In this project, a virtual file system is to be implemented. A data structure similar to INodes will be used to store data in an organised manner. The data should be both readable and writable. All of this data will be stored in a single file located on your computer. It is important to read the notes on this task sheet thoroughly.

A filesystem-file should be recognisable by the file extension `.fs` (although this is not required) and consists of the following blocks:

- Superblock, which contains:
  - the total number of blocks
  - the number of free blocks
- Free space, represented as an array of free blocks
- INodes
- Data blocks, which contain the file data

You can find the exact structure of each component in the struct definitions in filesystem.h or in the diagrams below.

The program is started using one of the following two commands:

```
./build/ha2 -c <FILESYSTEM_NAME> <FS_SIZE>

./build/ha2 -l <FILESYSTEM_NAME>
```

The first command creates a new filesystem (a file with the `.fs` extension) in which all further operations will be carried out. The second command opens an existing file system. `FILESYSTEM_NAME` is the name of the file, and `FS_SIZE` is the size of the filesystem, i.e., the number of 1024-byte blocks and INodes.

A shell will open, similar to the one you already know from the first exercise, in which you can enter the implemented commands.

The file system should be able to perform the operations listed in Table ??.

The entire shell, including reading user input and command-line arguments, as well as reading the filesystem file, is provided.

To exit the program, you can enter `quit` or `exit`.

Before you begin, familiarise yourself with the provided structure. Your tasks consist of implementing some of the functions in the `operations.c` file, so be sure to read the comments in the file carefully.

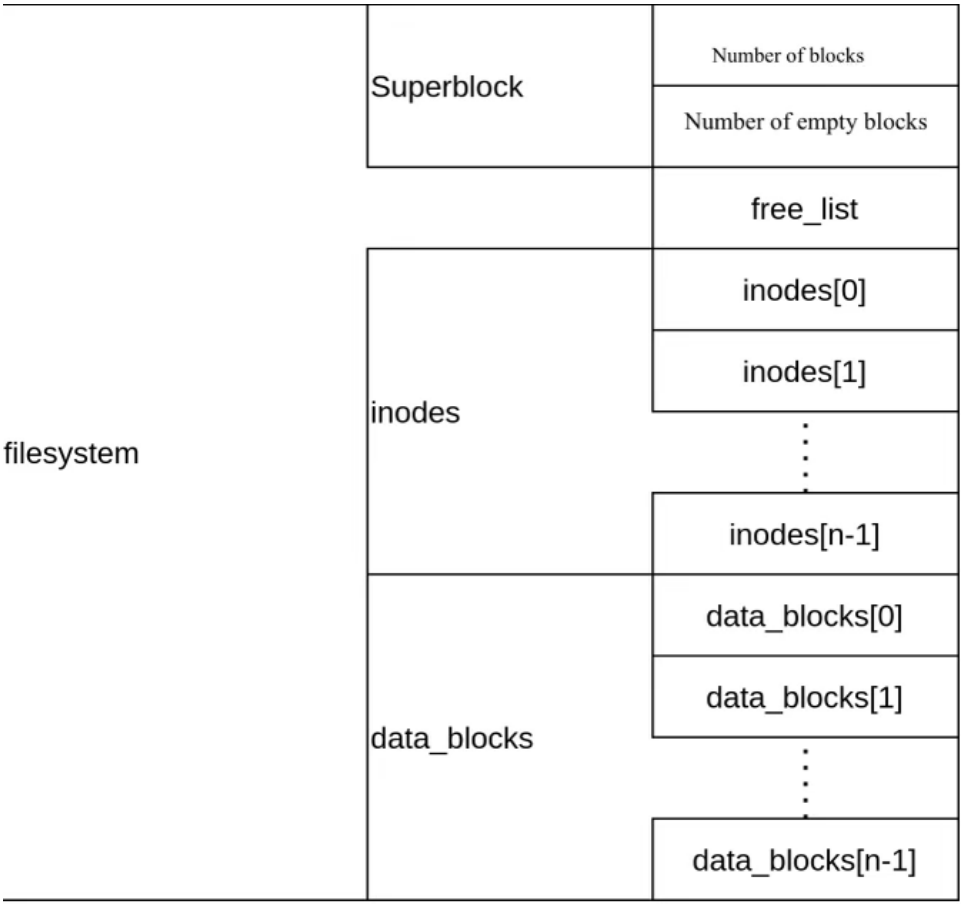

Figure 1: Overview of the file system structure

Command	Description
`mkdir <path>`	Create a new directory
`mkfile <path>`	Create a new file
`cp <path> <path/name>`	Copy an object to the specified path with the given name
`list <absolute path >`	Display the contents of a directory
`writef <path> <text>`	Write text to a file
`readf <path>`	Read text from a file
`rm <path>`	Delete a file or a directory
`export <internal path> <external path>`	Export a file
`import <internal path> <external path>`	Import a file
`dump`	Back up the file system to the hard drive

