Project Report
Group Ravioli

Java and C# in depth, Spring 2014

Michal Bang
mbang@student.ethz.ch
Pascal Fischli
fischlip@student.ethz.ch
Vladimir Grozman
grozmanv@student.ethz.ch

April 7, 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 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, both on 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

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).

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.

We use a structure called JCDDirEntry (see Figure 3) to represent files on disk. Each directory entry is 256KB, 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

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

```
                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.

$ceil(textentries/16) * 2^{12}$ 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 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.
- 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, $ceil(\frac{sizeinbytes}{2^{12}})$.
- 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'.

```
     Meta data
                           Size
+-----------------------+
| Magic number          | 4B
+-----------------------+
| Block size in         | 4B
| power-of-2 bytes      |
+-----------------------+
| Number of FAT blocks  | 4B
+-----------------------+
| Free blocks           | 4B
| (without expanding)   |
+-----------------------+
| 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.

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.

```
JCDDirEntry
                  Size
+----------------+
| Name           | 240B
+----------------+
| Size           | 8B
+----------------+
| isFolder       | 4B
+----------------+
| First block    | 4B
+----------------+

Total size: 256B
```

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

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.

5 Quick Start Guide

References