OPERATING SYSTEMS, DV1628/DV1629

—

LAB 3: FILE SYSTEMS

Håkan Grahn
Blekinge Institute of Technology

2022 fall
updated: 2022-11-25

The objective of this laboratory assignment is to study how a file system can be organized and implemented.

The laboratory assignments should be conducted, solved, implemented, and presented in groups of two students! It is ok to do them individually, but groups of more than two students are not allowed.

Home Assignment 1. Preparations

Read through these laboratory instructions and do the **home assignments**. The purpose of the home assignments is to do them *before* the lab sessions to prepare your work and make more efficient use of the lab sessions.

Read the following sections in the course book [1]:

- Section 4.1: Files
- Section 4.2: Directories
- Section 4.3: File-system implementation
- Section 4.4: File-system management and optimization
- Section 4.5.1: The MS-DOS file system
- Section 5.4: Disks
- Section 10.6: The Linux file system

End of home assignment 1.

Home Assignment 2. Plagiarism and collaboration

You are encouraged to work in groups of two. Groups larger than two are not accepted.

Discussions between laboratory groups are positive and can be fruitful. It is normally not a problem, but watch out so that you do not cross the border to cheating. For example, you are *not allowed* to share solution approaches, solutions to the different tasks, source code, output data, results, etc.

The submitted solution(s) to the laboratory assignments and tasks, should be developed by the group members only. You are not allowed to copy code from somewhere else, i.e., neither from your student fellows nor from the internet or any other source. The only other source code that is allowed to use is the one provided with the laboratory assignment.

End of home assignment 2.

1 Introduction

In all computer systems there is a need for persistent long-term storage of information. This persistent storage system also needs to be of substantial size to be able to hold large quantities of data. Different types of disks constitute this secondary storage. Further, we want the information to be accessible from several processes, thus the information storage must be independent of individual processes.

Thus, we have three essential requirements for long-term information storage [1]:

1. It must be possible to store a very large amount of information.
2. The information must survive the termination of the process using it.
3. Multiple processes must be able to access the information at once.

When storing information on disks, the information is usually stored as a sequence bytes. The physical disk itself is organized in a number of blocks, where each block contains a fixed number of bytes. Some questions that need to be addressed are, e.g.,

- How do we structure and find the information on disk?
- In which blocks do we store the information of a particular file?
- How do we keep track of which blocks free / available and which are allocated?

In this laboratory we will look into some of the aspects above by implementing a simple file system, very similar to the FAT file system. The file system we will implement should be organized as a file allocation table (linked allocation with an index table, see Sections 4.3 and 4.5.1 in [1])

The source code provided in this lab assignment is tested on Ubuntu Linux, and thus works in the lab room (G332). However, the code should also work on any Linux distribution you may have at home. The source code files necessary for the laboratory are available on Canvas.

2 Examination and grading

Present and discuss your solutions with a teacher / lab assistant when all tasks are done.

When you have discussed your solutions orally, prepare and submit a `tar`-file or `zip`-file containing:

- **Source code:** The source-code for working solutions to the tasks in this laboratory assignment. Specifically, your well-commented source code for **Task 1**, **Task 2**, **Task 3**, **Task 4**, and **Task 5**.
  Note: All your code should be in the files `fs.h` and `fs.cpp`. You only need to submit the final version of the code, since the tasks build upon each other.
- **Written report:** You should write a short report (approximately 2-3 pages) describing your implementations, with a general description of your solution, the data structures used, motivation of design decisions, etc. The format of the report must be pdf.

All material (except the code given to you in this assignment) must be produced by the laboratory group alone.

The examiner may contact you within a week, if he/she needs some oral clarifications on your code or report. In this case, all group members must be present at that oral occasion.

