Project #4: Recovering the Lost Bits

1 The Story

It is the distant future—the year is 2022. You work as a digital forensic investigator for the police department in the township of Grunfolle, the capital of the newly reorganized state of Outer Karolynia.

A series of major bank heists has just taken place, but evidence is scant. A suspect has been taken into custody, along with a large number of magnetic disk drives (dating back to the 1990s). After an initial perusal, they appear to contain only incredibly cute pictures of kittens, but you suspect that there is more there. Your coworkers (who never took CPSC/ECE 3220) think you're nuts and your boss has started suspecting that you have an unhealthy obsession with cats. It's up to you to prove them all wrong and keep your job.

2 Your Mission

In this project, you will create a tool, called **notjustcats**, that parses disk images (formatted using the FAT12 file system), and prints out information about the files that the image contains, including recoverable files that someone has tried to delete. It will work as described in Listing 1, being run from the terminal with two arguments: the filename of the FAT12 image to be analyzed and the directory where recovered files will be saved.

Your program should print out a single line (to `stdout`) for each file it finds, as shown. For deleted files, the first character of the filename (`0xE5`) should be replaced by an underscore ('_'). For each file that you recover (both normal and deleted), you should also write the contents of the recovered file into file in the specified output directory. The output directory should not contain any subdirectories, and the files should be named `file0.TXT`, `file1.JPG`, ..., numbered in the same order that they are listed in your programs output. The file extensions should be the same as the original files that were stored on the disk.

Listing 1: Syntax and output format for your notjustcats program.

notjustcats <image filename> <output_directory>
Example: `./notjustcats floppys/evidence5.img ./recovered_files`

Output: (to stdout)
`FILE<tab>NORMAL<tab><file path><tab><size(bytes)>`
`FILE<tab>DELETED<tab><file path><tab><size(bytes)>`

Listing 2: Organization of the FAT12 file system.

Section	Boot Sector	FAT1	FAT2	Root Directory	Data
Sector(s)	0	1 – 9	10 – 18	19 – 32	33 ———— 2879
Cluster(s)	—	—	—	—	2 ———— 2848

3 About FAT12

The FAT file system is a fairly simple file system that was used on nearly all disks during the DOS and Windows 9x era. It is still commonly used on flash drives and supported by nearly all operating systems. A File Allocation Table (FAT) is a data structure used on a disk (hard disk, flash drive, or floppy disk) to store that status and location of the data "clusters" that are stored on the disk. Without the FAT, which acts as a table of contents, the disk is unreadable. There are several versions of the FAT file system (*e.g.,* FAT12, FAT16, FAT32). The number refers to the size of the FAT entries (12, 16, or 32 bits). FAT12 was the version used on floppy disks, and will be the focus of this assignment.

The organization of a FAT12-formatted disk image is shown in Listing 2. The storage space is divided up into "sectors," each of which are 512 bytes. The first sector (the boot sector) contains various information about the disk. The next 18 sectors contain two copies of the FAT (there are 2 copies, in case one gets corrupted). The next 14 sectors contain the root directory. Note that the root directory has a fixed size, which limits the number of entries that it can contain. Finally, the remaining sectors contain the actual file data, or data "clusters." The first segment after the root directory, is cluster 0x002 (0x000 is used to indicate an empty cluster, and 0x001 isn't used).

3.1 The File Allocation Table

The File Allocation Table contains one entry for every cluster in the data area. In FAT12, each entry is 12-bits long.

Entries: One tricky thing about FAT12, is that 12-bits is not an integral number of bytes (it's 3 nibbles). So, two entries are stored in three bytes (six nibbles). The first entry is the second nibble of the second byte followed by the two nibbles in the first byte, and the second entry is the two nibbles in the third byte followed by the first nibble in the second byte. So, given the bytes `0xab`,`0xcd`,`0xef`, the entries would be `0xdab` and `0xefc`.

