COSC 301: Operating Systems
Fall 2019
Homework 5: File system consistency checker and fixer
Due: 12 December 2019, 11pm
Extension given: 13 December 2019, 4pm

Overview

The objective of this project is to write the equivalent of scandisk (aka chkdsk or fsck) for a FAT-12 filesystem. Scandisk is a filesystem checker for DOS filesystems. It can determine whether a filesystem is internally consistent, and correct any errors found. Scandisk for FAT file systems is analogous to the fsck program on (non-journal-based) UNIX file systems.

The FAT-12 filesystem may be rather old, but versions of FAT very similar to it are used on most USB sticks. Its main advantage is simplicity; attempting to write fsck for an FFS-based file system is actually quite hard. To avoid having to deal with a real device, we'll use a disk image that contains an entire FAT-12 file system within a 1.44 MB file. Note that the file systems that we'll be working with are *real*: they could be written to a physical 3.5 inch floppy disk and would work in any (old) system with a floppy drive (that is, if you could actually find a floppy disk and drive!)

Note: you can actually mount these images on a Linux system[1]. In a terminal, type:

```
mount -t msdos -o loop goodimage.img /mnt/floppy
```

(assuming the directory /mnt/floppy exists). This is not really recommended, since Linux may add contents to the disk images that make them more difficult to use for your scandisk program. Also, you should definitely not use the `fsck` program in Linux to detect and fix any problems on the image. That's your job!

The goals for this project are as follows:

- To learn more about how real file systems work, and how file system consistency checkers (and fixers) work.
- To learn how a FAT works, how free clusters are denoted, and how directories and files are represented.

As with past homeworks, you are welcome to work with another student on this homework**.** Again, please indicate the names of those who worked on the homework in the `README.md` in the top-level directory of the repo.

Note: there are *three* kinds of file system inconsistencies to detect and resolve in this project

[1] Note that you can also mount the image in MacOS X, but MacOS will add various files (e.g., hidden trash folder) and otherwise modify the image in ways that make it more difficult to read from your code. I'd **strongly** recommend that you not mount the image in MacOS X.

(and two of the inconsistencies are so similar, that you can really consider them to be the same situation). There are no explicit stages for this project. Rather, points will be awarded depending on how many of the bad file system images you are able to fix.

To get started, click on the Github classroom link on Moodle, and clone the repo on birds.cs.colgate.edu in the location where you intend to work on this project.

Detailed Description

An overview of the FAT-12 file system

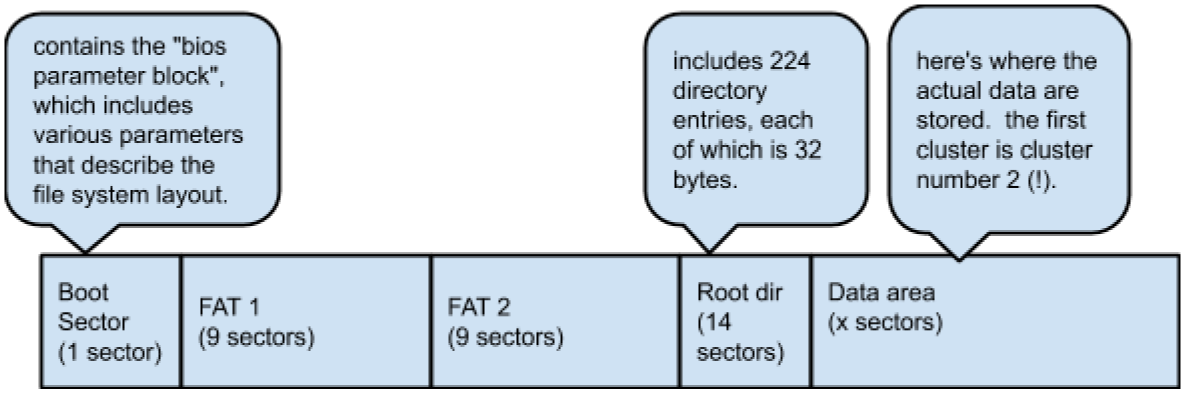
A floppy disk consists of 1.44 MBytes of data, arranged as 512 byte sectors. A FAT-12 filesystem groups these sectors into clusters (1 cluster = 1 block; these terms mean the same thing) and then maintains a File Allocation Table, with one entry per cluster. This allows a larger block size than 512 bytes. Note however, that on a floppy disk, the block size is normally 512 bytes (one sector per cluster).

