Andrew Craze
CSE 430 - Syrotiuk
8 December 2014

CSE 430 Project 3

Andrew Craze

Project Notes

For Project 3 we were given the task of implementing a file system library called LibFS. The overall goal was to emulate a simple file system which supported read and write operations for both files and directories. We were given an outline of the file system along with an emulated disk library and then told to develop our file system around the predefined API. In order to complete this project, there were numerous key conventions and data structures which will be explained in greater detail.

At a high level, the “disk” was an array of 10,000 by 512 bytes, where we can read and write similar to dealing with sectors (but without timing issues). The file system can interact with the “disk” through the LibDisk API, which was given at the start of the project. The disk is separated into 5 types of sections: a superblock, an inode bitmap, a data bitmap, sectors reserved for inodes and lastly sectors reserved for data. The superblock always belongs to the first ‘sector’ on disk and contains a ‘magic number’ which is used to validate the disk image when booting up. The next four sectors were reserved to hold bitmaps, then the following 250 sectors were dedicated to storing inodes and the last 9745 sectors were used as storage for data blocks.

Bitmaps were used to keep track of where space had been allocated for both inodes and data blocks. In addition, four inodes can be stored inside a single sector, which required developing a policy to perform allocations. My approach followed a best-fit policy, where both directories and files had global counter variables which were updated upon creation of a new inode. I decided to not mix file types within a sector, which allowed for greater ease when searching through subdirectories to find a file. The inode bitmap only took up one sector, however the data block bitmap took up three additional sectors since each file is allotted at most 30 data blocks for storage.

In order to map between addressed and memory objects, I stored file and directory names inside the ‘pointers’ for a given inode. By using the name as a key, I mapped the position of the inode pointer element to it’s corresponding data block using a global DATA_OFFSET constant which allowed me to get the inode of a subdirectory. This allowed to quickly look to see if a file existed in a specific directory or not. When a file is opened, it’s data block index is computed and then stored as an integer in an open-file table. This table is an integer array and the element’s index is returned as an integer file descriptor.

Ambiguities

The first ambiguity I encountered in the spec was how to model the Open File Table. Since I had made the decision to use file and directory names as keys in mapping between address and memory objects, I needed to constantly compute the corresponding data block index which is not really efficient. I decided to compute the data index once and then store it as an integer element inside the Open File Table. With this approach, I treated directories and files the same and stored both types of data indices within the table. Once a file is closed, it's corresponding directory would also close.

Another ambiguity was how to best allocate space for the inodes and directory files. Since there are 4 inodes to a sector and 16 directories to a data sector, I felt that it would be best to do a straight-forward approach and follow a best-fit policy. This involved having separate global variables which acted as counters for storing each file type. I did not mix file inodes with directory inodes in the same sector, instead i separated them using the counters which made searching for a specific file type that much simpler. However this approach does have it's drawbacks; tons of moving pieces which can be difficult to debug.

Features Not Implemented:

Here is a list of features that either are not working or were not implemented:

- The program does not handle file sizes larger than a single sector currently.
- It also does not have a current file pointer feature to allow.
- It currently only supports 2 levels deep of sub directories.
- File_Read()
- File_Seek()
- File_Unlink()
- Dir_Size()
- Dir_Read()
- Dir_unlink()