The first two entries: The first cluster of the data area is cluster #2, leaving the first two entries of the FAT unused. You can safely ignore them, but in case you are curious — the first byte of the first entry contains the media descriptor (`0xF0` for a 1.4MB floppy), and its remaining bits are 1. The second entry stores the end-of-file marker. The remaining entries in the FAT correspond to a data cluster in the data area, and can have the following values:

Value	Meaning
`0x000`	Cluster is free
`0x002–0xfef`	Cluster in use. Specifies the next cluster in the file
`0xff0–0xff6`	reserved (shouldn't be there)
`0xff7`	Bad cluster
`0xff8–0xfff`	Cluster in use. Last cluster in the file.

Since the first cluster in the data area is numbered 2, the value `0x001` does not occur...ever.

Remember that all FAT entries refer to logical clusters. You will need to compute the sector number: **sector = 33 + FAT_entry - 2**.

3.2 Directories

Directories are really just files with a special format. Each 512-byte sector in a directory contains 16 directory entries (each of which is 32 bytes long). Each directory entry represents some file or subdirectory, within that directory. A directory entry consists of the following information.

Starts at (offset)	Length (in bytes)	Description
0	8	Filename
8	3	Extension
11	1	Attributes (see below)
12	2	Reserved
14	2	Creation Time
16	2	Creation Date
18	2	Last Access Date
22	2	Last Write Time
24	2	Last Write Date
26	2	First Logical Cluster
28	4	File Size (in bytes)

Note that if the first byte of the filename is 0xE5, then the directory entry is free (not a valid file or subdirectory). If the first byte in the filename is 0x00, then the entry is free, and all remaining entries are also free.

File (or directory) names and extensions are always 8 and 3 bytes respectively. They are also not null-terminated. If the file name is shorter than 8 characters, it is padded with spaces (0x20). This also applies for extensions. File (and directory) names are always uppercase in FAT12. Subdirectories will also contain two special entries named "." and "..", which link to the current directory and the parent directory respectively.

The Attributes field has 8 bits where each bit refers to a different attribute of the entry. The bits have the following meaning.

Bit	Attribute
0	Read Only
1	Hidden
2	System
3	Volume Label
4	Directory
5	Archive
6	—
7	—

Bit zero is the *least significant bit*, so if a directory entry has the Attribute `0x12`, then the entry is a hidden directory. If you see an attribute byte of 0x0F, you can safely ignore it. This is part of a long-file name (an extension that was added later), which is something you don't have to handle in this assignment. The autograder will not test this.

The **First Logical Cluster** tells you where the file's (or subdirectory's) content is stored. This just tells you where to start. If the file has more than 1 cluster, you will need to use the FAT to find the remaining clusters. If the First Logical Cluster is 0x000, then this refers to the root directory (remember that the first cluster in the data area is logical cluster 2). If the First Logical Cluster were `0x004`, then you know that the first cluster worth of data is in cluster #4. If you look up the 4th entry in the FAT, you might see `0xFFF` indicating that the file (or directory) only has one cluster. Or, you might see `0x005` indicating that the next cluster in the file is cluster #5. Each FAT entry points to the next cluster in that file or directory. To get the rest of the data, you continue iterating through the FAT, until you have found all clusters in the file.

4 Recovering Deleted Files

I recommend first getting your analyzer working with the valid files that have been stored in the file system. Next, you will look for files that have been "deleted." If they were really deleted from the image, we would be out of luck, but what really happens when a file is deleted is that the first byte of its filename (in the directory entry) is set to `0xE5` and it's pointers in the FAT are set to `0x000`.

So, until another file or directory is written over the top of the data, it's all still there (minus the first character of the filename). You can still read out

Listing 3: Making your own test images.

```
cp blankfloppy.img tmp.img
mcopy -i tmp.img test_content/test.txt ::A.TXT
mcopy -i tmp.img test_content/deleted.txt ::B.TXT
mcopy -i tmp.img test_content/kitty.jpg ::CAT.JPG
mmd -i tmp.img ::IMGS
mcopy -i tmp.img test_content/kitty.jpg ::IMGS/KITTY.JPG
mdel -i tmp.img ::B.TXT
mv tmp.img images/simple.img
```

the directory entry, see what size the file was, when it was modified, and what cluster it used to start on. If that cluster's entry in the FAT is `0x000`, then the deleted data is probably still there.

