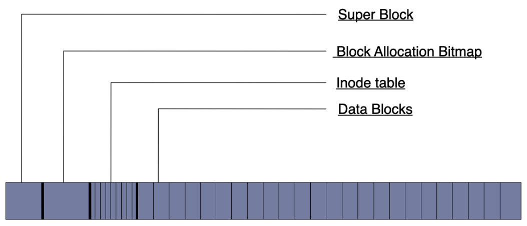
INF-2201: Operating Systems
Exam 3 / Assignment 6

Tarek Lein
UiT id: Tle044@uit.no
GitHub users: TrakeLean

May 31, 2023

1 Introduction

In this exam we were tasked with creating a file system, and a shell that's able to manage this. We decided to mimic the ext2 unix file system. This file system is simple and that easy to implement. Where each block is 512 bytes.



The first block is the super-block, which contains information about the file system.
The second block is the bitmap, which contains information about which blocks are free and which are not.
The inode table calculates how many blocks it needs, lets say it uses 15. so it uses the block from 3 to 17, to hold the inodes for the files. (image originally from the precept PowerPoint, but slightly modified to fit my file system)
The rest of the blocks are data blocks, which is where we store all the data for the files and directories. The picture above is an illustration of how the blocks are setup on the disk.

2 Technical Background

fs_init: Initialize the file system. This function should be called once at the beginning of the program and check if there is a file system on the disk. If it doesn't find anything it will create it.

fs_mkfs: This function is responsible of creating the file system and writing it to the disk. This function should be called once in the fs_init, and never again. It sets up the disk space required for the file system, such as the super-block, inode table, and bitmap.

fs_open: Opens a file for reading and writing. If the file exists, the cursor will be set to the beginning of the file. If the file does not exist, it will be created so that data can be written to it.

fs_close: Closes an open file. This function should be called when the file is no longer needed in memory. It also checks if the dirty bit has been flipped. If the dirty bit is set, the function writes the updated data to the disk, making sure that any changes are saved.

fs_read: Read data from an open file. This function allows retrieving data from an open file. By specifying the number of bytes to read, the cursor is updated to indicate the next position to read from within the file.

fs_write: Write data to an open file. This function writes a given amount of bytes to a file.

fs_lseek: Move the cursor to a new position determined by the user. This function enables users to change the current position of the cursor within an open file.

fs_mkdir: Create a new directory. This function is used to create a new directory within the file system.

fs_rmdir: Remove a directory. This function is used to delete a directory from the file system.

fs_chdir: Change the current directory. This function allows changing of the current working directory.

fs_link: Create a hard link to a file. This function creates a mirror copy of a text file, the only difference is the name given to the copied file.

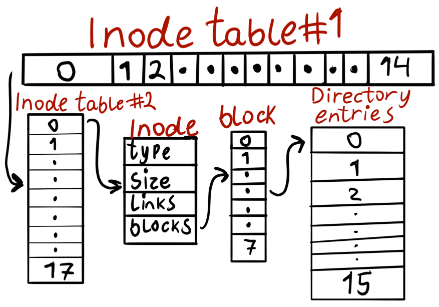
fs_unlink: Remove a hard link from a file. We delete the file if there are no more links connected to the file

fs_stat: Get information about a file.

fs_name2inode: Get the inode number of a file from the path given. This function maps a file's path to its inode number.

3 Design

Our file system is designed to be able to hold 256 inodes and 256 data blocks, this limitation comes from the bitmap that is used to keep track of which blocks are free and which are in use. The bitmap is 512 bytes, and each byte represents a block. it is a little misleading that it's called a bitmap in this case, because it is not used as a bitmap, but rather a byte-map. If it was a true bitmap, we could have 4096 blocks, but instead we have 512 blocks. The reason for this is unknown to us, but it came with the precode for the exam, so we just went with it.



