```
Sectors/cluster 37
# of FATs 2
Media descriptor 0xA8
```

  - Use "`0x`" to denote a hexadecimal value.
  - You can pad variables with zeros – see `bytedump`.
  - Be careful when printing the name – it can be 8 characters without a binary zero at the end, so you have to ensure that you never print more than 8.
  - Your variables will probably have to be declared "unsigned" to print correctly.
- A comment every 3-4 lines on average should be fine for algorithms. For declarations, please put a comment by each that explains what the variable or structure is used for. You might look at `bytedump.c` for an example of the style we're looking for.
- Give your variables meaningful names. Indexes are usually i, j, and k, but other variables should have more descriptive names.
- Avoid using global variables unless you really need them. Global variables can be modified anywhere in a program, making code difficult to read and to debug.
- When printing out your code, ensure that the lines do not wrap around. Format your source so it fits within the limits of the printer. Not doing so makes it hard to read which defeats the purpose of indenting.

4.1 Packed structures

The simplest way to read in the data from disk is to interpret it one byte at a time, and store the data in a structure. However, since structures in C are simply a definition of some binary data in memory, it is possible to define a structure that describes the FAT12 header.

The file `stdint.h` defines a number of useful types such as `uint16_t`, which is an unsigned integer. Integers are always stored in memory in the byte order of the system, so you do not need to convert between little and big endian if you use this.

Because several of the fields in the FAT12 header are not aligned to an even or odd byte boundary, the definition of a C structure may include padding to allow faster memory access. Since this is undesirable, you will need to add `__attribute__((__packed__))` after the structure definition. See http://sig9.com/articles/gcc-packed-structures or a C manual for more information.

There is no `uint24_t` type; the simplest way to represent a 3-byte field is by using an array of 3 `uint8_t` values. Alternatively, you can use a bitfield of length 24 – for an introduction to the bitfield feature of C structures, see http://publications.gbdirect.co.uk/c_book/chapter6/bitfields.html. Bitfields are rarely used in data structures as the slow access times due to compiler-generated bitmasking will outweigh the memory savings.

You may choose to either use a packed structure or to read the data byte by byte; you do not need to do both.