Operating Systems, Coursework 2

November 2016

The objective of this coursework is to write the equivalent of *scandisk* for a DOS (FAT12) filesystem on a floppy disk. *Scandisk* is a filesystem checker for DOS filesystems. It can check that a filesystem is internally consistent, and correct any errors found.

The FAT-12 filesystem may be rather old, but it is still used on floppy disk drives. Its main advantage is simplicity, which makes it a good subject for a coursework. Trying to write *fsck* for a Unix filesystem is far less trivial! Also, to avoid having to deal with real floppy drive hardware, we will use a disk image of a floppy that contains an entire DOS filesystem within a regular 1.44MB file. It is still a real DOS filesystem though, not something dumbed down for a coursework.

The information that follows, first gives you some more background on FAT-12, then on filesystem inconsistencies, and finally provides the actual question for the assignment.

FAT 12, as used on Floppies

A floppy disk consists of 1.44MBytes of data, arranged as 512 byte sectors. A FAT-12 filesystem groups these sectors into clusters and then maintains a File Allocation Table, with one entry per cluster. This allows a larger block size than 512 bytes. However, on a floppy disk, the block size is normally 512 bytes (i.e. one sector per cluster).

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, how 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). 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 is rarely used in practice, and you will ignore it for the purposes of this coursework.
- **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 indicted 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 clusters**. For some reason the first cluster number is 2. These clusters can be free (as indicated in the FAT), or can contain parts of files and directories.

You might choose to look at http://wiki.osdev.org/FAT for more information on the FAT format.

Provided below are slightly trimmed down versions of the C header files from FreeBSD's implementation of the FAT-12 filesystem. These give the precise layout of all these datastructures and you should treat them as authoritative.

- `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 will work with dates from version 3.3 of DOS.
- `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.
- `fat.h`: This gives some special values for FAT entries. Note that there are 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.

Filesystem Inconsistencies

A FAT-12 filesystem is quite simple, so there are only a limited number of ways it can become internally inconsistent. We will consider only two for this coursework:

- There is a file that is not referenced from any directory. To detect this, you need to walk the directory tree and follow the FAT linked list for every file, in order to discover the clusters that are neither free in the FAT nor used in any file.

  Unreferenced clusters will be parts of lost files. They will form linked lists in the FAT, one per lost file. You need to figure out how many lost files there are, how long each is, and what the first block of each is.

  Once you have discovered such unreferenced files, you will need to create a directory entry so the files can be recovered.
- The length in the directory entry can be inconsistent with the length in the FAT itself. For example, the FAT may indicate a file contains four clusters of 512 bytes, but the directory entry indicates the file is less than 1537 bytes.

Once you have discovered such an inconsistency, modify the FAT to free up the clusters that are past the end of the file (as indicated by the dirent).

Provided Code

Writing scandisk from scratch would take more time than it is worth spending, because you would have to figure out the precise layout of a FAT-12 filesystem for yourself. To help you, here is the source code to two programs:

- `dos_ls:` 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_cp:` This can be used to copy files into and out of the disk image file.

  For example, "`./dos_cp floppy.img a:foo.txt bar.txt`" will copy the file foo.txt from the disk image called floppy.img and save it as bar.txt in the regular filesystem.

This code is intended to help you understand the details of interacting with the FAT-12 disk image files. You can also re-use parts of these files as you wish to help you write your *scandisk* utility.

The question

You are expected to write a program called *dos_scandisk* that takes a disk image file, checks it for the two types of inconsistency listed above, and fixes the inconsistencies.

Your code should:

1. Print out a list of clusters that are not referenced from any file. Print these out separated by spaces in a single line, preceded by "Unreferenced: ". For example:

   ```
   Unreferenced: 5 6 7 8
   ```

2. From the unreferenced blocks your code found in part 1., print out the files that make up these blocks. Print out the starting block of the file and the number of blocks in the file in the following format:

   ```
   Lost File: 58 3
   ```

   This indicates that a missing file starts at block 58 and is three blocks long.

