Project Report
Group Pandas

Java and C# in depth, Spring 2013

Christian Klauser
Sander Schaffner
Roger Walt

April 7, 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 (assembly Panda.Core) is the actual implementation of the virtual file system. It is a library that manages directories and files within virtual as we are used to it from other file systems. Clients are presented with a simple API that models the virtual file system as a hierarchical structure. Implementation details are well hidden and the libraries design lends itself to isolated testing of its individual components.

2.1 Design

The VFS Core is divided in three layers. This makes the library easier to understand and maintain, as the individual layers need not concern themselves with the entire complexity of virtual file systems as a whole.

We chose to simplify storage management by dividing the available space in a virtual disk into evenly sized *blocks*. One of our layers, the block API, is exclusively task with the reading and writing of these blocks to and from "raw" storage space. The boundaries of the block layer define the two other layers: the file system layer, implemented on top of the block API, and the I/O layer, which manages the actual storage (reading/writing from the host file system)

2.1.1 I/O layer

On this level we store raw data. Implementations of the I/O layer implement the `IPersistenceSpace` and `IRawPersistenceSpace` interfaces in the `Panda.Core.IO` namespace. Our main implementation uses *memory mapped files* to read from and write to the disk. From the perspective of the block layer, a *storage space* is simply a large block of main memory that gets "magically" persisted to disk. The I/O abstraction is not perfect, as it hands out raw (unsafe) pointers and therefore must be backed by contiguous memory.

For debugging and unit testing purposes, we also implemented a persistence space that is not backed by a file on disk. This has proven useful as we did not have to deal with the creation and clean up of temporary files.

2.1.2 Block layer

The purpose of the block layer is to present a structured view of the raw storage space below. It divides the underlying space into evenly sized block (block size can be configured per disk), which it uses to store both file system meta information as well as the actual file contents themselves.

Blocks are addressed by their index (how many blocks come before them) as opposed to their absolute byte offset. This addressing has the advantage that it is independent of the underlying block size.

The block layer is exposed via the interface `Panda.Core.Blocks.IBlockManager`. Implementations of this interface hand out instances of `Panda.Core.Blocks.IBlock` (or derived interfaces), which are each responsible for a single block of the virtual disk.

Given such an `IBlock`, the user can read/write fields of the block in a safe manner. Pointer arithmetic and serialization/deserialization of data is handled by the block layer and thus completely hidden from upper layers.

Generally, the block layer only manipulates a single block at a time. Operations that span multiple blocks need to be coordinated on a higher level. The one exception to this rule is empty space management. Block managers

are expected to keep track of empty blocks (blocks that are ready for allocation) and they generally use blocks themselves to store this information.

As with the I/O layer, we created two implementations of the block layer. Our actual virtual disks use the `Panda.Core.IO.RawBlockManager`, which is backed by a `IRawPersistenceSpace` (in memory or based on a memory mapped file). That implementation persists blocks into the unmanaged memory provided by the raw block manager.

The second implementation, `Panda.Test.InMemory.Blocks`, simply models blocks as a collection plain old C# objects. Again, this second implementation proved very useful, because we were able to test the implementation of the upper layer before our `RawBlockManager` was ready for prime time.

2.1.3 Filesystem layer

This is the most abstract level. Here we implement the most of the actual functionality of the file system in terms of the block API.

This layer needs to worry about how the many different blocks relate to one another. For instance, if the block that is backing a large directory has not enough room for another directory entry, the file system is responsible for allocating an overflow or "continuation" block and spill the new entry there.

When implementing the file system API, we chose to follow the composite pattern. We have `VirtualFile` as the leaf and `VirtualDirectory` as the inner node of our virtual directory tree (both inherit from `VirtualNode`). Common operations like move, rename, delete are of course available on the `VirtualNode`-level.

The API that users of the Core consume is located in the `Panda` namespace whereas our implementation can be found in the `Panda.Core.Internal` namespace.

The entire system is wired up (instantiating persistence spaces, supplying them to block managers and injecting those into virtual disk implementations) by a pair of factory methods in `VirtualDisk`.

3 Requirements

Each requirement which is listed is also implemented. The software elements which are involved can be found in the test class Panda.Test/Integration/Specification.cs. The test method 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.
While our system defaults to creating disks in the process's working directory, it is not limited to that directory. All information about a virtual disk is stored as part of that virtual disk. As a consequence, we also do not have a central list of existing virtual disks.
The actual storage is handled by `Panda.Core.IO.MemoryMapped.MemoryMappedFileSpace` whereas the file format is implemented by `Panda.Core.IO. RawBlockManager`.
Test method: 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.
   Our implementation currently stores the maximum disk capacity as part of the disk, but also pre-allocates the virtual disk file to that size.
   Test method: Req2_1_1_and_2

3. *VFS must support several virtual disks in the host file system.*
   We can create any number of different disks each with its own file.
   Disks are completely isolated from one another. Exchanging files and directories means exporting from one disk and importing into the other.
   Test method: Req2_1_3

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

5. *VFS must support creating/deleting/renaming directories and files.*
   Files and directories support all these methods after they are created.
   See `VirtualFileImpl` and `VirtualDirectoryImpl`.
   Test method: 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 they are stored in an list and we can loop trough them. With the method `Navigate()` on `VirtualDirectory` (implemented in `VirtualDirectoryImpl`) we can go to any concrete path, absolute or relative.
   Test method: Req2_1_6

7. *VFS must support moving/copying directories and files, including hierarchy.*

`Move()` and `Copy()` are implemented as methods on `VirtualNode`.
Test method: Req2_1_7

8. *VFS must support importing files and directories from the host file system.*
   `Import()` is implemented on `VirtualDirectoryImpl`. It imports host directories recursively and uses our `System.IO.Stream`-based `CreateFile` method to import files.
   Test method: Req2_1_8_and_9

9. *VFS must support exporting files and directories to the host file system.*
   `Export()` is implemented as a method on `VirtualNode` and works recursively for virtual directories and via our own (read-only) `Stream` for virtual files.
   Test method: Req2_1_8_and_9

10. *VFS must support querying of free/occupied space in the virtual disk.*
    Size is implemented as by recursively querying the individual directories. To get the occupied space just ask root.
    Test method: Req2_1_10

11. **Bonus Basic:** *Elastic disk: Virtual disk can dynamically grow or shrink, depending on its occupied space.*
    The internal data structure can already grow and shrink dynamically. Section 4.3 discusses how keep track of how much space our file system actually needs. However, at this time, the space on disk is pre-allocated to the capacity that the user specified and our disk refuses to grow beyond that size.

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).*
    We use the operating system's virtual memory mechanism to get the relevant pieces of our file system in memory. We were careful not to hold on to data structures (blocks, virtual nodes) but instead re-create them whenever required. That way, our file system implementation only uses as much RAM as the user wielding it.

    We do, however, require a 64bit virtual address space to operate on large files. While it would be technically possible to map just parts of a large file into the much smaller 32bit address space, we chose not implement re-mapping.

4 Implementation details

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

4.1 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

4.2 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.

4.3 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.

5 VFS Browser

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

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

5.1 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.

5.2 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.

5.3 Integration

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

6 Synchronization Server

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

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

6.1 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.

6.2 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.

6.3 Integration

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

7 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.*