(Best-Effort) File Recovery in FAT32

2013-2014 CENG 3150 - Programming Assignment 2

Specification version 1.0, Nov 6, 2013.

Abstract

Recovering deleted files can be a business, a profession, or just a fun experience. Nevertheless, why does it seem to be so hard to get it done? In this assignment, We will explore this problem by writing a file recovery tool.

Contents

1 Introduction

The FAT32 file system is one of the file systems adopted by Microsoft since Windows 95 OSR2. Because of its simplicity, this file system is later adopted in various system / devices, e.g., the USB drives, SD cards, etc. Hence, knowing the internals of the FAT32 file system becomes **essential**.

In this assignment, you are going to implement a partial set of the FAT32 file system operations.

You are supposed to learn the following set of hard skills from this assignment:

- Understand the good, the bad, and the ugly of the FAT32 file system.
- Learn how to operate on a device formatted with the FAT32 file system.
- Learn how to write a C program that operates data in a byte-by-byte manner.
- Understand the alignment issue when you are operating with structures in C.

The one and only one soft skill that you must have is: **time management**. *How to manage the time with more than one assignment deadlines when we are close to the end of the semester?*

1.1 Accessing FAT32 without kernel support

In this assignment, you are going to work on the data stored in the FAT32 file system **directly**. The idea is shown in Figure 1. Since the concept of reading and writing the disk device file may be new to you, a brief comparison between the old way and the Assignment-2 way is provided.

Scenarios under a normal process

- **Open**. When a normal process opens a file, the operating system processes its request. The `open()` system call provided by the OS will check whether the file exists or not. If yes, the OS will create a structure in the kernel which memorizes vital information of the opened file.

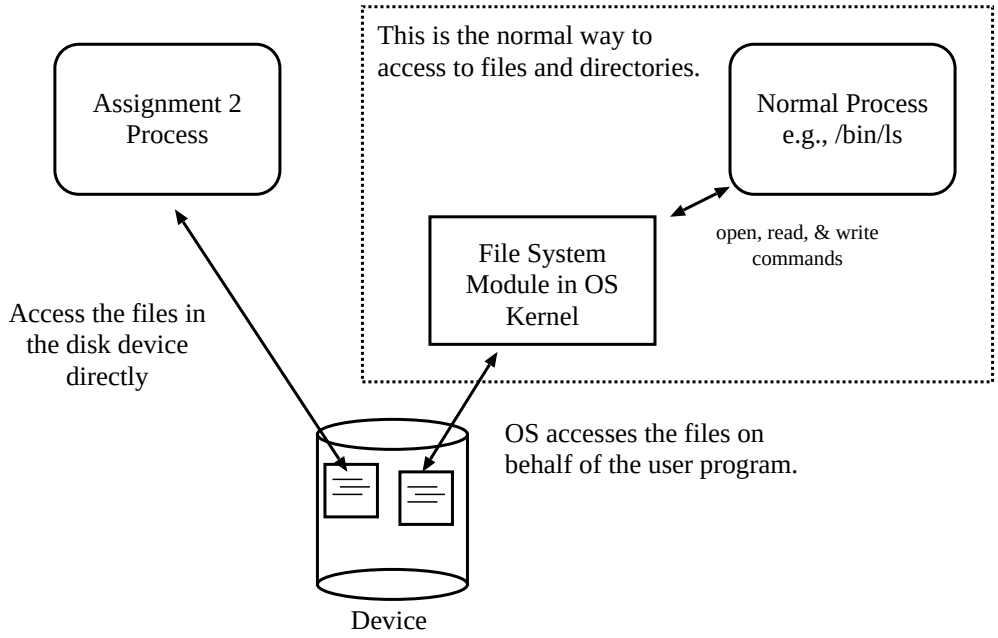

Figure 1: The theme of this assignment is to access to the disk directory. In other words, you will be implementing the jobs done by the file system module in the OS kernel.

