O.S. Lab 3: A Simple File System

In order to access a file stored on a disk you have to know its location and sectors used. For user convenience we prefer to have the o.s. track this information so the user can simply refer to files by name. This is the purpose of a file system; to match a file name with its location and footprint on the disk. File systems greatly vary in complexity so our purpose is to work with a very simple one.

Along with this lab writeup are two files, **floppya.img** and **filesys.c**. The first of these is a simulated 3½" 1.44Mb floppy disk formatted with our file system and including a couple of application programs. The second is a starter file for the program you will write to manipulate the files stored on the disk.

Introduction to the File System

The primary purpose of a file system is to keep a record of the names and sectors of files on the disk. The file system in this operating system is managed by two sectors toward the beginning of the disk. The *disk map* sits at sector 256, and the *disk directory* sits at sector 257.

Run the command **hexdump -C floppya.img** and note the following output toward the bottom:

```
      …
Map   *
↳     00020000  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff 00  |................|
      00020010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
      *
↱     00020200  63 61 6c 00 00 00 00 00  78 01 03 00 00 00 00 00  |cal.....x.......|
Dir   00020210  66 69 62 00 00 00 00 00  78 04 01 00 00 00 00 00  |fib.....x.......|
      00020220  6d 73 67 00 00 00 00 00  74 05 01 00 00 00 00 00  |msg.....t.......|
      00020230  74 33 00 00 00 00 00 00  78 06 09 00 00 00 00 00  |t3......x.......|
      00020240  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
      *
      00168000
      …
```

We're displaying the contents of the disk as though it were one big file. The far-left column is the starting address of the subsequent 16-byte quantity. Each byte's contents is displayed in hex, then in ASCII on the far right.

The map (starting at 0x20000) tells which sectors are available and which sectors are currently used by files. This makes it easy to find a free sector when writing a file. Each sector on the disk is represented by one byte in the map. A byte entry of -1 (0xFF) means that the sector is used. A byte entry of 0 (0x00) means that the sector is free. In this example sectors 0 through 14 are in use by miscellaneous files including the boot loader (sector 0). Sectors 15 through 511 are free.

The directory (starting at 0x20200) lists the names and locations of files stored on the disk. There are 32 file entries in the directory, each of which contains 16 bytes (32 × 16 = 512, which is the storage capacity of a sector). The first eight bytes of each directory entry is the file name. (This is an historic hold-over; MSDOS file names followed an 8-dot-3 pattern.) The next byte indicates file type: "t" for a text/printable/viewable file, "x" for an executable binary. The two bytes after that are the starting position and number of sectors, respectively, which tell where the file is on the disk (per our choice of contiguous allocation of files for simplicity). If the first byte of the entry is zero (0x0), then there is no file at that entry.

Consider the boldfaced directory entries in our disk image. These indicate that there are valid files (with legal names indicated by the visible hex characters) at sectors 1, 4, 5 and 6. (Zero is not a valid sector number but a filler since every entry must be 16 bytes). If a file name is less than eight bytes, the remainder of the first eight bytes should be padded out with zeros. You should note, by the way, that this file system is very restrictive. Since one byte represents a sector, there can be no more than 512 sectors used on the disk (256K of storage). It's likely worse than this since any file's initial sector must lie in the first 256 of these 512 sectors. Additionally, since a file can have no more than 255 sectors, file sizes are limited to this 128K. We can expand the amount of useable storage by using multiple sectors for the map and directory, but for this project this is adequate storage. For a modern operating system, this would be grossly inadequate.

```
Disk usage map:
      0 1 2 3 4 5 6 7 8 9 A B C D E F
     --------------------------------
0x0_  X X X X X X X X X X X X X X X .
0x1_  . . . . . . . . . . . . . . . .
0x2_  . . . . . . . . . . . . . . . .
0x3_  . . . . . . . . . . . . . . . .
0x4_  . . . . . . . . . . . . . . . .
0x5_  . . . . . . . . . . . . . . . .
0x6_  . . . . . . . . . . . . . . . .
0x7_  . . . . . . . . . . . . . . . .
0x8_  . . . . . . . . . . . . . . . .
0x9_  . . . . . . . . . . . . . . . .
0xA_  . . . . . . . . . . . . . . . .
0xB_  . . . . . . . . . . . . . . . .
0xC_  . . . . . . . . . . . . . . . .
0xD_  . . . . . . . . . . . . . . . .
0xE_  . . . . . . . . . . . . . . . .
0xF_  . . . . . . . . . . . . . . . .

Disk directory:
Name     Type Start Length
cal      exec     1   1536 bytes
fib      exec     4    512 bytes
msg      text     5    512 bytes
t3       exec     6   4608 bytes
```

The filesys.c Program

Included with this lab is the **filesys.c** starter file. All it does now is open the disk image, read the map and directory sectors into arrays, print the information (in the format as seen at right), write the two sectors back to the disk image (in code which is commented out) and close the disk. This program will provide the starting point for this lab. Most of what you will need to do in this lab can be figured out by reverse engineering this starter file.

