Lab 4: File Recovery

Introduction

FAT32 has been around for 25 years. Because of its simplicity, it is the most widely compatible file system. Although recent computers have adopted newer file systems, FAT32 is still dominant in SD cards and USB flash drives due to its compatibility.

Have you ever accidentally deleted a file? Do you know that it could be recovered? In this lab, you will build a FAT32 file recovery tool called **Need You to Undelete my FILE**, or `nyufile` for short.

Objectives

Through this lab, you will:

- Learn the internals of the FAT32 file system.
- Learn how to access and recover files from a raw disk.
- Get a better understanding of key file system concepts.
- Be a better C programmer. Learn how to write code that manipulates data at the byte level and understand the alignment issue.

Overview

In this lab, you will work on the data stored in the FAT32 file system **directly**, without the OS file system support. You will implement a tool that recovers a deleted file specified by the user.

For simplicity, you can assume that **the deleted file is in the root directory.** Therefore, you don't need to search subdirectories.

Preparation

Unfortunately, non-root users cannot mount a file system on the CIMS compute servers. If you want to mount a file system, you have to use your own Linux machine or virtual machine.

If you do not already have a Linux system installed, you can follow these steps to install a virtual machine running CentOS 7.

If your machine has an Intel CPU...

1. Install VirtualBox.
2. Install Vagrant.
3. Install the Vagrant `scp` plugin, which allows you to transfer files between your host machine and the virtual machine:

```
$ vagrant plugin install vagrant-scp
```

4. Download and extract the configuration file:

```
$ tar xf centos7.tar.gz
```

5. Start the virtual machine (all Vagrant commands must be run in the `centos7` directory):

```
$ cd centos7
$ vagrant up
```

6. Log into the virtual machine:

```
$ vagrant ssh
```

7. Now, you can work on this lab in the virtual machine. It already has `gcc`, `gdb`, and `vim` installed. If you want to install other packages, you can run this command inside the virtual machine:

```
$ sudo yum install <packages>
```

8. To copy files from your host machine to the virtual machine, run this command on your host machine:

```
$ vagrant scp <src> :<dest>
```

Conversely, to copy files from the virtual machine to your host machine, run:

```
$ vagrant scp :<src> <dest>
```

9. To pause the virtual machine, run:

```
$ vagrant suspend
```

To shut down the virtual machine, run:

```
$ vagrant halt
```

If you have a Mac with an M1 CPU…

Unfortunately, the virtual machine does not work on a Mac with an M1 CPU. You can run the virtual machine on one of the CIMS compute servers. VirtualBox and Vagrant are already installed. You just need to load the module:

```
$ module load vagrant
```

Then, follow steps 3–9 above.

Working with a FAT32 disk image

Before going through the details of this lab, let's first create a FAT32 disk image. Follow these steps:

Step 1: create an empty file of a certain size

On Linux, `/dev/zero` is a special file that provides as many `\0` as are read from it. The `dd` command performs low-level copying of raw data. Therefore, you can use it to generate an arbitrary-size file full of zeros.

For example, to create a 256KB empty file named `fat32.disk`:

```
$ dd if=/dev/zero of=fat32.disk bs=256k count=1
```

Read `man dd` for its usage. You will use this file as the disk image.

Step 2: format the disk with FAT32

You can use the `mkfs.fat` command to create a FAT32 file system. The most basic usage is:

```
$ mkfs.fat -F 32 fat32.disk
```

You can specify a variety of options. For example:

```
$ mkfs.fat -F 32 -f 2 -S 512 -s 1 -R 32 fat32.disk
```

Here are the meanings of each option:

- `-F`: type of FAT (FAT12, FAT16, or FAT32).
- `-f`: number of FATs.
- `-S`: number of bytes per sector.
- `-s`: number of sectors per cluster.
- `-R`: number of reserved sectors.

