DM510
Operating systems
File Systems with FUSE

Mads Mikael Keinicke

May 2020

1 Introduction

Finding data on a harddisk is much slower than finding data in memory. However memory is wiped when turning off a computer, and as such having a harddisk or SSD has always been a necessity. Harddisks are much slower than SSDs, but also much cheaper per byte. This is why a computer will read the data stored on the disk, and load it into memory when it needs to handle said memory more than once, or if it is able to read the data, before it's needed.

However, the more memory the average computer has is every increasing, and as such it might one time be possible to store all the data of a filesystem in memory. This would require a single read of the harddisk at startup, and a single write at shutdown. Obviously, such a simple method would have a big risk involved, losing all changed data if the computer were to crash.

Another thing to keep in mind would be the overhead of managing files. Having all files in memory poses the risk of filling up memory, and causing a crash. At this point you would have to prioritize what to keep in memory, and it's mostly back to square one. Another thing, that could hardly become an issue, would be if the harddisk, was smaller than memory. A weird combination, for sure, but it would mean the filesystem would have to keep track of how much space is left on the harddisk, compared to the memory, causing even more overhead.

When storing data in a filesystem, you have to account for the fact that data might get added or removed from a file. This means you cannot just store large files in one long sequence. Data might get added to the file, which would overwrite another file, that was placed right after the last data from the previous file. As such a filesystem has to divide data up in blocks. If a block was 512 bytes, but a file is 513 bytes, it would have to fill 1024 bytes, as the it would spill into the next block. Even more, would be the data that keeps track of the file. There different ways of handling this, one way would be to place a "link" to the next file, at the end of the old file. This would mean, it would fill more than 1 byte in the second block.

In general, the filesystem, is not just about how fast it is to read a certain file, but also how the filesystem keeps track of updated data, old data and where it has to read it. This is a big deal for the harddisk, in terms of read-time, compared to memory. A harddisk has an arm, which moves back and forth, reading whatever data is below the end of the arm. One can risk having part of a file on one side of the disk, and after reading 512 bytes, the

arm will have to go all the way to the other side of the disk, in order to read the next 512 bytes.

2 Design

LFS filesystem, is made with the intent of reading from the disk once, and then being run entirely in memory, until it needs to save data. It demonstrates one of the biggers issues with such a filesystem, by losing all changed data, if the system does not close down naturally. Data is stored as entries, either containing a table, describing it's "subentries, " or it will contain data and a size of the data.

After reading entries, it will read the data associated with each entry, and allocate space in memory for said space and save it to the entry. After this, every command given through terminal will be handling data in memory, until the filesystem closes, at which point it will write all data to a disk. Either the one specified when starting, or it will make it's own file "lfs.img." This file will be overwritten if it exists already, and the filesystem was told to read from a file that does not exist.

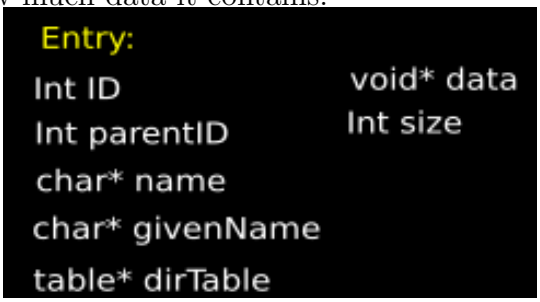
2.1 structure

An entry to the filesystem contains meta data describing either a file or a directory. It contains meta-data, such as when its contents was last changed, files only, and when it was created, both directories and files.

It contains it's own ID and the ID of it's parent directory.

The string typed upon it's creation, along with it's final name.

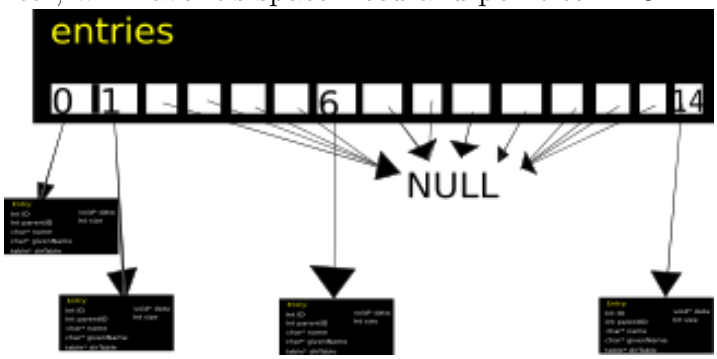
Both directories and files have a field that is able to contain data, however only directories will be able to be assigned data, along with a value specifying how much data it contains.