Table 1: Commands that the filesystem should understand and their descriptions. All paths must be specified as absolute paths from the root of the filesystem.“Internal path” refers to the path inside the filesystem, and “external path” refers to the one on your computer. The `dump` command is already implemented.

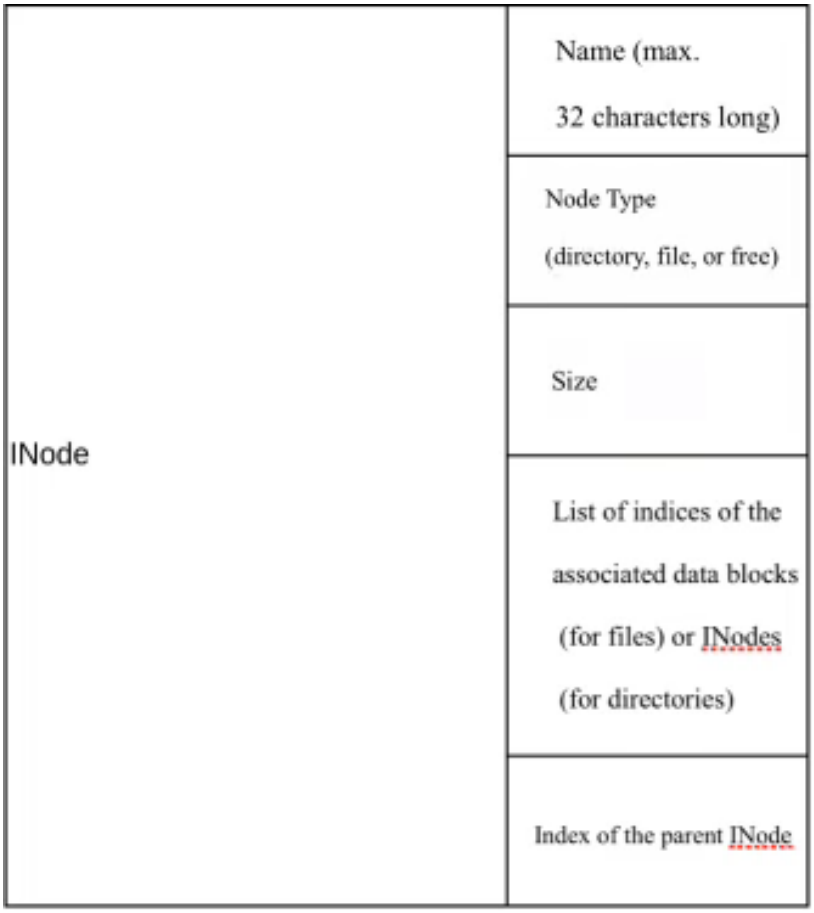

Figure 2: Structure of an INode. If an INode represents a directory, the array “Array der Indizes” contains the indices of the child INodes (i.e. other directories or files). If the INode represents a file, this array contains the indices of the associated data blocks.