Here are some general comments on submission of the code and the testing that will done:

- All your code should be in the files `fs.h` and `fs.cpp`, i.e., those are the only files copied to the test directory where all tests are executed.
- A general requirement of your implementation is that it should follow the C++11 standard, and correctly compile and execute on Ubuntu Linux (see Task 1). That means:
  - The tests will be compiled and executed on Ubuntu Linux, failing to do that means grade 'U'.
  - We will use exactly the same version of all other files that you got with the lab assignment.
  - We will use exactly the same `Makefile` that you got with the lab assignment.
- Paths to directories are written as '`/dir1/subdir2`', and the root directory should be '`/`' (not 'root')

- You cannot assume that the disk (i.e., the file `diskfile.bin`) is formatted or formatted with your file system.
  - We have tested other file systems before, i.e., the disk may have a formatting or content that is inconsistent with your file system.
  - The first command we execute at the tests is '`format`', i.e., we format an empty disk with your file system before we execute the rest of the tests.
- Basic error checks must be done, e.g., it should *not* be possible to create a new file or directory with the same name as an existing file or directory, a directory and a file cannot have the same name (leads to a number of strange behaviors and errors), and the access rights on a directory must be correct for various file operations (e.g., moving/copying a file to a directory requires write access on that directory), etc.
- Access rights of a file or directory should be '`rw-`' or '`rwx`' when the file or directory is created.
- No extra or unnecessary text should be printed on the terminal, e.g., the initial help text in the provided functions that just prints that a function is called should be removed.

To your help, we provide a file with test commands that your program should handle (`test_commands.txt`). In addition, we have written some test programs that do some basic automatic tests for each task. The test programs are named `test_script1.cpp` for Task 1, etc. You compile and run them as:

```
mymachine$ make test1
mymachine$ ./test1
```

3 Properties of the system

When designing a file system, we need to define a number of properties of the disk system. One of the first parameters is the block size to use. In our file system, we will use a disk block size of 4096 bytes (4kB). In our simple file system this will have an impact on how large disk partitions we will be able to handle with our file system.

In this laboratory we will look into some of the aspects above by implementing a simple file system. The file system we will implement should be organized as a file allocation table, FAT, (linked allocation with an index table, see Sections 4.3 and 4.5.1 in [1]). In our system, we assume that the root directory always is located in disk block 0 and the FAT is always located in disk block 1.

A disk partition consists of a sequence of disk blocks. Since the FAT in our system is stored in one block, this limits how many blocks we can address in the FAT. One FAT entry in our file system is 2 bytes and each entry in the FAT corresponds to one disk block. This means that we can address 4096 / 2 = 2048 disk blocks in a partition (since there is one FAT per partition), i.e., a partition can be at most: 2048 entries in the FAT * 4 kB per block $\Rightarrow$ 8 MB per partition.

A file / directory entry keeps information about a file or a sub-directory. The structure of a directory entry in our simple file system is shown in Listing 1. Figure 1 shows how the FAT is organized and how a file/directory entry stores the information about a file and points out the starting location for the file. The file `file1` is 9432 bytes in size, which means that it needs to be stored in three disk blocks ($4096 + 4096 + 1240 = 9432$), i.e., blocks 4, 7, and 8. Since we don't know anything about the other entires of the FAT, they are marked with a '?'. When you initialized your FAT with the `format` command (Task 1), all entries in the FAT should be marked as 'free', except entry 0 (the root directory) and entry 1 (the FAT block).

Listing 1. `direntry.h`

```
struct dir_entry {
    char file_name[56]; // name of the file / sub-directory
    uint32_t size; // size of the file in bytes
    uint16_t first_blk; // index in the FAT for the first block of the file
    uint8_t type; // directory (1) or file (0)
    uint8_t access_rights; // read (0x04), write (0x02), execute (0x01)
};
```