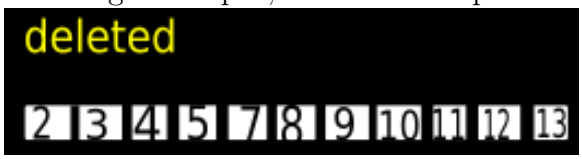
A directory will also have an a pointer to an intitialized struct called table.

A table has 4 variables. 2 are simple intergers keeping track of the sizes of two pointers. And two different pointers. One to integers, and one pointer pointing to another pointer, of an entry.

entries is an array of pointers, each one that can get allocated when needed, and freed when not needed anymore. When adding an entry, the array get reallocated space, adding space for an extra pointer, which is then also allocated space for an entry. When deleting an entry, the corresponding pointer, will have it's space freed and point to "NULL."



The int pointer, or int array, is an array keeping track of all deleted indexes. Each index contains the number of an emtpy index, which can be reused for a new index. When making a new entry, after deleting entries, it will be assigned a spot, from the last position in *deleted*.



Both arrays are able to allocate more space, but they do not deallocate unused space, it would have been wise to have them deallocated space as well, and free up unused memory. However this is not implemented in LFS, but will happen upon a restart of the filesystem.

Aside from each entry having a table, the filesystem has one large handling all entries. This is called the *OmegaTable*, and is the table the the entries are originally saved to. The tables contained within directories, all has pointers pointing to entries stored in the *OmegaTable*.

Both types of tables will assign a new entry, to the last spot of *entries*, specified in *deleted*. However the ID of each entry, is the index it has in the *OmegaTable*. This allows to find a fast very fast, if one has the ID. Upon the creation of a directory, it's name will not only be the string give as argument, but it will get it's ID added, to the start of it's name, a dash, and then the name specified by the user. When finding an entry, the filesystem will parse as much of the name given as an integer as it can, and return the entry in the given index. If the entry does not exist, it will return "NULL".

2.2 Reading and writing the filesystem

When you have started the LFS filesystem, made directories and/or files, and you close the system, it will write it's data to it's "disk". The disk is split into blocks of 512 bytes. Each entry will be given one block for it's meta data. There is currently no limitations for how long an entry name can be, however no code has been written to handle such an case. However, when writing each entry, it will move the pointer it writes from to the start of the next block, overwriting whatever data was spilled over to the currently empty block. And as such, an entry with too long a filename, will not be saved to the filesystem.

The first write will be 3 bytes away from the start of the disk, as the check for the LFS system is 3 bytes, and there is no point wasting 509 bytes with nothing. From there it will move to the next block, and write another entry. Since the memory contains the amount of entries created, it can also calculate the end of entry blocks. And is able to write data to these blocks, in between writing entries to the disk. This is simply 512 times the amount of entries, away from the beginning of the disk.

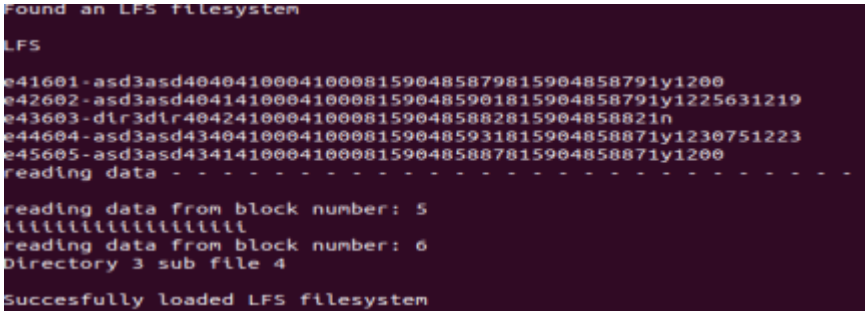
More specifically, LFS will first write the byte "01100101" before each entry, marking that this block is an entry. After this it will first write the size of the next data to be read, and then write the corresponding meta-data of the entry. Data written is written in a specific order, which is also the order it is meant to be read back again.