- **Read**. When a normal process reads an opened file, the operating system performs the lookup of the data clusters. Then, the kernel replies the process with the read data.
- **Write**. When a normal process writes an opened file, the operating system performs the allocation of the data clusters. Then, the kernel replies the process with the number of bytes of data written.

Scenarios under an Assignment-2 process

- **Open**. The Assignment-2 process will not open any files inside the device, but opening the **device file** itself. Remember, a device file is just a representation of the device. By opening the device file, it means the process is going to access the device directly, addressed in terms of bytes.
- **Read and Write**. When an Assignment-2 process reads/writes a file, the process locates and reads/writes the required data (or data clusters) by accessing the disk device directly. Of course, you need to invoke `open()`, `read()`, `write()`, `close()`, etc. in order to work with the device file.

Note that the only file that the Assignment-2 process should open is the device file, in addition to the standard I/O streams.

1.2 Goal of this assignment

In this assignment, you are going to open the device file, which is described above. Then, it searches for deleted files and recovers it. Nevertheless, you are not going to recover all deleted files, but just the file specified by a filename provided by the user.

Therefore, you are going to implement a file recovery tool with restrictions, and the restrictions are so follows:

- **A filename must be given**. The tool is not going to recover all deleted files. Based on the input filename, the tool looks for that deleted file in the device file.
- **Traversing root directory only**.

  It would become complicated if you have to look into all directories in order to look for the file needed. In this assignment, you are required to recover files in the root directory only. Note importantly that the file system contains one layer of directories only.
- **Files with one cluster only**. In the lecture, we discussed the difficulty to recover a file containing more than one cluster. Therefore, in this assignment, you are going to recover a file with **one cluster only**.

2 Milestones

You are required to submit only one C/C++ program, which is allowed to contain more than one source file. Suppose the executable of the program is called "`recover`". Such a program is restricted to be **running on the Linux operating system only**.

Note very important that, in most Linux distributions, the program "`mount`" requires the "`root`" privilege to invoke. In other words, in this assignment, **you must use the Linux virtual machine**. Note importantly that you could never use any Linux workstations provided by our department to run this assignment since you are not the root user of those workstations. By the way, we will use the distribution: 32-bit Ubuntu Linux 12.04 to grade your assignment. Please make sure that you are using the same development environment.

2.1 Milestone 1 - Detecting valid arguments

The program "`recover`" should take a set of program arguments.

Sample Screen Capture #1

```
root@linux:~# ./recover
Usage: ./recover -d [device filename] [other arguments]
-i                    Print boot sector information
-l                    List all the directory entries
-r filename [-m md5]  File recovery with 8.3 filename
-R filename           File recovery with long filename
root@linux:~# _
```

"*Sample Screen Capture #1*" shows the set of program arguments required, or we call it "**usage of the program**". If the requirements are not met, the above output will then be shown. For the sake of our marking, please print the output to the **standard output stream** (stdout).

On the other hand, the above list of arguments required the existences of the **device file** and therefore must be presented. Other arguments are specifying the executions of different milestones.

- "`-d [filename]`". It is an *filename* to a device containing a FAT32 file system. It can be an image file, which is just an ordinary file, or a device file. You can always assume

that the input file is always a device file in the FAT32 format. You can also assume that the filename provided always refer to a valid device file.

- "`[other arguments]`". There can be 5 possible combinations, either:

`-i`	`-l`	`-r filename`
`-r filename -m md5`	`-R filename`	

But, the above 3 cases must not appear together in one command line. E.g.,

```
root@linux:~# ./recover -d fat32.disk -i -l
```

is **wrong**. Then, the program should print the usage of the program.

- Last but not least, the order between "the device file" and "the other arguments" is not important. That means:

```
root@linux:~# ./recover -i -d fat32.disk   ## correct
root@linux:~# ./recover -d fat32.disk -i   ## correct
root@linux:~# ./recover -d -i fat32.disk   ## wrong
```

2.2 Milestone 2 - Printing file system information

In Milestone 2, you need to print the following information about the FAT32 file system.

