# Project Report
# Group Ravioli

Java and C# in depth, Spring 2014

Michael Bang (mbang@student.ethz.ch)
Pascal Fischli (fischlip@student.ethz.ch)
Vladimir Grozman (grozmanv@student.ethz.ch)

April 29, 2014

## 1 Introduction

This document describes the design and implementation of the *Personal Virtual File System* of group *Ravioli*. The project is part of the course *Java and C# in depth* at ETH Zurich. The following sections describe each project phase, listing the requirements that were implemented and the design decisions taken. The last section describes a use case of using the *Personal Virtual File System*.

## 2 VFS Core

VFS Core is the first step towards a Personal Virtual File System. It operates on virtual disks, of which each one is stored in a single file in the host file system. Its API not only offers functionalities to create, mount and delete virtual disks, but also to operate inside opened disks. This ranges from basic console operations, like navigating through directories, renaming and removing, to the virtual disk operations of importing and exporting directories, files or both together.
The most important task in the process of creating the VFS Core is the design of the storage structure in the virtual disk files. The main aspect in this is efficiency. Not mainly in speed, but in usage of the available space. Two basic approaches are storing the files in one block, which of course could result in lots of space lost due to fragmentation, or splitting files into parts

of predefined size that are always linking to the next part, resulting in a very space efficient but most likely slower system.

## 2.1 Requirements

In this part we indicate and describe what requirements we have implemented. We also list the main software elements involved in the respective implementation.

1. The virtual disk is stored in a single file in the host file system. The creation happens in the creators of `JCDFAT`, which are creating a FileStream for the new file that is then written to in the creators of `JCDFAT`.

2. The creation of virtual disk files happens in method `Create(string hfsPath, ulong size)` of class `JCDFAT`. The parameters specify the location and the initial size.

3. Several virtual file systems on the host are allowed. Creation and opening, which is done with method `Open(string hfsPath)` of class `JCDFAT`, can happen an unlimited amount of times and with the file at any place on the host file system.

4. Virtual disks can be disposed from the host file system, which can be done through the method `Delete(string hfsPath)` of class `JCDFAT`. Only files containing a virtual disk that can be opened are disposable through this.

5. On a mounted VFS files and directories can be created, deleted and renamed with the methods `CreateFile(ulong size, string path, bool isFolder)`, `DeleteFile(string path, bool recursive)` and `RenameFile(string vfsPath, string newName)` in class `JCDFAT`.

6. For basic navigation functionalities going to a specific location can be done with method `SetCurrentDirectory(string path)` in class `JCDFAT`. To support that, listing of files and directories can be done with the method `ListDirectory(string vfsPath)` in class `JCDFAT`, which is called by the method with the same signature in class `JCDFAT`.

7. Moving of files and directories can be done with method `MoveFile(string vfsPath, string newVfsPath)` and copying with method `CopyFile(string vfsPath, string newVfsPath)` in `JCDFAT`. Both preserve the internal file/directory hierarchy.

8. Files and directories can be imported from the host into the virtual file system. This is done through method `ImportFile(string hfsPath, string vfsPath)` in `JCDFAT` which is then calling `ImportFolder(string hfsFolderPath, string vfsPath)` or `ImportFile(Stream file, string path, string fileName)` in `JCDFAT`.

9. Exporting of files and directories from the virtual to the host file system is done in methods `ExportFile(Stream outputFile, JCDFile file)` and `ExportFolderRecursive(JCDFolder folder, string hfsPath)` in `JCDFAT`.

10. How much space is free in the virtual disk can be queried and is calculated in method `GetFreeSpace()` in `JCDFAT`, which is multiplying the number of free blocks times the block size. The occupied space can be queried too and is calculated in method `OccupiedSpace()` in `JCDFAT` which is subtracting the free space from the virtual disk size.

### 2.1.1 Bonus points

We've implemented two of the bonus features of this milestone. The first one is "Elastic disk". Our virtual disk initially takes up only the space required by the meta data (and FAT), and hereafter expands itself as needed when importing files. We currently have an untested implementation of automatic shrinking of the virtual disk.
The second bonus feature we've implemented is "Large data". This feature is implemented by using buffers of a predefined, constant size, when reading/writing files to/from the file system, instead of having the whole VFS in memory. The current implementation uses quite big buffers (20 MB), trying to mitigate the problem of constantly reading and writing from different parts of the harddisk.

