DV1629 Lab 3

Emil Karlström and Samuel Jonsson

January 14, 2022

Contents

1 Task 1. Implementation of basic file operations

1.1 `format`

In `format`, we begin with setting `fat[0]` and `fat[1]` to `FAT_EOF` to signify that the `root` and `FAT` block are used, as well as setting all other blocks to `FAT_FREE` to make them available for use, as per instructions.

Then set everything in the `root` directory to 0 to avoid fragmented data, after which we write the `root` and `FAT` block to the disk.

1.2 `create`

In `create` we start with doing some error handling; first we make sure the file name is not too long (> 55 characters). Then we make sure the file does not already exist in the target directory.

In order to find out what directory to put the file in (the target directory), we "chop off" the file name from the file path, that is to say, the last part of the path. This we do using a self-made function called `chop_file_name()` together with a function called `find_final_block()` that finds the number of the last directory in the path consisting of only directories (after we chop of the file name).

After this, we read user input line by line into a string accumulator and stop after an empty line.

We then proceed to write the text to the disk in chunks of size of one block and update the `FAT` table, and we stop when there is no more text to write. After which we find an empty directory entry in the target directory, which we populate with the file name, size of the file, the number of the first file block, the type of the file (`TYPE_FILE`), as well as read/write access rights (`READ|WRITE` on initial creation).

Lastly, we write the directory data to the target directory's block and update the `FAT` table on the disk.

1.3 `cat`

In `cat` we start with finding the file name and the path, as well as the target directory's block in the same way we did in `create`. We do error checking, making sure the file exists in the target directory (if a path is provided, else in the current directory). After this we make sure the file has `READ` access rights enabled and that it is not a directory.

Then we read the file block-wise and print it to the terminal, until we reach the end of the file.

1.4 `ls`

We start by reading in the current directory's block and then proceed to iterate over the block and find all files that are "visible", using a self-made function called `file_is_visible()`. For each file we print out a formatted string that represents the file in a systematic way.

2 Task 2. Implementation of file operations, part 2

2.1 `cp`

In `cp` we begin wit making sure that the source file exists and has `READ` access rights. We then decide which block is the target directory block to write to, as well as what will become the name of the copied file. In order to do this, we break it down into such:

- If there is a "/" in the destination path, then we trace the path to find the destination block number using our self-made function `find_final_block()`. The name of the copied file will just be the same name as the source file in this case.
- Else, we know that `destpath` is just a file name in the current directory.
  - If there is a file with the name `destpath` in the current directory, that means we have to check whether that is a file or a directory. If it is a file then we abort as there is already a file with the name `destpath`, however if that file is a directory then we will copy the source file into that directory and give the new file the same name as the source file.
  - If no file with the name `destpath` exists in the current directory, that means we are simply copying the source file into a new file which will be named `destpath` and it will be located in the current directory.

After we have decided the intended name of the new file and which directory to copy it to we must make sure that there is not already a file by that name in the target directory, if such is the case then we abort.

Next up we load in the target directory so we have both the source and the destination loaded into memory. We look through the target directory for an empty directory entry to populate with our new data. If no such entry can be found then it's a sign that the directory is full and we abort. We copy over the data from the source file directory entry into the new directory entry. After such we copy the source file by reading it block-by-block and writing the data to new blocks, we also update the FAT table accordingly.

Finally we write the new target directory data to the disk as well as update the FAT table on the disk with the new updated data.

2.2 `mv`

In order to move or rename a file on the disk we must first make sure the file exists, we do this by evaluating which directory block on the disk its directory entry lies in. If we find the file we proceed by loading the directory into memory and make sure that the file is a `TYPE_FILE` and not a directory as `mv` should only work on files.

Afterwards we check whether or not `destpath` includes any "/", this is an easy way to quickly find out if we are working with paths that are not the current working directory or a sub-directory of it. We also check if there is a file named `destpath` in the current directory in order to not have any name collisions.

- If there is no `/` in `destpath` and there is no file named `destpath`:

  We proceed by making sure that `destpath` is not longer than 55 characters and then we simply change the name of the file to `destpath` and write the new data to the disk.

- However, in any other case we are moving the file to another directory. We begin by making sure that `destpath` is a valid potential path, that is to say that all directories that appear in `destpath` are valid sub-directories. If `destpath` is a valid path then we evaluate the block number of the final directory in the path and load that block into memory.