Step 3: verify the file system information

The `fsck.fat` command can check and repair FAT file systems. You can invoke it with `-v` to see the FAT details. For example:

```
$ fsck.fat -v fat32.disk
fsck.fat 3.0.20 (12 Jun 2013)
fsck.fat 3.0.20 (12 Jun 2013)
Checking we can access the last sector of the filesystem
Warning: Filesystem is FAT32 according to fat_length and fat32_length fields,
  but has only 472 clusters, less than the required minimum of 65525.
  This may lead to problems on some systems.
Boot sector contents:
System ID "mkfs.fat"
Media byte 0xf8 (hard disk)
       512 bytes per logical sector
       512 bytes per cluster
        32 reserved sectors
First FAT starts at byte 16384 (sector 32)
         2 FATs, 32 bit entries
      2048 bytes per FAT (= 4 sectors)
Root directory start at cluster 2 (arbitrary size)
Data area starts at byte 20480 (sector 40)
       472 data clusters (241664 bytes)
32 sectors/track, 64 heads
         0 hidden sectors
       512 sectors total
Checking for unused clusters.
Checking free cluster summary.
fat32.disk: 0 files, 1/472 clusters
```

You can see that there are 2 FATs, 512 bytes per sector, 512 bytes per cluster, and 32 reserved sectors. These numbers match our specified options in Step 2. You can try different options yourself.

Step 4: mount the file system

You can use the `mount` command to mount a file system to a **mount point**. The mount point can be any empty directory. For example, you can create one at `/mnt/disk`:

```
$ sudo mkdir /mnt/disk
```

Then, you can mount `fat32.disk` at that mount point:

```
$ sudo mount -o umask=0 fat32.disk /mnt/disk
```

Step 5: play with the file system

After the file system is mounted, you can do whatever you like on it, such as creating files, editing files, or deleting files. In order to avoid the hassle of having long filenames in your directory entries, it is recommended that you use only 8.3 filenames, which means:

- The filename contains at most eight characters, followed optionally by a `.` and at most three more characters.
- The filename contains only **uppercase** letters, numbers, and the following special characters: `! # $ % & ' ( ) - @ ^ _ ` { } ~`.

For example, you can create a file named `HELLO.TXT`:

```
$ echo "Hello, world." > /mnt/disk/HELLO.TXT
$ mkdir /mnt/disk/DIR
$ touch /mnt/disk/EMPTY
```

For the purpose of this lab, after you write anything to the disk, make sure to **flush the file system cache** using the `sync` command:

```
$ sync
```

(Otherwise, if you create a file and immediately delete it, the file may not be written to the disk at all and is unrecoverable.)

Step 6: unmount the file system

When you finish playing with the file system, you can unmount it:

```
$ sudo umount /mnt/disk
```

Step 7: examine the file system

You can examine the file system using the `xxd` command. You can specify a range using the `-s` (starting offset) and `-l` (length) options.

For example, to examine the root directory:

```
$ xxd -s 20480 -l 96 fat32.disk
00005000: 4845 4c4c 4f20 2020 5458 5420 0000 0000  HELLO   TXT ....
00005010: 6e53 6e53 0000 0000 6e53 0300 0e00 0000  nSnS....nS......
00005020: 4449 5220 2020 2020 2020 2010 0000 0000  DIR        .....
00005030: 6e53 6e53 0000 0000 6e53 0400 0000 0000  nSnS....nS......
00005040: 454d 5054 5920 2020 2020 2020 0000 0000  EMPTY       ....
00005050: 6e53 6e53 0000 0000 6e53 0000 0000 0000  nSnS....nS......
```

To examine the contents of `HELLO.TXT`:

```
$ xxd -s 20992 -l 14 fat32.disk
0005200: 4865 6c6c 6f2c 2077 6f72 6c64 2e0a       Hello, world..
```

Note that the offsets may vary depending on how the file system is formatted.

Your tasks

Important: before running your `nyufile` program, please make sure that your FAT32 disk is **unmounted**.

Milestone 1: validate usage

There are several ways to invoke your `nyufile` program. Here is its usage:

```
$ ./nyufile
Usage: ./nyufile disk <options>
  -i                     Print the file system information.
  -l                     List the root directory.
  -r filename [-s sha1]  Recover a contiguous file.
  -R filename -s sha1    Recover a possibly non-contiguous file.
```

The first argument is the filename of the disk image. After that, the options can be one of the following:

- `-i`
- `-l`
- `-r filename`
- `-r filename -s sha1`
- `-R filename -s sha1`

You need to check if the command-line arguments are valid. If not, your program should print the above usage information and exit.

Milestone 2: print the file system information

If your `nyufile` program is invoked with option `-i`, it should print the following information about the FAT32 file system:

- Number of FATs;
- Number of bytes per sector;

- Number of sectors per cluster;
- Number of reserved sectors.