Overview of the Lab

The purpose of this lab is to turn **filesys.c** into a simple tool for manipulating the files stored on the provided disk. It will accept options from the Linux command line and alter the disk accordingly. In short, once compiled (**gcc -o filesys filesys.c**) you will implement these four commands:

- **./filesys D** *filename*   delete the named file from the disk
- **./filesys L**   list the files on the disk
- **./filesys M** *filename*   create a text file and store it to disk

- **./filesys P** *filename* read the named file and print it to screen

Any other option yields an error message. Let us review each of these in turn.

Option L: List files

Tweak the existing code to get rid of the disk map and list the files so that the names print out in the traditional MSDOS 8-dot-3 format (**cal.x**, **msg.t**, etc.) without spaces in the middle. Also remove the starting sectors, keep only a list of file names and bytes used. Finally, by the way we designed the file system, a total of 511 sectors (or 511 x 512 = 261,632 bytes) is being tracked. At the end of the list of files, print out the total space used by the files and the total free space remaining, both in terms of the number of bytes.

Option P: Print file

Start with the disk directory and map loaded as in the current program. Do the following:

1. Go through the directory trying to match the file name (without "t" or "x" extension). If you don't find it, return with a "file not found" error. If you do find it but it's an executable file and not printable return with an error message.
2. Using the starting sector number and sector count in the directory, load the file into a new buffer of size 12288 (our max file size) via *fseek* and *fgetc* as shown in the starter file.
3. Starting from index zero print each individual character until you run into a zero (the end-of-file delimiter). Return.

This is easy to test given the four files on the disk image, only one of which (**msg**) should be viewable. Also remember to test the "file not found" error.

Option M: Create and store a text file

Prompt the user for a string of text and create a 1-sector (512 byte) file storing it. Writing a file means finding a free directory entry and setting it up, finding free space on the disk for the file, and setting the appropriate map byte(s). Your function should do the following:

1. Search through the directory, doing two things simultaneously:
   a. If you find the file name already exists, terminate with a "duplicate or invalid file name" error.
   b. Otherwise find and note a free directory entry (one that begins with zero).
2. Copy the name to that directory entry. If the name is shorter than 8 characters, fill in the remaining bytes with zeros. If the name is longer than 8 characters keep only the first 8. Include the "t" file type at the ninth location in the entry.
3. To write the actual file to disk:
   a. Find a free sector on the disk by searching through the map for a zero.
   b. Set its map entry to 255.

   c. Add the starting sector number and length (1) to the file's directory entry.
   d. Write the buffer holding the file to the correct sector.

4. Write the map and directory sectors back to the disk.

If there are no free directory entries or not enough free sectors left, your function should terminate with an “insufficient disk space” error. Testing should be obvious (create a file and try printing it out) but remember the error cases.

Function D: Delete file

Deleting a file takes two steps. First, you need to change all the sectors reserved for the file in the disk map to free. Second, you need to set the first byte in the file's directory entry to zero.

Your function should find the file in the directory and delete it if it exists. To accomplish this, do the following:

1. Search through the directory and try to find the file name. If you can’t find it terminate with a “file not found” error.

2. Set the first byte of the file name to zero.

3. Based on the starting sector and file length, step through the sectors numbers listed as belonging to the file. For each sector, set the corresponding map byte to zero. For example, if sector 7 belongs to the file, set the *eighth* map byte to zero (index 7, since the map starts at sector 0).

4. Write the character arrays holding the directory and map back to their appropriate sectors.

Notice that this does not actually delete the file from the disk. It just makes the disk space used by the file available to be overwritten by something else. This is typically done in operating systems; it makes deletion fast and un-deletion possible. Test as above: create a file and then delete it, followed by a check of the **hexdump**.

Conclusion

There are a lot of other things a file system has to do (like copying files) but this should give you an idea of what’s involved in making a file system work in an o.s.. When finished submit your revised **filesys.c** program to the drop box.

Last updated 11.4.2020 by T. O’Neil.