CprE 308 Lab 8: Unix File Interface

Iowa State University

Fall 2012

Due Tuesday 12/4/2012, 11.59pm

Purpose

The purpose of this lab is to gain an understanding of the file system interface by implementing a subset of this interface on a file system that uses the FAT-12 format.

You will be given three C files: *main.c*, *sfs.c* and *sfs.h*. You will also be given a FAT-12 floppy disk image. You need to modify the files *sfs.c* and *sfs.h* and provide a working implementation for a FAT-12 interface. The *main.c* can be used as a driver program to test your implementation. You are encouraged to modify it and test your interface in as many ways as you may like.

Submission

Please submit a tar of your modified *sfs.c sfs.h*, in addition to a lab summary.

Tasks

Using the supplied skeleton code, implement the functions *sfs_initfs*, *sfs_concludefs*, *sfs_open*, *sfs_read*, and *sfs_close*. You may use the structure "SFS" defined at the top of sfs.h to maintain any state variables you wish, and you will notice that a reference to this structure is passed to all of the functions. You can organize your code so that all the information of the current state of the file system can be stored here.

For the functions sfs_open, sfs_read, and sfs_close, please ensure that they conform to the return values specified in the man pages for the functions open, read, and close. Also, note that you do not have to handle writing to a file. Pay special attention to error handling. For instance, if the pathname specified as an argument to "sfs_open" does not exist, "sfs_open" should return -1.

Hints

The code you wrote for lab 7 will be useful here. The boot sector contains a lot of the information used to open a file, and seeking through more than one cluster is impossible without knowing the “sectors per cluster” and “bytes per sector” values from the boot sector.

The FAT stores a list of entries corresponding to the next cluster used by a file. If a file starts at logical cluster 0, the 0^{th} entry of the FAT will contain the value of the next logical cluster used by the file. Generally the value -1 is used to signify that the current cluster is the last cluster in the file, though any value from 0xFF8 to 0xFFF can be used. Reading these values from a FAT-12 device can be problematic because FAT stores numbers in little endian. Remember that in FAT-12 the cluster number is stored using 12 bits. As an example, let the first few entries in the FAT be:

0	0x123
1	0x456
2	0x789
3	0xABC

On the disk, this would be represented as:

Offset	Stored Value
0x00	0x23
0x01	0x61
0x02	0x45
0x03	0x89
0x04	0xC7
0x05	0xAB

Also, there is a difference between logical and physical addressing in FAT. The cluster information in the FAT is saved by logical cluster, but the information on the disk is saved according to physical cluster. The physical cluster number equals the corresponding logical cluster number minus 2.

For example, Entry 0 in the FAT above would actually says that the physical cluster 0x121 is the cluster that continues the file in cluster 0.

For ease of parsing the input, you can assume that there will not be a filename with whitespace in it, and in addition, there will not be a file and a subdirectory with identical names within the same directory.

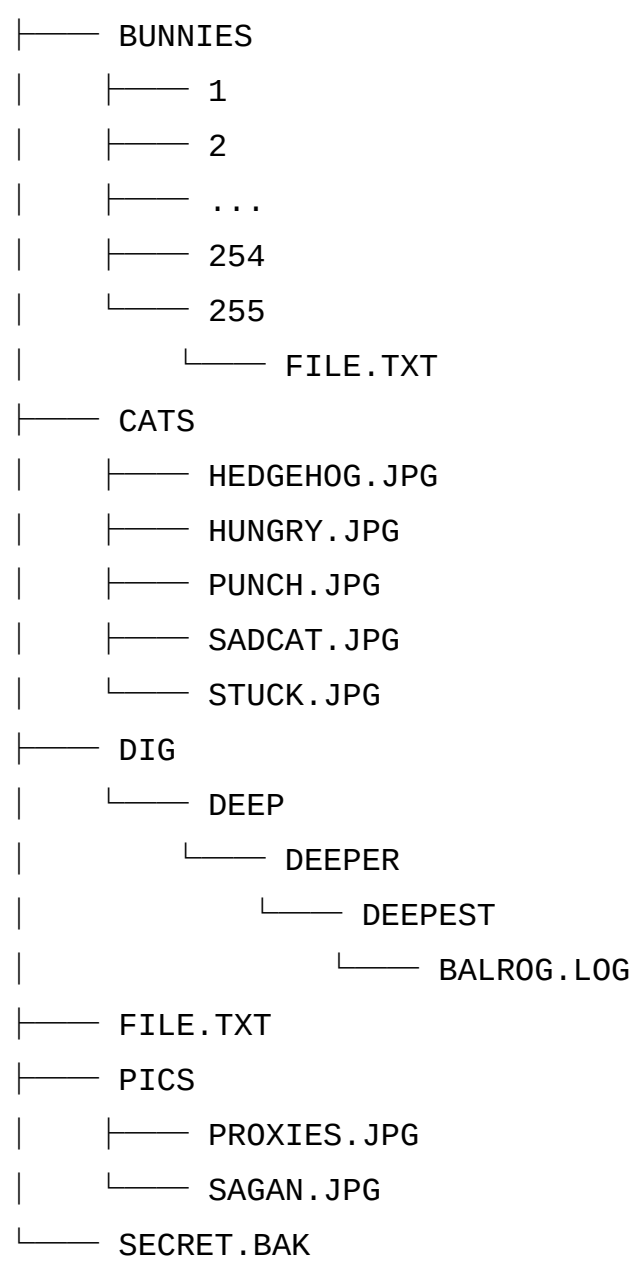
The root directory will span exactly 32*(number of root directory entries) bytes in FAT12. The first sector after the root directory is physical cluster 0. As mentioned previously, this is also logical cluster 2.

Note that the EOF character is not physically saved to the drive. You must read the file size from the directory listing once you find the file requested, otherwise there is no way to detect that the file has ended.

For testing, in the image we gave you, there is a file “FILE.TXT” in the root directory with the contents:

```
Hello! You've found a text file.
Good job.
```

For testing directory traversal, the contents of the image look like this.



The file /BUNNIES/255/FILE.TXT can be used to check whether or not you handle multi-cluster directories correctly, and the file /DIG/DEEP/DEEPER/DEEPEST/BALROG.LOG can be used to check if you are traversing directories correctly. Any of the .JPG files can be used to check whether or not you are reading file contents correctly.

If you have implemented your functions correctly, dumping the contents of any of the .JPG files to a file in your native file structure should result in a picture that you can open and view without errors. Look at the file main.c for an implementation of this.