## 2.2 Design

We decided to take a lot of inspiration from the FAT file system [1], also naming our file system JCDFAT. The overall structure of JCDFAT can be seen on Figure 1.

The smallest unit of allocation in JCDFAT is one block, which currently is $2^{12}$ bytes (4KB).

[1] http://en.wikipedia.org/wiki/File_Allocation_Table

The first block of the file system contains meta data (see Figure 2). Even though the meta data currently only is 28 bytes, it has a full block allocated to it, as described above.

The next block(s) contain(s) the File Allocation Table (FAT). Depending on how large the file system is, the FAT will span one or more blocks. Since we're using 32 bit integers, and blocks of size 4KB, each FAT block allows us to address 4MB. This means that the smallest JCDFAT file system allowed is 4MB. It also means that the largest JCDFAT file system possible theoretically is 16 TB, but due to an implementation detail, it currently is 2 TB. The size of the FAT is *included* in the size of the file system. The FAT takes up around 1/1024th of the file system; this means that if the file system is 16 TB, the FAT will be 16 GB.
There is no bound on the size of individual files, except the size of the file system.

The two blocks following the FAT are reserved for the root directory and the search file. The root directory is the root folder of the file system. The search file is the file in which we wish to store meta data for indexing the file system, allowing us to implement file search in milestone 2 of the project.

```
                File system structure
          (‘|‘ represents a block boundary)

| Meta data | FAT block(s) | Root directory | Search file |
| data | data | data | data | data | data | data |  data  |
                           . . .
| data | data | data | data | data | data | data |  data  |
```

Figure 1: Block structure of the JCDFAT file system.

We use a structure called JCDDirEntry (see Figure 3) to represent files on disk. Each directory entry is 256 bytes, meaning that we can have $\frac{2^{12}}{2^8} = 16$ entries in each block. Since the smallest unit of allocation is 4 KB, a folder takes up at least 4 KB of space. More generally, a folder takes up ceil(entries/16) $* 2^{12}$ bytes.

As can be seen on Figure 3, the maximum length of a file name is 240 bytes. The string is interpreted as unicode, though, which means that the length of

Meta data

| | Size |
|---|---|
| Magic number | 4B |
| Block size in power-of-2 bytes | 4B |
| Number of FAT blocks | 4B |
| Free blocks (without expanding) | 4B |
| First free block | 4B |
| Root directory block | 4B |
| Search file block | 4B |

Total size: 28 B

Figure 2: Meta data for the JCDFAT file system.

the file name is at most 120 characters.

## 2.3 File system events

The following is an explanation of how the implementation of our file system behaves in different scenarios.

In the following we assume that there is always enough free space, and that the source and destination files and folders exist/don't exist as required.

### 2.3.1 Finding a specific folder or file

- Starting at the root folder, recursively identify the directory entry of the next folder specified by the given path.

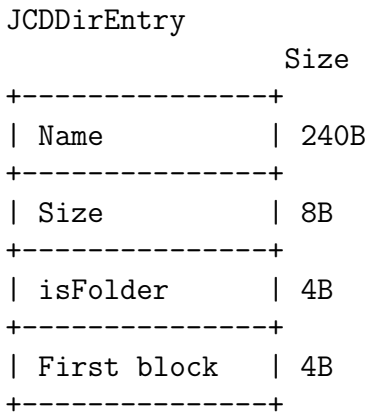


Figure 3: Structure of a directory entry stored on disk in JCDFAT.

- Once the parent folder of the specified file or folder has been found, find the specified file or folder.

### 2.3.2 Allocating space for a file

- Figure out how many blocks the file requires, $ceil(\frac{\text{size in bytes}}{2^{12}})$.
- Starting from the first free block, walk the FAT in increments of 1, chaining free blocks.
- Mark the last used block as 'end of chain'.

### 2.3.3 Creating a file or folder

- Allocate enough blocks to store the file. This is done by finding free entries in the FAT and chaining them.
- Store the file in the newly allocated blocks.
- Write a directory entry in the destination folder.

### 2.3.4 Deleting a file (or folder)

- (Loop over all files/folders and perform delete file.)
- Starting from the first block of the file, walk the FAT chain and delete all entries.
- Delete the directory entry from the parent folder.

