(Best-Effort) File Recovery in FAT32

COMP3060- Programming Assignment2

Due:23:59,Dec16, 2012.

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.

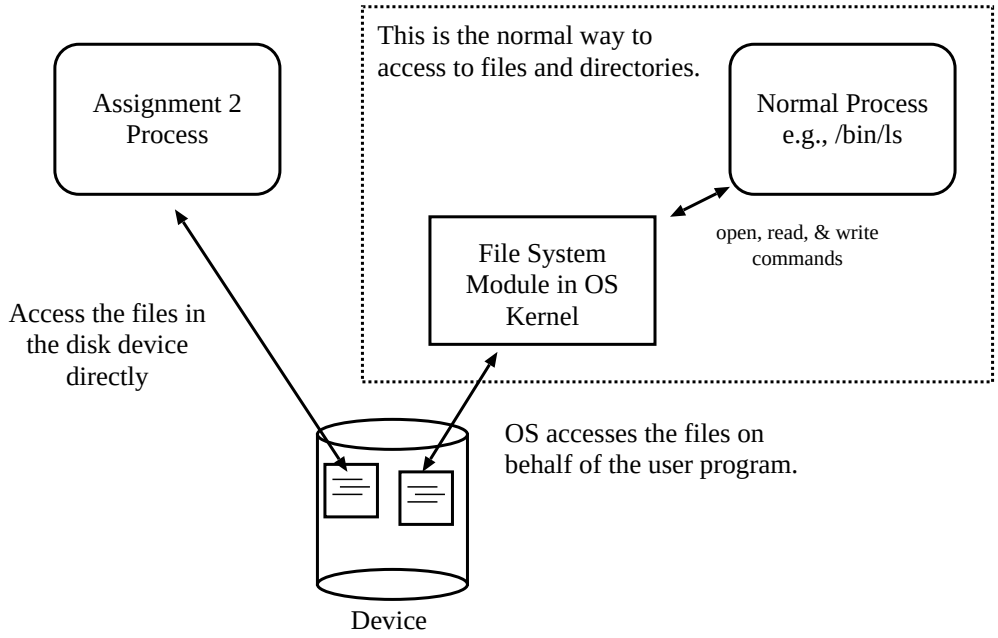

Figure 1: The theme of this assignment is to access to the disk directory. In other words, you will be implementing the jobs done by the file system module in the OS kernel.

- **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 asfollows:

- **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 directories**.

  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 on the root directory as well as the sub-directories under the root directory. Note importantly that the file system contains two layers of directories only.
- **Number of clusters may be more than one**. In the lecture, we discussed the possibility to recover a files containing more than one cluster. In this assignment, you are going to recover a file with **multiple, contiguous clusters**.

  Of course, if the clusters are not contiguous, the difficulty will be increased. Therefore, recovering files with non-contiguous clusters is the not required.

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.

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
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 3 choices, either:

`-i`	`-l`	`-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   ## wrong command
```

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

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 = 1
Number of reserved sectors = 32
root@linux:~# _
```

Keep in your mind that you should **never hardcode the above result**.

