Operating Systems '15 - CS 323
Assignment #4

FAT32 Filesystem

April 23, 2015

1 Objectives

1. Learn basics of VFS APIs, especially about FUSE, Filesystem in USErspace
2. Learn the structure of FAT (File Allocation Table), the most widely used file system to date

2 Introduction

In this assignment, you will develop a read-only file system driver for the FAT32 file system. FAT is very widely used these days and is present in most consumer electronics devices which offer storage. For example, USB keys or SD cards come with FAT32 format, and most MP3 players also use the FAT32 file system. During the exercise, your are encouraged to try your file system driver on your own FAT32 media.

There exist as many file system APIs (the so-called VFS API) as there are different operating system kernels. In this exercise you learn how to use FUSE(Filesystem in USErspace) API. FUSE allows the development and execution of file system drivers in userland instead of the kernel space. The advantage of this approach is that less code is run in the kernel, which reduces the risk of crashes or exploits. Since FUSE presents a simple, abstracted file system interface, it is easily possible to use FUSE file system applications on different operating systems, such as Linux, FreeBSD, Mac OS, OpenSolaris or even Windows. Running in userland also enables the use of various language bindings, allowing the implementation of FUSE file systems in different progamming languages such as C, C++, Java, C#, Python, Perl, OCaml, etc.

3 Assignment

Your goal is to write a **read-only** file system driver for **FAT32**. The implementation has to support the `getattr`, `readdir`, and `read` FUSE file system operations. You need to return information as precise as possible, in particular deal with **long file names**, decode correct timestamps and file permissions. Your submission does not have to be able to run as multi-threaded FUSE application. You should use the skeleton implementation provided by us

and you should use C programming language for the assignment.

As usual, please make sure that we can compile and run your submission in VMchecker. You need to include all relevant files in your submission (Makefile, .c and .h files).

4 The FAT file system

As primary source, please refer to the official Microsoft specification. Because the specification is sometimes hard to read without previous context, we encourage you to look also Wikipedia page on FAT and other available resources. You should also have a look at existing FAT32 file systems with a hex editor (of course not limited to our sample file system), *e.g.*:

```
> hd testfs.fat | less
```

5 Running and testing

5.1 Development environment

To build the code, you will need the FUSE library in userland. On a Linux distribution that supports deb packages, such as Debian and Ubuntu, you can install it by the following command:

```
> sudo apt-get install libfuse2 libfuse-dev
```

5.2 Mounting

The supplied skeleton already performs basic FUSE argument parsing. To mount the file system, run the application with file system and destination mount point as arguments:

```
> mkdir dest
> ./vfat -s -f -odirect_io ./testfs.fat dest
> (other console) ls -l dest
```

The argument `-s` specifies single-threaded operation, `-f` instructs FUSE to stay in the foreground and not to place itself in the background (which is extremely useful for debugging). Finally, option `-odirect_io` tells the fuse not to cache anything (not using direct I/O can sometimes mask your programming errors so we recommend you should test your program with this option enabled).

5.3 Unmounting

In case your file system driver does not shutdown properly (*e.g.*, segfaults), you might experience an error message, such as

```
ls: cannot access dest: Transport endpoint is not connected.
```

In this case you will have to unmount the file system manually:

```
> fusermount -u -z dest
```

5.4 Testing

You can create new, empty FAT file systems using `dd` and `mkdosfs`:

```
> dd if=/dev/zero of=newfs.fat bs=1M count=33
> (sudo) mkdosfs -F 32 newfs.fat
```

We also provide a compressed sample file system to test against: `testfs.fat`. Bear in mind that this file will expand to 500MB when extracted[1] . You can test your implementation with standard mount by comparing the structure of the mounted sample file system.

```
> sudo mount testfs.dat dest2 -o ro
```

[1] Note: check the file size after expanding, the file is sparse and some archive managers (*e.g.*, Midnight Commander) do not unpack it properly

```
> diff -r dest/ dest2/
```

You are also encouraged to use your own FAT32 devices as test file system. For example, you could try to list the contents of your digital camera, or listen to music stored on your MP3 player. As long as you open the device read-only, your program will not be able to destroy your data.

6 References

FUSE file system operations http://fuse.sourceforge.net/doxygen/structfuse˙operations.html
Official FAT specification http://msdn.microsoft.com/en-us/library/gg463080.aspx
FAT at WikiPedia http://en.wikipedia.org/wiki/File˙Allocation˙Table
FAT for microcontrollers http://www.pjrc.com/tech/8051/ide/fat32.html
Microsoft Technet How FAT works http://technet.microsoft.com/en-us/library/cc776720(v=ws.10).aspx

7 Deliverables

Deliverables for this project are:

- Full source code for the assignment
- MANDATORY: Status of your implementation (what you implemented and works, what does not work) so we know where to focus on during the grading

You should submit these deliverables as a single zip archive . The deadline for this assignment is Thursday, 23:59 CET, May 21, 2015.

8 Hints and notes

Implementing a filesystem is hard and tedious task. We therefore encourage you to start early and work on a small piece of functionality at a time. In particular, we recommend using the following implementation sequence with proper testing between moving to the next stage:

1. Read the first 512 bytes of the device
2. Parse BPB sector, read basic information (sector size, sectors per cluster, FAT size, current FAT id, etc.) and verify the filesystem is FAT32. Export this information via `.debug/` (the code skeleton contains helpers for this)
3. Locate and read raw data from the correct FAT table (implement `vfat_next_cluster`), then make sure you can follow cluster allocation chains.
4. Implement basic root directory parsing make sure you can enumerate raw entries from the root directory until the final entry is indicated
5. Implement basic short entry handling parse short name (remove space-padding), attributes, size, first cluster, skip all other entries
6. Add an ability to read the content of a file and a directory given the number of their first cluster.

7. Now you can start integrating with fuse make sure you can list root (e.g., "/") directory and fill stat entries correctly (especially attribute `S_IFREG` and `S_IFDIR`). Make sure you can read top-level files

8. Add multi-level directory resolution (implement `vfat_resolve`)— traverse directory structure (i.e., handle "/dir1/dir2/file")

9. Add support for long names. You will need to keep some state before you finally read the short entry holding the information about the file and you should check that long name entries are correct.

10. Fill other stat fields (atime/mtime/ctime)

8.1 Other notes

- You may assume that we will ignore DST shift. More precisely, you do not need to compensate for daylight saving time change between the date on which the file was saved and the current date.

- As we do not have a way to test your code on non-standard architectures, you may safely assume that the machine we will use for grading will be using little-endian. You therefore do not need to use endian-conversion functions in your code.