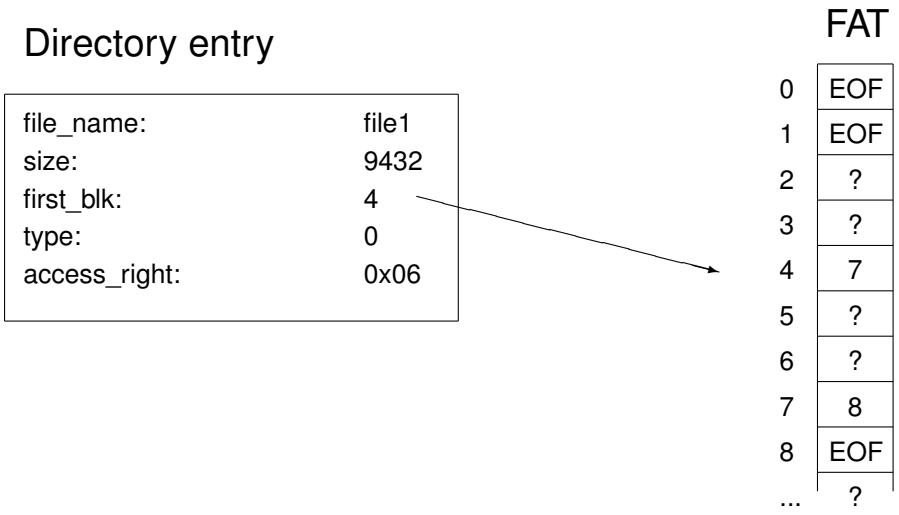

Figure 1: Overview of how a file / directory entry and the FAT are related to each other. The example file `file1` is 9432 bytes, is stored in blocks 4, 7, and 8, and we have read and write access.

Home Assignment 3. Study the provided source code

Read through the provided source code so you understand the different parts of it. It contains four parts:

- **main**: Initialization and starting up the shell.
- **shell**: A simple interactive shell that reads commands from the user and calls the file system.
- **fs**: API for the file system, implements the file system and does block reads from / writes to the disk.
- **disk**: Implementation of a disk (block device), the disk is simulated by a binary file (`diskfile.bin`).

In the laboratory assignment you should modify *only* the files `fs.h` and `fs.cpp` (i.e., the file system part).

End of home assignment 3.

Home Assignment 4. Directory design

How long file or directory names can be stored (including file extention) in a directory entry?

How many files or directories can each directory contain?

End of home assignment 4.

4 Basic functionality and one-level directory

In the first part of the laboratory assignment we are going to implement basic functionality of a single-level directory. This means that we need to implement commands for:

- formatting the disk, i.e., initializing the FAT and marking all blocks as free (except block 0 (the root directory) and block 1 (the FAT))
- creating a file and reading a file
- listing the contents of a directory

We are going to develop our file system step-wise, including changing the implementations of previously implemented commands. Therefore, we strongly recommend that you use some kind of version control system, e.g., github, or at least take backups / snapshots of your code base regularly. There is a significant risk that you introduce errors in previously working versions, and need to step back to a working version of your code.

Task 1. Implementation of basic file operations

Implement the following commands:

- `format`
  Formats the disk, i.e., initializes the FAT and marks all blocks as free (except block 0 (the root directory) and block 1 (the FAT block)).
- `create <filename>`
  Creates a new file with the name *filename* on the disk. The data contents are written on the following rows. An empty row ends the user input data. This enables the user to write several lines of input.
  When all input data is written, the data is stored in the file *filename*. Remember to handle the case when a file is larger than one disk block.
  If there already exists a file with the same name, an error should be detected.
- `cat <filename>`
  Reads the contents of the file *filename* and prints it on the screen.
- `ls`
  Lists the contents of the current directory (in this case the root directory) on the form:

```
name        size
file1       37
file2       128
file3       9432
```

Note: A general requirement of your implementation is that it should follow the C++11 standard, and correctly compile and execute on Ubuntu Linux.

Remember to *always* keep the FAT updated on the disk *after each file system operation*.

Keep in mind that you only can read (and write) full disk blocks from (and to) the simulated disk.

Since the disk will be modeled as a binary file (`diskfile.bin`), you can e.g. use the program `xxd` to inspect at the contents of the disk.

End of task 1.

The next step in our file system development is to implement operations for copying files, changing names of files, deleting files, and appending the content of one file at the end of another file.

Task 2. Implementation of file operations, part 2

Implement the following commands:

- `cp <sourcefilename> <destfilename>`
  Makes an exact copy of the file *sourcefilename* to a new file *destfilename*.
  If *destfilename* already exists, you shall give an error message (i.e., noclobber semantics). Existing files shall never be overwritten.
- `mv <sourcefilename> <destfilename>`
  Renames the file *sourcefilename* to the name *destfilename*.
  If *destfilename* already exists, you shall give an error message (i.e., noclobber semantics). Existing files shall never be overwritten.
  Note that it is *not* ok to first copy the source file using `cp` and then remove the source file using `rm`.
- `rm <filename>`
  Deletes the file *filename*, i.e., removes its directory entry and marks the corresponding disk blocks as free.
- `append <filename1> <filename2>`
  Appends the contents of file *filename1* to the end of file *filename2*. The file *filename1* is unchanged. You do *not* need to cover the special case when appending a file to itself, i.e., '`append <filename1> <filename1>`'.