2.3 Milestone 3 - Listing all directory entries

Milestone 3 is doing a similar job as the command "`ls -R`". 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, 4321, 10
3, TEST.C, 1023, 12
4, HELLO.MP3, 4194304, 14
5, BACKUP/, 0, 100
6, BACKUP/., 0, 100
7, BACKUP/.., 0, 0
8, BACKUP/TEST.C, 1023, 18
9, EMTPY/, 0, 200
10, EMPTY/., 0, 200
11, EMPTY/.., 0, 0
12, WHATSUP.TXT, 0, 0
root@linux:~# _
```

- The order of the printout must be the order that the files are found in the directory that you are working on.
- For each entry, the output should be the following format:
  1. the **order number**, followed by a comma and a space character;
  2. the **filename**, followed by a comma and a space character;
  3. the **file size** in bytes, followed by a command and a space character;
  4. the **starting cluster number**, followed by a newline character.
- 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:

1. First, you have to add a trailing character '/' to the end of directory name. Then, print out the information of the sub-directory.
2. Then, your program should look into the content of that sub-directory. Note that even though a directory does not contain any file, **at least one cluster** will be allocated.
3. For each directory entry in the sub-directory,
   - There are two default directory entries in every sub-directory:
     * the current directory '.' (without the quotes), which points to itself, and
     * the parent directory '..' (without the quotes), which points to the root directory, and it is **special**[1] that the value is starting cluster value for parent directory is 0, only for sub-directories of the root directory.

     You are required to print out the information of the above two special directory entries.
   - If the directory entry is a regular file, then print out the related information.

2.3.1 Filename: 8.3 or LFN?

You may notice that in the above output, the 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:

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

[1] http://en.wikipedia.org/wiki/File_Allocation_Table#File_Allocation_Table

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 What to do if I meet long filename?

You have to **skip all the LFN entries**. The flag in the directory entry should specify whether an entry is a long filename or not. If yes, just ignore that directory entry. For details, please refer to our tutorials. Nonetheless, while we are creating the FAT32-formatted disk, we may use the following kinds of filenames:

- 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.

2.4 Milestone 4 - Recovering small 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, 4321, 10
3, TEST.C, 1023, 12
4, HELLO.MP3, 4194304, 14
5, BACKUP/, 0, 100
6, BACKUP/., 0, 100
7, BACKUP/.., 0, 0
8, BACKUP/TEST.C, 1023, 18
root@linux:~# mount -o fat32.disk tmp
root@linux:~# ls tmp/
BEST.C   BACKUP/   HELLO.MP3   MAKEFILE   TEST.C
root@linux:~# /bin/rm tmp/MAKEFILE
root@linux:~# ls tmp/
BEST.C   BACKUP/   HELLO.MP3   TEST.C
root@linux:~# umount tmp
root@linux:~# ./recover -d fat32.disk -l
1, BEST.C, 4321, 10
2, TEST.C, 1023, 12
3, HELLO.MP3, 4194304, 14
5, BACKUP/, 0, 100
6, BACKUP/., 0, 100
7, BACKUP/.., 0, 0
8, BACKUP/TEST.C, 1023, 18
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 in /
root@linux:~# ./recover -d fat32.disk -l
1, MAKEFILE, 21, 11
2, BEST.C, 4321, 10
3, TEST.C, 1023, 12
4, HELLO.MP3, 4194304, 14
5, BACKUP/, 0, 100
6, BACKUP/., 0, 100
7, BACKUP/.., 0, 0
8, BACKUP/TEST.C, 1023, 18
root@linux:~# mount -o fat32.disk tmp
root@linux:~# cd tmp
root@linux:~/tmp# ls
BEST.C   BACKUP/   HELLO.MP3   MAKEFILE   TEST.C
root@linux:~/tmp# _
```

Requirements

- The filename in FAT32 is **case-insensitive**. Note that only the LFN entries stores case-sensitive filenames while the filenames provided by the user are case-insensitive. The requirement applies to all further milestones.
- 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 in [directory name]
```

is printed when the input filename can be found in the file system. It also displays the **location that the deleted file is found**. In the example, '`/`' denotes the root directory. 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.
- You are **NOT** required to handle the case when there are **multiple deleted directory entries with identical names**.

Suggested steps

1. Check if the filename provided by the user contains the dot character.
   - If yes, it is assume that:
     - the file 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 file 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 check if the original cluster is occupied by inspecting the FAT. Else, report an error message: "`error - file not found`" to the `stdout` stream.

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.

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 - Recovering small files in sub-directories

In this milestone, you are again required to recover a file that occupies one cluster only. On top of that, you have to **search all sub-directories for the target file**. It is illustrated in "*Sample Screen Capture #5*".

Sample Screen Capture #5 (1 of 2)

