Lab 3
File Systems

Jonathan Dell'Ova

August 19, 2022

Contents

1 Introduction

In this report I will go through my implementation of creating a **FAT filesystem** on a simulated disk. I will go through the most essential parts of the program and what the thought process behind those were. These chapters will also briefly discuss how the code was constructed in those relevant parts.

2 The project

2.1 Helper Functions

A conscious choice was made to have several helper functions that all of the public functions should derive from. This meant that every time a certain *concept* was noticed, such as making a FAT entry or writing contents of a file to a string object, a new function was created for that purpose as these would most certainly be used several times in the program. This did not only make the code easier to read but also easier to manage. This helped with reducing variety in implementation for the same structures which made changes to the code easier to accomplish.

2.2 Filepaths

It was decided that filepaths were internally going to be represented as a vector of strings (and will continue to be throughout this report) where each element would represented the file name of an existing directory entry (dir_entry). Each filepath could also have a **path type** evaluated which was stored as an enum. This path type could either be root, relative, absolute or invalid. With this combination of information any type of directory entry could be found no matter where the users current working directory (CWD) is, which is exactly what is wanted from a path.

The decision to have a vector of strings is that it is easy to access each element of a vector and it is also easy to modify the path. Most notably, it was often seen that both the file at the end of a filepath and its parent directory should be accessed. In these cases the last element is always the name of the file and every element *but* the last is the directory the file should exist in. Having a filepath as a string vector also helped when validating filenames as the same validation function could be ran on each element.

2.3 Error handling

This project is prone to many errors, mostly to do with writing or reading from the wrong blocks or using indexes into the FAT which are out of bound. The codebase is constructed in such a way that easy validation checks are made first, such as checking if a path exists when copying or if a certain string is a valid filename.

Sometimes an action has to be taken that changes the filesystem in some way to be able to proceed with the next step of a function. This makes error handling difficult because if any error occurs after a change has happened, the filesystem might become corrupted or cause unintended consequences.

The code was therefore consciously constructed in a way that most of the functionality that did not make any changes was done *before* making any changes. Throughout development it was noticed that by constructing the code in a certain way, several guarantees could be made for the arguments to future function calls because they would have already caused errors at previous points. This does not absolve the risk of errors though as there could be some undefined behaviour that was not properly accounted for or not seen.

This could be fixed by having a section for cleanup if anything goes wrong in a function to restore changes that were made but this proved to be a difficult problem to handle and was not explicitly part of the assignment and was therefore not implemented. Only the most common errors where therefore checked and handled in the final implementation.

2.3.1 Guard Clauses

Guard clauses where used extensively in this project and were more naturally developed as part of the code than decided upon from the beginning. This is because there are several sources of errors that can occur in this program and each have to be checked before being allowed to proceed. These error checks could be handled through nested if-statements but when these error checks appear more than a couple of times in each public function call it can become very messy. It then made sense to return an early error code if something went wrong during the program which made the structure of the code much cleaner.

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

    if(dirpath == "/")
    {
        m_cwdBlock = ROOT_BLOCK;
        return 0;
    }

    if(!FilenamesAreValid(dirpath)) { return ERROR_CODE; }

    dir_entry newCWD;
    GetDirEntry(ParseDirPath(dirpath), newCWD);
    if(!DirEntryExists(newCWD) || newCWD.type != TYPE_DIR) { return ERROR_CODE; }

    m_cwdBlock = newCWD.first_blk;

    return 0;
}
```

Figure 1: Code snippet of *cd()* function.

In figure 1 an example is shown of how guard clauses was used in the project and is very similar in all other parts of the code. Some check will be made and return error immediately if a undesirable result was to occur.

2.4 Removal of files

Two ways were thought of on how to deal with removed files. Firstly was that when a file is to be *removed*, the program simply marks the FAT in a way so that it should be regarded as a *dirty* but free block. This means that when a new block needs to be allocated (for any reason) then it first checks if it is free and then clears it if its dirty before finally writing to the block. This makes deletion incredibly fast but additional information and rules needs to be implemented for the program to handle this situation.

The second solution was the easiest and is the one that was actually implemented which is to simply clear the block and mark it as free on the FAT without any special *dirty* flags. This makes it so removal of files are slower but any time a new block needs to be allocated, no additional checks has to be done and handled.

The second solution was chosen because it did not require any implementation of special cases when allocating files. It also did not need to define how a dirty but free block should be flagged or searched for.

3 Summary

This was a moderately big project which required much time to be spent on thinking about the structure of the code and making definitions before even starting to write it. Making these decisions helped the development of the project immensely both in writing and understanding the code but also in debugging and solving problems.

This was mostly due to having many helper functions with names that matched their high cohesion so that the flow of execution was easy to follow throughout each function. Also having guard clauses to handle errors made the code less cluttered than what it could have been which further increased readability. Better implementations of cleaning up when errors caused could be implemented but would require some restructuring to each core function to work properly.