INF-2201-1 23V: Operating system fundamentals

Assignment 6

Amund Strøm

Email: amund.strom@gmail.com

GitHub user: ast283

May 31, 2023

1 Introduction

This report describes the design and implementation of a greatly simplified UNIX-like file system. The file system supports some standard system calls, which will be discussed in this report. For many reasons is this a simplified version of a file system, for instance it does not support admin/user privileges, everything runs within the same privilege level.

2 Technical Details

These are some concepts and problems that play a large role in the assignment, and is discussed in this report.

- The superblock contain general meta data about the file system, such as number of inodes and datablocks and where to find the inode table and bitmaps. The superblock must be updated regularly to secure consistency in the file system, since it plays a large role when powering the operating system on and off. Whenever the system is turned on, it reads a specific byte in the file system memory to check if a "magic number" exists. This magic number is unique and resides inside the superblock. If the magic number is found it means a file system already exist on disk, and the superblock should be read into memory. If it does not exist, a new file system must be created, which means it will overwrite old data.
- The bitmap record the state of inodes and datablocks, if they are taken or not. As the name suggests, each bit in the map represent an index, which goes into the inode table or datablock table. If a bit is set, the position of that bit, represents an occupied slot in the inode/datablock table. The bitmap is used to allocate new data and free non important data.
- The inode contain meta data about a file or directory, such as the disk datablock locations, number of links for that inode, the current size it occupies on disk, and what type it is. Whenever a new directory or file is created, a new inode is assigned to that directory or file. Therefore you could argue that an inode is an abstraction for a directory or file.
- The inode table is a consecutive row of blocks that stores inodes. This table is accessed whenever a new inode is assigned, whenever an inode needs to be read to access information about a directory or file, whenever an inode needs to be updated, etc.
- Datablock contain raw data about a directory or file. If the datablock belongs to a directory it will only contain directory entries which points to other directories or files. If the datablock belongs to a file it would contain the text inside that file. Datablocks is assigned on demand, whenever the size of a directory or file exceeds more than a block, a new datablock is assigned to that directory or file.
- A directory entry is a data-structure that is stored in the datablocks belonging to a directory. An entry contain the filename and inode number for a directory or file. The filename is used to differentiate between entries and each filename must be unique in a directory. Due to the fact that multiple entries may share the same inode. The inode number is used to acquire the inode for a given directory or file, via the inode table.

- File descriptor is used whenever a directory or file is "open" in memory. A file descriptor table puts the inode into memory so it would be less time consuming to access the inode. Eventually the file is closed and at that point the inode is updated on disk if necessary.

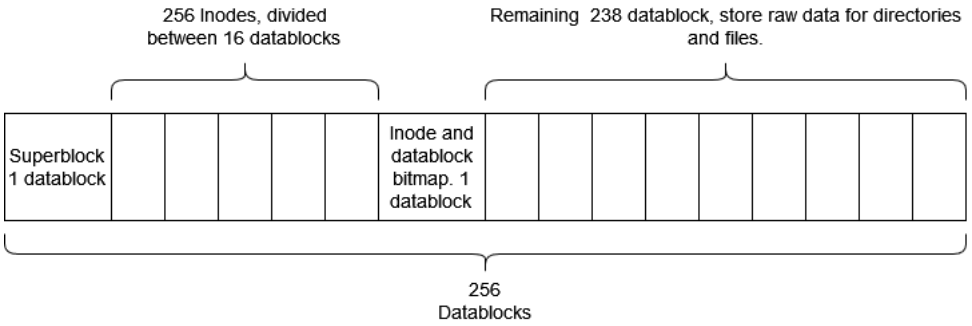

Figure 1: Memory layout for the files system on disk.

3 Design

To support some standard UNIX-like system calls required implementation of a handful of functions. These function works largely independently and serve one specific purpose. This section will therefore go through each function and give a high-level view of the functionality they posses. It will not go into every detail about the different "checks" that must be done to ensure consistency and completeness in the file system.

3.1 fs_init and fs_mkfs

fs_init initializes values that is used during a session, such as the file descriptor table and memory inode table, both of which is used to "open" and "close" files (will be discussed). It will initialize the superblock with bitmaps and other meta data, and set the current working directory to root. The final important step is to check for the magic number on disk. If the number exists, the bitmap must be read onto memory so the superblock knows what datablocks and inodes are in use. If the number does not exist, fs_mkfs is called.

fs_mkfs initializes the actual file system on disk, refer to figure [1] to get a visualization of how the memory layout looks on disk. The superblock, inode table and bitmap must be initialized and acquire private datablocks which will store their values, even when the system is powered off. Thereafter the root directory is created, which is somewhat the same process as fs_mkdir and will be explained in the next section. Finally the the superblock, bitmap, and root directory is written onto disk.

3.2 fs_mkdir

fs_mkdir will create a sub-directory in the current working directory. Creating a directory consists of acquiring a free inode and datablock. The inode will hold meta data about the directory, while the datablocks will hold directory entries. A newly created directory will have two entries, "." and ".." which holds the inode number to itself and its parent accordingly. The created sub-directory must also insert an entry into the current working directory so that it will be accessible.

3.3 fs_chdir

fs_chdir changes the current working directory by parsing the given path whenever it encounters a "/". The parsed path holds each directory name in the given path, and uses these names to iterate through directories to search for matching names. When it finds a matching name, it updates the current working directory corresponding to that name, which holds the inode number into the inode table on disk. This system call ensures that the user is not allowed to change current working directory to a file.