```
root@linux:~# ./recover -d fat32.disk -l
1, MAKEFILE, 21, 11
2, BEST.C, 4321, 10
3, TEST.C, 1023, 12
4, HELLO.MP3, 4194304, 14
5, BACKUP/, 0, 100
6, BACKUP/., 0, 100
7, BACKUP/.., 0, 0
8, BACKUP/TEST.C, 1023, 18
root@linux:~# mount -o fat32.disk tmp
root@linux:~# ls tmp/BACKUP
TEST.C
root@linux:~# /bin/rm tmp/BACKUP/TEST.C
root@linux:~# ls tmp/BACKUP
root@linux:~# umount tmp
root@linux:~# ./recover -d fat32.disk -l
1, MAKEFILE, 21, 11
2, BEST.C, 4321, 10
3, TEST.C, 1023, 12
4, HELLO.MP3, 4194304, 14
5, BACKUP/, 0, 100
6, BACKUP/., 0, 100
7, BACKUP/.., 0, 0
root@linux:~# _
```

Sample Screen Capture #5 (2 of 2)

```
root@linux:~# ./recover -d fat32.disk -r TEST.C
TEST.C: recovered in /BACKUP
root@linux:~# ./recover -d fat32.disk -l
1, MAKEFILE, 21, 11
2, BEST.C, 4321, 10
3, TEST.C, 1023, 12
4, HELLO.MP3, 4194304, 14
5, BACKUP/, 0, 100
6, BACKUP/., 0, 100
7, BACKUP/.., 0, 0
8, BACKUP/TEST.C, 1023, 18
root@linux:~# mount -o fat32.disk tmp
root@linux:~# ls tmp/BACKUP
TEST.C
root@linux:~/tmp/BACKUP# _
```

Requirements.

- All requirements listed in Milestone 4 must be upheld.
- The file system contains two layers of directories only. Therefore, there is no directory in the sub-directories under the root.

2.6 Milestone 6 - Recovering large files

In this milestone, you are required to recover **a file that occupies more than one cluster**. Nevertheless, we assume that such a file was allocated contiguously. You may ask: "*How can we guarantee that a file was allocated contiguously?*" We suggest the following way:

1. Format the disk or the disk image.
2. Mount the disk.
3. Copy a large file into the disk.

Then, such a newly-created file is guaranteed to be contiguously allocated. Of course, you need to create more files into the disk in order to make your own testcase more meaningful.

Note that the way to recover a deleted large file is similar to the way specified in Milestones 4 and 5. On top of that, the **file size** in the deleted directory entry tells you **how many clusters you need** for the large file recovery operation. Of course, you have to think about the implementation by yourself.

Requirements

- All requirements listed in Milestones 4 and 5 must be upheld.
- The deleted large file can be either in the root directory or any one of the sub-directories under the root directory.
- There is one exception in the "**contiguous cluster**" assumption. What if there is an arbitrary cluster allocated inside the range of contiguous clusters?

  Let us give you an example to illustrate.

  - Say, the clusters of the deleted file are in the range [200, 300].
  - By inspecting the FAT, your program finds that cluster 222 is already allocated.
  - Then, your program stops and should report the error message:

    ```
    [filename]:  error - fail to recover
    ```

    as shown in "*Sample Screen Capture #6*".

Sample Screen Capture #6

```
root@linux:~# ./recover -d fat32.disk -r BIG.GUY
BIG.GUY: error - fail to recover
root@linux:~# _
```

2.7 Some general hints

- Beware of empty files.
- You are not required to recover directory files, just regular files.
- The root directory may span across more than one cluster.
- Be aware of the file size, especially the large ones.
- You are free to use `fopen()`, `fclose()`, etc. to access to the device file.
- 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 programs to grade your assignments. If any discrepancy is found, we will grade your assignment manually.
- A reminder: there is one root directory. Inside the root directory, there can be more than one sub-directory. However, there were **no further sub-directories**.

3 Mark Distributions

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

Milestone 1 - valid arguments detection	5%
Milestone 2 - printing file system information	10%
Milestone 3 - listing all directory entries	15%
Milestone 4 - recovering small files in the root	20%
Milestone 5 - recovering small files in sub-directories	20%
Milestone 6 - recovering large files	30%
Total	100%

Submission

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

–END–