Project Report
Group Pandas

Java and C# in depth, Spring 2013

Christian Klauser
Sander Schaffner
Roger Walt

April 6, 2013

1 Introduction

This document describes the design and implementation of the *Panda Virtual File System* of group *Pandas*. 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 *Panda Virtual File System*.

2 VFS Core

The VFS Core is the system part of the virtual file system. It manages directories and files as we are used to it in other systems like linux.

We introduced a block system. Each block can be specified in one of 4 different types. To achieve code which can be good maintained we designed 3 layers. These block types are described in the Specification section and the layers are in the Design section.

2.1 Requirements

Each requirement which is listed is also implemented. The software elements which are involved can be found in the testmethod

Panda.Test/Integration/Specification.cs. The testclass is noted to each requirement.

1. *The virtual disk must be stored in a single file in the working directory in the host file system.*
   The whole VFS is stored in the instance disk which is created at the beginning.
   Testclass: Req2_1_1_and_2

2. *VFS must support the creation of a new disk with the specified maximum size at the specified location in the host file system.*
   Possible since we can create a VFS.
   Testclass: Req2_1_1_and_2

3. *VFS must support several virtual disks in the host file system.*
   We can create n different disks with different files.
   Testclass: Req2_1_3

4. *VFS must support disposing of the virtual disk.*
   Possible with Dispose()
   Testclass: Req2_1_4

5. *VFS must support creating/deleting/renaming directories and files.*
   Files support all these methods after they are initialized.
   Testclass: Req2_1_5

6. *VFS must support navigation: listing of files and folders, and going to a location expressed by a concrete path.*
   Since files and folders inherit from nodes we can store them in an list and loop trough them. With navigate() we can go to a concrete path.
   Testclass: Req2_1_6

7. *VFS must support moving/copying directories and files, including hierarchy.*
   move() and copy() are implemented.
   Testclass: Req2_1_7

8. *VFS must support importing files and directories from the host file system.*
   import() is implemented and reads from the system.io.stream
   Testclass: Req2_1_8_and_9

9. *VFS must support exporting files and directories to the host file system.*
   import() is implemented and writes to the system.io.stream
   Testclass: Req2_1_8_and_9

10. *VFS must support querying of free/occupied space in the virtual disk.*
    Size is implemented as an Field of diretory and file. To get the occupied space just ask root.
    Testclass: Req2_1_10

11. **Bonus Basic:** *Elastic disk: Virtual disk can dynamically grow or shrink, depending on its occupied space.*
    The intern data structure can already grow and shrink dynamically since we work with blocks.

12. **Bonus Advanced:** *Large data: This means, that VFS core can store & operate amount of data, that can't fit to PC RAM (typically, more than 4Gb).*
    To fullfill this taks we decided to work with a block structure. This implies automaticaly that we can sotre more than 4Gb.

2.2 Design

Give an overview of the design of this part and describe in general terms how the implementation works. You can mention design patterns used, class diagrams, definition of custom file formats, network protocols, or anything else that helps understand the implementation.

The VFS Core is divided in 3 layers. Each of it accesses the layer below and ensures good maintenance and readability.

2.2.1 I/O layer

On this level we store the raw data. The methods related to this abstraction level are stored in Panda.Core/Core/IO.

2.2.2 Block layer

We have abstract blocks of uniform size. They are specified in the section Specification. The class blockManager manages the acces to these blocks. It communicates with the I/O layer.

With the blockManager we can allocate new blocks get data out blocks or access the blockOffset which contains block relevant data. The methods related to this abstraction level are stored in Panda.Core/Core/Blocks.

2.2.3 Filesystem layer

This is the most abstract level. Here we implemented the functionality of the VFS which are based on the requirements. It communicates with the Block layer via the blockManager.

The most important Classes are the VirtualDirectory.cs and VirtualFile.cs which inherit from VirtualNode.cs. With the node concept we realize the fact that a directory and file have many things in commen (they can be renamed, deleted, etc.). Due to the integration of an enumerator we can acces a list of nodes very efficient.

The methods related to this abstraction level are stored in Panda.Core and Panda.Core/Core/Internal.

2.3 Specification

This section lists the detailed specification of the blocksystem of the virtual file system.

2.3.1 General Remarks

- Offsets & lengths in bytes.
- The VFS is organized in blocks with fixed BLOCK_SIZE.
- All addresses are in number of blocks from 0 and of length 4 bytes.
- Only single links to blocks (not more than one hard-link) are allowed. This means that one file or directory can only be in one directory.
- Block address 0 is illegal, it means absence of a block.
- B_S := BLOCK_SIZE & d-t := data-type
- Offsets are absolute
- Strings are encoded in UTF-8

2.3.2 Metadata

Metadata of the whole VFS starts at address 0.