3.4 fs_open and fs_close

fs_open opens a directory or file and puts it into a global memory inode table and a file descriptor, both of which exists on memory. This reduces the time it takes to access the inode, which happens frequently whenever a data-structure is open. This system call is used for several scenarios, such as reading the content of a directory, file or inode, and create a writable file or continue writing to an existing file. The only difference between creating a file and directory, is that they differ in types and that a file does not need "." and ".." in its datablock, since it should only store raw data such as text.

fs_close closes a directory or file, freeing them from memory and writes any new values in the inode onto disk. Before freeing something at all, it must check if other processes have the directory or file open.

3.5 fs_read and fs_write

fs_read reads the content of an open directory or file. The content refers to the datablocks on disk that a directory or file owns. The system call must check if the caller have privilege to read the given data-structure, and what type the data-structure is, since different types differ in allowed read sizes. For example a directory is only allowed to read one directory entry at a time.

fs_write writes data to a file. It follows much of the same logic as fs_read, but on the contrary only files are allowed to be written too. It must therefore check if the caller have permission to write to the given data-structure and if the data-structure is of type file. If the write will span over to a new datablock, the system call must acquire a new datablock for the file.

3.6 fs_rmdir

fs_rmdir removes a sub-directory from the current working directory. This is achieved by checking if the given filename is of type directory and if it does not contain any data. Since this implementation does not support recursive deletion, the directory must be empty. To delete a directory, simply set the entry into the inode bitmap and datablock bitmap to 0, this indicates that there is no important data on these locations. The current working directory must also remove the entry that correlates to the removed directory. Whenever an entry is removed from a directory, the last entry in the inode is moved to the slot that just got removed. This secures a continuous flow of information, and an accurate representation of the current size in the directory.

3.7 fs_link and fs_unlink

fs_link creates a copy of a file into the current working directory. The system call requires two inputs, the path to the file which should be copied, it can either be absolute or relative. And a filename for the copied version of the file. To make a copy of a file, simply insert a new directory entry with a new filename, and reuse the same inode number for the file that should be copied. Since the inode holds the meta data and datablocks for a file. This means that when editing either version (the original or copied) both will change equally, since they contain the same inode. This is referred to as an "hard link". The system call only works on files, and not directories.

fs_unlink removes a file from the current working directory. This is accomplished by checking if the given filename is of type file, then followed by removing the directory entry for the given file in the current working directory, using the same logic as fs_rmdir. Before removing the inode and datablocks for the given file, it must check if there still exists links to that file. If that is the case it cannot remove the inode and datablocks yet, since they are still in use. If there are no more links, the inode and datablocks that the file occupies can be freed.

3.8 fs_lseek and fs_stat

fs_lseek is used to set the position of a cursor in a open file. Using the given offset and whence to dictate where to set the cursor. The whence selects position the cursor starts, either from the beginning of the file, the current position, or the end of the file. The offset is added into this position.

fs_stat reads meta data from the inode in the given file descriptor, and prints it on screen. This system call require that a directory or file is opened.

4 Discussion

There is many other solutions to solve the assignment, and this solution is in no way the best. However, many of the design choices were made thoughtfully, to either increase performance or readability. In order to enhance readability, I've implemented functions that serve specific purposes. This effectively reduces the amount of visible code, providing a more coherent understanding of the main function's purpose.

Whenever a sub-directory entry is removed, the last entry in the parent directory is moved to the slot that just got removed. This solution provides some pros and cons. As mentioned in the design section "This secures a continuous flow of information, and an accurate representation of the current size in the directory." It will also make inserting new entries effortless, since there is no empty spaces in the directory that needs to be filled. And the size of the inode works as an offset to the next available space. The same logic applies to reading a directory, as the size serves as an endpoint, and there is no empty spaces to skip a read. The downside to this solution is that it will take longer to remove entries, since it must move the last entry, but this sacrifice does not outweigh the positive effects.

I have chosen to increase the size of the disk inode data-structure to 32 bytes. This makes the data-structure sector aligned, which makes it much easier to write and read inodes to/from disk, since there is no need to split up the data. This causes a minor negative effect, since supporting 256 inodes with the size of 32 bytes, increases the size of the inode table to span 16 blocks. Compared to keeping the original size (28 bytes) which makes the inode table span 14 blocks, but this require extra logic since it must occasionally write and read data over two blocks.

Contradicting myself, I've implemented a function that supports writing and reading across multiple blocks. This function is only used on datablocks with directory entries or file data, regardless of its ability to also handle writing and reading inodes. But i choose to not utilize this function on inodes, since in my opinion it does not make a huge difference and it would reduce readability.

4.1 Testing

My methodology for testing the implemented file system consisted of manually testing each shell command as each command got implemented. This was a tedious process and got increasingly challenging as more commands were implemented, since each command spawned new edge cases. After each shell command was implemented and worked as intended according to the manual testing, the right decision was to create a test file. This file tests every edge case that I could think of, but there is probably more cases that I did not think of.

The test file also stress tests most of the shell commands, for example the depth of the file system (via "cd"), generating a substantial amount of directories and files within the same directory, and writing a considerably amount of data into a file. This created bugs that I was previously unaware of, and made things much easier to test. The test file is part of the hand-in and does not create any bugs, but as mentioned, there is probably edge cases and other stress tests that is not tested.

5 Conclusion

This report gave a high-level view of how I implemented a simple UNIX-like file system. It described the functionality for each function given in the assignment. While also discussing positive and negative consequences for some of the design choices made in the implementation. A small testing section which describes the methodology for testing the thoroughness of the file system.

References

[1] Tanenbaum, Andrew S. Modern Operating Systems, 4th edition. Pearson, 2015. Chapters 4.1-4.3.