Your output should be in the following format:

```
$ ./nyufile fat32.disk -i
Number of FATs = 2
Number of bytes per sector = 512
Number of sectors per cluster = 1
Number of reserved sectors = 32
```

For all milestones, you can assume that `nyufile` is invoked **while the disk is unmounted.**

Milestone 3: list the root directory

If your `nyufile` program is invoked with option `-l`, it should list all valid entries in the root directory with the following information:

- **Filename.** Similar to `/bin/ls -p`, if the entry is a directory, you should **append a `/` indicator**.
- **File size.**
- **Starting cluster.**

You should also print the total number of entries at the end. Your output should be in the following format:

```
$ ./nyufile fat32.disk -l
HELLO.TXT (size = 14, starting cluster = 3)
DIR/ (size = 0, starting cluster = 4)
EMPTY (size = 0, starting cluster = 0)
Total number of entries = 3
```

Here are a few assumptions:

- You **should not** list entries marked as deleted.
- You don't need to print the details inside subdirectories.
- For all milestones, there will be no long filename (LFN) entries. (If you have accidentally created LFN entries when you test your program, don't worry. You can just skip the LFN entries and print only the 8.3 filename entries.)
- All files and directories, including the root directory, may span across **more than one cluster**.
- There may be **empty** files.

Milestone 4: recover a small file

If your `nyufile` program is invoked with option `-r filename`, it should recover the deleted file with the specified name. The workflow is better illustrated through an example:

```
$ sudo mount -o umask=0 fat32.disk /mnt/disk
$ ls -p /mnt/disk
DIR/  EMPTY  HELLO.TXT
$ cat /mnt/disk/HELLO.TXT
Hello, world.
$ rm /mnt/disk/HELLO.TXT
$ ls -p /mnt/disk
DIR/  EMPTY
$ sudo umount /mnt/disk
$ ./nyufile fat32.disk -l
DIR/ (size = 0, starting cluster = 4)
EMPTY (size = 0, starting cluster = 0)
Total number of entries = 2
$ ./nyufile fat32.disk -r HELLO
HELLO: file not found
$ ./nyufile fat32.disk -r HELLO.TXT
HELLO.TXT: successfully recovered
$ ./nyufile fat32.disk -l
HELLO.TXT (size = 14, starting cluster = 3)
DIR/ (size = 0, starting cluster = 4)
EMPTY (size = 0, starting cluster = 0)
Total number of entries = 3
$ sudo mount -o umask=0 fat32.disk /mnt/disk
$ ls -p /mnt/disk
DIR/  EMPTY  HELLO.TXT
$ cat /mnt/disk/HELLO.TXT
Hello, world.
```

For all milestones, you only need to recover **regular files** (including empty files, but not directory files) in the **root directory**. When the file is successfully recovered, your program should print `filename: successfully recovered`.

For all milestones, you can assume that no other files or directories are created or modified since the deletion of the target file. However, multiple files may be deleted.

Besides, for all milestones, you don't need to update the `FSINFO` structure because most operating systems don't care about it.

Here are a few assumptions specifically for Milestone 4:

- The size of the deleted file is no more than the size of a cluster.
- At most one deleted directory entry matches the given filename. If no such entry exists, your program should print `filename: file not found`.

Milestone 5: recover a large contiguously-allocated file

Now, you will recover a file that is larger than one cluster. Nevertheless, for Milestone 5, you can assume that such a file is allocated contiguously. You can continue to assume that at most one deleted directory entry matches the given filename. If no such entry exists, your program should print `filename: file not found`.

Milestone 6: detect ambiguous file recovery requests

In Milestones 4 and 5, you assumed that at most one deleted directory entry matches the given filename. However, multiple files whose names differ only in the first character would end up having the same name when deleted. Therefore, you may encounter more than one deleted directory entry matching the given filename. When that happens, your program should print `filename: multiple candidates found` and abort.

This scenario is illustrated in the following example:

```
$ sudo mount -o umask=0 fat32.disk /mnt/disk
$ echo "My last name is Tang." > /mnt/disk/TANG.TXT
$ echo "My first name is Yang." > /mnt/disk/YANG.TXT
$ sync
$ rm /mnt/disk/TANG.TXT /mnt/disk/YANG.TXT
$ sudo umount /mnt/disk
$ ./nyufile fat32.disk -r TANG.TXT
TANG.TXT: multiple candidates found
```

Milestone 7: recover a contiguously-allocated file with SHA-1 hash

To solve the aforementioned ambiguity, the user can provide a SHA-1 hash via command-line option `-s sha1` to help identify which deleted directory entry should be the target file.

