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. While the majority of the program is complete, some features are not fully operational (see section 4 for more information).

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) which sector that inode is actually stored in on the disk (since each sector contains four inodes). 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, 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 calls itself:

```
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 map 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.

```
int findFirstAvailableInode()
{
    for (int i = 0; i < NUM_CHARS; i++)
    {
        char c = inodeBitmap->bits[i];
        for (int j = 7; j >= 0; j--) //check from highest to lowest bit
        {
            int comp = pow(2, j);
            if ((c & comp) != comp) //theres a 0 at that position
            {
                //we want to flip the jth bit
                c |= 1 << j;
                inodeBitmap->bits[i] = c;

                //now return the inode num
                return ((i * 8) + abs(j - 7));
            }
        }
    }

    return -1; //if we never find anything
}
```

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.

4 Problems

fdfsdfsd