project for KIV/ZOS

A simple i-node based file system

Jakub Šilhavý
A17B032P
silhavyj@students.zcu.cz

12. 10. 2020

Contents

1 Assignment description

The assignment is all about designing and implementing a simple file system based on i-nodes. The entire document describing all the functionality of the system should provide is available on Courseware - `https://courseware.zcu.cz/portal/studium/courseware/kiv/zos/samostatna-prace.html`. Essentially, the system should allow the user to execute simple commands that are available on almost any operating system. Depending on the operating system, the syntax of the commands may differ, but their purpose remains the same. A list of all of them is shown bellow.

Table 1.1: List of all the commands

Shortcut	Example	Description
cp	`cp s1 s2`	copies file in location s1 into location s2
mv	`mv s1 s2`	moves file in location s1 into location s2
rm	`rm s1`	deletes file s1
mkdir	`mkdir a1`	creates a new folder a1
rmdir	`rmdir a1`	deletes folder a1
ls	`ls a1`	prints out the content of folder a1
cat	`cat s1`	prints out the content of file s1
cd	`cd a1`	changes the current directory to location a1
pwd	`pwd`	prints out the current location (absolute path)
info	`info a1/s1`	prints out information about the i-node
incp	`incp s1 s2`	imports a file from the HDD (path s1) into location s2 within the file system
outcp	`outcp s1 s2`	exports file s1 from the file system on to the HDD (path s2)
load	`load s1`	loads file s1 from the HDD containing commands to execute
format	`format 600MB`	formats the file system with a new size

Additionally, each student was assigned an extra feature they were supposed to implement within this project. As far as mine is concerned, I had to implement symbolic links (`slink s1 s2`), meaning the user can create a symbolic link that points at a file within the file system itself. Some more information on symbolic links can be found at `https://en.wikipedia.org/wiki/Symbolic_link`

2 Analysis

2.1 The structure of the virtual disk

As the project is supposed to be a virtual file system, it is essential to realize what the storage of the file system looks like. Basically, it is a binary file of a pre-defined size that is stored somewhere on the user's physical disk (HDD) and contains all the necessary information about the file system as well as the data itself. As shown down below, the disk is consist of four different sections - **superblock** (2.2), **bitmap** (2.3), **i-nodes** (2.4), and **clusters** (2.5).

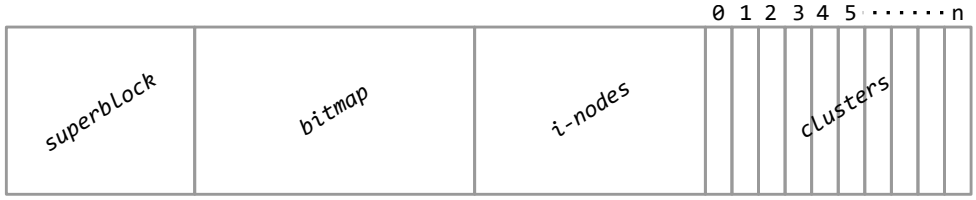

Figure 2.1: The structure of the disk (the storage of the file system)

2.2 Superblock

The first block in the file is called a *Superblock*. A superblock holds all the information about the file system that is needed when the system boots up. The superblock is the main section where the program can find out who the owner of the file system is, the total number of clusters, the size of a cluster, or for example, a short description of the file system. Most importantly, it also holds all the start addresses of the next sections.

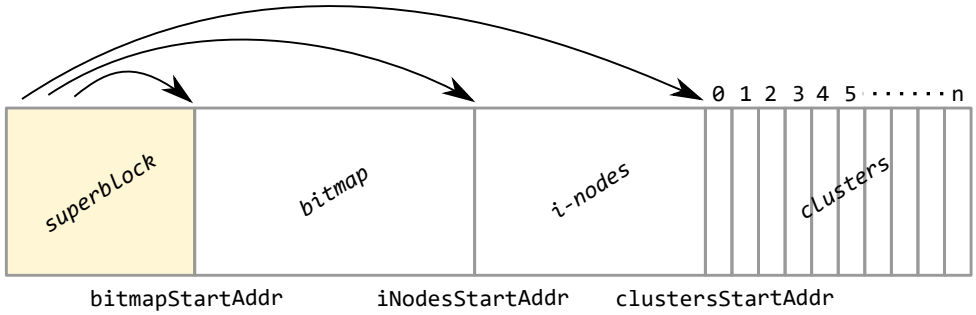

Figure 2.2: The start addresses of the next sections stored in the superblock

2.3 Bitmap

The purpose of the bitmap is to keep track of which clusters are free and can be used to store a file or a folder, and which are currently occupied. The system takes advantage of the constant look-up time this structure provides when finding out whether or not a cluster is free. It is nothing but an array of values where each value is either `true` or `false`. The size of the array is unknown, meaning it is a variable that needs to be worked out from the size of the disk. The way it is done is explained in section 2.5.

