(Best-Effort) File Recovery in FAT32

2015-16 Term 2, CSCI 3150 - Programming Assignment 2

Last update: 1:00am, 2016 March 28

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

- **Device File**. In *nix systems, a device is a file and can usually be found under the directory `/dev`. For example:
  - `/dev/hda` and `/dev/hdb` are the first two ATA hard disks in the system;
  - `/dev/sda` and `/dev/sdb` are the first two SATA disks / USB in the system;

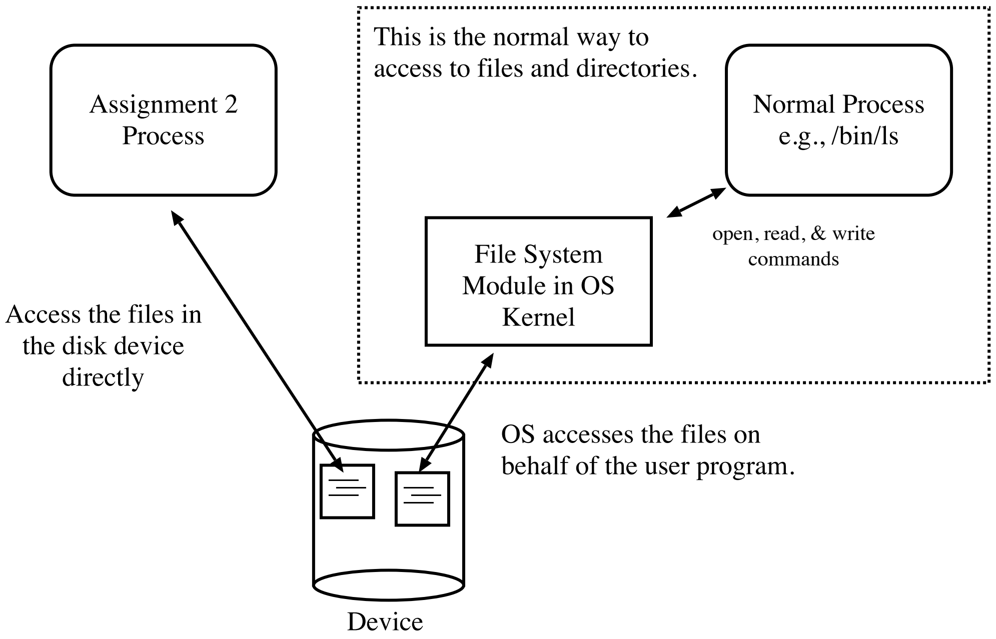
This is the normal way to access to files and directories.
Assignment 2 Process
Normal Process e.g., /bin/ls
File System Module in OS Kernel
open, read, & write commands
Access the files in the disk device directly
OS accesses the files on behalf of the user program.
Device

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.

- `/dev/ram*` are ram disks in the system; etc.

In this assignment, you are going to create a file and format it as a disk containing a file system.

- **Disk creation**.

```
dd if=/dev/zero of=fat32.disk bs=1M count=64
```

This command create a file named “`fat32.disk`” of size 64MBytes. In the tutorials, our tutors will explain this in details.

- **Disk format**.

```
mkfs.vfat -F32 fat32.disk
```

This command formats the disk “`fat32.disk`” into FAT32 format.

- **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.

- **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.

  Useful system calls: `pread()`, `pwrite()`, `lseek64()`, `open()`, and `close()`.

  Useful library calls: `fread()`, `fwrite()`, `fseek()`, `fopen()`, and `fclose()`.

  You are free use the GNU C library calls or the corresponding system calls to implement this assignment.

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 pathname provided by the user.

- **Root directory only**. The tool is not going to recover all deleted files in the entry file system. It is a limited tool which only works in the root directory.

- **Recover one file at a time**. Your program will be given a file name of the deleted file. Then, your program will look for that file in the root directory. If it is found, recover that deleted file. Else, report an appropriate error.

- **Files with one cluster only**. In the lecture, we will discuss the difficulty to recover a file containing more than one cluster. 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 may 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 should not use any Linux workstations provided by your department to run this assignment since you are not the root user of those workstations. During demonstrations, we will use the distributed Ubuntu Linux 14.04 virtual machines to grade your assignment. Please make sure that you are using the same development environment.