End of task 2.

5 Hierarchical directories

Any file system needs to support hierarchical directories, i.e., directories and sub-directories at different levels. We limit our implementation in the sense that one (sub-)directory is not larger than one disk block.

Task 3. Implementation of directory hierarchies

Implement the following commands:

- `mkdir <dirname>`
  Creates a new sub-directory named *dirname* in the current directory. When a sub-directory is created, it should always contain a "sub-directory" named `'..'` which refers to the parent directory (i.e., the directory where we created the sub-directory).
  Change the `ls` command so it also prints the type, i.e., if a directory entry is a regular file or a directory. The data structure `struct dir_entry` already has support for this, see Listing 1 and the file `fs.h`.
- `cd <dirname>`
  Changes the current (working) directory to the directory named *dirname*. Remember to cover the case when *dirname* is `'..'`
- `pwd`
  Prints the full path, i.e., from the root directory, to the current directory (including the current directory name). A path to a directory is written as '`/dir1/subdir2`', and the root directory should be '`/`'

In addition, *some of the previous commands may need to be updated*. For example, `cp` and `mv` can now have a directory as destination, i.e., `mv <sourcefilename> <destdirname>`, which means that the file `sourcefilename` is not renamed but *moved* to a sub-directory `destdirname`. Other examples are `cat <filename>` which should give an error message if the given `<filename>` is a directory, and `rm <dirname>` that removes the directory `dirname` if the directory is empty (otherwise it shall give an error message).

Further, the `ls` command shall print also the type (file/dir) of each directory entry in the form:

```
name          type        size
dir1          dir         -
file1         file        37
file2         file        128
file3         file        9432
subdir2       dir         -
```

End of task 3.

The final step when implementing hierarchical directories is to make sure that we can use absolute and relative directory paths. An absolute path always starts at the root directory, while a relative path starts at the current directory. We use `'/'` as directory separator (as in Unix/Linux) and the "sub-directory" `'..'` always refers to the parent directory (also as in Unix/Linux), see Task 3.

Task 4. Implementation of absolute and relative paths

Implement absolute and relative paths in your file system. Use `'/'` as directory separator and the "sub-directory" `'..'` always refers to the parent directory.

Note that several of your previous commands may need to be modified to account for absolute/relative file paths as arguments. Examples of commands that may need to be updated are `cd`, `mv`, `cp`, `cat`, etc.

End of task 4.

6 Access rights

Any proper file system needs to handle access rights for the files and directories. For example, we should not be able to write to a file which is read-only. Therefore, we need to implement that in our file system.

In the file `fs.h` we have defined the different access rights and the data structure `struct dir_entry` in Listing 1 supports it:

#define READ 0x04
#define WRITE 0x02
#define EXECUTE 0x01

If we would like a file to have, e.g., both read and write permissions we set the access rights by performing a bitwise or-operation:

```
(0x04 | 0x02) = 0x06
```

This means that the command:

```
chmod 6 fileone
```

enables read and write access on the file `fileone`.

Task 5. Implementation of access rights

Implement the following command:

- `chmod <accessrights> <filepath>`
  Changes the access rights for the file <filepath> to <accessrights>.

Note that several of your other commands may need to be modified to check the access rights, e.g., that you have read permission on `filename` when `cat <filename>` is executed, that you have read permission on `filename1` and write permission on `filename2` when you execute `append <filename1> <filename2>`, etc .

Further, the `ls` command shall print also the access rights for each file/directory:

```
name          type       accessrights       size
dir1          dir        rwx                -
file1         file       r-                 37
file2         file       rw-                128
file3         file       rw-                9432
subdir2       dir        r-x                -
```

The access rights should be printed as a triplet `rwx`. If a particular file/directory does have a particular access right set, you should print a `'-'`. For example, a file with read and execute permissions has the access rights `r-x`.