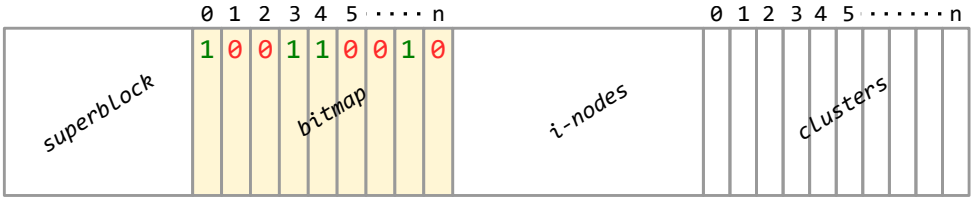

Figure 2.3: Bitmap in the file system

2.4 I-Nodes

In an i-node-based file system, as well as some other UNIX-based file systems, folders are treated as files. This gives us the advantage of using only one data structure for both entities. An i-node holds various information about the file/folder such as permissions, size, the date-time when it was created, etc. However, since an i-node is a tiny-sized block, it does not contain directly the content of the file. Instead, it holds indexes of the clusters that physically make up the content of the file. Surprisingly, the name of the file/folder is not stored in the i-node itself either. This is taken care of using `DirectoryItems`(2.6).

2.4.1 Direct pointers

There are three types of pointers to clusters in an i-node. The first type is called direct pointers and they directly point at the clusters that make up the content of the file. This comes in handy for files of a relatively small size.

2.4.2 First indirect pointer

Within this project, I refer to the second type of pointers of an i-node as the first indirect pointer. In this case, the pointer holds an index of a cluster that

contains indexes of the clusters that make up the file. This quickly increases the number of clusters we can use to store a file if the direct pointers are not enough.

2.4.3 Second indirect pointer

As the first indirect pointer represents one "jump" over a cluster to get to the final clusters that make up the file, the second indirect pointer takes this whole idea up to another level, meaning there are two "jumps".

