Abhijai Singh
Luis Alfaro
Michael Schroeder
Peter Ijeoma
CSC415 FSP SP2021

File System Project

Github

https://github.com/CSC415-Spring2021/filesystemproject-schrodobaggins

Introduction

Our filesystem is a demonstration of functionality of a hierarchical filesystem, similar to the Linux File System. The concept behind this file system is that the directory model communicates with our directory services interface, or our file services interface through an open file table system. These interfaces talk with the shell and the directory entry model to pass data to and between them. In order to implement this, some design choices were made. A directory entry system exists as an array structure, while the hierarchy is implemented similarly to a linked list structure. Free space is organized and managed using a bitmap to represent logical blocks that are used with a 1, and free blocks with a 0. Memory is allocated in contiguous blocks, with file memory allocation being allocated using the concept of extents as well.

Directory Entries(dirEntry.c):

Directory hierarchy with a max of fifty (50) nodes - 48 subdirectories. Organized in an array of struct.

```
typedef struct DirectoryEntry
{
	unsigned long locationLBA;		//location of this directory entry, logical block
	unsigned long childLBA;			//location of the directory this entry points to
	short entryIndex;			//the index in the directory array
	//unsigned long id;			// id of the file
	unsigned long dataLocation;		// location where file data starts
	char name[256];				// directory name for the directory it points to
	uint64_t sizeOfFile;				// the number of bytes of the file data
	unsigned long numBlocks;			// the number of blocks occupied by the file
	time_t dateCreated;				// the date the file was created
	time_t dateModified;				// date the file was last modified
	time_t dateAccessed;				// date the file was last accessed
	unsigned long locationMetadata; // 512 file per directory
	unsigned long extents;		// an LBA location that stores array of 64 lBAs and sizes
	unsigned short numExtents;	// tracks how many extents currently allocated
	unsigned short numExtentBlocks; // the number of blocks reserved in all extents
	unsigned short isBeingUsed;		// tells whether this entry is currently in use or not
	unsigned char type;				// f for file, d for directory
} dirEntry;
```

Free Space Structure(bitMap.c):

We use a bitmap for free space management. 0 (zero) means the block is free and 1 means the block is used. We used two functions for this file, one that takes in the number of free space blocks, and returns the LBA which contains the free blocks that were not used. It also sets all of the previous blocks to used, and flips the bitmaps back to 0 if they were not used.

Formatting the Drive:

FILE for a file and DIR for a directory. The MBR is created the first time we format the volume. We initialize the root directory with a “.” and a “..” entry system. The “.” points to the starting location in memory, while the “..” points to the entry points to its parent directory. Although they point to different directory locations, their location in memory is the exact same. So if you were to do cd “.” or cd “..”, they would both take you to the same exact location. Freespace is created with booleans (makeRemove.c), they are set to zero, and then initialized to 1 when they are in use.

The bitmap is stored on disc, in the block immediately following the master boot record, being read and written whenever needed. Ordinarily the dot entry (the first entry) points to its start location in memory, and the dot-dot entry points to its parent directory. With the root directory, however, it does not have a parent so the dot-dot and dot both point to the same location in memory. This means that when the "cd .." command is used in the shell, it would behave exactly the same as "cd .", provided the current working directory is root.

The free space is initialized by creating an array of booleans all initialized to 0 other than the blocks corresponding to MBR and space being used for the current bitmap which are initialized to 1. The bitmap is stored on disc, in the block immediately following the MBR, being read and written to whenever needed.

Directory Services Interface (makeRemove.c):

Implemented basic directory functions - make directory (md), change directory (cd), and keep track of the current working directory. We also have rm to remove a directory.

File Management Interface (readWrite.c):

File management interface is responsible for handling all of the operations related to the files in our file system. In our readWrite.c file, we use a struct that gets altered before, during, and after a file is being opened and closed. The open files are referred to as “fd”, and they have their own allocated pointer file. This same file also contains other functions related to the file management of our file system. For example, they are known as “b_functions”. These b functions can do many things such as read, write, open, seek, and close. These system calls are pretty normal, everyday use system calls within linux, but we modified the read and write functions to interface with the file system.

Issues That Presented a Challenge:

Copying a file from our file system to Linux is a challenge. We have the copy showing as zero B in size but the file still is represented within the directory. We can manipulate the blob of data between directories, and even remove directories, but we are unable to actually see what is inside of the file. While we are still hoping to have this problem fixed, we probably will not have it fixed before the due date is reached.

Before that happened, however, the single biggest challenge of this project was figuring out exactly what everything should do, what major components or phases are required, and how to implement each feature. It was stated by the client that the starter code did not need to be understood fully in order to complete the assignment, however, by the end of it most of the code had been dissected and analyzed enough to have a good understanding of how most of it works.

How Driver Program Works (fsshell.c):

The driver program takes basic commands:

Ls, md (make directory), mv, cwd, and cd with options.

There are also the cp2fs and cp2l commands, which copy a file to the file system (cp2fs), and the command cp2l which copies said file back to linux. In total, we have implemented the complete list of commands,

ls - Lists the file in a directory

cp - Copies a file - source [dest]

mv - Moves a file - source dest

md - Make a new directory

rm - Removes a file or directory

cp2l - Copies a file from the test file system to the linux file system

cp2fs - Copies a file from the Linux file system to the test file system

cd - Changes directory

pwd - Prints the working directory

history - Prints out the history

help - Prints out help

Testing basic commands

Here I am testing the make directory function, and the copy to file system function. Below you can see the 0 bytes error we have been experiencing, although we are still able to manipulate the file between directories regardless.

```
FSP.SP2021~$ md Michael

FSP.SP2021~$ ls -l

D      17920   Michael Thu May  6 21:32:54 2021

FSP.SP2021~$ cp2fs DecOfInd.txt doi
Copy to filesystem complete

FSP.SP2021~$ ls -l

F          0       doi Thu May  6 21:33:30 2021
D      17920   Michael Thu May  6 21:32:54 2021

FSP.SP2021~$ mv doi /Michael
trying to move from doi to /Michael

FSP.SP2021~$ ls -l

D      17920   Michael Thu May  6 21:32:54 2021

FSP.SP2021~$ cd Michael

FSP.SP2021~/Michael/$ ls -l

F          0       doi Thu May  6 21:33:30 2021

FSP.SP2021~/Michael/$ cd ..

FSP.SP2021~$ ls -l

D      17920   Michael Thu May  6 21:32:54 2021

FSP.SP2021~$
```

Testing mkdir and removing a directory

Below I created two separate directories, and removed a directory to test that the “rm” command works successfully.