End of task 5.

References

[1] Andrew S. Tanenbaum and Herbert Bos, "Modern Operating Systems, 4th ed", *Pearson Education Limited*, ISBN-10: 0-13-359162-X, 2015.

A Program listings

Listing 2. `main.cpp`

```
#include "shell.h"
#include "fs.h"
#include "disk.h"

int
main(int argc, char **argv)
{
    Shell shell;
    shell.run();
    return 0;
}
```

Listing 3. `disk.h`

```
#include <iostream>
#include <fstream>

#ifndef __DISK_H__
#define __DISK_H__

#define DISKNAME "diskfile.bin"
#define BLOCK_SIZE 4096
#define DEBUG false

class Disk {
private:
    std::fstream diskfile;
    const unsigned no_blocks = 2048;
    const unsigned disk_size = BLOCK_SIZE * no_blocks;
    bool disk_file_exists (const std::string& name);
public:
    Disk();
    ~Disk();
    unsigned get_no_blocks() { return no_blocks; }
    unsigned get_disk_size() { return disk_size; }
    // writes one block to the disk
    int write(unsigned block_no, uint8_t *blk);
    // reads one block from the disk
    int read(unsigned block_no, uint8_t *blk);
};

#endif // __DISK_H__
```

Listing 4. `disk.cpp`

```
#include <iostream>
#include "disk.h"

Disk::Disk()
{
    // first check if the disk file exists, otherwise create it.
    if (!disk_file_exists(DISKNAME)) {
```

```
            std::cout << "No disk file found...\n";
            std::cout << "Creating disk file: " << DISKNAME << std::endl;
            std::ofstream f(DISKNAME, std::ios::binary | std::ios::out);
            f.seekp((1<<23)-1);
            f.write("", 1);
        }
        // the disk is simulated as a binary file
        diskfile.open(DISKNAME, std::ios::in | std::ios::out | std::ios::binary);
        if (!diskfile.is_open()) {
            std::cerr << "ERROR: Can't open diskfile: " << DISKNAME << ", exiting..."<< std::endl;
            exit(-1);
        }
}

Disk::~Disk()
{
    diskfile.close();
}

bool
Disk::disk_file_exists (const std::string& name) {
    std::ifstream f(name.c_str());
    return f.good();
}

// writes one block to the disk
int
Disk::write(unsigned block_no, uint8_t *blk)
{
    if (DEBUG)
        std::cout << "Disk::write(" << block_no << ")\n";
    // check if valid block number
    if (block_no >= no_blocks) {
        std::cout << "Disk::write - ERROR: Invalid block number (" << block_no << ")\n";
        return -1;
    }
    unsigned offset = block_no * BLOCK_SIZE;
    diskfile.seekp(offset, std::ios_base::beg);
    diskfile.write((char*)blk, BLOCK_SIZE);
    diskfile.flush();
    return 0;
}

// reads one block from the disk
int
Disk::read(unsigned block_no, uint8_t *blk)
{
    if (DEBUG)
        std::cout << "Disk::read(" << block_no << ")\n";
    // check if valid block number
    if (block_no >= no_blocks) {
        std::cout << "Disk::write - ERROR: Invalid block number (" << block_no << ")\n";
        return -1;
    }
    unsigned offset = block_no * BLOCK_SIZE;
    diskfile.seekg(offset, std::ios_base::beg);
    diskfile.read((char*)blk, BLOCK_SIZE);
    return 0;
}
```

Listing 5. `shell.h`

```
#include <iostream>
#include "fs.h"

#ifndef __SHELL_H__
#define __SHELL_H__

class Shell {
private:
    FS filesystem;
public:
    Shell();
    ~Shell();
    void run();
};

#endif // __SHELL_H__
```