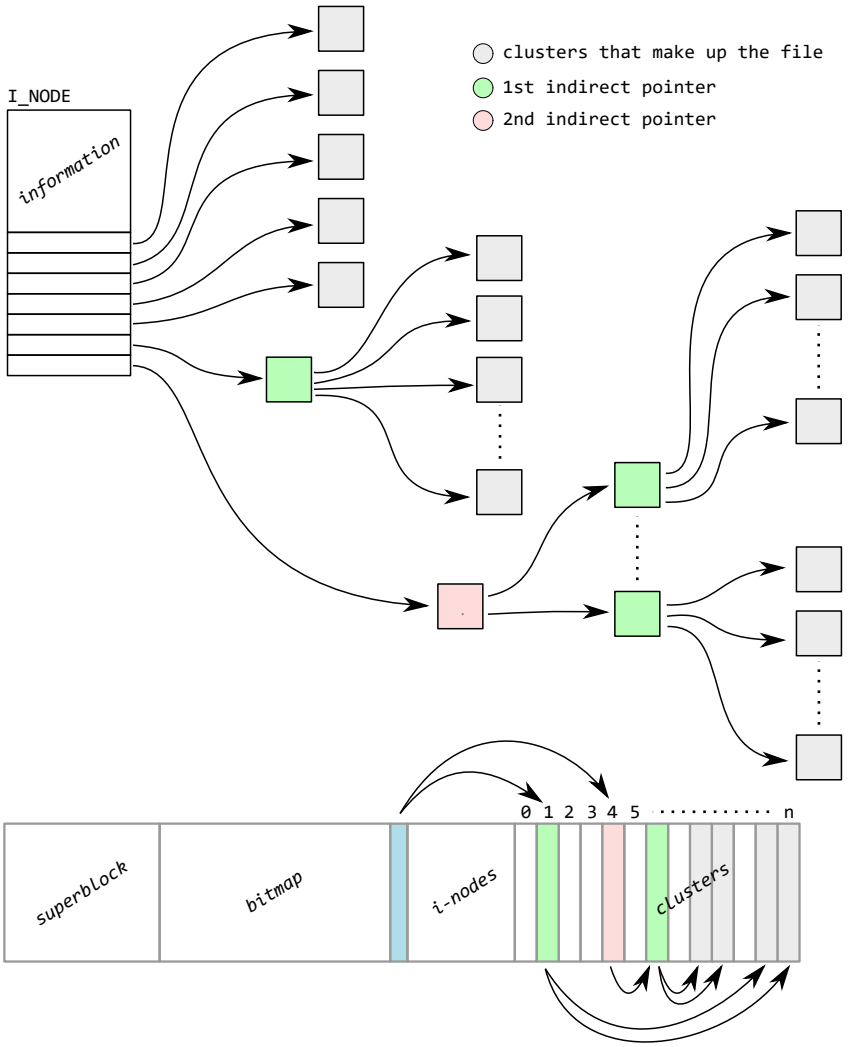

Figure 2.4: An i-node representation

To clear this up, there is a picture above visualizing a simple concept of an i-node along with its representation in terms of the structure of the disk.

2.5 Clusters

The rest of the disk is meant to be used for clusters. A cluster is a fixed-sized block used to store a file or a folder. Whether a cluster is free or occupied could be found using the bitmap (2.3). As the user has the option of formatting the file system with a size they wish, the number of clusters changes along with that. Therefore, we need to work out the number of clusters manually whenever the size of the disk changes. Fortunately, the number of clusters is the only variable we need to work out. The rest of the values is known as it is shown in the picture below.

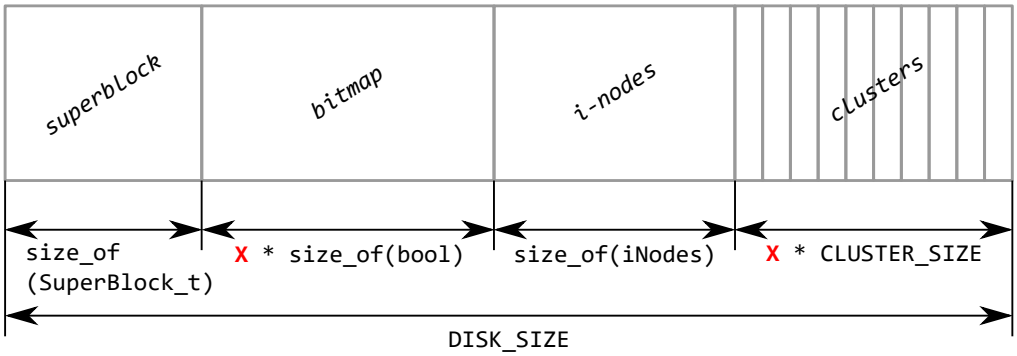

Figure 2.5: The process of working out the number of clusters

The reason why we need to work it out this way is that we can not simply do `DISK_SIZE / NUMBER_OF_CLUSTER` as there are also other sections partially occupying the disk.

From looking at the picture above, we can come up with a simple formula that gives us the value of `X`, which is the total number of clusters in the file system. All the values, such as the number of i-node or the size of a superblock, can be found in the source file called `Setup.h`.

2.5.1 Storing a file

When reading a file from the user's physical disk, it needs to be read as a binary file block by block, where each of the blocks has a size equal to `CLUSTER_SIZE`. The program then finds free clusters and copies the content of the file into them. Finally, all the clusters that make up the file will be attach to the appropriate i-node.

2.6 Directory items

This structure represents a folder containing items where each item can be either a file or another directory. As mentioned in section 2.4, the folders are treated as files, meaning they also have their own i-nodes. In order to distinguish whether an i-node represents a directory or a file, we can add another attribute to the i-node structure which can be either set to `true` or `false`. The entire structure is made up by the number of items in the directory and the items themselves. Each item holds a name and an id of i-node that belongs to it.