An inode table entry is 512 bytes, and each inode is 28 bytes. This means that we can have 18 inodes in the inode table. We then know that there is room for 256 inodes in total, and we can calculate that we need 14,2 blocks for the inode table, since we cant just allocate 1/5 of a block, we have to allocate an extra block that will be partially used. This means that we need 15 blocks for the inode table.
Each inode has 8 direct blocks that is used to hold either text for a file or directory entries for a directory. Each block is 512 bytes, so we can hold 4096 bytes in each inode. This means that we can hold 4096 bytes in each file, and $4096/28 = 200$ directory entries in each inode.

When opening a file we have to calculate our way through the tables and to the correct inode, this calculation will be shown in the Implementation part below. When the file is open and the inode has been found, we use our memory inode table, where we will input the file we're trying to open into the first open spot, this is done to allow multiple read and writes to the same file without having to read and write from disk every time we access it. It also allows multiple processes to access the same file at the same time.

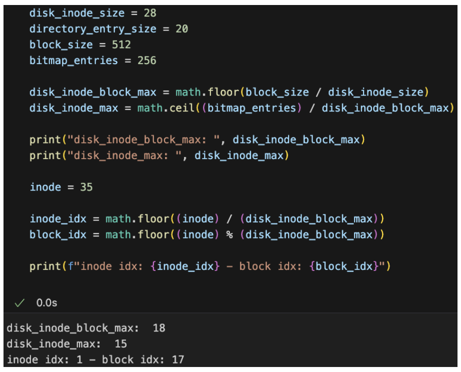
We also chose to make every command in the shell operate recursively, this means you're able to do "mkdir a/b/c/d" and it'll make all the directories in the path. This also means that you can do "rmdir a/b/c/d" and it'll remove all the directories in the path. This is a design choice that we made, and it is not a requirement for the exam, we are aware that this would normally require a flag to be set, but we chose to do it this way to make it easier to test the file system, and to make it easier to use. Plus it's a nice feature to have.

When creating a file we chose to not give it a block, but rather wait until the user writes to the file, and then allocate a block for the file. This is because we don't know how big the file is going to be, and we don't want to allocate a block that is not going to be used.

We did not find any use case for the "fs_lseek" function, but we still implemented it, because it was a requirement for the exam. However we "used" it in the "fs_read" function where we just seeked 0 bytes from the current position, just to show that we know how to use the function and that it seeks correctly.

The same goes for "ino2blk" and "idx2blk", we felt that instead of using "idx2blk" we would just do the changes in the "block_read" and "block_write" functions, rather than having to use the idx2blk to calculate block offset every time.
The "ino2blk" felt a little useless considering we used eight blocks for the direct blocks, which means we would only be able to get hold of the first block in the inode. We still implemented the function and tweaked it to allow a offset to be passed in, so we could chose what block in the inode we wanted, but we still did not find any real use case for it.

4 Implementation



```
disk_inode_size = 28
directory_entry_size = 20
block_size = 512
bitmap_entries = 256

disk_inode_block_max = math.floor(block_size / disk_inode_size)
disk_inode_max = math.ceil((bitmap_entries) / disk_inode_block_max)

print("disk_inode_block_max: ", disk_inode_block_max)
print("disk_inode_max: ", disk_inode_max)

inode = 35

inode_idx = math.floor((inode) / (disk_inode_block_max))
block_idx = math.floor((inode) % (disk_inode_block_max))

print(f"inode idx: {inode_idx} - block idx: {block_idx}")
```

✓ 0.0s

```
disk_inode_block_max:  18
disk_inode_max:  15
inode idx: 1 - block idx: 17
```

Here you can see that we're trying to find the 35th inode, to find the inode table we divide 35 with the the maximum amounts of inodes in one table(18). Since only 0 to 17 fits in the first table, that means 18 to 35 would be in the second table, and since we're dealing with indexes it would mean that 1 is the second table, which is what we got on the picture.
Then we have to find the block index, this is simply done by taking the inode number 35 modulo the maximum amount of inodes in a table(18), which would give us 17. which is also correct.
These two calculations allow us the usage of one number being used as an index to find the correct inode, and the correct block in the inode table.