In short, a SHA-1 hash is a 160-bit fingerprint of a file, often represented as 40 hexadecimal digits. For the purpose of this lab, you can assume that identical files always have the same SHA-1 hash, and different files always have vastly different SHA-1 hashes. Therefore, even if multiple candidates are found during recovery, at most one will match the given SHA-1 hash.

This scenario is illustrated in the following example:

```
$ ./nyufile fat32.disk -r TANG.TXT -s c91761a2cc1562d36585614c8c680ecf5712e875
TANG.TXT: successfully recovered with SHA-1
$ ./nyufile fat32.disk -l
HELLO.TXT (size = 14, starting cluster = 3)
DIR/ (size = 0, starting cluster = 4)
EMPTY (size = 0, starting cluster = 0)
TANG.TXT (size = 22, starting cluster = 5)
Total number of entries = 4
```

When the file is successfully recovered with SHA-1, your program should print `filename: successfully recovered with SHA-1`.

Note that you can use the `sha1sum` command to compute the SHA-1 hash of a file:

```
$ sha1sum /mnt/disk/TANG.TXT
c91761a2cc1562d36585614c8c680ecf5712e875  /mnt/disk/TANG.TXT
```

Also note that it is possible that the file is empty or occupies only one cluster. The SHA-1 hash for an empty file is `da39a3ee5e6b4b0d3255bfef95601890afd80709`.

If no such file matches the given SHA-1 hash, your program should print `filename: file not found`. For example:

```
$ ./nyufile fat32.disk -r TANG.TXT -s 0123456789abcdef0123456789abcdef01234567
TANG.TXT: file not found
```

The OpenSSL library provides a function `SHA1()`, which computes the SHA-1 hash of `d[0...n-1]` and stores the result in `md[0...SHA_DIGEST_LENGTH-1]`:

```
#include <openssl/sha.h>

#define SHA_DIGEST_LENGTH 20

unsigned char *SHA1(const unsigned char *d, size_t n, unsigned char *md);
```

You need to add the compiler option `-l crypto` to link with the OpenSSL library.

Milestone 8: recover a non-contiguously allocated file

Finally, the clusters of a file are no longer assumed to be contiguous. You have to try every permutation of unallocated clusters on the file system in order to find the one that matches the SHA-1 hash.

The command-line option is `-R filename -s sha1`. The SHA-1 hash must be given.

Note that it is possible that the file is empty or occupies only one cluster. If so, `-R` behaves the same as `-r`, as described in Milestone 7.

For Milestone 8, you can assume that the entire file is within the first 12 clusters, so that a brute-force search is feasible.

If you cannot find a file that matches the given SHA-1 hash, your program should print `filename: file not found`.

FAT32 data structures

For your convenience, here are some data structures that you can copy and paste. Please refer to the lecture slides for details on the FAT32 file system layout.

Boot sector

```c
#pragma pack(push,1)
typedef struct BootEntry {
  unsigned char  BS_jmpBoot[3];     // Assembly instruction to jump to boot code
  unsigned char  BS_OEMName[8];     // OEM Name in ASCII
  unsigned short BPB_BytsPerSec;    // Bytes per sector. Allowed values include 512, 1024, 2048, and 409
  unsigned char  BPB_SecPerClus;    // Sectors per cluster (data unit). Allowed values are powers of 2,
  unsigned short BPB_RsvdSecCnt;    // Size in sectors of the reserved area
  unsigned char  BPB_NumFATs;       // Number of FATs
  unsigned short BPB_RootEntCnt;    // Maximum number of files in the root directory for FAT12 and FAT16
  unsigned short BPB_TotSec16;      // 16-bit value of number of sectors in file system
  unsigned char  BPB_Media;         // Media type
  unsigned short BPB_FATSz16;       // 16-bit size in sectors of each FAT for FAT12 and FAT16. For FAT32
  unsigned short BPB_SecPerTrk;     // Sectors per track of storage device
  unsigned short BPB_NumHeads;      // Number of heads in storage device
  unsigned int   BPB_HiddSec;       // Number of sectors before the start of partition
  unsigned int   BPB_TotSec32;      // 32-bit value of number of sectors in file system. Either this val
  unsigned int   BPB_FATSz32;       // 32-bit size in sectors of one FAT
  unsigned short BPB_ExtFlags;      // A flag for FAT
  unsigned short BPB_FSVer;         // The major and minor version number
  unsigned int   BPB_RootClus;      // Cluster where the root directory can be found
  unsigned short BPB_FSInfo;        // Sector where FSINFO structure can be found
  unsigned short BPB_BkBootSec;     // Sector where backup copy of boot sector is located
  unsigned char  BPB_Reserved[12];  // Reserved
  unsigned char  BS_DrvNum;         // BIOS INT13h drive number
  unsigned char  BS_Reserved1;      // Not used
  unsigned char  BS_BootSig;        // Extended boot signature to identify if the next three values are
  unsigned int   BS_VolID;          // Volume serial number
  unsigned char  BS_VolLab[11];     // Volume label in ASCII. User defines when creating the file system
  unsigned char  BS_FilSysType[8];  // File system type label in ASCII
} BootEntry;
#pragma pack(pop)
```

