CSCI 3150 Assignment 3 - (Best-Effort) File Recovery in FAT32

Updated: ~~18:45, 2011 Nov 23.~~ 21:50, 2011 Dec 5.

Abstract

Recovering deleted files can be a business, a profession, or just a funny 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: **how can you tune yourself back to the working mode after the Christmas holiday?**

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-3 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.
- **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.

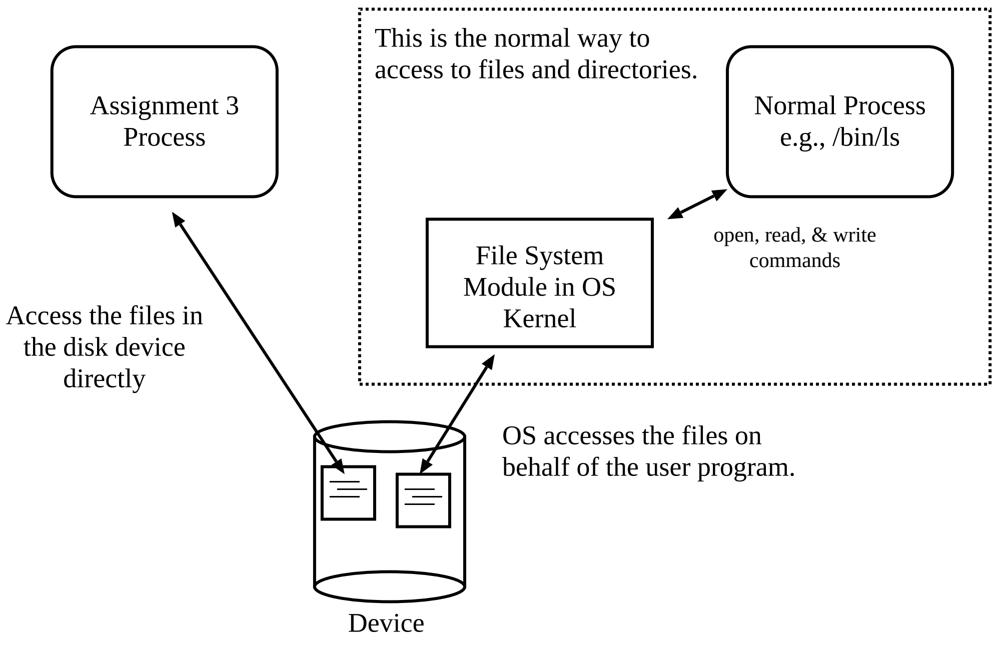
This is the normal way to access to files and directories.
Assignment 3 Process
Normal Process e.g., /bin/ls
open, read, & write commands
File System Module in OS Kernel
Access the files in the disk device directly
OS accesses the files on behalf of the user program.
Device

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.

- **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-3 process

- **Open**. The Assignment-3 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, in a byte-by-byte manner.
- **Read and Write**. When an Assignment-3 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 `open()`, `read()`, `write()`, `close()`, etc. to work with the device file.

Note that the only file that the Assignment-3 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.

- **Root directory only**. It would become complicated if you have to look into all directories and sub-directories in order to look for the file needed. In this assignment, you are required to tackle a file system with one directory only, i.e., the root directory.

- **File checksum may be provided**. Because of the deletion mechanism on FAT32, there are chances that there would be more than one file that are possible to be the deleted file you want. Under this case, we will not and should not recover all those possible files. But, what shall we do?

  Don't worry. You will be provided a file checksum. Note that a checksum is a hash value of the file. Therefore, with the checksum, we can determine if the recovered file is the one that the user wants.

- **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 contiguous clusters is one of the tasks of this assignment. 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 "a3". 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**, and, since you are not the root user, you could never use any Linux workstations provided by our department to run this assignment.

2.1 Milestone 1: detecting valid arguments

The program "a3" should take a set of program arguments.

Sample Screen Capture #1

```
root@linux:~# ./a3
Usage: ./a3 -d [device filename] [other arguments]
-i                     Print boot sector information.
-l                     List the root directory.
-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.

- "`[other arguments]`". There can be ~~6~~ 4 choices, either:

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

But, the above ~~6~~ 4 cases must not appear together in one command line. E.g.,

```
root@linux:~# ./a3 -d fat32.disk -i -l
```

is wrong, and 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:~# ./a3 -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 the root directory

Milestone 3 is doing a similar job as the program "`/bin/ls`". 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:~# ./a3 -d fat32.disk -l
1, MAKEFILE, 21, 11
2, BEST.C, 4321, 10
3, TEST.C, 1023, 12
4, HELLO.MP3, 4194304, 14
5, TEMP/, 0, 100
root@linux:~# _
```

- The order of the printout must be the order that the files are found 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 **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, you have to add a trailing character '`/`' to the end of filename

Other requirements.

- You are only required to print out the contents of the root directory.
- You are not required to print out the details inside sub-directories.
- You are not required to print out entries marked as deleted.

2.3.1 Filename: 8.3 or LFN?

You may notice that in the above output, the filenames are all in upper-case characters. 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.