The process of deleting directory entries proved to be trickier than expected. Initially, our approach involved simply removing the entries, which resulted in a "hole" in the block. This seemed to be good at first, since when we introduce a new entry, we simply look for the first empty block to fill. However this method of removing presented a problem. When we attempted to use the "ls" command in the shell, it would stop reading after the hole, because "ls" continues to read until it encounters a "0".

To address this issue, we decided to relocate the last entry in the directory to the position of the entry we were removing. This ensured that there would be no gaps. The first step in this process was to iterate the entire directory and identify the last entry while also preserving its data. We then searched for the entry to be deleted. Once found, this entry was replaced with the last entry. Finally, we cleared the position previously occupied by the last entry to ensure we didn't keep any garbage data that could potentially cause problems.

5 Discussion

We might have a few extra unnecessary bugs in our code because we implemented recursive functions, but we have not been able to find these bugs which means they aren't a huge problem since everything still works, but it might be a problem if we were to expand on the file system. The trade of is also quite good, because it allows us to do things like "mkdir a/b/c/d" and "rmdir a/b/c/d" which is a quality of life feature that we felt was worth the trade of.

We had an annoying bug that was really difficult to pinpoint. To make it occur, we had to start the OS, close the OS, open it, then create a new directory and cd into that, when we did that we noticed that "ls" gave us nothing, the directory was somehow empty. When the directory entry struct wasn't a size of 2^x it would not work, and we could not figure out why. That was until we opened the image_sim binary file and looked at the disk data live, and saw that when we created that new file, it wrote over the entire super-block, which could only mean one thing, and that was that we did not initialize the bitmap correctly on startup. After finding that out and fixing it, everything worked as intended.

5.1 Testing

We used a lot of prints when testing the code initially to see if everything was working as intended, but we removed most of them before handing in the exam. We also used the image_sim binary file to look at the image data updating live, to see if everything was working as intended. There was also created a little debug printer, that would print where in the code it was, and a counter for how many times it had been used. The built in GDB debugger was also huge help when debugging the code. Seeing the values of the variables at any given time and being able to step through the code was invaluable, it helped us through segfaults and read errors that we would not have been able to find otherwise.

Then when we were "done" with the code, we used the given python test files to check if there were any errors that we couldn't find by ourselves. This is where we noticed the bug with the bitmap, and we were able to fix it accordingly.

When we felt like a system-call was working properly, we went ahead and tested edge cases and things that should not be allowed, to see if they were handled correctly, if they weren't we would go back and fix it. This was done for every system-call, and it was done multiple times for each system-call, to make sure that it was working as intended.

We've also decided to include the python test file which shows that the basic functionality of the file system works. There are also a few special case tests that could be tricky to come up with on the spot. there are also checks for illegal shell usage. you're free to use it as you please.

6 Conclusion

The main parts of this system is a super-block, a bitmap, an inode table, and data blocks. Important functions were implemented to setup, create, open, close, read from, and write to the file system. The system also provides features for handling directories and hard links.

The file system's design is limited to 256 inodes and data blocks due to the bitmap structure. While a file is open, the inode is stored in memory for efficient reading and writing. Every command in the shell operates recursively, which makes testing easier and betters usability.

To optimize resource use we only allocate blocks when needed, that means if a file has been created but not written to, it wont have a data block allocated. Functions like "fs_lseek," "ino2blk," and "idx2blk" were implemented to fulfill exam requirements, even though no real use cases were found.

There are potential bugs due to recursive functions implementation, but they did not impact the overall functionality during testing. Multiple testing strategies, including live binary inspection, GDB debugging, and given Python test files, were used to ensure the stability of each implementation.

7 Sources

References

[1] https://wiki.osdev.org/File_Systems

[2] https://www.cbtnuggets.com/blog/certifications/open-source/linux-hard-links-versus-soft-links-explained

[3] powerpoint from precode.