Simple Filesystem Implementation Detailed Review

Marco La Gamba

July 2018

Problems with this implementation

Crash consistency problem

The implementation proposed doesn't handle the case in which a crash occurrs. In fact a possible crash of the system could leave the file system in an inconsistent state, causing damages from the loss of user datas, to inconsistencies of inodes or making the entire filesys unmountable due to an uninterpretable super block.
Usually real file systems deal with this problem with different solutions. For example fsck (file system checker) is a program that scans the entire filesystem on reboot after a crash, searching for inconsistencies and makes the needed changes to fix the filesys.
However, to scan the entire filesys is onerous and modern filesys (for example Linux etx3) prefer a different approach, such as journaling, which consist in some extra operations per write to store in the filesys "write ahead-logs".
Whenever a crash happens, on reboot the operating sys will have to only look the logs committed and redo the operations stored in these logs. It permits to leave the system in a consistent state even if the crash happens while storing the logs obviously, cause the log are stored in transactions, and invalid logs (which don't have the transaction end block stored) are skipped.

Atomical set operations in bitmap

When we ask the bitmap for a new block, we want it find the designated block and set it to 1 (mark it as used) atomically in the bitmap, so if another process issues a write operation that also asks the bitmap for a block, the two processes won't get the same block, and they will work according to the specifics.
Otherwise it could happen that the two concurrent process get the same block to save their data and one of the two would overwrite the other, cause two different logic blocks would be mapped to the same physical block.
Currently my implementation doesn't handle this case, which should be resolved with the appropriate synchronization method to access the shared resource (the bitmap), such as semaphores.

Performance problems

Problems with directory structure

Files and directories entries stored in a directory's block are represented by their inode block number and stored as an unordered list. This means that in the worst case we need to browse all the directory's blocks to found a file. Also when creating a file we will scan all the blocks of the directory to be sure the file doesn't already exist. The cost is $\theta(n+m)$, where n is the number of entries in the directory and m is the number of the blocks allocated to store the entries in the directory.
This means that every operation which requires to find a file in the directory (such as open), implies to access all the blocks of the directory and all the file control blocks of the file entries in the worst case, half in the common case. Meanwhile a create operation requires the same accesses as in the open worst case, to check that the filename isn't already in use. We need to notice that all these accesses are I/O operations, and so very expensive.
The main problem is that we store only the inode block number of the files/directories contained by a directory, which means that to scan the above-mentioned directory we need to access every single inode to check the filename, requiring n read on disk operations, where n is the number of entries on the directory. That is extremely expensive talking about interactions with the disk.
A solution that real filesystems choose is to store also the namefile in the entry of the directory. An entry will be then composed by `<inode_block_number, namefile>`, so each we will have fewer entries per block, but a lot less disk operations involved.
A further approach to boost performances would be using Binary search trees to improve the cost to $\theta(logh)$, where h is the height of the tree (statistically near to n).

Problems with allocation method

In my implementation i use a linked list to retrieve the blocks which belongs to a specific file. The next and previous block numbers in the list are stored in each block in the block header. This approach is easy to implement, but is extremely inefficient for random access on files or seek operation.
In fact to move to the n-th byte on file we will have to browse the linked list and read all the needed blocks from disk. In particular access to the last block of a file require $\theta(n)$ read operations on disk, where n is the number of blocks of the file.
A better solution, used by many modern filesystems is a multi-level indexed allocation. In particular it saves the pointers to the file's blocks in the inode of the file. Usually it provides a constant number of direct pointers and then some indirect pointer. Indirect pointers leads to block that contains a list of other direct pointers. Note that we can iterate this process to allow also big-sized files.
The proposed method decrease the cost to O(k), where k is constant and is the number of iteration of pointers adopted (3 iterations is enough to contain very big files).