However, the last part of the entry, that will be read is the data, and the

size of the data. It will however write the byte offset from the beginning of the file, where the data will be written, and then also the size of the data. The data written, as said, will be written at the end of all entries, spilling over to however many blocks it will need to fill up.

Since it's not decided if it's a directory or a file yet, it will first write either a "y" or a "n", 121 "1111001" or 110 "1101110", a "y" indicating a file, and "n" indicating a directory. If it is a file without any data, it will simply write a "0" after the "y".

When reading the data, it will, once again, start after the first 3 bytes. Check if it can read an "e", and then act accordingly. after reading an entry it will move to the start of the next block. Once it gets to reading the possible data of a file. It will first read a check, and read data if it is a file. The data read, will instead of the actual data, be a pointer to the start of the first block containing the data. And after that it will read the size of the data in the file. Such that once it reads the data, it will simply read until it has read bytes equal to that of its given size.



This is done after reading each entry. Once each entry has been read. It will have to account for the deleted table in the previous open session. It will iterate through each index in the loaded entries, and check if it's own ID matches the given index upong reading, as well as if it's parent is in the right spot as well. It will change it's parent ID to the new index of it's parent.

After iterating through and changing parent ID's, LFS will iterate through

the indexes again, and change ID to the new index. It will also rename all files, such that the first part of the string matches the new ID.

3 Discussion

LFS cannot recover from a power failure with data changed during the failed session. It only writes upon closing naturally. This can cause many issues if there are bufs that can crash the system. Or if the user closes the filesystem, in an bad manner. If the system crashed while writing, it might open up again with faulty data. And can also read the data of a file as a new entry. However, the data will not result in a new entry, it will simply attempt to create an entry and fail.

A system could be made, where one could mark all files that has changed since loading the system, and routinely writing these changes to the disk. This could either be done with a simple variable saying if a file has been changed or not, and then iterating through *OmegaTable* checking what files to write. Another way would be using a pointer like *deleted* and then routinely write the entries to disk, noted by their ID.

Another fault of LFS, is that it changes the ID of each file upon reading a system again. This means that data from an installed program, would not be able to find other files it has referenced, it would need in order to start.

There are severall ways this could be solved, one way would be to change every reference to the ID when reading the disk. This would require a lot of work, though. And an easier implementation, would be to just never change the ID of an entry, and instead recreate the original *OmegaTable* of the old system. This would require more overhead on the disk, however, it would ensure consistency, when reading the disk again.

This consistency would also be important from the user's perspective. He is maybe used to having the ID "212" refer to his folder used for storing pictures in one session, and having to relearn the ID each session, would get tiring, and defeat the entire purpose of solely using ID's for navigating in the system. Names by string, is purely for remembering what they refer to, and becomes even more important when LFS will be changing ID's. But since there is nothing telling what entries got their ID's changed, there is no way of telling you how to find old files, when booting up a larger filesystem again.

Saving the *OmegaTable* would also make it even more important to manage the size of *entries* and *deleted*, as these can grow infinitely large, or rather as large as your have memory left. LFS does not account for the size of the disk either. It assumes that the memory is of smaller size than the Disk. And will as such allways have space for what is stored in memory.

To save on performance, one could save data with a seperate thread or process running a cleanup method, constantly, until told to stop.

It does however not keep track of how much memory is left on the computer. And will as such always allocate more space, when asked to. This could cause a crash on a computer, if the user does not keep track of it himself. This can happen when *OmegaTable* becomes too large, simply by deleting many files or creating many files.

4 Conclusion

In conclusion it was possible to create a filesystem running entirely in memory, however the implementation is pretty volatile, and not very user friendly. It can be useful for storing smaller filesystem, with not too many directories. However it does not keep track of how much memory that is left. Doing so would require the system to either limit how much data it is willing to allocate. Or by having to do what many filesystems already do, writing routinely to disk, and keeping track of where data is stored on disk, somewhat defeating the use of LFS, which is to have ALL of the filesystem in memory.