But what about the other clusters? Well, we know (from the file size) how many clusters there should be, but we really don't know where they are. So, we're going to guess, and look at the clusters that immediately follow the first cluster. If their FAT entries are `0x000`, then we will assume that they belong to the file. Sometimes, we will be wrong, but all recovery tools get confused, at some point, when you start fragmenting files. Make sure you don't take more clusters than were in the original file, and if you run into a cluster with a nonzero FAT entry, then your program should stop, and truncate the file.

One other thing. What if the deleted file is a directory? The file size will be zero, so you won't know how many clusters of directory entries you might have to recover. In this case, just recover the first cluster in the directory and any files that that cluster contains.

4.1 Testing your program

You should test your program on a variety of disk images. I will give you a few test images (including a blank image). You can and should also create your own. The `mtools`, a collection of tools for manipulating DOS disk images, are installed on the lab machines. The `man` page for `mtools` is a good place to start.

Listing 3 shows one of my scripts for creating a test image (**images/simple.img**) from a blank image (called **blankfloppy.img**) and some random files located in the `test_content` directory.

The `mtools` has one peculiarity you should be aware of. Depending on how they were compiled, you can run into strange behaviors related to

long file names. If you use all uppercase filenames and directory names in your image, and make sure that all names are shorter than 8 characters (plus a 3 char extension) it should always use short filenames, which is all you are required to support.

Your program should compile and run on the lab machines. Lack of thorough testing on the lab machines is asking for an extremely undesirable outcome.

5 Submission Instructions

This project is due by 4:00 PM on April 25^{th}. Absolutely no late assignments will be accepted. Extensions will not be granted.

Your program should be written in C. When your project is complete, archive your source materials, using the following command:

```
> tar cvzf project4.tgz README Makefile <list of source files>
```

The **Makefile** should compile your program (by running **make**). Your program should compile without producing any errors or warnings.

The **README** file should include your name, a short description of your project, and any other comments you think are relevant to the grading. It should have two clearly labeled sections titled with KNOWN PROBLEMS, and DESIGN. The KNOWN PROBLEMS section should include a specific description of all known problems or program limitations. This will not hurt your grade, and may improve it. I am more sympathetic when students have thoroughly tested their code and clearly understand their shortcomings, and less so when I find bugs that students are ignorant of or have tried to hide. The DESIGN section should include a short description of how you designed your solution (especially anything you thought was interesting or clever about your solution). You should also include references to any materials that you referred to while working on the project. Please do not include special instructions for compilation. The compilation and running of the tests will be automated (see details below) and your instructions will not be followed.

Please make sure you include only those source files, not the object files. Before you submit this single file, please use the following command to check and make sure you have everything in the `project4.tgz` file, using the following script:

```
> tar xvzf project4.tgz
```

This command should put all of your source files in the current directory. It should not create any subdirectories. Running `make` after extracting the files should build your **`notjustcats`** program, which should end up in the current directory.

Submit your `project4.tgz` file via **handin.cs.clemson.edu**. You must name your archive **project4.tgz**, and your program must compile and work. We will compile your program using your **Makefile**, and it must be named "`notjustcats`".

6 Grading

Your project will be graded based on the results of functional tests the design of your code. We will run a variety of tests to make sure it works properly. Your program should not crash. Specifically, you will receive 10% credit if your code successfully compiles, and 10% for code style and readability. The rest of your score will be determined by your library's functionality.

Your source materials should be readable, properly documented and use good coding practices (avoid magic numbers, use appropriately-named variables, etc). You code should be `-Wall` clean (you should not have any warnings during compilation). **Our testing of your code will be thorough. Be sure you test your application thoroughly.**

7 Collaboration

You may work independently or with one other person on this project. You must *not* discuss the problem or the solution with any classmates who are not your partner. All of your code must be your own code. **If you work with a partner, only one of you should submit and both of your names should be listed clearly at the top of your README file.**

You may, of course, discuss the project with me, and you may discuss conceptual issues related to the project that we have already discussed in lecture, via Piazza (do not post any code or implementation details of the algorithms). **Collaborating with any peers (who are not your partner) on this project, in any other manner will be treated as academic misconduct.**