Listing 6. `shell.cpp`

```
#include <iostream>
#include <sstream>
#include <string>
#include <vector>
#include "shell.h"
#include "fs.h"

std::string commands_str[] = {
    "format", "create", "cat", "ls",
    "cp", "mv", "rm", "append",
    "mkdir", "cd", "pwd",
    "chmod",
    "help", "quit"
};

Shell::Shell()
{
    std::cout << "Starting shell...\n";
}

Shell::~Shell()
{
    std::cout << "Exiting shell...\n";
}

void
Shell::run()
{
    bool running = true;
    std::string line;
    std::string str;
    char c;
    std::vector<std::string> cmd_line;
    std::string cmd, arg1, arg2;
    int ret_val = 0;
    while (running) {
        std::cout << "filesystem> ";
        std::getline(std::cin, line);
        std::stringstream linestream(line);
        cmd_line.clear();
        str.clear();
        while (linestream.get(c)) {
            //std::cout << "parsing cmd line: " << c << "\n";
```

```
        if (c != ' ') {
            str += c;
        } else {
            // strip multiple blanks
            if (!str.empty()) {
                cmd_line.push_back(str);
                str.clear();
            }
        }
    }
    if (!str.empty())
        cmd_line.push_back(str);
    if (line.empty())
        cmd = "";
    else
        cmd = cmd_line[0];

    if (DEBUG) {
        std::cout << "Line: " << line << std::endl;
        std::cout << "cmd: " << cmd << std::endl;
        for (unsigned i = 0; i < cmd_line.size(); ++i)
            std::cout << "cmd/arg: " << cmd_line[i] << "\n";
    }

    if (cmd == "format") {
        if (cmd_line.size() != 1) {
            std::cout << "Usage: format\n";
            continue;
        }
        // check return value so everything is ok
        ret_val = filesystem.format();
        if (ret_val) {
            std::cout << "Error: format failed, error code " << ret_val << std::endl;
        }
    }

    else if (cmd == "create") {
        if (cmd_line.size() != 2) {
            std::cout << "Usage: create <file>\n";
            continue;
        }
        arg1 = cmd_line[1];
        std::cout << "Enter data. Empty line to end.\n";
        // check return value so everything is ok
        ret_val = filesystem.create(arg1);
        if (ret_val) {
            std::cout << "Error: create " << arg1;
            std::cout << " failed, error code " << ret_val << std::endl;
        }
    }

    else if (cmd == "cat") {
        if (cmd_line.size() != 2) {
            std::cout << "Usage: cat <file>\n";
            continue;
        }
        arg1 = cmd_line[1];
        // check return value so everything is ok
        ret_val = filesystem.cat(arg1);
        if (ret_val) {
            std::cout << "Error: cat " << arg1;
            std::cout << " failed, error code " << ret_val << std::endl;
        }
    }

    else if (cmd == "ls") {
        if (cmd_line.size() != 1) {
            std::cout << "Usage: ls\n";
            continue;
        }
        // check return value so everything is ok
        ret_val = filesystem.ls();
```

```
        if (ret_val) {
            std::cout << "Error: ls failed, error code " << ret_val << std::endl;
        }
    }

    else if (cmd == "cp") {
        if (cmd_line.size() != 3) {
            std::cout << "Usage: <oldfile> <newfile>\n";
            continue;
        }
        arg1 = cmd_line[1];
        arg2 = cmd_line[2];
        // check return value so everything is ok
        ret_val = filesystem.cp(arg1, arg2);
        if (ret_val) {
            std::cout << "Error: cp " << arg1 << " " << arg2;
            std::cout << " failed, error code " << ret_val << std::endl;
        }
    }

    else if (cmd == "mv") {
        if (cmd_line.size() != 3) {
            std::cout << "Usage: mv <sourcepath> <destpath>\n";
            continue;
        }
        arg1 = cmd_line[1];
        arg2 = cmd_line[2];
        // check return value so everything is ok
        ret_val = filesystem.mv(arg1, arg2);
        if (ret_val) {
            std::cout << "Error: mv " << arg1 << " " << arg2;
            std::cout << " failed, error code " << ret_val << std::endl;
        }
    }

    else if (cmd == "rm") {
        if (cmd_line.size() != 2) {
            std::cout << "Usage: rm <file>\n";
            continue;
        }
        arg1 = cmd_line[1];
        // check return value so everything is ok
        ret_val = filesystem.rm(arg1);
        if (ret_val) {
            std::cout << "Error: rm " << arg1;
            std::cout << " failed, error code " << ret_val << std::endl;
        }
    }

    else if (cmd == "append") {
        if (cmd_line.size() != 3) {
            std::cout << "Usage: append <filepath1> <filepath2>\n";
            continue;
        }
        arg1 = cmd_line[1];
        arg2 = cmd_line[2];
        // check return value so everything is ok
        ret_val = filesystem.append(arg1, arg2);
        if (ret_val) {
            std::cout << "Error: append " << arg1 << " " << arg2;
            std::cout << " failed, error code " << ret_val << std::endl;
        }
    }

    else if (cmd == "mkdir") {
        if (cmd_line.size() != 2) {
            std::cout << "Usage: mkdir <dirpath>\n";
            continue;
        }
        arg1 = cmd_line[1];
        // check return value so everything is ok
        ret_val = filesystem.mkdir(arg1);
```

```
            if (ret_val) {
                std::cout << "Error: mkdir " << arg1;
                std::cout << " failed, error code " << ret_val << std::endl;
            }
        }

        else if (cmd == "cd") {
            if (cmd_line.size() != 2) {
                std::cout << "Usage: cd <dirpath>\n";
                continue;
            }
            arg1 = cmd_line[1];
            // check return value so everything is ok
            ret_val = filesystem.cd(arg1);
            if (ret_val) {
                std::cout << "Error: cd " << arg1;
                std::cout << " failed, error code " << ret_val << std::endl;
            }
        }

        else if (cmd == "pwd") {
            if (cmd_line.size() != 1) {
                std::cout << "Usage: pwd\n";
                continue;
            }
            // check return value so everything is ok
            ret_val = filesystem.pwd();
            if (ret_val) {
                std::cout << "Error: pwd failed, error code " << ret_val << std::endl;
            }
        }

        else if (cmd == "chmod") {
            if (cmd_line.size() != 3) {
                std::cout << "Usage: chmod <accessrights> <filepath>\n";
                continue;
            }
            arg1 = cmd_line[1];
            arg2 = cmd_line[2];
            // check return value so everything is ok
            ret_val = filesystem.chmod(arg1, arg2);
            if (ret_val) {
                std::cout << "Error: chmod " << arg1 << " " << arg2;
                std::cout << " failed, error code " << ret_val << std::endl;
            }
        }

        else if (cmd == "quit")
            running = false;

        else if (cmd == "help") {
            std::cout << "Available commands:\n";
            std::cout << "format, create, cat, ls, cp, mv, rm, append, mkdir, cd, pwd, chmod, help, quit\n";
        }

        else if (cmd == "") {
            ; // do nothing
        }

        else {
            std::cout << "Available commands:\n";
            std::cout << "format, create, cat, ls, cp, mv, rm, append, mkdir, cd, pwd, chmod, help, quit\n";
        }
    }
}
```