2.1 Milestone 1 - Detect 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 file system information
-l                    List the root directory
-r target -o dest     Recover the target deleted file
-x target             Cleanse the target deleted file
root@linux:~# _
```

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

According to the above, the **device file** must be presented while other arguments are specifying the executions of different milestones.

- In the above sample, you see the name "`./recover`". Note that it is not done by hard-coding the name there. You should print the name through the use of "`argv[0]`".

- “-d [filename]”. It is an *filename* to a “*disk*” containing a FAT32 file system. It can be an image file, which is just an ordinary file, or a device file.

  Assumptions on the device file.

  - You can always assume that the input file is a file carrying the FAT32 disk format.
  - You can also assume that the filename provided always refers to an existing, readable device file.

- “[other arguments]”. The following lists the possible valid program arguments to your program:

```
./recover -d fat32.disk -i                   # print information
./recover -d fat32.disk -l                   # list the root dir
./recover -d fat32.disk -r HI.TXT -o out.txt # Recover "HI.TXT"
./recover -d fat32.disk -x OH.TXT            # Cleanse "OH.TXT"
```

  In the assignment, when we refer to a file stored inside the FAT32 file system, we always work with filenames in 8.3 format. That means your program is not going to support long-filename-formatted targets, such as:

```
./recover -d fat32.disk -r counter-strike.exe  # not supported
./recover -d fat32.disk -r hi.docx ...         # not supported
```

- Last but not least, the order between “-d [device file]” and “[other arguments]” is important. The device file input always comes first. That means:

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

Useful but optional library call. If you are interested in exploring the proper way to process the program arguments, please feel free to use the library call getopt().

2.2 Filename: 8.3 or LFN?

You may notice that in the above, we mentioned 8.3 and long filenames. What are they?

Let us cover 8.3 filenames first. Under Linux, there are two conditions that have to be satisfied in order to guarantee a filename is stored as the 8.3 format in a FAT32 disk.

---

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.

Note that the above restrictions apply to files in the FAT32 file system only, but not for other file systems. Henceforth, while you are creating testing disks, you have to aware of the filenames of the created files.

2.2.1 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 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.

2.2.2 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, you are not required to read LFN entries.

2.3 Milestone 2 - Print 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
First FAT starts at byte = 16384
Data area starts at byte = 15400960
root@linux:~# _
```

Hints.

- You should **never hard code 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 directly available in the boot sector. Your program has to calculate the results.
- In order to check the correctness of your program, you can use the command: `dosfsck -v fat32.disk`.

2.4 Milestone 3 - List directory content

Milestone 3 is a similar job as the command "`ls -l`". However, you are required to print out a different set of data, as shown in "*Sample Screen Capture #3*". This milestone should be invoked by the following command line:

```
./recover -d fat32.disk -l   # list the root directory
```

Sample Screen Capture #3

```
root@linux:~# ./recover -d fat32.disk -l
 1, MAKEFILE, 21, 11
 2, BEST.C, 4096, 10
 3, TEST.C, 1023, 12
 4, HELLO.MP3, 4194304, 14
 5, LFN entry
 6, TEMPOR~1/, 0, 100
 7, LFN entry
 8, LFN entry
 9, THISIS~1.TXT, 1000, 19
10, ?ELLO.TXT, 12, 17
root@linux:~# _
```

- The order of the printout must be the same order as the directory entries stored in the corresponding directory.
- For each directory entry in the root directory, print its information on the same line. Each line should be in the following format:

  1. The **order number**, starting from 1, and is followed by a comma and a space character.

  2. The **8.3 filename**, followed by a comma and a space character. If the entry is a deleted one, print the leading character as a question mark '?', instead of 0xE5,

as shown in the tenth output in the sample screen capture.

Note 1. You may find two strange outputs: “`TEMPOR~1/`” and “`THISIS~1.TXT`”. They are the 8.3 names for their long filenames counterpart.

Note 2. In this assignment, you are not going to read the LFN structures. If you encountered any LFN structures, **please print “`LFN entry`”**. However, you still need to print the 8.3 name of that LFN file.

