OS Labb 3 - rapport

Noah håkansson och Gabriel Ivarsson

The general concepts we have used are based on the course book “modern operating systems”. This includes using a FAT to handle blocks for the files and directories in the FS. The FAT (File allocation table) is in the form of an array of integers, where each entry's index represents a block on the disk device, and what the index contains is the index of the next block. This then points us to the next block and so on, until we reach an entry in the array with a value of -1 which means that it is the last block, so we stop there. This can be easier explained through a figure such as:
[-1][-1][3][4][-1][0][0][0][0][0] - 0 = free, -1 = EOF (end of file)

In this figure you can clearly see that index 0 and index 1 both has EOF in their index as a value, this means that once either index 0 or 1 is read, it shall not go any further to read the contents of that file. Another example can be seen starting in index 2 where the value in the index points to the next index in the array, and from index 3 it points to index 4 at which it then points to EOF. In this way we have linked each block through a sequence of integers. In this specific example the FAT has space for 10 entries but of course most FATs are much larger.

We use the included code for a directory entry that came with the assignment. A tree is used to handle the actual structure of the FS when it is read in from disk to help keep track of the current working directory and where we are in the tree. Each tree node consists of a pointer to the parent node, a pointer to a directory entry and an std::vector of nodes called children that contains all the nodes children nodes. In the actual FS we have a root node that always points to root and a currentNode node that points to the node corresponding to the current working directory. As well as the currentNode pointing to the directory entry of the directory we are currently in. We have a std::vector of directory entry pointers that contains all the entry’s, both files and directories that are in the current working directory. This std::vector is always read in from disk whenever the current working directory is changed to make sure it is always up to date. Whenever a file or directory is created we make sure to always write it to disk and then update the FAT, since only keeping changes in RAM can lead to inconsistencies if you were to open another shell for example. This also means that if the shell were to crash for any reason no changes should be lost. The working directory, root directory and FAT are also always saved on an exit using the “quit” command for good measure.

But how are we then supposed to write and read back in the information coming from the FAT, root and working dir? What we know from our given DISK class is that each position on the binary-file is represented by a byte of information, in other words 8 bits. Each FAT entry stores a 16 bit, or 2 byte large integer. So in other words we need to split it up into two values from one 16-bit integer into 2 8-bit integers. Which is in it’s own quite simple, take in for example the value:
0000000000000011 this is the value 3 in binary and the integer is 16 bits large. What we can do is split it in half, other words splitting it into:

00000000 and 00000011, we now have two values which we can store at two positions in the disk-file. This same technique is used for the rest of the values too. Let's say then that we would like to read from the disk, if so we simply smash these values back together into one central 16-bit integer. This technique is used for each value in the FAT array. This is also very heavily used when writing down the data in our dir_entry data structure. This data structure consists of a few variables: filename, size, access rights, first block, type.

To start filename is an array of chars, meaning it’s not more than 8 bits large per value, this can be easily written into the array without any conversion bitwise. Size is a little bit different as it’s an integer. Most integers are represented by a 32-bit value and so is this one, meaning we split it into 4 distinct values. This goes on til each value has its subparts on the disk array and can be written. In total each dir entry consists of 64 bytes on the disk, 56 chars to start with, 4 bytes for the size int, 2 bytes for the first block (first_blk), 1 byte for the type (it’s an 8-bit integer) and 1 byte for access rights (also 8-bit integer).

To keep the filesystem consistent not only does the working_dir vector have to be kept consistent, so too must the tree for us to know where we are in the directory tree. Saving the tree structure is not needed, as long as the root is kept consistent between runs as much as other directory files when operations are done on them (e.g., create file and so on) there’s no problem with consistency. The challenge arises when we have to read it in. This is done through a recursive algorithm we have used, namely DFS (depth first search). It’s implemented in the functions initTree() and initTreeContinued(). What it does is that it starts at the root dir, reads in it’s children and creates treeNode variables for each, adding it to its children vector. This is done recursively on each directory file, reading in its block containing directory entries and adding the dir entries with type dir to the children vector.

Another integral part of the Shell and FS that is used all the time is how we change working directory and move around in the FS using CD and its helper functions. We have a function called changeWorkingDir() that takes a block as an argument, this is also the function partly described above that reads in the working directory from disk. But what it also does before reading in all the working directory entries is look in the tree for the tree node containing the directory entry that exists in the same block as the one given as an argument. It then changes the currentNode to point to that tree node so that we have the right currentNode. It does this by first checking if the argument block is 0, if it is 0 then we instantly know that it is the root and can just set the currentNode as root. If it is not the root node we check if it is the current node's parent, and if so we set that as currentNode. If neither of the earlier two cases are right then we check the current node's children and see if any of them is a match and as before, if they are we set it as currentNode. When all of the above cases fail, that means we are most certainly not calling this function from CD or any of its help functions, but instead from a number of other functions trying to call

changeWorkingDir() directly trying to reset the working directory after moving around a lot in the FS. When this happens we do a BFS (Breadth first search) in the tree until we find the node we are looking for and set it as currentNode.

So how does the function CD() work and how does it use ChangeWorkingDir()? When CD is called it instantly calls parsePath() that error checks the path and returns -1 throwing an error if something is wrong or if the entered path is just “/” meaning we only call changeWorkingDir() with block 0 and change to the root directory and then return successfully. If no error is caught and we are not changing to root parsePath() calls the function parseTilFile() with the path as an argument, parsePath() first checks if the first character of the path is a “/” to decide if it is a absolute or relative path, if it is the first option it changes working directory to root before we continue, otherwise the working directory stays the same. After that it goes through the path string extracting each directory and calling the function changeDirectory() with the singular directory as argument one by one except for the last entry in the path which is instead returned as a string so that we can handle it how we want in other functions, for example CP, MV, CAT, Chmod etc.. use parseTilFile() to both CD into the right working directory and get the last entry in the path which could be both a file or directory and handles that accordingly. What changeDirectory() does is check if the directory it is called with exists in the current workingDir array containing directory entries, if not it returns -1 to the function that called indicating that this directory does not exist so the calling function can handle it. If it does exist it checks that we have permissions to CD into it and also that its a directory and not a file. If everything is ok it calls changeWorkingDir() mentioned earlier that changes to that directory, while also changing currentNode and reading the directories and files from disk into the workingDir array. When it is done it returns 0, indicating everything went ok. If we then take a step back and return to pareTilFile() that was the caller and returned the last directory or file, we don't know yet. In this case the caller of parseTilFile() was parsePath() so it is up to that function to handle the last returned file or directory. In this case we expect it to be a directory so we call changeDirectory() again but this time from parsePath(). If changeDirectory() returns -1 we know it either doesn't exist, is a file or we don't have permissions. So parsePath resets the workingDir to the directory that everything was originally called from and returns -1 indicating something went wrong. If changeDirectory() instead returns 0 we know that we changed the working directory successfully, we then return to the first function CD() that only contains one if statement calling parsePath, CD() then returns 1 for error or 0 for success depending on what parsePath() returned and we are done.