3. Create a directory entry in the root directory for any unreferenced files. These should be named "found1.dat", "found2.dat", etc.

4. Print out a list of files whose length in the directory entry is inconsistent with their length in the FAT. Print out the filename, its length in the dirent and its length in the FAT. For example:

   ```
   foo.txt 23567 8192
   ```

```
bar.txt 4721 4096
```

5. Free any clusters that are beyond the end of a file (as indicated by the directory entry for that file). Make sure you terminate the file correctly in the FAT.

If your program has succeeded in fixing the filesystem with 3 and 5, re-running on the fixed filesystem should produce no output for 1, 2 or 4.

The version of your code you hand in should not print any other output to stdout (output to stderr will be ignored by the marking script). If you produce other output to stdout, the marking script may fail, and you may miss out on marks if we do not happen to notice this.

You will be supplied with two disk image files:

- `floppy1.img` contains a few files and a single directory (in addition to the root directory). This is a consistent filesystem that you can use to test that your code does not detect any problems that do not exist.
- `badfloppy1.img` is similar to floppy1.img, but contains an unreferenced file. This is for you to test against. Your code should be able to fix this, as described in 3 above.
- `badfloppy2.img` is similar to floppy1.img but contains unreferenced files and a length inconsistency. *Your code should be able to fix this disk image*.

In addition to these, your code will be tested on another inconsistent disk image file that you will not be given.

What to hand in.

1. The commented, properly indented source code for your `dos_scandisk` utility. You are expected to subnit a single C file called `dos_scandisk.c` which links against `dos.o` in the same way that `dos_ls` and `dos_cp` do. You must adapt the `makefile` to go with this. **Your code will be compiled by typing 'make'; if this doesn't work, your code is deemed not to compile.**
2. The output from running your code on `badfloppy2.img`, listing all the unreferenced clusters and the files with a length inconsistency.
3. A one-page description of how *your code* detects and fixes the inconsistencies. This should be in a file called `description.pdf`

The reason for this coursework.

- Filesystems are a core part of operating systems. Although FAT-12 is somewhat obsolete, it is still in use today. It is simple enough to understand quite quickly.

- As you do not have root access on the lab machines, you are not asked to write a real filesystem, but a filesystem in a disk image file is close enough, and captures the important parts of the problem without wasting time messing about with the hardware itself.
- Scandisk is a nice problem because it involves understanding how the FAT works, how free clusters are represented, how directories are represented, and the general concept of filesystem consistency.
- You are given a lot of code to make the problem tractable in the time available. But also the best way to learn a programming language is to learn it by example.

Marking.

Your programs will be both run and read. As a consequence, you will be awarded marks for two things: (i) the ability of your program to compile and execute the commands it should and (ii) the structure and generality of the code, and the quality of documentation provided.

Your code must compile and run on frontal.cs.ucl.ac.uk (or . It will be tested on this machine, and if it fails to compile or run, then you will not be awarded marks for any of the first part. The code will be tested by running the same sequence of commands for everyone and you will be marked on the outcome of that test. You will not be told what this sequence is, so you are strongly advised to test your program, and even to get others to attempt to find its faults before you hand it in.

Your code will also be read. It should be professional: neatly laid out, well commented and come with explanations of what each function is trying to achieve and how. You are recommended to use doxygen formatting or something similar. Even if it doesn't run, it may still gain some marks for attempting to solve the problem in a reasonable way.

The coursework is worth 10% of the marks for the course. Not submitting a reasonable attempt at the coursework will render you incomplete for the course, rather than only losing 10%. While you can discuss general ideas with your friends, your code must be your own. Electronic submission makes it simple to run automated plagiarism detection software.

Your program must be submitted electronically on the Moodle page. To do this, you should zip together a directory named firstName_lastName containing the source, makefile, profile file and anything else you want to submit. You should include a 'README' file that describes each file included in that directory.

You can resubmit your coursework more than once if you need to, up until the deadline – only the last version will be marked.