Offset	Length	C# d-t	Description
0	4	UInt32	Number of blocks in entire VFS
4	4	UInt32	BLOCK_SIZE in bytes
8	4	UInt32	Address of root directory node
12	4	UInt32	Address of empty page block. Must never be 0
16	4	UInt32	break in number of blocks, see empty space management.
20	B_S -20	UInt32	Empty (initialized with 0)

Normal blocks are everywhere but at address 0.

Block Types

- Directory blocks (many different blocks, with optional continuation blocks)
- File blocks (many different blocks, with optional continuation blocks)
- Data blocks (many different blocks)
- Empty space block (exactly one block, with optional continuation blocks)

Directory blocks

Contain file / directory names of current directory and their block addresses.

Offset	Length	C# d-t	Description
0	?	-	Arbitrary number of directory entries
B_S - 4	4	UInt32	Link to directory continuation block. 0 here marks absence of continuation blocks.

Directory continuation blocks look the same as directory blocks and can link to other directory continuation blocks.

Directory entry

Offset	Length	C# d-t	Description
0	1	UInt8	If first bit (the least significant) set (== 1), following address points to directory. Else to file.
2	1	UInt8	Number of bytes in file name. 0 here marks end of directory block.
3	X	String	File / directory name
X	X + 4	UInt32	Address to file / directory block

File blocks

Contain addresses to data blocks.

Offset	Length	C# d-t	Description
0	8	UInt64	File size in bytes (to manage files smaller than block size)
8	?	UInt32	Arbitrary number of addresses to data blocks
B_S - 4	4	UInt32	Link to file continuation block. 0 here marks absence of continuation blocks.

File continuation blocks have file size 0 and can link to other file continuation blocks.

File blocks

Contain only plain binary data.

Empty space block

Contains addresses to empty blocks.

Offset	Length	C# d-t	Description
0	4	UInt32	Number of empty blocks in number of blocks
4	?	UInt32	Arbitrary number of addresses to empty blocks
B_S - 4	4	UInt32	Link to empty space continuation block. 0 here marks absence of continuation blocks.

2.3.3 Empty space management

The VFS is designed to maintain an index of unused blocks. The addresses of the unused blocks are stored in the empty space block. Its address is stored in the VFS meta-data. This empty space block may also have empty space continuation blocks. But not every address to an empty block in the whole VFS can be stored in this empty space block. Instead, only addresses of empty blocks up to a maximum address, which is called break, is stored in this block. If there are no empty blocks left, the break must be increased by 1, and the new empty block addresses must be added to the empty space block. If the block next to break is freed, decrease the break, otherwise the address of this block to the empty space block or its last continuation block.

3 VFS Browser

[This section has to be completed by April 22nd.]

Give a short (1-2 paragraphs) description of what VFS Browser is.

3.1 Requirements

Describe which requirements (and possibly bonus requirements) you have implemented in this part. Give a quick description (1-2 sentences) of each requirement. List the software elements (classes and or functions) that are mainly involved in implementing each requirement.

3.2 Design

Give an overview of the design of this part and describe in general terms how the implementation works. You can mention design patterns used, class diagrams, definition of custom file formats, network protocols, or anything else that helps understand the implementation.

3.3 Integration

If you had to change the design or API of the previous part, describe the changes and the reasons for each change here.

4 Synchronization Server

[This section has to be completed by May 13th.]

Give a short (1-2 paragraphs) description of what VFS Browser is.

4.1 Requirements

Describe which requirements (and possibly bonus requirements) you have implemented in this part. Give a quick description (1-2 sentences) of each requirement. List the software elements (classes and or functions) that are mainly involved in implementing each requirement.

4.2 Design

Give an overview of the design of this part and describe in general terms how the implementation works. You can mention design patterns used, class diagrams, definition of custom file formats, network protocols, or anything else that helps understand the implementation.

4.3 Integration

If you had to change the design or API of the previous part, describe the changes and the reasons for each change here.

5 Quick Start Guide

[optional: This part has to be completed by April 8th.]

If you have a command line interface for your VFS, describe here the commands available (e.g. ls, copy, import).

[This part has to be completed by May 13th.]

Describe how to realize the following use case with your system. Describe the steps involved and how to perform each action (e.g. command line executions and arguments, menu entries, keyboard shortcuts, screenshots). The use case is the following:

1. *Start synchronization server on localhost.*
2. *Create account on synchronization server.*
3. *Create two VFS disks (on the same machine) and link them to the new account.*

4. *Import a directory (recursively) from the host file system into Disk 1.*
5. *Dispose Disk 1 after the synchronization finished.*
6. *Export the directory (recursively) from Disk 2 into the host file system.*
7. *Stop synchronization server.*