Sample Screen Capture #2

```
root@linux:~# ./recover -d fat32.disk -i
Number of FATs = 2
Number of bytes per sector = 512
Number of sectors per cluster = 8
Number of reserved sectors = 32
Number of allocated clusters = 1000
Number of free clusters = 8000
root@linux:~# _
```

Hints.

- Keep in your mind that you should **never hardcode the above results**.
- For the first four pieces of information, they are available in the boot sector.
- For the last two pieces of information, they are not available in the boot sector. You have to count the numbers of the allocated and the free clusters, respectively, inside the FAT.

2.3 Milestone 3 - Listing all directory entries

Milestone 3 is doing a similar job as the command "`ls -l`". However, you are required to print a different set of data out, as shown in "*Sample Screen Capture #3*". Every directory entry should be printed in a row-by-row manner.

Sample Screen Capture #3

```
root@linux:~# ./recover -d fat32.disk -l
1, MAKEFILE, 21, 11
2, BEST.C, best.c, 4096, 10
3, TEST.C, test.c, 1023, 12
4, HELLO.MP3, hello.mp3, 4194304, 14
5, TEMPOR~1/, temporary/, 0, 100
6, THISIS~1.TXT, this is a LFN entry.txt, 1000, 19
root@linux:~# _
```

- The order of the printout must be the same order as the directory entries listed in the root directory.
- For each entry, the output should be the following format:
  1. the **order number**, followed by a comma and a space character;
  2. the **8.3 filename**, followed by a comma and a space character;
  3. the **long filename**, if any, followed by a comma and a space character;
  4. the **file size** in bytes, followed by a comma and a space character;
  5. the **starting cluster number**, followed by a newline character.

  E.g., the first file "`MAKEFILE`" does not have its long filename. Therefore, you should print the 8.3 filename only.
- The output only shows existing files. You should never print out deleted entries.
- In case you meet a sub-directory entry in the root directory, then:
  - You have to add a trailing character '`/`' to the end of directory name. Then, print out the information of the sub-directory.
  - You do not need to go into the sub-directory.

2.3.1 Filename: 8.3 or LFN?

You may notice that in the above output, nearly all filenames are all in upper-case. You may wonder what the reason(s) is(are). The issue is about the **long filename (LFN) support** in FAT32. Before FAT32, the filename is restricted to the so-called **8.3 filename format**. The long filename support introduces strange directory entries and we want to avoid that.

Under Linux, two conditions have to be satisfied in order to guarantee a filename is stored as the 8.3 format.

---

1. **Set of characters in the filename**.

   - uppercase alphabets,
   - digits, and
   - any of the following special characters:

     `$ % ' ` - { } ~ ! # ( ) & _ ^`

   Other characters are considered to be invalid and should be avoided, e.g., '`/`', '\', the space character, '`:`', etc.

2. **Length and format of the filename**.

   - The filename contains one '`.`' (0x2E) character and the '`.`' character is not the first character in the filename; and
   - The number of characters before the '`.`' character is between 1 and 8; and
   - The number of characters after the '`.`' character is between 1 and 3.

Therefore, LFN entries may exist and your program has to skip them.

Note that the above restrictions apply to files in the FAT32 file system, but not other files.

2.3.2 How to guarantee 8.3-format filename?

While we are using the FAT32-formatted disk, you may use the following kinds of filenames in order to guarantee the use of 8.3 format.

- The characters used in the filenames include upper-case characters, lower-case characters, and digits.
- If the filenames carry file extensions, then
  - The number of characters before the dot character is [1,8] (i.e., between 1 and 8 inclusively).
  - The number of characters after the dot character is [1,3].
- If the filenames carry no file extensions, then:
  - The number of characters in the filename is [1,8].
  - The dot character must be absent.