FAT-12 on-disk structure

The filesystem consists of the following, in order starting at the first sector:

- **The boot sector.** This contains a lot of metadata about the file system, including useful information such as what the sector size is, what the cluster size is, now many FATs are on the disk, how many sectors are in each FAT, how much space is in the root directory, and how large the whole disk is.
- **The File Allocation Table**. This consists of one entry per cluster on the disk. Each entry is 12 bits long (3 bytes stores 2 entries), so we'll need to do some bit shifting to get the data into and out of an entry. The entries form a linked list for each file, indicating which clusters are in that file. A special value indicates the last cluster in a file. Another special value indicates a free cluster.
- **A copy of the File Allocation Table**. The purpose of this is to allow the disk to be recovered in the event that the primary FAT gets corrupted. However, it's rarely used in practice, and you should entirely ignore it for this project.
- **The Root Directory**. This consists of a sequence of 32-byte directory entries, one per file in the root directory. The space for this is pre-reserved, as indicated in the boot sector. Typically on a floppy, enough space for 224 entries is reserved. A special value in the first byte of the filename indicates when an entry is the last entry in the directory, or that this directory entry has been deleted and should be skipped over.
- **The actual data clusters**. For reasons that only Bill Gates knows, the first cluster number is 2. These clusters can be free (as indicated in the FAT), or can contain parts of files and directories.

Pictographically, here's how the disk layout looks (from left to right):


contains the "bios parameter block", which includes various parameters that describe the file system layout.
includes 224 directory entries, each of which is 32 bytes.
here's where the actual data are stored. the first cluster is cluster number 2 (!).
Boot Sector (1 sector)
FAT 1 (9 sectors)
FAT 2 (9 sectors)
Root dir (14 sectors)
Data area (x sectors)

Reading and manipulating the FAT-12 filesystem

For the starter files for this project, you will have somewhat trimmed-down versions of C header files from the FreeBSD (a UNIX-based OS[2]) implementation of the FAT-12 filesystem. We'll use these files in order to more easily read and follow the filesystem structures.

A description of these header files follows:

- **`bootsect.h`**: This gives the layout of the boot sector. One of the entries in the boot sector is the Bios Parameter Block (bpb). This differs between versions of FAT filesystems, but the one we'll work with dates from version 3.3 of DOS. (Side note: I actually remember being rather excited when MS-DOS 3.3 came out in the late '80s. Needless to say, the excitement has worn off by now.)
- **`bpb.h`**: This gives the layout of the bpb. All sorts of useful information is in here, allowing you to figure out the layout of the rest of the disk. There are also some very helpful macros for copying chunks of data in and out of FAT-12 structures. (Note: the FAT-12 structures are not "word-aligned", i.e., not aligned on 4-byte units. Copying data in and out of these structures can thus be a pain. There are a series of macros in `bpb.h` that can do the ugly work of getting or putting various sizes of integers into the on-disk structures, *e.g.*, `getushort`, `putushort`, `getulong`, and `putulong`. Use them! See the example programs described below for instances of how they can be used.)
- **`fat.h`**: This gives some special values for FAT entries. Note that there is more than one code for end of file. Bitwise-AND these values with `FAT12_MASK` to get the correct values for a FAT-12 filesystem.
- **`direntry.h`**: This gives the layout of a directory entry, and the values for many of the attribute bits. If `deName[0]` contains `SLOT_EMPTY`, there are no more directory entries after this one in this directory. If `deName[0]` contains `SLOT_DELETED`, this directory entry has been deleted, and should be skipped over. (Note: the `print_dirent` function in the sample program `dos_ls.c` is a good example to show how to read a set of directory

[2] See http://www.freebsd.org/. FreeBSD is a really nice version of UNIX, in my humble opinion.

entries.)

Additional starter files

Writing scandisk from scratch is actually a non-trivial task primarily because of having to manipulate the on-disk structures. Writing all the code to do this would take more time than I want you to take, because you'd have to figure out the precise layout of a FAT-12 filesystem for yourself, and you'd never know if you got it right. To help you, I'm giving you the source code to three programs I have written. You can and should reuse parts of the following three programs in your own code!

