DM510 Assignment 3

Viktor B. Nordstrøm (24/03/1999)
Vinor19

15. May 2021

Contents

1 Introduction

This report details the design choices, implementation and testing of a filesystem, created with the FUSE library.
The filesystem will implement directories, files and inodes.
It will be able to create, delete and list both directories and files, and files will also have the ability to be written to and read from.
It will not have the option of renaming or having multiple links, and a lot of data from inodes will not be implemented.
The filesystem must be written/read to/from a semihuge file.
So it will be a very basic filesystem.
There will also be some discussion on how the system handles power failures.
Lastly I'll conclude on the entirety of the report, with possible improvements mentioned.

2 Design

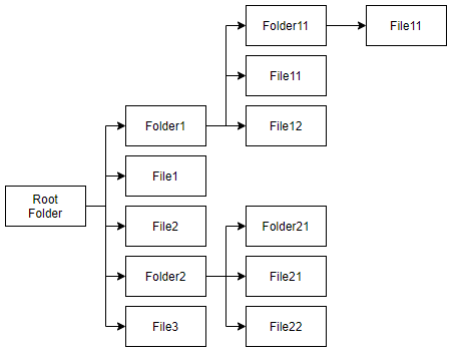
Designing the filesystem. Firstly there are directories, files and inodes, so I want to have a way to represent those structures in my code.
Since inodes holds information about the given file/directory, I can save the needed data in this structure too. Important, somethings will be omitted for simplicity, such as owner, permission, and more. Inodes will be, what kind of file it is, what's the size of it, and acces/modification times.

2.1 Structure of files and directories

There are some things I want both files and directories to have. These are as follows name and path of the file, filetype(File or Directory) access and modification time, size and current max size, the depth in the filesystem of the given file.
Files and Directories have things that are not in common, files have contents, while directories have entries and a list of the files and directories they store.
I also don't want size limitations on the system, so ideally the files and directories can become as large as the user wants, and no size limits on names or paths to allow as much freedom as possible.

2.2 Filesystem

The filesystem is completely stored in memory whenever it is loaded, and only gets written to a file whenever it gets unmounted. This is not only to make the implementation simple, but also for fast reads and writes due to memory being faster than a disk.
Below is a quick representation of a filesystem.
Every folder (directory) have access to what is stored inside of it.



I simply want every directory to point to whatever is stored in it.
Whenever a file or directory is made, it will also update the access time and modification time of its parent directory, since I see that as modifying that directory too.

2.3 Inodes

The plan is, to have the data for inodes saved in a structure, together with the other data the file/directory will need, and whenever the attributes of the file/directory gets read, it will return the attributes from the structure as needed.

2.4 Disk

The file that saves the filesystem saves the parts of the files/directories that are needed to recreate them. Which would be their path, what kind of file it is, access and modification times. Files also has to have their contents saved.

3 Implementation

This section details the implementation of the filesystem. This includes saving and loading the filesystem, how files and directories are created and dynamic allocation.

3.1 lfs_file

This is the datastructure that stores both files and directories. It holds all the attributes they have in common and not in common. If the attribute is not needed for the given file, it will either be set as -1 or null.
I have chosen to store paths as a list of strings, instead of just a singular string, this is due to me wanting an easy way of reading every directory in that path, and wanting to have the name seperately from the path.

3.2 Directories

So the directories has a list of pointers to lfs_files stored. If a new file or directory is created, it will update the parent directory with a pointer to it, increment the number of entires and size, access and modification time of the parent directory.
The reason size and entries are not the same is due to the fact, that I have not implemented defragging. When a lfs_file gets removed from the filesystem, it just sets the parent directory's pointer to null and then frees the lfs_file, and when one gets added it just gets put at whatever size is. So if there are three files, and I remove the second one, it will be {File1, Null, File3} which means there are potentially holes in the list of lfs_files. The only way to really defrag the filesystem is to save it and load it again, more on that later.
Whenever size gets to the max size of the directory, it will automatically allocate a new list of double the size, and then copy the old on into it, this is done by using memcpy, so it does not fix defragging here either.

3.3 Files

A file's standard size is 8192, this was chosen because of interaction with vim before I made the file size dynamic.
Files has contents, which is a list of chars (a string), that gets reallocated whenever someone writes more to the file, than it can currently store. The new allocated size is double the size of what the file size would be after the write. This is done so I don't have to allocate as often, but will also result in a lot of allocated memory, which could be wasted. Another way of dealing with this would be allocating in specific sized chunks, and then doing some simple modulus calculus to figure out how many extra blocks is needed, and then allocating that. This is more computation and would require more reallocations.

3.4 Inodes

The way Inodes have been implemented is very simple, the only time Inodes gets read is when getAttr gets called, so some of the things are hard coded in, so when it gets called, it checks whether or not

it's an file or directory, and then it assigns to the stbuf (st_mode), S_IFDIR | 0755, if it's a folder and S_IFREG | 0777 if it's a file.
The size for file is just the variable size, while the size for a directories is the number of entries, so it sets stbuf->size to entries.
The number of links for all files is just one, since there can't be more than that in my implementation, and for folders it's 2.
It also sets access time and modification time in the stbuf to whatever the file/folder has.
That's all of inodes that gets implemented.