2.3.3 How about long filenames?

Names that do not follow the above rules are considered as long filenames (LFNs). E.g.,

"`hello.docx`", "`hello world.txt`", etc.

In this assignment, we will create filename with ASCII characters only. Therefore, in a LFN entry, you are guaranteed that, for each two-byte character in a LFN entry,

- The most significant byte is zero.
- The least significant byte is usually non-zero.
- If both bytes are zero, that means the end of a long filename.
- Note that the longest filename in FAT32 is 255 two-bytes characters.

2.4 Milestone 4 - Recovering one-cluster-sized files

In this milestone, you are required to recover **a file that occupies one cluster only**. It is illustrated in "*Sample Screen Capture #4*".

Sample Screen Capture #4 (1 of 2)

```
root@linux:~# ./recover -d fat32.disk -l
1, MAKEFILE, 21, 11
2, BEST.C, best.c, 4096, 10
3, TEST.C, test.c, 1023, 12
4, HELLO.MP3, hello.mp3, 4194304, 14
5, TEMPOR~1/, temporary/, 0, 100
6, THISIS~1.TXT, this is a LFN entry.txt, 1000, 19
root@linux:~# mount -o fat32.disk tmp
root@linux:~# ls tmp/
best.c       hello.mp3   MAKEFILE
temporary/   test.c      this is a LFN entry.txt
root@linux:~# /bin/rm tmp/MAKEFILE
root@linux:~# ls tmp/
best.c       hello.mp3
temporary/   test.c      this is a LFN entry.txt
root@linux:~# umount tmp
root@linux:~# ./recover -d fat32.disk -l
1, BEST.C, best.c, 4096, 10
2, TEST.C, test.c, 1023, 12
3, HELLO.MP3, hello.mp3, 4194304, 14
4, TEMPOR~1/, temporary/, 0, 100
5, THISIS~1.TXT, this is a LFN entry.txt, 1000, 19
root@linux:~# _
```

Sample Screen Capture #4 (2 of 2)

```
root@linux:~# ./recover -d fat32.disk -r MAKEFIL
MAKEFIL: error - file not found
root@linux:~# ./recover -d fat32.disk -r MAKEFILE
MAKEFILE: recovered
root@linux:~# ./recover -d fat32.disk -l
1, MAKEFILE, 21, 11
2, BEST.C, best.c, 4096, 10
3, TEST.C, test.c, 1023, 12
4, HELLO.MP3, hello.mp3, 4194304, 14
5, TEMPOR~1/, temporary/, 0, 100
6, THISIS~1.TXT, this is a LFN entry.txt, 1000, 19
root@linux:~# mount -o fat32.disk tmp
root@linux:~# cd tmp
root@linux:~/tmp# ls
best.c       hello.mp3   MAKEFILE
temporary/   test.c      this is a LFN entry.txt
root@linux:~/tmp# _
```

Requirements

- The filename in this milestone is of 8.3 format. In order to match the characters in the directory entries, we will provide filename with upper-case characters.
- The error message:

  `[filename]: error - file not found`

  is printed when the name provided by the user does not match any one of the deleted directory entries. Note that the error should be written to the standard output stream.
- The error message:

  `[filename]: error - fail to recover`

is printed when the cluster originally belonged to the deleted file is occupied. Such a checking can be done by inspecting the FAT. Note that the error should be written to the standard output stream.

- The message

  `[filename]: recovered`

  is printed when the input filename can be found in the file system. Note that this message should be written to the standard output stream.

  Note also that in this milestone, we will input names of deleted files which were in the root directory only.

- The set of outputs shown using "`./recover -d fat32.disk -l`" suggests that the order of the files inside the directory should be **preserved** after the recovery.