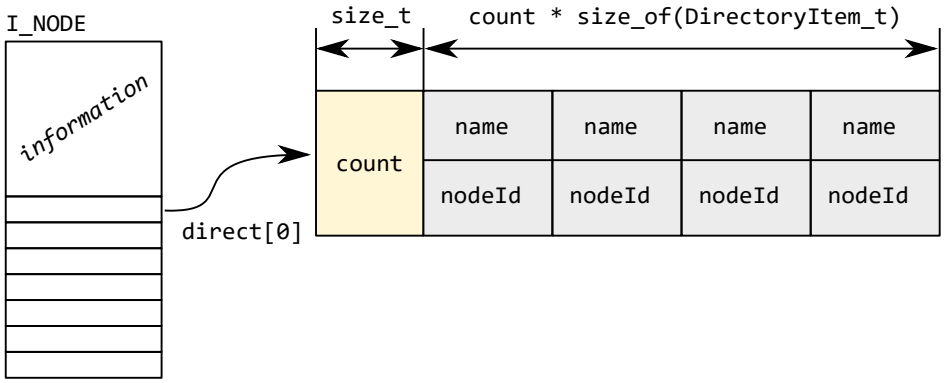

Figure 2.6: The way a forlder is sotred in the file system

When we want to access a directory, first, we need to get the i-node belonging to it, then need to check out whether it is a file or a directory and treat it accordingly. In the case of directory, the number of directory items is stored as the first block in the cluster the first indirect pointer points at.

2.6.1 Root directory

When the program starts, the root directory is the entry point of the hierarchy of the file system. Therefore, the location of the root directory must be known. The root i-node is always the first i-node. This way, the program knows where to start when reading some other directory if the user requires.

3 Implementation

3.1 Programming language

As required, the whole program was implemented using programming language `C/C++`. Particularly, I decided to use `C++14`.

3.2 UML diagram

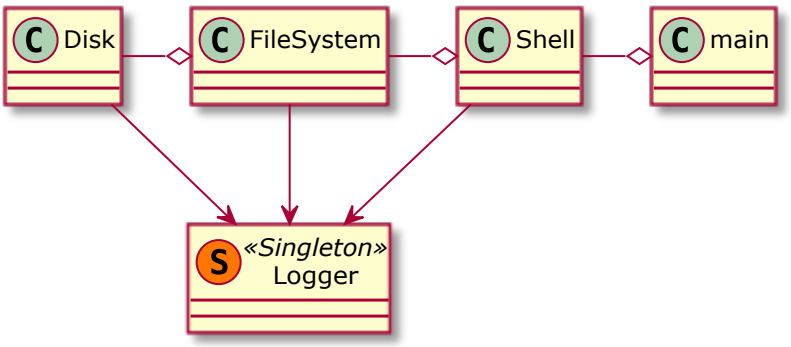

Figure 3.1: UML diagram

The UML diagram is more or less self explaining. Class called `Disk` provides all the operations that can be performed on the file system such as copying files, collecting clusters etc. However, this class is situated too low in terms of abstractness of this project. Therefore, there is another class called `FileSystem` that works as a wrapper for class `Disk`, meaning it uses only the functionality necessary to execute commands without all the ancillary methods that are called down in class `Disk`. Finally, there is class called `Shell` that deals with everything the user types down. This class works as an interface between the user and the rest of the system. Furthermore, there is class called `Shell` that was widely used in the process of debugging.

A more detailed explanation of how the classes are implemented can be found in the documentation generated by Doxygen (`https://www.doxygen.nl/index.html`) located in the following directory `doc/doxygen/index.html`.

4 Compilation and execution

4.1 Compilation

From the user's point of view, they do not have to care about anything as all the compilation and linking rules are defined in the **`Makefile`**. If they want to compile all the source files in order to get an executable file, they need to go into folder called **`app`** where the **`Makefile`** is stored. Once in the folder, all they need to do is run command **`make`** from the terminal. Upon successful compilation, there will be an executable file created along with folder called **`bin`** containing all the binary files.