- `dos_ls.c`: This will do a recursive listing of all files and directories in a FAT12 disk image file. For example `./dos_ls floppy.img` would list all the files and directories in the disk image file called `floppy.img`.

- `dos_cat.c`: This can be used to cat (print) files from the disk image file. You can use this program to copy files out of the disk image onto the "regular" filesystem (e.g., using redirection). This code is intended to help you understand the details of interacting with the FAT-12 disk image files.

- `dos_cp.c`: This program can be used to copy files "into" the disk image or to copy files "out" from the disk image. This code is intended to give you an idea of how to modify FAT-12 on-disk structures (e.g., in the case of writing a file onto the image).

Besides the source to the above three programs, the files `dos.c` and `dos.h` contain some very useful functions for you to employ when writing your scandisk program:

`uint8_t *mmap_file(char *filename, int *fd)` (also `unmmap_file()`)

Memory-map a disk image, returning (effectively) a pointer to unsigned char. This pointer is something you can then pass along to additional functions for reading and manipulating different structures of the filesystem.

`struct bpb33* check_bootsector(uint8_t *imgbuf)`

Read the bootsector and ensure that it looks right. Returns a pointer to the bios parameter block (bpb).

`uint16_t get_fat_entry(uint16_t cluster, uint8_t *imgbuf, struct bpb33* bpb)`

Return the FAT entry, given a FAT cluster number, a pointer to a memory-mapped disk image, and the bios parameter block. You can test the return value to see whether the next cluster is a free cluster, another data block, or something else.

`void set_fat_entry(uint16_t cluster, uint16_t value, uint8_t *imgbuf, struct bpb33* bpb)`

Set the FAT entry at clusternum to point to the given value. Useful for creating a link between an existing data block and the next data block in the cluster chain.

`int is_end_of_file(uint16_t cluster)`

Test whether the given cluster number indicates that this is the last cluster in the chain.

`int is_valid_cluster(uint16_t, struct bpb33 *)`

Test whether the given cluster is a valid data cluster number. (Generally, you can just use `is_end_of_file()`. This function does a bit more checking than `is_end_of_file()`, which may or may not be overkill.)

`uint8_t *root_dir_addr(uint8_t *imgbuf, struct bpb33* bpb)`

Get the address in the filesystem image where the root directory is located.

`uint8_t *cluster_to_addr(uint16_t cluster, uint8_t *imgbuf, struct bpb33* bpb)`

Get the address in the filesystem image where the given data cluster is actually located (i.e., get a pointer to the data block indicated by the given cluster number).

A skeleton file for your scandisk program is found in `scandisk.c`.

The starter files also contain three disk images for you to use as you develop and test your scandisk program.

- `goodimage.img`: A consistent file system for testing.
- `badimage1.img`: A file system with one consistency problem. You should find that for one file, the file size in the metadata is smaller than the cluster chain length for the file would suggest.
- `badimage2.img`: Another inconsistent disk image, with a similar problem to the first: there is one file that has a file size in the metadata that is larger than the cluster chain for the file would suggest.
- `badimage3.img`: Yet another inconsistent disk image, with one problem: you should find a cluster that is not marked as free, but does not have any cluster chain referring to it (it's an "orphan block").
- `badimage4.img`: An image with a few problems: there are two files with inconsistent metadata file size/cluster length, a few orphan blocks, and a block in the middle of a FAT chain marked as BAD.
- `badimage5.img`. A bad gadget. Problems beyond the three issues described below plague this image. Can you figure out what happened and recover from the problem(s)?

Tasks to Accomplish

There are three basic problems that you will have to detect and fix to complete this project; as noted above two of the problems are extremely similar. There are no explicit "stages", but you will receive increasing credit for being able to detect and fix each problem. You can choose which problem to attack first, and go from there. Note that your `scandisk` program should not only work to fix the problems in these images, but should generally work for other images with similar inconsistency problems.