Suggested steps

1. Check if the filename provided by the user contains the dot character.

   - If yes, it is assumed that:
     - the filename always contains **at least one character and at most eight characters** before the '.' character, and we called this the "*name part*" of the filename;
     - the filename always contains **at least one character and at most three characters** after the '.' character, and we called this the "*extension part*" of the filename.
   - Else, it has no extension and it is considered as a valid filename with an empty extension part. It is assumed that the filename will contain **at least one character and at most eight characters** in the *name part*.

2. For each directory entry in the root directory,

   (a) See if the filename starts with `0xe5`. If yes, it is a deleted entry. Else, continue with the next directory entry.

(b) Match both the *name part* and the *extension* part of the filename stored in the directory against those provided by the user, respectively.

- For the *name part*, the matching process starts with the **second** character.
- For the *extension part*, an exact matching will be performed.

If they do not match, continue with the next directory entry.

Note that if the length of the *name part* stored in the directory entry is fewer than eight characters, then the *name part* will be filled with space (`0x20`) characters until the length reaches 8 characters. E.g., if the *name part* is four-character long, then 4 space characters will be followed. The same case happens for the *extension part*: space characters will be filled until the *extension part* reaches 3 characters.

The following figure gives an illustration in representing the filename "`TEST.C`" inside the directory entry.

← *Name Part* →								← *Extension Part* →		
0x54	0x45	0x53	0x54	0x20	0x20	0x20	0x20	0x43	0x20	0x20
T	E	S	T					C		

If such an entry is deleted, the first character will be marked as `0xE5`. For example, if "`TEST.C`" is deleted, then the entry is updated as follows.

0xE5	0x45	0x53	0x54	0x20	0x20	0x20	0x20	0x43	0x20	0x20
?	E	S	T					C		

For more details, please refer to the lectures and the tutorials.

---

3. If a matched entry is found, then continue with the checking if the original cluster is occupied by inspecting the FAT.

4. If the target cluster is not occupied, then change the filename of the directory entry to the filename supplied by the user. Else, report an error message: "`error - fail to recover`" to the `stdout` stream.

5. Also, report an error message: "`error - file not found`" to the `stdout` stream when all directory entries in the root directory have been processed.

Note that you are **not required to update the `FSINFO` structure** in the device file after the recovery is completed. The reasons are:

- Most operating systems do not care about the values stored in the `FSINFO` structure.
- This simplifies your implementation.

In further milestones, you are not required to update the `FSINFO` structure.

2.5 Milestone 5 - Detecting ambiguous file recovery request

In Milestone 4, we can always recover the target file if it is found. However, when there are multiple deleted directory entries with identical names, then the program does not know how to proceed.

Under such a case, the program should report the following message to the STDOUT stream and the program should then terminate:

```
[filename]:  error - ambiguous
```

"*Sample Screen Capture #5*" illustrates such a scenario. Let us assume that we are using the original disk image "`fat32.disk`".

Sample Screen Capture #5

```
root@linux:~# mount -o fat32.disk tmp
root@linux:~# cd tmp
root@linux:~/tmp# /bin/rm *.c
root@linux:~/tmp# ls
hello.mp3   MAKEFILE   temporary/
this is a LFN entry.txt
root@linux:~/tmp# cd ..
root@linux:~# umount tmp
root@linux:~# ./recover -d fat32.disk -r TEST.C
TEST.C: error - ambiguous
root@linux:~# _
```

2.6 Milestone 6 - Disambiguity: using long filename

As a matter of fact, the long filename structures are not removed after the corresponding file is deleted. With the similar set of delete actions taken in regular directory entries, the first byte of a LFN directory entry will be changed to `0xE5`. Therefore, the entire long filename is still in the file system! Let us show what you should do in "*Sample Screen Capture #6*".

