

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 occurs. In fact a possible crash of the system could leave the file system in an inconsistent state, causing damages from the loss of user data, to inconsistencies of inodes or making the entire filesystem 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 searching for inconsistencies and makes the needed changes to fix the filesystem. However scanning the entire filesystem is onerous and modern filesystems (for example Linux ext3) prefer a different approach, such as journaling, which consists in some extra operations per write to store in the filesystem "write ahead-logs". Whenever a crash happens, on reboot the operating system will have to only look at the logs committed and redo the operations stored. It permits to leave the system in a consistent state even if the crash happens while storing the logs obviously, because the logs are stored in transactions, and invalid logs (that doesn't have the transaction end block stored) are skipped.

Atomic set operations in bitmap

When we ask the bitmap for a new block, we want it to work atomically find the designated block and set it to 1 atomically, 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.

Performance problems

Problems with directory structure

Files and directories entries stored in a directory's blocks 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 $\theta(n)$, where n is the number of entries in the directory, but above all it will require to access all the blocks of the directory (that can be a lot of I/O operations). A better approach would be using Binary search trees to improve the cost to $\theta(\log h)$, where h is the height of the tree (statistically near to n). Then another great problem is that we store only the inode 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 k read on disk operations, where k 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 $\langle \text{inodeblocknumber}, \text{namefile} \rangle$.

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 program, 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

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 related to operation on blocks of the disk itself. 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 filesystem) running on my machine.

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, maintaining 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 simplicity. 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.