CS342 Spring 2017 – Project 5
Parsing FAT32 File System

Assigned: April 30, 2017
Due date: May 14, 2017, 23:55

Objectives: *Learning FAT32 file system in detail. Understanding on-disk file system structures. Practicing formatting and mounting a file system, and parsing on-disk file system structures.*

In this project you will explore the FAT32 file system and write a program that will read and and parse a FAT32 volume directly in raw mode. The project will be done in Linux and using C programming language. You can work in groups of 2 students each.

First, read and learn about FAT32 file system and its on-disk structures: its volume information structure, directory record structure, FAT structure, etc. You can find a lot of documentation about FAT32 in Internet. We recommend starting with [1]. Note that FAT32 terminology is using the term "*cluster*" instead of the term "*block*". A typical cluster size is 8 sectors, that means 4 KB. In this case, a cluster will occupy 8 consecutive sectors. A sector is 512 bytes long. A disk that has a file system on it is called a *volume*.

In this project you will work with a FAT32 volume that will not sit on a separate physical media but on a regular Linux file. Therefore, you first need to create a regular Linux binary file that will act as your disk, i.e. a virtual disk, with Linux dd command. The dd command will initialize the file to all zeros. Block size will be 4KB. You can create such a file with as many blocks as you wish, of course there is a limit, but in this example we will create a file that has 1024 blocks. Hence the size of the file will be 4KB x 1024 = 4 MB. The name of the virtual disk will be "vdisk".

dd if=/dev/zero of=vdisk bs=4k count=1024

Linux file vdisk will act as a disk: a virtual or pseudo disk. We can format it with FAT32 file system and mount it to an empty directory in our local Linux file system.

Now use the **losetup** command to associate a loop device file (/dev/loop0) with the virtual disk file. In this way you make the virtual disk file look like a block storage device instead of just a regular binary file within the file system [5].

losetup /dev/loop0 vdisk

Each device in a Linux computer has a device file associated with it. A device file starts with the /dev prefix. After running losetup command as above, the virtual disk file is now threated as a block oriented storage device that can be mounted with the mount command. To delete the association between the loop device file and the virtual disk file, we can use the –d option of losetup as: loosetup –d vdisk.

You can now create a FAT32 file system in your vdisk file. This is called (logical) formatting the disk. You can use the mkfs.fat command of Linux for this purpose, which may take several options. Run the command with options given in the example below. We will always use the same options in our tests. You are suggested to read the man page of mkfs.fat or mkdosfs commands.

mkfs.fat -a -v -S 512 -s 8 -F 32 -n CS342 vdisk

Above, the –F option says that the file system will be FAT32, not FAT16 or FAT12, -s 8 options says that the block size will be 8 sectors, that means 4KB, -S 512 says that a logical sector is 512 bytes long, and -n option gives a name for your

volume. Hence after creating the file system, you now have a FAT32 volume which you can mount and use; or you can write a program to analyze the volume directly without mounting.

Create a mount point, that means an empty directory with the mkdir command. The name of the empty directory can be, for example, "myfat32fs". Then mount your volume on this mount point. Read the man page of the mount command.

sudo **mount** -t msdos /dev/loop0 myfat32fs

You can now change into myfat32fs directory. That means you are changing into the root directory of the FAT32 volume you mounted. There you can create and use files with any tools that you like: touch command, dd command, cp command, an editor, or something else. You can also create subdirectories.

Now, unmount your volume.

sudo **umount** vdisk

You can now analyze the virtual disk file. You will write a program, called fatfs.c, that will read and parse it directly one byte at time, or 512 bytes at a time (one sector at a time), or 4096 bytes at a time (one block at a time). The executable will be called **fatfs**.

As said, your fatfs program will read the virtual disk file block by block, or sector by sector and will parse the file system structures directly. That means you will be accessing your virtual disk in raw mode in your program. In our test cases, we will have just have one directory, i.e., a root directory, in a FAT32 virtual disk. There will be no subdirectories and the number of files in the root directory will not exceed 50. Hence the root directory will be fitting into a single block. You can assume that only *short filenames* will be used. A short filename has at most 8 characters for the filename and 3 characters for its extension. "project.pdf", for example, is a short filename. We will not use long filenames.

