SIMPLE FILE SYSTEM

Alexander Einshøj

May 25, 2016

Introduction

In this project, I will describe the implementation of a greatly simplified version of a UNIX-like file system.

Requirements

To fulfill the requirements for this assignment, I had to implement the following system calls: fs_init, fs_mkfs, fs_open, fs_close, fs_read, fs_write, fs_lseek, fs_mkdir, fs_rmdir, fs_chdir, fs_link, fs_unlink, and fs_stat.

Technical Background

EXT2

EXT2 is the file system that this project is vaguely based on. In order to even begin understanding this assignment, one needs to have a pretty good overview considering EXT2 and its structure. As it usually works, EXT2 operates with block sizes between 1KiB and 8 or 16KiB. In this assignment, however, the block size is 512 Bytes, which makes the structure of the file system quite different than it would be in practice. The superblock takes up one block, and the following bitmaps share one block between themselves, as they are both 512 Bytes. There are 13 blocks available to inodes. The superblock contains information about the inodes, datablocks, location of root inode and the bitmaps. In an actual file system, there would be a lot more information in the superblock, but as previously stated, this is a very simplified version of a file system.
The inodes contain metadata, that is, they tell you data about the actual data, e.g. where to find the data, how much data there is and so forth. Their function is to easily find any given data within the file system, given that you know the inode number which points to it. The data blocks contain actual data. The data may be represented as directory entries (accessing a directory's data), or as - in this file system - strings which means it's a file.

Design

In the currently implemented file system, the maximum allowed file size is 4096 bytes. That is an extremely low maximum size, if put in a realistic setting. With this as a factor, what one is able to do with the file system is greatly limited compared to having eight times as much, through indirect blocks.

When first initializing the filesystem, it will try to read from disk where the superblock is supposed to be, and end up not finding it, because the magic signature isn't set/the system doesn't exist yet. When this is the case, we initiate the fresh file system with nothing but a root directory and insert the magic signature.

No matter what syscall is provoked in a file system, even a small one like this, there are tons of factors to consider when deciding what to do with the information given. Taking open as an example, we will always get a path and a mode. The path might be relative or absolute, and different modes allow for different things. When writing 'ls' into the terminal, the syscall will always receive an absolute path and the mode is known. From thereon, we only need to parse the path to end up where we want. If we get a call with 'cat', the path could in theory be either absolute or relative, there is no way to tell unless the implementation only allows for one of those.

I've designed most of my functions to first of all check whether the path is absolute or relative, and the mode after that. However, after finishing the implementation, I realize that it's probably a much better idea to let the parser handle the path itself, in either case. This would make the code look more structured, as well as ending up with fewer lines of code, which would be a good thing.

Probably the most important aspect of my design, is the way I handle the position when reading and writing. In the given precode, this value resides in the mem_inode struct, although in my implementation I've added an extra field to the fd_entry struct, allowing for easier handling of position.

Handling position this way shouldn't be necessary at all, but since I don't have any global mem_inodes, I don't have a good way to get the position of a given process in a given file without doing it the way I did.

Another big part of my design, is the way I wrapped around both inode blocks and data blocks. The easy and most intuitive solution to this, is to only write or read everything you want on the same block, and if what you want to write requires you to wrap onto the next block, you simply skip the remaining bytes in the first block and write everything on the last block. This way, you won't have to worry about how to read or write anything, as everything can be done in one simple read or write.

In my implementation, when I realize that what i want to write will require one more block, I write as much as there is room to write on the first block, and the rest on the second block. This means that when I read something, I have to know whether it will wrap around the next index or not. As a result, I end up with the possibility of 4 more inodes than the first alternative, which is about a 2% increase. That might not sound like a lot, and it isn't in this file system. If my implementation was to be used in a larger file system, it would have a bigger and more visible impact.

A weak part of my implementation, is that I haven't implemented all the functions, which means not all the syscalls are available. The functions rmdir, lseek and unlink are not yet implemented, which basically just means that you can't remove anything from the file system nor seek to a certain point in a file. In a real file system, where you want to remove elements pretty often, my implementation would be very weak.

The link function in my implementation only accepts absolute filenames, and whatever is the input for linkname is, will be the name of the link, in the current working directory.

Implementation

This project was written in C.

Discussion

In the design part, I closed with the way I handled position in this assignment. As a consequence of not having any global mem_inodes, I have to read all the inodes from disk every time I want to read anything. This means a significant decrease in response time for anything, really. However, if I had gotten to the point of implementing extra credits in form of a cache, that

would improve my implementation a huge amount, because I would have a lot less reads from disk compared to what I currently do.

When doing an assignment like this, one thought has hit me pretty hard. You have to make a decision on quantity or quality in terms of the functions you want to implement. If I had implemented all the functions, the rest of my code would not be as stable as it is. On the other hand, if I had skipped a couple of the functions I have implemented, the rest of the functions would probably be very stable, as it would allow me to test for more faults. That's the thing with file systems, everything can go wrong. It's only the things that you don't think on before hand which become a hassle to solve sometimes. I've definitely chosen quality over quantity for this assignment, just based on the thought that I would rather have a file system which is stable, than one which might not be, at the cost of functionality, of course.

Conclusion

Even though not all the functions were implemented, the ones which were, were implemented to the very best of my ability, and I think the way my implementation is, it's incredibly easy to build further on, which is a very important aspect to me.