#### 2.3.5 Moving a file or folder

- Move the file entry from the source folder to the destination folder. (File contents are not actually moved.)

#### 2.3.6 Renaming a file or folder

- Rewrite the name in the file's directory entry.

### 2.4 Unit Tests

After having defined the API and prior to implementing its functionalities we have created a set of unit tests. For each method in the interface we have identified the "normal", and also the possible "wrong" use cases and the expected reaction of the system to them. Each such use case resulted in an own test, which can be run independently of the others. The test framework that we have used is MSTest.
Since we are running the tests on the same files that are created and then deleted for each test run, our tests cannot be run or analysed for code coverage in parallel.

## 3 VFS Browser

In this part of the project we're creating one or several user interfaces that are more sophisticated than the console client from the previous part, which means Graphical User Interfaces (GUI), for the VFS core. We decided to create a Windows desktop client and a web application written in ASP.NET. The browsers support both mouse and keyboard navigation.
The clients support all of the previously created operations of the VFS core and additionally a functionality to search for file/directory names with the options search location and case sensitivity.

### 3.1 Requirements

In this part we indicate and describe what requirements we have implemented. We also list the main software elements involved in the respective implementation. When requirements are described for both browser clients separately, we do so first for the desktop (a) and then the web client (b).

1. The platforms we have created our browsers for are the Windows desktop and web (ASP.NET).

2. (a) All operations from VFS core are supported, some through buttons, especially operations on VFSs, but most through the list view, which is the main component of the browser, and its menu. The event handling is done mostly in MainForm.cs and the other Form classes.

   (b) TODO web

3. (a) In the list view single and multiple items can be selected. The operations where this multiple selection makes sense are then run on all those items.

   (b) TODO web

4. (a) Keyboard navigation in the application is done mostly through the TAB key. In the list view going to child folders is done through pressing ENTER when a folder is selected. Going back can be done trough the Back-Button or the BACKSPACE key. Common Windows key functionalities (e.g. Delete)as well as combinations (e.g. Ctrl+C, Ctrl+X, Ctrl+V, Ctrl+A) are all supported.

   (b) TODO web

5. (a) Mouse navigation is supported in the expected way. In the list view double click on a folder means going into this folder. Going to the parent folder can be done by clicking on the Back-Button.

   (b) TODO web

6. (a) Search can be done by entering the key word into the text box at the top right corner and then pressing ENTER. Selection of the search options can be done in the menu that appears when a click on the v-Button besides the text box happens.

   (b) TODO web

### 3.1.1 Bonus points

As a general bonus feature we have implemented browsers for two different platforms. The platform specific implemented bonus features are:

(a) In the desktop client we have implemented especially nice-to-have features like drag-and-drop. Inside of the list view, dragging with the left mouse button results in a move and with the right mouse button in a copy operation. Dragging from the Windows Explorer and dropping into the list view results in an Import. Dragging an item of the list view and dropping it outside of this view results in an Export.

(b) TODO web

## 3.2 Design

### 3.2.1 Search functionality

In implementing search functionality on a file system, there are some decisions to be made. There are many different data structures to use, and different ways to use each data structure. In this project, we decided to optimize our implementation for full file-system search, i.e. looking for a file name throughout the whole file system. There, of course, are trade-offs being made when making decisions like the one just mentioned; optimizing for full-file system search has an impact on the time it takes to search through only a small subset of folders of the file system. In our implementation, it requires (in the order of) the same amount of operations to search through the entire file system as it does to search through only a single folder. This is a fact that we were aware of when making the decision, and found to be acceptable. It would be possible to have multiple data structures, each optimizing for a different type of search, but we decided to go with just one. Having multiple data structures adds much more complexity and overhead, both w.r.t. processing power and storage.
We found that B+-trees are a perfect match to implement an efficient full-file system search. Thus, our implementation relies on this data structure. We did not implement B+-trees ourselves, but sought help from the open source community, where we found an implementation we could use called 'bplusdotnet'. The code for this project can be found at `https://github.com/benaston/BTree`.