Problem of fragmentation on disk

The performances of the operations on filesystem are linked to the physical disposition of blocks on disk. In general access to contiguous blocks are less expensive, so we would like to allocate contiguous blocks for the same file or directory to provide sequential access to them in less time.
However in my implementation the blocks are given to files by searching the first free block with get free block operation, so basically it can be a block on the other part of the disk, requiring a costly seek operation on disk at every block read.
Disk awareness is the key to guarantee better performances when accessing to disk. In particular the disk can be divided in cylinder groups (or block groups) and subsequent access to blocks in the same group are less expensive cause they don't require the expensive seek operation, that implies a mechanical move.
Modern filesystems divide the space in groups with their own bitmap and apply a serie of policies to overcome the problem. For example they allocate

the blocks of the same file in the same group, or the blocks of the files owned by a directory in the group in which is placed the directory.
Another practice is to allocate a constant number of contiguous block when a file is created in order to guarantee fast access to them, and so to all small sized files.

Observations

Remove file

Remove a file is one the most elaborate operations within filesystem. In particular when deleting a file we must:
1) set as free the blocks used by that file in the bitmap;
2) remove the entry in the directory to which the file belongs;
3) update the directory control block(inode), for example we will have to decrease num entries;
Note that in my implementation there aren't links, that should be cancelled aswell or handled in another way.
Moreover we must face another problem, let's consider the situation in which we have a directory composed of 2 blocks. When deleting all the file whose reference in directory data are in the second block, we have a block assigned to the directory which isn't used at all, and so a space leak. To handle this problem when deleting a file, i move the last entry of the directory (the last file in the last block) in the position of the entry deleted and deallocate the block if there are no entry in it, so i will never have a block allocated but not used by the directory.
This solution avoid space leaks, and internal fragmentation but isn't optimal for performances, in fact i have to browse the entire list of blocks of the directory, when removing a file, which is extremely expensive (cause of the list data structure employed) and requires several read on disk.

Operations on disk (Disk Driver operations)

The disk device is mocked by a simple file, which contain the datas that should be on disk. So in a real implementation the operations in DiskDriver, that are associated to the disk, which are init, read and write a block, would be modified so that they would call functions defined by the API of the disk, usually standard and defined according to the protocol of the bus and the interface of the disk to tranfer the data of a logical block to/from the disk.
Instead in my implementation the disk driver operations rely on the file API itself, calling read, and write to change the file that represents my disk. So we must notice that i am implementing a filesystem (and so the access operations related) using the access operations provided by the operating system (and so its filesytem) running on my machine.
The main task of the operating system is to provide an interface with the underlying hardware to the user, so a real implementation of the disk driver in which the disk is an actual disk would be pretty different and wouldn't rely on the file API.

Caching changes

In createFile and mkDir i can save with a write operation the changes on the block of the directory in which i am adding a file/dir. I could otherwise adopt a lazy approach, mantaining the changes only in main memory and writing them to the disk when changing the fileHandle structure (that caches in memory the changes done), so when issuing a changeDir (cd).
In my implementation i am currently using the first approach for semplicity. So all changes to files or directories are immediately saved with a write operation on disk, otherwise other functions such as printTree would launch an error cause they retrieve infos directly from disk. An approach that uses cache would require some addition management that i didn't implement.
In fact OS should also provide a disk scheduler which examines the requests and decides which one schedule next. Note that unlike job scheduling, we can guess with enough precision how much a disk request will take.
However reads to blocks contained in filehandle or dirhandle will not require a read from disk, cause i still cache the first and current blocks.

C library

In my implementation i rely on some function defined by the C standard library, such as those defined in string.h header, but the kernel doesn't have access to that library basically for speed and size reasons.
So in an working implementation we should change the references to the usual C functions implemented in the kernel. For example the common string functions are defined in "lib/string.c" and can be used including the header <linux/string.h>.
We should also notice that i dynamically allocate memory to return FileHandle and DirectoryHandle structures, but the standard malloc function isn't available aswell, so i should use kmalloc instead.
For example `kmalloc(sizeof(FileHandle),GFP_USER)` allocates memory of specified size on behalf of user.

Current process point of view

This implementation also lacks the update of the current process structure. In fact when opening a file it isn't sufficient to create a handler for it. We should also add a file reference in the process control block.
For example considering Linux, the structure of a process `task_struct` contains a struct `files_struct` which holds the struct `fdtable`, in which we have a set of the filedescriptors related to the above-mentioned process; in this case we should provide a filedescriptor for the opened file and add it to this structure.
I added this consideration for completeness, but the management from the file point of view fall outside the purposes of my project.