  We also make sure that there is no file with the same name as our source file in order to avoid name collisions, as well as making sure that our target directory is not the same as the current directory. From here we find an empty directory entry using a self-made function called `find_empty_dir_entry_id()` that we can populate, and then we copy over the source data. We mark the original file as "not used" by setting the `size` and `first_blk` attributes both to 0.

  Finally we write the new data to the disk and inform the user about the function's success.

2.3 `rm`

First we make sure that the file exists, if it does not we return error code 1 and we tell the user that the file does not exist. Then, if the file exists, we decide which file type it is:

- If it is a `TYPE_FILE`, e find the first block in the file and iterate over all blocks and mark them as `FAT_FREE`, after which we mark the dir entry as not used by setting `first_blk` and `size` to 0.

- Else, if it is a directory, we check if the directory is empty. If it is not, then we return error code 1 and tells the user that they cannot remove a non-empty directory.

  However, if it is empty, we mark the `dir_entry` as `FAT_FREE`, as well as marking it as not used by setting `first_blk` and `size` as 0.

  When we look through the directory to see if it is empty, we ignore the `..` directory, as it will always be there.

Finally, we write the new data to the disk.

2.4 `append`

In `append`, we first make sure both files exists. If either does not, we return error code 1 and announce it to the user.

After the check, we make sure that `file1` has `READ` permissions, and that `file2` has both `READ` and `WRITE` permission. If any permission is not set right, we tell the user what permission is missing and return error code 1.

After all the pre-work is done, we prepare a buffer that is `BLOCK_SIZE` $\cdot$ `2` bytes big. We then read the last block of `file1` into the buffer, as well as the first block of `file2`. We then, while there is data left to write:

- write one block of data in `buf` to the disk and mark the block as `FAT_EOF`. If the file data in `buf` is less than a block size, we quit the loop, as there is no more data to write or read.
- check if there is data left to read from `file2`, if there is, we read the next block of `file2` into `buf`.
- find an empty block and mark the previous block to point to the new empty block in the FAT.

Finally, we mark the final block as `FAT_EOF` and update `file2`'s `size` to the sum of the two file's sizes and write it to the disk.

3 Task 3. Implementation of directory hierarchies

3.1 `mkdir`

First we make sure the file exists, if it does not, we return error code 1 and announce it to the user.

After that we make sure there is enough space on the disk to create a new directory. If there is space, we create a new `dir_entry` and populate the data with correct `name`, `size`, `first_blk`, `type` and `access_rights`.

We then reset the new directory so that any old information is removed, after which we create a `..` directory within that directory that points to the parent directory and populates it wit the necessary information.

Finally we update FAT and write everything to the disk

3.2 `cd`

In `cd`, we begin to make sure that `dirpath` is a path on the system, and return error code 1 and an error message if it is not.

We then set `final_block` to the block that `dirpath` points to using a self-made function called `find_final_block`.

3.3 `pwd`

We begin with checking whether the current directory is `root`. If it is, we simply print `/`.

However, if it is not, then we iterate backwards through the directory hierarchy in order to build a string of directory names. We do this by finding the `dir_entry` in the parent directory that points to the current directory. We append its `file_name` to the start of the string and also append a `/` in position 0. We do this until we reach `root`.

4 Task 4. Implementation of absolute and relative paths

To implement absolute and relative paths, we created three functions:

- `find_final_block`
- `chop_file_name`
- `get_file_name_from_path`

`find_final_block` is mainly used to turn a path into a block number, which can the be read from the disk. `chop_file_name` is used to get the directory path of `filepath`, while `get_file_name_from_path` is used to get the file name from a full `filepath`.

4.1 `find_final_block`

`find_final_block` is used to find the final block on the file system in a path of only directories, and returns the block number. It takes into consideration special cases, such as

- `path` being empty: returns the starting block.
- if `path` is only `/`: returns `ROOT_BLOCK`.
- if `path` starts with `/`, where it sets the current block to `ROOT_BLOCK` and continues as normal.

4.2 `chop_file_name`

`chop_file_name` removes the file name from `filepath`. For example, turns `/dir/dir2/file` into `/dir/dir2`.

4.3 `get_file_name_from_path`

`get_file_name_from_path` removes everything from `filepath` up to and including the final `/`.

5 Task 5. Implementation of access rights

5.1 `chmod`

We start with making sure that the file exists, if it does not, we tell the user that it does not and returns error code 1.

we then make sure that `accessrights` is a valid access right, that is, it is a value between 0 and 7, inclusive. If it is not, we tell the user that their input is invalid, otherwise, we read the file from the disk, set `access_rights` to `accessrights` and write the new data to the disk.