The implementation of B-trees we use, 'bplusdotnet', uses two files for storage; one for storing the data structure and one for storing the actual data we put in the nodes of the tree. We decided to store these two files in the virtual file system itself, in order to comply with the requirement that the virtual file system should span only one file on the host file system. This means that additional storage space is required for storing the B-tree data structure on the virtual file system, and makes it more difficult to predict exactly how much space is required when adding a new file. This has the implication that, even though there may be X bytes of free space on the drive, it might not be possible to import a file of X bytes because of the metadata that is also written to the file system.

The implementation of the search functionality almost solely resides in the

namespace 'vfs.core.indexing', which wraps the implementation of B+-trees we use. This implementation is described further in Section 3.3 because this mostly consisted of integration work.

### 3.2.2 Desktop client

The most notable design decision was to make a clean separation between the View part, where event handling from the MainForm is done, and the Control part that is accessing the VFS core functionalities. In order to do this, we have defined a class VFSSession.cs which is the only class in the client that is referencing vfs.core. It contains a JCDFAT object, which represents an open VFS, and mostly calls methods of this object to fulfill its tasks. On the other hand, no Forms functionality is used in it, which makes this class easily reusable for other clients.

### 3.2.3 Web client

TODO

## 3.3 Integration

In order to integrate file search into JCDFAT, all we had to do was to add and maintain nodes in the B-tree when importing, renaming, moving, and deleting files on the virtual file system. These changes are not very interesting to comment on here, and make up only around a total of 30 lines of code in the classes JCDFAT, JCDFile, and JCDFolder. The only real integration work we had to do in this milestone, was to integrate the open source project 'bplusdotnet' into our project.

We had three obstacles to overcome during the integration of bplusdotnet into our project:

- bplusdotnet doesn't provide support for having multiple keys with the same value.
- bplusdotnet to writes its data to the host file system.
- bplusdotnet doesn't allow for case insensitive search.

In the following we describe how we overcame these three obstacles.

In solving the first obstacle, we started out with implementing a wrapper

(vfs.core.indexing.FileIndex) for the class SerializedTree of bplusdotnet. SerializedTree implements a B+-tree which maps unicode keys (which our file names are) to serializable objects. As already mentioned, SerializedTree doesn't support having multiple keys with the same value. But it does support using serializable objects as values, and since arrays of serializable objects are also serializable, our solution was to create a serializable class (vfs.core.indexing.IndexedFile) and use an array of this class as values. Our wrapper class (FileIndex) then implements the logic required to add and remove items from the array used as value in the B-tree.

Since bplusdotnet uses instances of Stream for writing data to disk, we figured that the correct way to solve our second problem was to create a class which implements the Stream interface on our virtual file system. This class can be found in vfs.core.JCDFileStream. After having implemented JCDFileStream, we realized that it might also be useful for the 3rd milestone of the project, and so were even more happy with our solution.

Even though the third obstacle wasn't the most difficult one to overcome, it probably was the most time consuming. Here, we decided to pull up our sleeves and give something back to the open source community; we went through the codebase of bplusdotnet and added support for case insensitive key comparisons. Since these modifications don't really have anything to do with our own project, this will not be described

# 5 Quick Start Guide

To facilitate debugging we have created a console client. The available commands are described in this section.

1. Can be called all the time:
    - help: Shows the help text with the available commands.
    - exit/quit: Exit the client.
2. Can be called only when NOT mounted:
    - create path size: Create a new VFS file at the given path and with the given size.
    - delete path: Delete the VFS file at the given location.
    - open path: Open/mount the VFS file at the given location.

3. Can be called only when a VFS is mounted:

    - close: Close the mounted VFS.
    - ls [path]: List the files/directories in the current or in the given directory.
    - cd path: Change to the given directory.
    - rm [-r] path: Remove the given file/directory (recursively with -r).
    - mk [-p] path size: Make a new file (and parents with -p) of the given size.
    - mkdir [-p] path: Make a new directory (and parents with -p).
    - cp source target: Copy the source to the target, both on the VFS.
    - mv -hv/-vh/-vv source target: Move the source to the target. In the first parameter the first letter indicates the source and the second the target file system. h stands for host and v for the virtual. Therefore -hv means import, -vh export and -vv inside the VFS.
    - rn path newName: Rename the file/directory to the given new name.
    - size: Show the size of the hole VFS file.
    - free: Show the free space.
    - occupied: Show the occupied space.
    - search [-i] [-a] [-n] filename: Search current directory for the given file (case insensitive with -i, -a search from root, -n non recursive search)