Listing 7. `fs.h`

```
#include <iostream>
#include <cstdint>
#include "disk.h"

#ifndef __FS_H__
#define __FS_H__

#define ROOT_BLOCK 0
#define FAT_BLOCK 1
#define FAT_FREE 0
#define FAT_EOF -1

#define TYPE_FILE 0
#define TYPE_DIR 1
#define READ 0x04
#define WRITE 0x02
#define EXECUTE 0x01

struct dir_entry {
    char file_name[56]; // name of the file / sub-directory
    uint32_t size; // size of the file in bytes
    uint16_t first_blk; // index in the FAT for the first block of the file
    uint8_t type; // directory (1) or file (0)
    uint8_t access_rights; // read (0x04), write (0x02), execute (0x01)
};

class FS {
private:
    Disk disk;
    // size of a FAT entry is 2 bytes
    int16_t fat[BLOCK_SIZE/2];

public:
    FS();
    ~FS();
    // formats the disk, i.e., creates an empty file system
    int format();
    // create <filepath> creates a new file on the disk, the data content is
    // written on the following rows (ended with an empty row)
    int create(std::string filepath);
    // cat <filepath> reads the content of a file and prints it on the screen
    int cat(std::string filepath);
    // ls lists the content in the current directory (files and sub-directories)
    int ls();

    // cp <sourcepath> <destpath> makes an exact copy of the file
    // <sourcepath> to a new file <destpath>
    int cp(std::string sourcepath, std::string destpath);
    // mv <sourcepath> <destpath> renames the file <sourcepath> to the name <destpath>,
    // or moves the file <sourcepath> to the directory <destpath> (if dest is a directory)
    int mv(std::string sourcepath, std::string destpath);
    // rm <filepath> removes / deletes the file <filepath>
    int rm(std::string filepath);
    // append <filepath1> <filepath2> appends the contents of file <filepath1> to
    // the end of file <filepath2>. The file <filepath1> is unchanged.
    int append(std::string filepath1, std::string filepath2);

    // mkdir <dirpath> creates a new sub-directory with the name <dirpath>
    // in the current directory
    int mkdir(std::string dirpath);
    // cd <dirpath> changes the current (working) directory to the directory named <dirpath>
    int cd(std::string dirpath);
    // pwd prints the full path, i.e., from the root directory, to the current
    // directory, including the current directory name
    int pwd();

    // chmod <accessrights> <filepath> changes the access rights for the
    // file <filepath> to <accessrights>.
```

```
    int chmod(std::string accessrights, std::string filepath);
};

#endif // __FS_H__
```