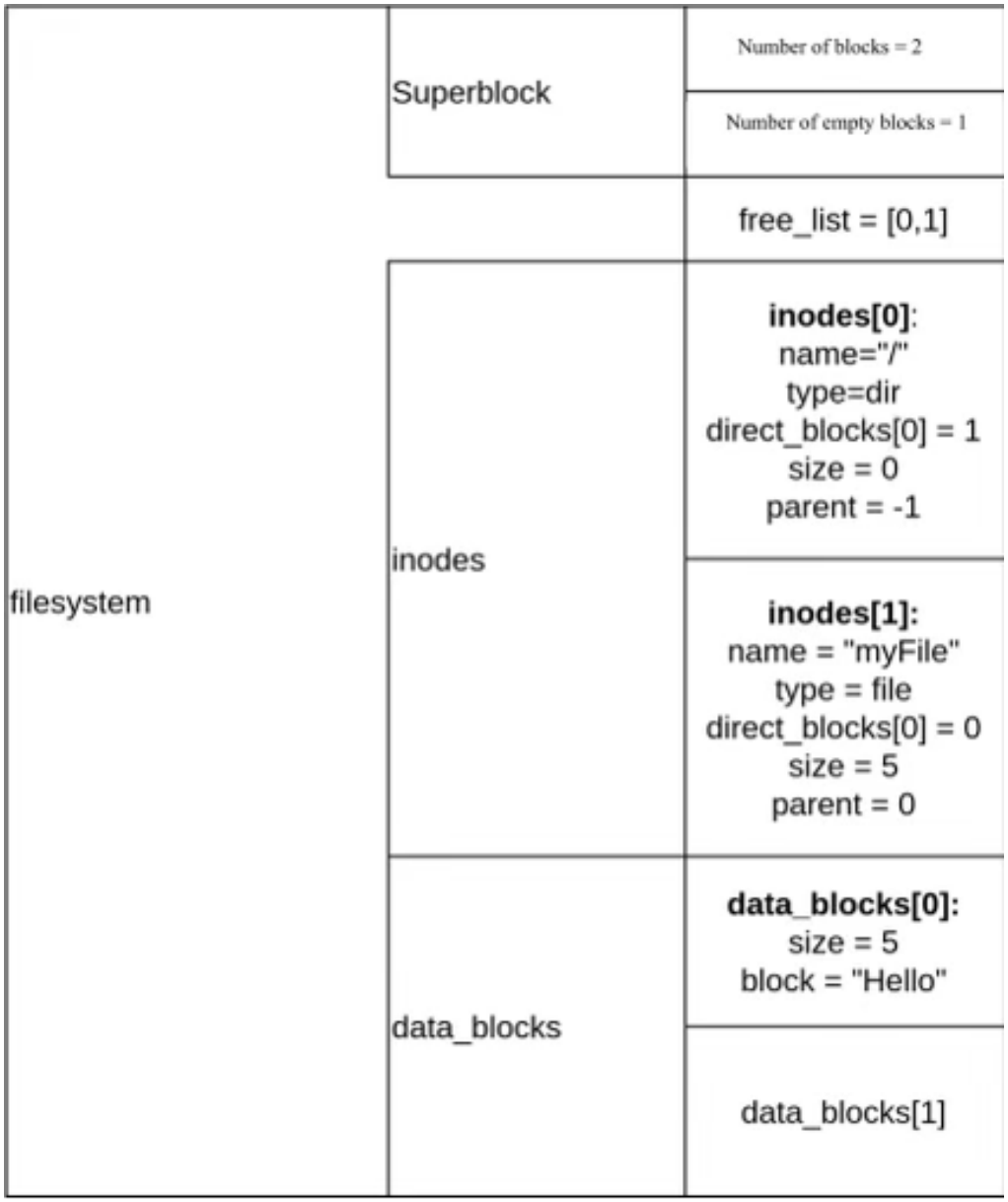

Figure 3: File system with 2 allocated blocks. The file `/myFile` has been created, and the data `Hello` has been assigned to it.

Carefully read the comments in the `operations.h` file. They provide valuable insights into how the functions work. The first argument, `file_system *fs`, is always the filesystem to be operated on.

Optionally, create additional test cases. You can use the provided examples as a guide.

Task 1: mkdir

Implement the function `fs_mkdir`. It should create a new directory in the filesystem. It receives an absolute path at which the directory should be created. To create a directory, use the smallest available free INode. Set its `n_type` to `directory`, copy the provided name into the INode's name field, and set its parent to the INode number of the superior directory. In the parent INode, the `direct_blocks`-array must be updated to include the new directory's INode number at the lowest available index.

Task 2: mkfile

Implement the function `fs_mkfile`. It should create a new file in the filesystem. It receives an absolute path where the file should be created. Attention: The folder where the file is to be created must exist beforehand. The creation of a file happens analogously to directory creation, however the INode-type must be set to reg_file.

Task 3: cp

Implement the function fs_cp. It should copy a file or directory in the filesystem. It receives the source path and the destination path (where the last part of the destination path is the new name). Attention: Directories must be copied recursively.

Example: `cp /a /b/c`: copy directory `/a` into directory `/b` and name it `c`.

(Note: `mkdir` and `mkfile` must be implemented first for the tests to work.)

Task 4: list

Implement the function `fs_list`. It receives an absolute path. It should return a string listing all files and directories contained in the given directory. The entries must be sorted by INode number. Each item should appear on its own line. Prefix `DIR` for directories and `FIL` for files, separated by a space from the name. Do not add any extra characters to the output string.