3.5 Save and load the filesystem

Currently the filesystem always gets loaded from a specific file in the tmp folder called "disk.img".
The way it stores the filesystem is by using a recursive call on the folders.
There are somethings that are stored whether they are files or directories, this is the file type, path and their access/modification times, since that's always needed.
Directories don't get anything else saved, while files also store the length of their contents and the content itself.
This is started whenever destroy() is called, in my implementation I have said it so, it will call lfs_destroy. This will do a loop over all contents of root, and afterwards free root.
lfs_writeFolderToDisk writes the file type, then the length of the path, the path and last access and modification times. Afterwards it enters a forloop that loops over all it's contents, and if it hits a directory, it calls lfs_writeFolderToDisk on that directory, if it hits a file it calls lfs_writeFileToDisk. after the loop it calls lfs_rmdir, which is my function for removing directories, to make sure it gets removed properly. This does not get called on Root, because I don't need to save roo, since it always needs to be made, and always has the same path and standard settings.
lfs_writeFileToDisk simply writes the same as lfs_writeFolderToDisk, and then just writes size of contents and then contents. Afterwards frees the file by calling lfs_unlink, which is the unlink function for files.
With this implementation, it means that it gets saved in a special way.
This is a quick representation of it the order it is saved on the disk:
{Folder1{File11}, File1, File2, Folder2{Folder21{File211}}File3}
Now when the filesystem gets loaded, it simply goes from left to right on the disk, read first the file type, then the path, then it creates a file with that exact path and sets the access and modification times according to what was stored. Since I call the actual lfs_mkdir and lfs_mknod functions, I don't need to worry about the size of the directories at all, since this will be sorted out after it's done loading the filesystem. This is also why it defrags, since there will no longer be any holes in the directories.
I also check if the contents of the files is larger than the standard size, if it is, I update the max size and reallocate contents to a larger size, before I copy the contents into the file.

4 Power failure

The system as it stands currently has no way to recover from a power failure, the best it does, is it has the previous version saved, the one that got saved last, which is not really the best for backups, since this only occurs whenever the system gets destroyed (which happens during unmounting).
There are some ways to combat this. One would be to write down all commands run on the filesystem with their contents saved in a seperate file whenever they occur. This would then get loaded when you reload the system, and you can have the option to apply the changes again.
A specific way of writing these things down would need to be designed, and then implemented into every function, but it is an option.
Another way would be to save it directly to a disk, instead of just writing it to memory, or doing both. Doing both would hinder the write times, but read times would be unaffected (since it will still be in memory, so it can read from there). But in this case I would have to redesign my entire disk file on how it's written and loaded.

5 Test

These are all the tests that I ran:

Test 1	Mounting the filesystem	0:04	Passed
Test 2	Making and reading folders	0:10	Passed
Test 3	Making, reading and writing files	0:24	Passed
Test 4	Unmouting and remounting the filesystem	1:27	Passed
Test 5	Deleting files and directories	2:06	Passed
Test 6	Testing if it can save empty filesystem	3:20	Passed
Test 7	Making and saving multi layered filesystem	3:38	Passed

Test 1: simply if it will mount the filesystem by using the command "./lfs /tmp/lfs-mountpoint". It does this properly.

Test 2: Testing if I can make directories, and whether I can read the contents of a directory.
This is achieved by using mkdir and ls, to see if I can create directories and then list them out.

Test 3: Used vim to test whether all the functions works, since vim will use mknod, unlink, read and write. As tested it can read and write through vim, and truncate properly in the case that lines/characters get removed.

Test 4: This was to show that I can unmount the system, and on remount it will load in the files.
This is done by calling "fusermount -u /tmp/lfs-mountpoint/" and checking with ls, whether there are any files at the mountpoint, and then going back to tmp and calling " /dm510/assignment3/sources/lfs /tmp/lfs-mountpoint", and going back into the mountpoint and seeing if the files where there.
After the unmount there were no files, afterwards the same files were back with the same content.

Test 5: Actually removing things from the filesystem.
Simply using rmdir and rm to remove files and directories.

Test 6: Removed everything from the filesystem, then unmounted and remounted it, and it was empty (like it was supposed to be)
Using rm -r to remove all files from the directory and its child directories recursively, and then unmounting and remounting the filesystem, to make sure the saved one is actually empty, which it is.

Test 7: The last test, I tested whether it could properly save the structure of a multi layered filesystem, and the contents of the files.
Essentially just making directories within directories, and files within those directories, and seeing if they can be saved and loaded in, in the same structure even in multilayered filesystems. It does this just fine.

6 Conclusion

A functional filesystem has been created, there are a lot of things that could be done differently and most likely better, and somethings that haven't even been implemented that would make it an even better filesystem.

One of these is the defragging of the directories while it is all running. A way to have done this, would be to assign the slot in the last entry in the list to where I removed the other one. This would also remove the need to have size and entries seperate, since size would be the number of entries anyway, and would make sure there are no NULL in the list of files for a directory (up until index size).

A lot of code could probably be more generalised, and/or more usage for auxilary functions, as an exmaple, I have a decent amount of copy paste code in this project, when I remove files/directories from the parents, that entire thing could just be an auxilary function, since it's the exact same if removing file or folder.
When loading and unloading the filesystem, if I wrote the access and modification times after contents, I could easily have reused the lfs_write function I have to write to a file, so I didn't have to use another special block of code to resize the contents of files, or just have the resizing be an auxilary function all together.

There is no real way to come back from a power failure, only from loading back in the last time it got unmounted.

So while everything works, without any issues, there are still a lot of things that could be done better.