Directory entry

```
#pragma pack(push,1)
typedef struct DirEntry {
  unsigned char  DIR_Name[11];      // File name
  unsigned char  DIR_Attr;          // File attributes
  unsigned char  DIR_NTRes;         // Reserved
  unsigned char  DIR_CrtTimeTenth;  // Created time (tenths of second)
  unsigned short DIR_CrtTime;       // Created time (hours, minutes, seconds)
  unsigned short DIR_CrtDate;       // Created day
  unsigned short DIR_LstAccDate;    // Accessed day
  unsigned short DIR_FstClusHI;     // High 2 bytes of the first cluster address
  unsigned short DIR_WrtTime;       // Written time (hours, minutes, seconds
  unsigned short DIR_WrtDate;       // Written day
  unsigned short DIR_FstClusLO;     // Low 2 bytes of the first cluster address
  unsigned int   DIR_FileSize;      // File size in bytes. (0 for directories)
} DirEntry;
#pragma pack(pop)
```

Compiling

We will grade your submission on a CentOS 7.9 system. We will compile your program using `gcc` 4.8.5. You must provide a `Makefile`, and by running `make`, it should generate an executable file named `nyufile` in the current working directory. Note that you need to add the compiler option `-l crypto`.

Testing

To get started with testing, you can download a sample FAT32 disk and expand it with the following command:

```
$ gunzip fat32.disk.gz
```

There are a few files on this disk:

- `HELLO.TXT` – a small text file.
- `DIR` – an empty directory.
- `EMPTY.TXT` – an empty file.
- `CONT.TXT` – a large contiguously-allocated file.
- `NON_CONT.TXT` – a large non-contiguously allocated file.

You should make your own test cases and test your program thoroughly. Make sure to test your program with disks formatted with different parameters.

~~We will provide sample test cases and grading script one week before the deadline.~~

The autograder

Please download the autograder to test your program. Note that the test cases are not exhaustive. However, if you can't pass these cases, you can't expect to pass the final grading.

Submission

You will submit an archive containing all files needed to compile `nyufile`. You can do so with the following command:

```
$ tar cvJf nyufile-Your_NetID.tar.xz Makefile *.h *.c
```

Rubric

The total of this lab is 100 points, mapped to 15% of your final grade of this course.

- Milestone 1: validate usage. (40 points)
- Milestone 2: print the file system information. (5 points)
- Milestone 3: list the root directory. (10 points)
- Milestone 4: recover a small file. (15 points)
- Milestone 5: recover a large contiguously-allocated file. (10 points)
- Milestone 6: detect ambiguous file recovery requests. (5 points)
- Milestone 7a: recover a small file with SHA-1 hash. (5 points)
- Milestone 7b: recover a large contiguously-allocated file with SHA-1 hash. (5 points)
- Milestone 8: recover a non-contiguously allocated file. (5 points)

Tips

Don't procrastinate

This lab requires significant programming effort. Therefore, **start as early as possible!** Don't wait until the last week.

Some general hints

- Before you start, use `xxd` to examine the disk image to get an idea of the FAT32 layout. Keep a backup of the hexdump.
- After you create a file or delete a file, use `xxd` to compare the hexdump of the disk image against your backup to see what has changed.
- You can also use `xxd -r` to convert a hexdump back to a binary file. You can use it to "hack" a disk image. In this way, you can try recovering a file manually before writing a program to do it. You can also create a non-contiguously allocated file artificially for testing in this way.
- Always **umount** before using `xxd` or running your `nyufile` program.
- When updating FAT, remember to update **all** FATs.
- Using `mmap()` to access the file system image is more convenient than `read()` or `fread()`.
- The milestones have diminishing returns. Easier milestones are worth more points. Make sure you get them right before trying to tackle the harder ones.

This lab has borrowed some ideas from Dr. T. Y. Wong.