Sample Screen Capture #6

```
root@linux:~# mount -o fat32.disk tmp
root@linux:~# cd tmp
root@linux:~/tmp# /bin/rm *.c
root@linux:~/tmp# ls
hello.mp3   MAKEFILE   temporary/
this is a LFN entry.txt
root@linux:~/tmp# cd ..
root@linux:~# umount tmp
root@linux:~# ./recover -d fat32.disk -r TEST.C
TEST.C: error - ambiguous
root@linux:~# ./recover -d fat32.disk -R test.c
test.c: recovered
root@linux:~# mount -o fat32.disk tmp
root@linux:~# cd tmp
root@linux:~/tmp# ls
hello.mp3   MAKEFILE   temporary/
test.c      this is a LFN entry.txt
root@linux:~/tmp# _
```

Note importantly that:

- The long filenames are case-sensitive.
- When you input a long filename in your command prompt, you should better use double-quotes to enclose the name. E.g.,

```
./recover -d fat32.disk -R "this is a file.txt"
```

2.7 Milestone 7 - Disambiguity: using MD5 checksum with 8.3 filename

Another way to solve the problem stated in Milestone 5 is using **MD5 checksum** to identify which deleted file is the target file.

In short, a MD5 checksum is a piece of 128-bit data representing a file (or other forms of data). The MD5 checksum guarantees that the probability that two different files have the same MD5 checksum value is extremely small. Therefore, it is always convenient to say that "*two identical files generates the same MD5 checksum*".

Based on the mentioned property, when we face an ambiguous file recovery request, we use the checksum value of the content of the target file as the key to identify which directory entry should be the target. "*Sample Screen Capture #7*" shows the example invocation.

Sample Screen Capture #7

```
root@linux:~# ./recover -d fat32.disk -r TEST.C -m "......"
TEST.C: recovered with MD5
root@linux:~# mount -o fat32.disk tmp
root@linux:~# cd tmp
root@linux:~/tmp# ls
hello.mp3   MAKEFILE   temporary/
test.c      this is a LFN entry.txt
root@linux:~/tmp# _
```

Note that "`......`" is a string representing the MD5 checksum value.

Suggested steps

1. For each directory entry that are found to be a candidate of the target file, read the content of the deleted file up to the file size specified.
2. Generate the MD5 checksum of the file content.
3. If the MD5 checksum matches the one supplied by the user, treat this directory entry to be the target one and recovery it with the steps similar in Milestone 4.

4. Otherwise, if none of the suspected directory entries return an exact MD5 checksum supplied by the user (and the user may do this *on purpose*), then the program should report "`error - file not found`" to the `stdout` stream.

5. Last, the program terminates.

2.8 Some general hints

- Beware of empty files.
- Every file that you are asked to recover is not larger that one cluster.
- You are not required to recover directory files, just regular files.
- The root directory may span across more than one cluster.
- You are suggested to "`mount`" the device file after an invocation of the program in order that you can check the correctness of your work.
- For the output format,
  - At the end of every message, there is no punctuation.
  - No tab characters were used.
  - Note that we will first use judge programs to grade your assignments. If any discrepancy is found, we will grade your assignment manually.
- According to our debugging experiences, many students forgot writing changes from the memory to the disk before terminating the program.

3 Mark Distributions

This assignment is a group-based assignment and the mark distribution is as follows.

Milestone 1 - Detecting valid arguments	5%
Milestone 2 - Printing file system information	5%
Milestone 3 - Listing all directory entries	15%
Milestone 4 - Recovery one-cluster-sized files	20%
Milestone 5 - Detecting ambiguous file recovery request	5%
Milestone 6 - Disambiguity: using long filename	25%
Milestone 7 - Disambiguity: using MD5 checksum with 8.3 filename	25%
Total	100%

Submission

For the submission of the assignment, please refer to our course homepage:

`http://www.cse.cuhk.edu.hk/ceng3150/`

Deadline: 23:59, December 6, 2013 (Fri).

Change Log

Time	Affected Version(s)	Details
2013 Nov 6	NIL	Release version 1.0.

–END–