CSE 430 Project 3

David Ganey

December 7, 2014

1 Introduction

This project simulates a file system with basic creation, writing, and persistence features. The project demonstrates knowledge of Unix-style file systems and uses a variety of data structures and programming techniques.

2 Data Structures and Utility Functions

This section will discuss some of the data structures and basic algorithms implemented for use in the File/Directory APIs.

2.a Bitmaps

The bitmaps are implemented as arrays of characters to save space. Each character stores 1 byte, which can represent the state of 8 inodes or directories. Two methods were written to work with these bitmaps, called `int findFirstAvailableInode()` and `int findFirstAvailableDataSector()`. Each of these functions loops through all characters in the bitmap, and for each one, checks each bit (from highest to lowest). When it finds a 0, the function flips it and returns the corresponding position. When the inode function returns, it returns the specific number of the inode. The caller then needs to deduce (using integer division) in which sector that inode is actually stored on the disk (since each sector contains four inodes). It also needs to know the offset to the first inode, since inode 0 is actually stored after the superblock and both bitmaps on the disk. In contrast, the data sector function simply returns the actual sector to be used.

2.b Searching

When creating files or directories, it is essential to traverse the file structure until the *parent* of the new node is found. A recursive function, called `int searchInodeForPath`, accomplishes this. The parameters given to this function are `int inodeToSearch`, which starts as 0 (the root inode number), a vector of `std::strings` which represents the path (passed by reference to avoid copies), and `int pathSegment`, which tells the function which index of the vector to check. At each call of this function, it first checks if `pathSegment` is at the end. If the current segment is the end of the path (the name of the new directory/file), then `inodeToSearch` already represents the parent, and we simply return that. Otherwise, the function loads the current directory inode and searches the directory contents for names which match the vector entry at the current segment. When it finds one, it makes the recursive call:

```
return searchInodeForPath(curEntry.inodeNum, path, pathSegment + 1);
```

2.c Arithmetic

Basic integer division and modulo functions are heavily used in this project. Most sectors contain more than one entry (e.g. 4 inodes per sector), so some arithmetic is required to determine how to reference the specific section of the block and the specific sector in which the block should be written. The code below demonstrates this arithmetic, showing how an inode number of 6 could be used to write an inode to index 2 of sector 1:

```
int inodeNum; //either a parameter or returned result of bitmap function.
Inode* inodeBlock = (Inode*)calloc(4, sizeof(Inode)); //allocate an entire block of Inodes
//to modify a particular inode:
inodeBlock[inodeNum % 4].fileType = 1;
//to save the whole block:
int inodeSector = inodeNum / 4 + ROOT_INODE_OFFSET;
Disk_Write(inodeSector, (char*)inodeBlock);
```

2.d Open File Table

The open file table is basically implemented as an array. However, to make removal of entries easier a map is used instead. The table maps from integers to a custom struct called `OpenFile`. This structure contains the inode number of the file and that file's current filepointer. When a file is opened, a new entry is added, and when a file is closed, that entry is removed.

2.e Other Structs

Each type of sector which could be stored has a corresponding structure which, through casting, produces a program-usable version of the bytes stored on the disk. The most basic of these structs stores the data in a file data block, and simply contains a `char[SECTOR_SIZE]`. More complicated structures, like the inode, contain multiple variables. For each structure which does not fully fill the sector, a character array called "garbage" was added to fill the remaining space. While C++ casting should simply truncate the end, this was helpful in ensuring that all structures are precisely the same size.

3 API Implementation

Most of the API functions share a similar structure. All functions with a `char* path` parameter start out by tokenizing the string, storing each "/" delimited substring as a string in a vector. Any function which needs to add a directory or inode uses the bitmap functions to flip the appropriate bit and to determine which sector to store the new file in. This means that the bitmap is filled in order – the first character starts as 00000000, then 10000000, then 11000000, etc. It is interesting to note, therefore, that the bitmap could simply be replaced with an integer to keep track of how many inodes are allocated (first 0, then 1, then 2 in the previous example). The bitmaps, however, are appropriate for the filesystem as they could be extended to support deletion, which could then free the inode sectors by toggling the bits once more.

4 Error Handling

Most unusual behavior should be handled in this application. For example, the search function makes sure to check that a file is a directory before searching into it recursively. This prevents users from creating "/dir1/one.txt" and then attempting to create "/dir1/one.txt/two.txt." It also correctly prevents duplicates by calling a helper function, `bool directoryContainsName(int directoryInodeNum, std::string name)` before creating files. If this returns true, the system will refuse to create the duplicate. Other error cases (such as running out of space) have implemented handlers, but these are more difficult cases to test and as a result may be less stable.

5 Problems

Testing this application was very difficult. In order to test `File_Write`, for example, I had to step through the file creation function to manually note the inode number for that function. I was then able to hardcode a read on that sector to grab the content written and verify in the debugger that the writes were completed. Using this method, I was able to verify that basic file writes work. I also validated that the writer is able to write in multiple sectors (and add pointers in the inode correspondingly) and I additionally confirmed that writes could append to a file after a call to `FS_Sync` and a restart of the application. I believe this

encompasses the most complex behavior and I am reasonably confident in the performance of the application, but I would not be surprised to learn that some features are less than 100% stable.

6 Conclusion

Ultimately, the project represents a functional file system. The system handles the vast majority of errors and has well-documented, extensible code backing it. Changing the size of the disk or the sector size would be trivial with this implementation, as few "magic numbers" are used. Manual testing has verified that even the more complex writing behaviors are correctly performed, and as such the system fits the given requirements.