The program will be invoked in one of the following ways, doing the specified things:

- **fatfs <vdiskfile> -p volumeinfo**: Print information from the first sector (VolumeID sector) of the FAT32 file system. Printed information will include some numbers, like the number of sectors that FAT occupies.
- **fatfs <vdiskfile> -p rootdir**: Print the content of the root directory. For each directory record you will print the filename, its extension, and also other information found in the record, as nicely as possible. You will use one output line per record.
- **fatds <vdiskfile> -p blocks <filename>**: Print the numbers of the clusters, i.e., blocks allocated to file <filename>. If file size is 0, nothing will be printed. One number per line will be printed.

Sample invocations of your program can be as follows:

- **fat32** vdisk –p volumeinfo : Print info from the volumeID sector.
- **fat32** vdisk –p rootdir : Print info about root directory content.
- **fat32** vdisk –p blocks x.bin : Print numbers of the blocks of the file with name "x.bin".

You will be provided an example virtual disk, called vdisk, in the webpage of the course. It is a regular Linux binary file. It acts as a FAT32 volume of size 4MB. That is like a 4MB disk formatted with FAT32. In that we have some files created and sitting. As an exercise you can start with parsing this file. You can see the content of the vdisk file in hex using the command xxd.

xxd vdisk

Submission:

Submit through Moodle. You will do a demo and you will be asked very detailed questions.

Clarifications:

--- In boot (volumeID) sector, Starting at offset 36 (in decimal), you need to read 4 bytes to obtain the Sectors per Fat information.

--- Note that integers (4 bytes or 2 bytes) stored in vdisk are in little-endian order; that means the byte at offset 36 is the least significant byte (LSB) of a 4-byte integer value, the byte at offset 39 is the most significant byte of the 4-byte (32 bit) integer value (do not worry about the bit order in a byte).

--- You can use memcpy to read 4 bytes (size of integer) from an array into an integer variable. For example:

```
unsigned int x;
unsigned char sector[512];
memcpy ( (void *) &x, (void *) (sector + 36), 4);
```

Since Intel machines also use little-endian byte order, memcpy above will obtain an integer value into variable x from the 4 bytes of the sector starting at offset 36.

--- You can also use a structure mapping the content of the boot sector in to the fields of the structure properly. Such a structure is defined in msdos_fs.h. It is struct fat_boot_sector. That means:

```
unsigned char sector[512];  // assume sector stores boot sector data
struct fat_boot_sector *bsp;
bsp = (struct fat_boot_sector *) sector;
```

Then access the fields if the structure like below:

```
bsp->afieldname
```

Learn the field names from the msdos_fs.h file.

--- Try to use unsigned integers and characters for various numbers stored in the boot sector.

--- You don't have the use the msdos_fs.h file in your program. If you want, you can use it, but you don't have to. You can define your own macros and structures instead of the ones available in msdos_fs.h header file.

--- A **project skeleton** is availabe at github: https://github.com/korpeoglu/cs342-spring2017-p5.git

--- FAT32 uses litle-endian format in storing integers in on-disk structures like FAT table. That means the least significant byte is stored first. If your machine is big-endian, you need to be extra careful. Intel x86 architecture is little-endian.

--- FAT32 normally installed on disks that have 65525 clusters at least. If block size is 4KB, that means the disk should be 256 MB at least. Hence we may get a warning in

FAT32 installations on smaller volumes like a volume having 1024 clusters. This should not be a problem in this project.

--- The number of reserved sectors is 32 by default. Hence in the given vdisk file, the number of reserved sectors is 32. Then starts FAT. That means after the first 32 sectors of the virtual disk, the FAT starts. Note that we have 2 FATs stored. After FATs we have clusters (blocks) storing the data of files and directories. For example, the first block (cluster) after the second FAT is storing the root directory records. Record size is 32 bytes. Hence each sector can store 16 directory records, i.e., directory entries.

--- The first sector is the boot sector and volumeID sector in the disk that we provide. This is because the disk we provide is not a bootable disk. Hence the first sector (sector 0) also containes volumeID information. Hence you only need to parse sector 0 to obtain important file system information, like for example, the number of sectors that FAT is occupying.

References:

[1]. Understanding FAT32 File Systems.
http://www.pjrc.com/tech/8051/ide/fat32.html.

[2.] FAT32 File Systems Specification.
https://msdn.microsoft.com/en-us/library/gg463080.aspx

[3]. Design of FAT Filesystems.
https://en.wikipedia.org/wiki/Design_of_the_FAT_file_system

[4]. FAT file systems.
http://wiki.osdev.org/FAT

[5]. Anatomy of the Linux File system.
http://www.ibm.com/developerworks/linux/library/l-linux-filesystem/