3. The **file size** in bytes, followed by a comma and a space character.

4. The **starting cluster number**, followed by a newline character.

- In case your program meets an entry pointing to a sub-directory, then:
  - Add a trailing character ‘`/`’ to the end of directory name. Then, print out the information of the sub-directory.
  - Your program is not required to recursively process the sub-directories.

2.5 Milestone 4 - Recover one-cluster-sized files only

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, 4096, 10
3, TEST.C, 1023, 12
4, HELLO.MP3, 4194304, 14
root@linux:~# mount fat32.disk tmp
root@linux:~# ls tmp/
BEST.C       HELLO.MP3   MAKEFILE   TEST.C
root@linux:~# /bin/cp tmp/MAKEFILE .  ## copy to current directory
root@linux:~# /bin/rm tmp/MAKEFILE    ## remove FAT32 file
root@linux:~# ls tmp/
BEST.C       HELLO.MP3   TEST.C
root@linux:~# umount tmp
root@linux:~# ./recover -d fat32.disk -l
1, ?AKEFILE, 21, 11
2, BEST.C, 4096, 10
3, TEST.C, 1023, 12
4, HELLO.MP3, 4194304, 14
root@linux:~# _
```

Sample Screen Capture #4 (2 of 2)

```
root@linux:~# ls -l output.txt
ls: cannot access output.txt: No such file or directory
root@linux:~# ./recover -d fat32.disk -r MAKEFIL -o output.txt
/MAKEFIL: error - file not found
root@linux:~# ls -l output.txt
ls: cannot access output.txt: No such file or directory
root@linux:~# ./recover -d fat32.disk -r MAKEFILE -o output.txt
MAKEFILE: recovered
root@linux:~# ls -l MAKEFILE output.txt
-rw-r--r-- ...  21 ... MAKEFILE   ## size of MAKEFILE is 21 bytes
-rw-r--r-- ...  21 ... output.txt
root@linux:~/tmp# diff MAKEFILE output.txt  ## they are the same
root@linux:~/tmp# _
```

Requirements

- This milestone is invoked through:

```
./recover -d fat32.disk -r TARGET -o output.txt   # TARGET: 8.3 format
```

- 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.

"*But, how can we generate such a testcase?*" You asked. We have a way:

1. Create another new file.
2. Grow that file until it occupies all available disk space.
3. Then, all free blocks will be occupied.

- 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.

- Last, the option "`-o output.txt`" specifies the filename that saves the content of the recovered file. As suggested in the sample screen capture, the program should not create or truncate "`output.txt`" if the recovery task returns an error.

  If the file "`output.txt`" does not exist, your program should create it. Else, your program should truncate the original file, i.e., removing all previous data in "`output.txt`".

  If your program fails to open "`output.txt`", please report:

  ```
  [output filename]: failed to open
  ```

  This message should be written to the standard output stream. Then, your program should be terminated.

Suggested implementation

1. Open the device file with the read-only mode.

2. 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*.

3. 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.

    Cases for 8.3 filename and extension.

    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 of how the filename "`TEST.C`" is stored 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.

---

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

    - If the target cluster is not occupied, then start reading the content of the deleted file from the target cluster up the size specified by the directory entry. Further, save the data into the file specified by “`-o`”, i.e., “`output.txt`” in our example.

      Last, print the message “`[filename]:  recovered`” to stdout.

    - Else, if the target cluster is occupied, report the error message “`[filename]: error - fail to recover`”.

5. If no matched entry is found after all directory entries in the root directory have been processed, then report an error message: “`[filename]:  error - file not found`” to the `stdout` stream.

Assumptions.

- No ambiguous recovery request. For example:
    - We have two files “`BEST.C`” and “`TEST.C`”.
    - We remove them.

- Then, both deleted directory entries would carry "*the same name*", i.e., "`0xE5 EST.C`".

  To ease your implementation, we would not test your program under such a scenario.

- If the target filename exists, it always points to a file, not a directory.

2.6 Milestone 5: Cleanse the deleted file

Last but not least, we know that deleting a file in FAT32 is not that “*secure*”. Let’s write add one more feature to our assignment program in order to **set the content of a deleted file to zeros**. The process 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, 4096, 10
3, TEST.C, 1023, 12
4, HELLO.MP3, 4194304, 14
root@linux:~# mount fat32.disk tmp
root@linux:~# ls tmp/
BEST.C       HELLO.MP3   MAKEFILE   TEST.C
root@linux:~# /bin/cp tmp/MAKEFILE .  ## copy to current directory
root@linux:~# /bin/rm tmp/MAKEFILE    ## remove FAT32 file
root@linux:~# ls tmp/
BEST.C       HELLO.MP3   TEST.C
root@linux:~# umount tmp
root@linux:~# ./recover -d fat32.disk -l
1, ?AKEFILE, 21, 11
2, BEST.C, 4096, 10
3, TEST.C, 1023, 12
4, HELLO.MP3, 4194304, 14
root@linux:~# _
```