Listing 8. `fs.cpp`

```
#include <iostream>
#include "fs.h"

FS::FS()
{
    std::cout << "FS::FS()... Creating file system\n";
}

FS::~FS()
{

}

// formats the disk, i.e., creates an empty file system
int
FS::format()
{
    std::cout << "FS::format()\n";
    return 0;
}

// create <filepath> creates a new file on the disk, the data content is
// written on the following rows (ended with an empty row)
int
FS::create(std::string filepath)
{
    std::cout << "FS::create(" << filepath << ")\n";
    return 0;
}

// cat <filepath> reads the content of a file and prints it on the screen
int
FS::cat(std::string filepath)
{
    std::cout << "FS::cat(" << filepath << ")\n";
    return 0;
}

// ls lists the content in the currect directory (files and sub-directories)
int
FS::ls()
{
    std::cout << "FS::ls()\n";
    return 0;
}

// cp <sourcepath> <destpath> makes an exact copy of the file
// <sourcepath> to a new file <destpath>
int
FS::cp(std::string sourcepath, std::string destpath)
{
    std::cout << "FS::cp(" << sourcepath << "," << destpath << ")\n";
    return 0;
}

// mv <sourcepath> <destpath> renames the file <sourcepath> to the name <destpath>,
// or moves the file <sourcepath> to the directory <destpath> (if dest is a directory)
int
FS::mv(std::string sourcepath, std::string destpath)
```

```
{
    std::cout << "FS::mv(" << sourcepath << "," << destpath << ")\n";
    return 0;
}

// rm <filepath> removes / deletes the file <filepath>
int
FS::rm(std::string filepath)
{
    std::cout << "FS::rm(" << filepath << ")\n";
    return 0;
}

// append <filepath1> <filepath2> appends the contents of file <filepath1> to
// the end of file <filepath2>. The file <filepath1> is unchanged.
int
FS::append(std::string filepath1, std::string filepath2)
{
    std::cout << "FS::append(" << filepath1 << "," << filepath2 << ")\n";
    return 0;
}

// mkdir <dirpath> creates a new sub−directory with the name <dirpath>
// in the current directory
int
FS::mkdir(std::string dirpath)
{
    std::cout << "FS::mkdir(" << dirpath << ")\n";
    return 0;
}

// cd <dirpath> changes the current (working) directory to the directory named <dirpath>
int
FS::cd(std::string dirpath)
{
    std::cout << "FS::cd(" << dirpath << ")\n";
    return 0;
}

// pwd prints the full path, i.e., from the root directory, to the current
// directory, including the currect directory name
int
FS::pwd()
{
    std::cout << "FS::pwd()\n";
    return 0;
}

// chmod <accessrights> <filepath> changes the access rights for the
// file <filepath> to <accessrights>.
int
FS::chmod(std::string accessrights, std::string filepath)
{
    std::cout << "FS::chmod(" << accessrights << "," << filepath << ")\n";
    return 0;
}
```