Note that the above restrictions are applied only to files stored in the FAT32 file system, but not the filename of the device file.

2.3.2 What if I still 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, we will avoid the use of long filenames in the grading process.

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:~# ./a3 -d fat32.disk -l
1, MAKEFILE, 21, 11
2, BEST.C, 4321, 10
3, TEST.C, 1023, 12
4, HELLO.MP3, 4194304, 14
5, TEMP/, 0, 100
root@linux:~# mount -o fat32.disk tmp
root@linux:~# ls tmp/
best.c   hello.mp3   makefile   temp/  test.c
root@linux:~# /bin/rm tmp/MAKEFILE
root@linux:~# ls tmp/
best.c   hello.mp3   temp/   test.c
root@linux:~# umount tmp
root@linux:~# ./a3 -d fat32.disk -l
1, BEST.C, 4321, 10
2, TEST.C, 1023, 12
3, HELLO.MP3, 4194304, 14
4, TEMP/, 0, 100
root@linux:~# _
```

Sample Screen Capture #4 (2 of 2)

```
root@linux:~# ./a3 -d fat32.disk -r MAKEFIL
MAKEFIL: error - file not found.
root@linux:~# ./a3 -d fat32.disk -r MAKEFILE
MAKEFILE: recovered.
root@linux:~# ./a3 -d fat32.disk -l
1, MAKEFILE, 21, 11
2, BEST.C, 4321, 10
3, TEST.C, 1023, 12
4, HELLO.MP3, 4194304, 14
5, TEMP/, 0, 100
root@linux:~# mount -o fat32.disk tmp
root@linux:~# cd tmp
root@linux:~/tmp# ls
best.c   hello.mp3   makefile   temp/  test.c
root@linux:~/tmp# _
```

Requirements

- The error message shown in *Sample Screen Capture #4 (2 of 2)* is **the only error message** that you have to implement for this milestone. The error should be written to the standard output stream.
- The set of outputs shown using "`./a3 -d fat32.disk -l`" suggests that the order of the files inside the directory should be **preserved** after the recovery.
- Note that you are not required to recover deleted sub-directories and we would not have such a testcase.

Suggested steps

1. Detect if the filename provided by the user contains the 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 change the filename of the directory entry to the filename supplied by the user.
4. Else, report an error message: "`error - file not found`" 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 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 Milestone 4. 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.

2.6 Milestone 6 - Detecting ambiguous file recovery request

In Milestones 4 and 5, 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 "`error - ambiguous`" to the `stdout` stream and the program then should terminate. "*Sample Screen Capture #5 (1 of 2)*" illustrates such a scenario. Let us assume that we are using the original disk image "`fat32.disk`".

Sample Screen Capture #5 (1 of 2)

```
root@linux:~# mount -o fat32.disk tmp
root@linux:~# /bin/rm tmp/*.C
root@linux:~# umount tmp
root@linux:~# ./a3 -d fat32.disk -l
1, MAKEFILE, 21, 11
2, HELLO.MP3, 4194304, 14
3, TEMP/, 0, 100
root@linux:~# ./a3 -d fat32.disk -r TEST.C
TEST.C: error - ambiguous.
root@linux:~# _
```

2.7 Milestone 7 - Recovering files with MD5 checksum

In order to solve the problem stated in Milestone 6, we introduce the use of the **MD5 checksum** to identify which deleted directory entry should be 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 target file as the key to identify which directory entry should be the target. "*Sample Screen Capture #5 (2 of 2)*" shows the example invocation.

Sample Screen Capture #5 (2 of 2)

```
root@linux:~# ./a3 -d fat32.disk -r TEST.C -m ......
TEST.C: recovered with MD5.
root@linux:~# ./a3 -d fat32.disk -l
1, MAKEFILE, 21, 11
2, TEST.C, 1023, 12
3, HELLO.MP3, 4194304, 14
4, TEMP/, 0, 100
root@linux:~# _
```

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 read 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 Milestones 4 and 5.

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.

Assumption

- For this milestone only, there is an upper limit on the size of the file to be recovered: **1 MBytes** and this assumption corresponds to the MD5 checksum calculation.

2.8 Some general hints

- Beware of empty files.
- You don't have 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.

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 the root directory	10%
Milestone 4 - recovering small files	20%
Milestone 5 - recovering large files	20%
Milestone 6 - detecting ambiguous recovery request	5%
Milestone 7.1 - recovering files with MD5 (small files)	15%
Milestone 7.2 - recovering files with MD5 (large files)	15%
Total	100%

4 Submission

For the submission of the assignment, please refer to the following link:

`http://appsrv.cse.cuhk.edu.hk/~csci3150/html/submit.php`

Deadline: 11:59 AM, January 4, 2012 (Wed).

Note very important that we only allow one submission method: **online submission with instant marking**. The submission system will be online on 2011 Dec 5 (Mon), and the submission details will also be available by the time the system is online.

If you want to appeal for the marking, our tutor will welcome you to look for them after the deadline:

SHB 120, 1:00pm - 5:00pm, January 4, 2012 (Wed).

If you want to appeal but you are not in campus, please send email to the course instructor and we can look into the case.