Sample Screen Capture #5 (2 of 2)

```
root@linux:~# ./recover -d fat32.disk -x MAKEFIL
MAKEFIL: error - file not found
root@linux:~# ./recover -d fat32.disk -x BEST.C
BEST.C: error - file not found
root@linux:~# ./recover -d fat32.disk -x MAKEFILE
MAKEFILE: cleansed
root@linux:~# ./recover -d fat32.disk -r MAKEFILE -o output.txt
MAKEFILE: recovered
root@linux:~# ls -l MAKEFILE output.txt
-rw-r--r-- ...  21 ... MAKEFILE   ## size of MAKEFILE is 21 bytes
-rw-r--r-- ...  21 ... output.txt
root@linux:~/tmp# diff MAKEFILE output.txt    ## NOT the same
Binary files MAKEFILE and output.txt differ   ## diff output
root@linux:~/tmp# _
```

Requirements

- This milestone is invoked through:

  ```
  ./recover -d fat32.disk -x TARGET        # 8.3 pathname
  ```

- 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.

  Note importantly that you cannot cleanse any existing files, as shown in the sample screen capture.

- The error message:

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

is printed when either one of the following cases happens:

1. the cluster originally belonged to the deleted file is occupied. Such a checking can be done by inspecting the FAT, or
2. the deleted file originally is of zero bytes.

Note that the error should be written to the standard output stream.

- The message

  ```
  [filename]: cleansed
  ```

  is printed when the input filename can be found in the file system. Then, you have to overwrite the original content of the deleted file to the value zero. You are required to overwrite the content of the deleted file. Do not alter the content of the directory entry.

  Note that this message should be written to the standard output stream.

By the way, in *Sample Screen Capture #5*, we use the recovery function to verify the deleted file is cleansed. Such an illustration shows:

1. the content of the deleted file is changed, and
2. the directory entry of the deleted file is kept unchanged.

As a matter of fact, during the grading of this assignment, we can directly look into the disk to see if the file content is reset to zero, not just using the recovery function.

Hints:

- Use the file-opening mode "`r+`" when you open the disk file using `fopen`.
- You can reuse most of the codes in the file recovery function.

2.7 General notes to all milestones

- Be aware of empty files.
- Every file that you are asked to recover is not larger than one cluster.
- You are not required to recover directory files, just regular files.
- A directory may span across more than one cluster.
- The program "`diff`" should always be used for checking whether your program produces a correct recovered file or not.
- For the output format,
  - At the end of every message, there is no punctuation.
  - For spacing, only space characters are used.
  - Note that we will first use judge programs to grade your assignments. If any discrepancy is found, we will grade your assignment manually.

3 Mark Distributions

This assignment is an individual assignment and the mark distribution is as follows.

Milestone 1 - Detect valid arguments	5%
Milestone 2 - Print file system information	15%
Milestone 3 - List all directory entries	20%
Sub-task #1: root directory has only 8.3 entries	
Sub-task #2: root directory has a mix of 8.3 and LFN entries	
Sub-task #3: root directory has more than one cluster	
Milestone 4 - Recover a one-cluster-sized file	30%
Sub-task #1: recover from the root directory	
Sub-task #2: handle all error conditions	
Milestone 5 - Cleanse the deleted file	30%
Sub-task #1: cleanse a deleted file	
Sub-task #2: handle all error conditions	
Total	100%

Submission & Demonstration

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

CUHK network only: `http://course.cse.cuhk.edu.hk/~csci3150/`

Deadline: 23:59, April 22, 2016 (Fri)

The demonstration will be held in the the semester break. Please stay tuned with our announcements.

–END–

Change Log

Time	**Details**
2016 Mar 28	Release the specification.