Guthrie Alexander

CS4414 (Spring 2016)

Project 4 writeup

Due: April 28 2016

Design and implementation of project 4

**Design**

After reading the assignment thoroughly, and looking at the disk layout pdf in the Collab resources page, I had a grasp of what I had to do: build a file system that would allow the creation and deletion of files, and allow writing to and reading from files, all while being persistent (i.e. dismount and remounting the file system would keep the files that were stored on disk to be available to be opened). The general approach was to follow the assignment instructions and implement one function at a time.

In retrospect, after putting together make\_fs, mount\_fs and fs\_create, I should have then implemented write and read, because otherwise the files are always going to be 0 in length, and it’s hard to test whether anything else works. I also ended up spending the most time troubleshooting my fs\_read and fs\_write functions, so developing those in conjunction with test strings would have been helpful.

**Make\_fs()**

Before anything else could be done, I would need to create a new file system. I followed the instructions in the assignment as well as the “disk layout for project 4.pdf” file in Collab; in other words, I implemented a FAT file system.

In the make\_fs() function, after some error checking, I call the make\_disk() function in the disc.c file, and then open it. First, I need to write the superblock, as this will contain the data needed for the other pieces of the file system. I wrote my superblock to be 0108091632630000, which is broken up into pieces. The first four bytes indicate where the directory is stored (here, its between blocks 01 and 08, inclusive). The next four bytes indicate where the FAT is stored (between 09 and 16 blocks, inclusive). Finally, the next four bytes indicate where the data should be stored (between 32 and 63 blocks, inclusive). The last 4 bytes are unused.

Once the superblock is created, the only other thing to do is make sure the space is allocated properly on the virtual disk. To do this, I just wrote 0s for each byte for the directory, FAT and data sections.

After that, the only thing needed is to close the disk.

**Mount\_fs()**

Mount\_fs() needs to not only open the disk, but also initialize the “meta data,” aka the OFT, directory and FAT that will be used while the disk is mounted. The first thing to do, though, is grab the starting and ending blocks for the directory, FAT and data from the superblock. These values are just stored in variables.

Next, the OFT is initialized. Every time a disk is mounted, the OFT will be initialized to all 0s, since it’s just relevant while the disk is open, and not when it’s closed. This is the next thing that’s done in the mount\_fs() function.

First, I read and populate the FAT that’s loaded “into memory” (i.e. while the disk is open), which is done by reading the contents on disk. I go through and load each block of the FAT into the char \* buffer. This char \* buffer contains 4 FAT blocks, each 4 bytes long. I iterate through each of the four parts of the block in buffer, find the status (note that in order to change it to an int, I did the following: status = buffer[p\*4] – ‘0’; I couldn’t get anything else to work for a single char). Then, I grab the next block that the FAT entry holds, which is stored in the subsequent 3 bytes of the block. I also store the block number in the FAT, but I don’t think I ever end up using it (I always just add or subtract 32 from the index or block number if I need to switch between using the two).

Finally, I set the next\_block variable for the FAT entry using the atoi() function.

Loading the directory from disk into memory is done in a similar fashion, I load a block from disk at a location specified by the superblock, one at a time, and parse the block into 4 sections: 1 byte for the status, 2 bytes for the first data block, 4 bytes for the file name, and 3 bytes for the file length.

**Dismount\_fs()**

For dismount\_fs(), I just needed to write back all the data from the directory and FAT to disk. The OFT didn’t matter. Since I was doing this a few times, I wrote updateDir() and updateFAT() so I could just call those functions rather than copy/paste the same writeback functionality in several different methods.

**Fs\_create()**

One thing to note here – because 4 bytes were allocated in the OFT and in the directory for file name, I filled in 0s into empty spaces of the file name (e.g. if the file name is fa, on disk its stored 00fa). I was having a problem storing the file names on disk on a dismount, and 0s are not a legal character for file names anyway, so it works out.

After dealing with setting the file name and error checking, the only thing creating a file really does is find an open spot in the directory, and set the status equal to 1, and set the file name. Allocating blocks is only done when writing to the file.

**Fs\_open()**

After setting the filename (described above), and some error checking, and going through the directory to make sure the file exists, fs\_open() goes through the OFT and finds an available file descriptor (i.e. an index in the OFT). If the OFT is full, it won’t allow another file to be open. Once a file descriptor is located, the status is set to 1, offset set to 0, and the directory index for the OFT is set. Now the file is ready to be changed or read.

**Fs\_close()**

When a file is closed, its entry in the OFT is cleared. After error checking, and updating the directory, I just set the OFT at index fildes entry equal to the initial state (status equal to 0, offset = 0, dir\_index = 0). This makes the fildes available for another file to be opened.

**Fs\_delete()**

After error checking (if the file exists, or if the file is open), the fs\_delete function locates the file in the directory first. Then, it writes zeroes for all the data blocks that were associated with that file (which it grabs from the FAT). It continues to write zeroes for the blocks allocated to that file according to the next\_block value for the corresponding FAT entry, until next\_block is 0 (meaning that’s the last block in the file). In the FAT, each entry that corresponded to a block in use by the file has its status set to 0, meaning its available again.

Once the data is cleared, the directory entry in the directory table is cleared. Both of the changes in the FAT and directory tables are then written to disk.

**Fs\_write()**

Write does some error checking, then it determines how many blocks the file will take up after the write (offset + nbyte / 16). Then, it allocates a block in the FAT for the file to use, if it has length 0. This is the first time a file has data space allocated to it (i.e. it’s not done in fs\_create or fs\_open). Then, since it’s already determined how many blocks will be needed for the file, it sets the FAT so that each entry points to the next block that the file will need, allocating the correct number of blocks for the file.

Then, fs\_write checks to see if the offset is 0. If so, it just writes the data from buf (which has been converted to a char \* buf\_char) to disk, one block at a time, using the number of blocks needed from before to iterate through to write all the data.

If the offset is not 0, then it’s a little more complicated. The function has to go through the blocks until it finds the block where the offset is located. Then, the block is read, and concatenated with the beginning of buf\_char, starting at offset+1 in that particular block.

Overwriting data is allowed this way; fs\_write() doesn’t care what data is currently in the file, unless it’s before the offset.

**Fs\_read()**

Read does some error checking at the start, then it locates the block where the offset is found. Then, starting at the offset value (which is now a 0-15 value, corresponding to an individual byte in the starting block for the read), the values read into the tmp char\* are copied to the readString array. Note how they are have different indicies (the char \* tmp is being iterated through starting at offset, the readString starting at 0). If the end of the block is reached but more bytes are requested to be read, the next block is grabbed from the FAT table, and the process continues.

Eventually, the readString char \* is copied into buf using the memcpy function, and the function returns.

**Fs\_get\_filesize()**

This function just returns the length of the file as listed in the directory. The length of the file in directory is updated every time there is a write to the file, so it’s always up to date.

**Fs\_lseek()**

Really all lseek does is manipulate the offset in the OFT. It will only do this if the file exists, and the offset in the argument isn’t out of bounds (it checks both the length of the file and that the offset, if negative, isn’t greater than the current offset for the file).

**Fs\_truncate()**

Since I had already implemented read and write by the time I came to truncate, I just used those to implement this function. After some error checking (making sure it was a valid fildes, and that length was less than the file’s size), I moved the file offset to the beginning, read the first **length** bytes of the file into the char \* newString, and then called fs\_write, passing that newString.

With how fs\_write is implemented, the FAT and directory tables are updated so that the data and metadata for that file are cut off after the end of the write.

Finally, the length of the file is updated, and the offset is set to 0.