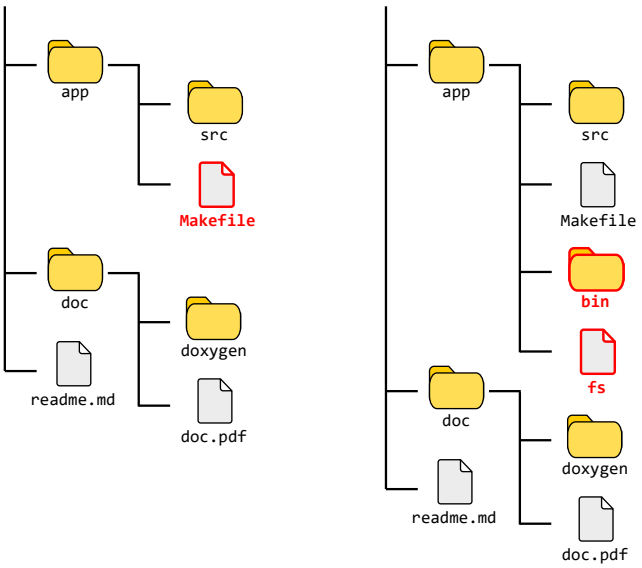

Figure 4.1: Root folder before and after compilation

4.2 Execution

Once successfully compiled, the user can now run the program executing command **`./fs`** in the same folder.

4.3 Clean

Also, there is a function called **`clean`** in the **`Makefile`** in case the user wants to delete all the files and folders created by the compilation process. If they want to do so, all they need to do is run the following command from terminal **`make clean`**.

5 User manual

5.1 Start of the application

When running the command to start the application, the user is supposed to enter one parameter, which is the name of the file system. For example, "data.dat" or "storage.dat". If they do not do so, an error message will be printed out as you can see down below.

```
kuba@LenovoT540p:/mnt/d/ZOS_A17B0362P_silhavyj$ ./fs
You are supposed to run the program with one parameter, which is the name
of the file system (e.g. data.dat).
kuba@LenovoT540p:/mnt/d/ZOS_A17B0362P_silhavyj$
```

Figure 5.1: The user forgot to enter the name of the file system

Once they do enter a name of the file system, the program can start two different ways. If the file already exists, the program will attempt to load it and restore all the data that it contains. The other way is that the file does not exist yet. In that case, the program will create an empty file of the pre-defined size, which can be found in the source file called `Setup.h`.

```
kuba@LenovoT540p:/mnt/d/ZOS_A17B0362P_silhavyj$ ./fs data.dat
FORMATTING DISK (50000000B)
OK
/>
```

Figure 5.2: The program creates a new storage file

```
kuba@LenovoT540p:/mnt/d/ZOS_A17B0362P_silhavyj$ ./fs data.dat
/>
```

Figure 5.3: The program loads up data from the storage file

5.2 Help

In case the user does not remember a command they want to use, they can use the help menu of the application by running command **`help`**. As soon as they do so, all the commands with their descriptions will be printed out.

```
kuba@LenovoT540p:/mnt/d/ZOS_A17B0362P_silhavyj$ ./fs data.dat
/> help
cat s1          - prints out the content of file s1
cd a1           - changes the current path into folder a1
cp s1 s2        - copies file s1 into file s2
exit            - closes the application
format 600MB    - formats the file given as a parameter
help            - prints out help
incp s1 s2      - load file s1 into the file system (directory s2)
info a1/s1      - prints out information about the i-node
load s1         - loads commands stored in file s1 and executes them
ls a1           - prints out the content of folder a1
mkdir a1        - creates a new folder a1
mv s1 s2        - moves file s1 into file s2
outcp s1 s2     - exports file s1 out onto the physical disk (directory s2)
pwd             - prints out the current path
rm s1           - removes file s1
rmdir a1        - removes folder a1
slink s1 s2     - creates a symbolic link s2 pointing at file s1
/>
```

Figure 5.4: Printing out the help of the program

5.3 Command execution

The user can run all the commands mentioned in the Assignment section (1). They should also get feedback as they execute individual commands. If they run a command and everything goes as expected, the program will print out `OK` as an acknowledgment that everything went well. On the other hand, if they, for example, try to import a file using command `incp`, and the file happens to be too big for the file system, the program will print out an error message informing the user about what just happened.

```
kuba@LenovoT540p:/mnt/d/ZOS_A17B0362P_silhavyj$ ./fs data.dat
/> format 12KB
FORMATTING DISK (12000B)
OK
/> incp test_file.zip
[0454][ERROR] There's not enough free clusters in the file system
/>
```

Figure 5.5: Printing out a feedback message for the user

6 Conclusion

The application was tested using various types of files such as `.pdf`, `.docx`, `.zip`, `.txt`, `.png`, and so on. Beside trivial operations such as `mkdir` or `cd`, the testing process mainly focused on moving the files around the file system. I tried copying files, moving files, and exporting files using command `outcp` in order to find out whether the content as well as the size of the exported file is the same as the original file. Everything seemed to work just fine.