Project correctness credit will be awarded as follows:

- For detecting and fixing the problem in badimage1.img, you will receive a maximum of 56% correctness credit.
- For detecting and fixing the problem in badimage2.img, you will receive an additional 13% correctness credit.
- For detecting and fixing the problem in badimage3.img, you will receive an additional 13% correctness credit.
- For detecting and fixing the problem in badimage4.img, you will receive an additional 13% correctness credit.
- For detecting and fixing the problem in badimage5.img, you'll receive an additional 5% correctness credit.

Your goal in all of this is to try to salvage as much user data as is reasonably possible and return the disk to a consistent state. In particular, wiping the image clean leaves the disk in a consistent state, but is not a valid solution.

FAT-12 filesystem inconsistencies

A FAT-12 filesystem is quite simple, so there are only a limited number of ways it can become internally inconsistent. To help you think through some of the issues and to give you concrete ideas for how to fix certain problems, here are some example scenarios:

1. An existing file increases (or decreases) in size by at least one block. The directory entry and metadata (e.g., the file size in the direntry, etc.) are modified, but the OS crashes and the FAT is not updated. Thus, the size of the file in the metadata residing in a directory is inconsistent with the length of the file in clusters that can be detected in the FAT. For this kind of situation, your code should:
    a. Print out a list of files whose length in the directory entry is inconsistent with their length in data blocks (clusters).
    b. Free any clusters that are beyond the end of a file, but to which the FAT chain still points.
    c. Adjust the size entry for a file if there is a free or bad cluster in the FAT chain.
2. A new file is added to a directory entry. The new direntry is created, so *some* metadata is on disk, but the OS crashes before the FAT is entirely updated. You can assume that

the data blocks were written to disk first, so at least they're somewhere on disk. This situation is nearly identical to the one above. In this case, your code should:

   a. Perform similar actions as in the previous case as necessary (i.e., print the file name for any cases where the length is inconsistent with the length of the file in clusters, free any clusters beyond the end of the file, adjust file size entry).
   b. Create a directory entry in the root directory for any unreferenced file data. These should be named "found1.dat", "found2.dat", etc.

3. A file is being deleted. The directory entry for the file is removed, then the OS crashes. FAT entries are not updated, so there are data blocks that are "orphaned". For this kind of situation, your code should:
   a. Print out a list of clusters that are not referenced from any file, but not marked as free (i.e., identify any orphaned clusters).
   b. Create a directory entry in the root directory for any unreferenced file data. As above, these should be named "found1.dat", "found2.dat", etc.
4. The OS detects a bad cluster in the middle of a cluster chain and marks it as bad in the FAT, but crashes before it is able to make any other adjustments to the FAT or to the file metadata. You'll need to decide about the best way to fix this sort of problem.

Examples and Hints

- Be sure you understand how a FAT file system is organized. Once you do, detecting and fixing the various inconsistencies should be conceptually straightforward. The challenge will be in translating the concept to code.
- Read and understand the `dos_ls.c` code, as well as the code that it uses, `dos.c` (and `dos.h`). Same goes for `dos_cat.c` and `dos_cp.c`. These programs do a superset of the manipulation of file system structures that your scandisk program will eventually have to do. Thus, a really good starting point is to understand these programs.
- In general, the cluster size may not be the same as sector size. The size of a cluster is the sector size multiplied by the number of sectors per cluster. These items can be obtained from the bios parameter block (bpb). You're lucky in that for the images you're working with, the number of sectors per cluster is just 1 (as you'll be able to see from the bpb).
- One thing to perhaps start with is to create a structure that counts references to all the clusters. From that, you can discover which clusters are in use, which are free, and which ones fall "in between"...
- Work on detecting problems first, *then* on recovering from them. (That is, get your `scandisk` working to the point of being able to identify and print out information about problems before trying to do anything about them.) Develop a good understanding of the problems and how to detect them and get your code working well for that before starting to modify the images.
- Make copies of the disk images before using your scandisk program on them, or become proficient at rolling back changes to files within git. If you accidentally modify an image in a "bad" way, you'll want to be able to restore it to its previous state.

- When you fix a file, don't necessarily expect the contents to be valid. For example, if you fix the metadata for an image (jpeg) file to make its cluster chain consistent with the metadata file size, the file contents still may not be readable. (That's mainly due to the way in which the disk images are put in a corrupt state.)

Submission

Just commit your code to git and push to github.