Example: The directories „newDir" (`inode[1]`) and „newerDir" (`inode[3]`) and the file „myFile.txt" (`inode[2]`) were created in „/". Then `list /` will result in the following output:

```
DIR newDir
FIL myFile.txt
DIR newerDir
```

Task 5: writef

Implement the function `fs_writef`. It receives an absolute path pointing to the file to be written to, and the text that should be written. The text must now be written into that file. Attention: The file must already exist. If the same file is written to multiple times, the new data should be appended — not overwrite the existing content.

If the file is currently empty:

- Find the free `data_block` with the lowest index
- Write the data into the `block`-field of that block
- Set the `size` in both the `data_block` and the INode to the length of the written data
- Store the index of the used block at position 0 of the array `inode->direct_blocks`

If the data is larger than a single block, use as many free blocks as needed. Mark each used block as "not free" by setting the corresponding entry in the `free_list` array to 0.

Task 6: readf

Implement the function `fs_readf`. It should read the content of a file. It receives an absolute path pointing to the file to read, and a pointer to an integer variable, into which the file size should be written. The data to read is that stored in the `direct_blocks` of the INode, i.e. the data previously written with `writef`.

Task 7: rm

Implement the function `fs_rm`. It should delete a file, or a directory and its entire content recursively (including all subfolders and their contents). It receives an absolute path pointing to the file or directory to be deleted. Make sure to mark deleted INodes as free, and free up any used data blocks for future use.

Task 8: import

Implement the function `fs_import`. The parameter `int_path` (internal path) is the path to a file within the virtual filesystem. The parameter ext_path (external path) is the path to a file (not necessarily existing) on your real system (computer). The function should read the file at `ext_path` from your real system and write its contents into the internal file at int_path. Attention: Similar like with `writef`, the internal file must already exist.

Task 9: export

Implements the `fs_export` function. The argument `int_path`, the internal path, is the path to a file in the filesystem, while the argument `ext_path`, the external path, is the path to a (not necessarily existing) file on your computer. The function should then export the file with the internal path to the file system of your computer, at the location of the external path. The external file does not necessarily have to exist beforehand.

Notes:

- Function Descriptions: You can find more detailed descriptions of the individual functions in the .h files located in the lib folder.
- Program Limitations:
  - File names are limited to 32 characters (including the null byte).
  - Each file can address a maximum of 12 blocks.
  - You do not need to check whether these limits are exceeded in your implementation. Assume that:
    - names will not exceed the limit,
    - files will not be too large, and
    - the total space will not exceed the available number of blocks.
- File and Directory Names: A directory cannot contain both a file and a subdirectory with the same name.
- Specifications:
  - Do not change any existing data structures, function names, header files etc.
  - Only modify the file `src/operations.c`.
  - You are allowed to define and implement additional helper functions or data structures within this file.
- Makefile:
  - Use the provided Makefile to build your program.
  - Run make in the main directory to compile the project with clang.
  - On Ubuntu, you can install make and clang via: sudo apt install make clang
  - Use make clean to remove compiled files.
  - You are free to add extra flags or modify the Makefile, as long as your program still compiles correctly using the provided Makefile.
- Memory Leaks:
  - Finally, check your program for memory leaks to ensure all allocated memory is properly freed.
  - Recommended tool: valgrind (http://valgrind.org/)
- Tests:
  - Some test cases for individual functions are provided in the tests directory.
  - However, creating your own tests is strongly recommended as the provided ones do not cover everything.

    To use the provided tests:

  - Make sure Python and the pytest module are installed. On Ubuntu:
    - Install python3: `sudo apt install python3`
    - Install pip (Python Paketmanager): `sudo apt install python3-pip`
    - Install pytest: `python3 -m pip install pytest`

  - You can run these test with `make test`. This generates a `.so`-file that is then loaded by the test framework and executes all available tests. To run only specific tests (e.g. for `mkdir`), use: `make test_mkdir`
  - You can find all test names in the tests directory.
  - If you use a different Python version or command than `python3`, adjust line 23 and 26 in the `Makefile` accordingly.

- There is no test provided for the function `fs_export`. You can write your own tests or test it manually by running the program.
- For manual testing, the file `SysProgFiles.fs` is available. Load this file with your program and locate an image file within. If you export this image to your host system, you should be able to view it.

---

[1] http://valgrind.org/