Java C# in depth – Part 1: VFS core

Fabian Meier
ETH ID 10-919-280
meiefabi@student.ethz.ch

Andrea Canonica
ETH ID 10-293-116
canandre@student.ethz.ch

Lukas Häfliger
ETH ID 11-916-376
haelukas@student.ethz.ch

ABSTRACT

In this part we developed the VFS core. All requirements were implemented as stated in the problem description. The disk is stored in a single file and can be configured with multiple parameters when creating it. Also hosting multiple disks and disposing them is possible without limitations. A rich variety of commands is implemented to create, delete, rename, move and copying files and directories. There is rich support for navigating through the filesystem and querying of free and occupied space is possible with a nice textual representation. Finally import and export for arbitrary files is possible without limitations. As bonus features we implemented several additions. Firstly we implemented compression and encryption with a third-party library[1]. The implemented filesystem is an elastic disk. It supports dynamical growing and shrinking when the disk is defragmentated with the `defrag` command. The last point implemented is a support for large data so the filesystem is not bound by the size of the RAM.

1. INTRODUCTION

In this part the VFS core was implemented. We used ANTLR4[2] to implement the command line. This led to a great extensibility of the command line. The VFS core was implemented in Visual Studio 2013 Ultimate using the Resharper[3] and dotCover[4] plugins. The goal of this part was to develop an API which is used in later parts.

2. VIRTUAL DISK LAYOUT

In this part the core layout of the disk is described.

- 4 bytes: address of the root directory
- 4 bytes: number of blocks in the disk
- 4 bytes: number of used blocks in the disk
- 8 bytes: the maximum size of the disk in bytes
- 4 bytes: the length of the name of the disk without the `.vdi` ending
- 128 bytes: the name of the disk as a char array

After this so called preamble of the disk, the bitmap is written. The bitmap provides information about which blocks of the disk are free and which are occupied. It uses exactly one bit per block so the actual size of this part may span over multiple blocks. The next block after the preamble and the bitmap is reserved for the root directory. The root directory has the same name as the disk. So when changing the working directory one can always change to a directory on another disk (which needs to be loaded at the time) by just accessing the root directory of said disk.

2.1 Enhancements

To improve the performance of the system, only the preamble, the bit map and the root directory are loaded on start up. From then on all operations are directly forwarded to disk to allow files which are even bigger than the RAM.

3. FILE LAYOUT

A file stored on the disk always occupies at least a single block. The first block of a file has always the following layout:

- 4 bytes: the address of the next block of the file. If there is no next block, the value of this field is zero
- 4 bytes: size of the file in bytes
- 4 bytes: the number of blocks
- 1 byte: signals if this entry is a directory. Always 0 for files
- 4 bytes: the address of the parent directory
- 1 byte: the length of the file name
- 106 bytes: the name as a char array

After the first 128 bytes, the file header, the actual data starts.
If the file extends over multiple blocks, only the first block stores the meta information. Any consecutive block has the following layout:

- 4 bytes: the address of the next block of the file. If there is no next block, the value of this field is zero
- 4 bytes: the address of the parent directory

3.1 Enhancements

To improve performance when navigating through the filesystem, only the header of a file is loaded. The actual data is only loaded when exporting the file to the host filesystem.

4. DIRECTORY LAYOUT

The layout of a directory is exactly the same as for files. Only the file size field is used to denote the number of entries in the directory.

5. DESIGN

The main application is built around the `VFSManager` class which provides an API for all operations supported. It then uses multiple classes such as `DiskFactory` and `EntryFactory` which provide low level commands to create/load disks or create/load files/directories respectively. Files and directories are abstracted through the classes `VfsFile` and `VfsDirectory` which inherits from `VfsFile`. `VfsFile` inherits from `VfsEntry` which is just an abstraction of an entry which is either a file or a folder.

[1] We used the .NET library with `GZipStream` and `Rijndael-Managed` which provide good compression and great security with AES encryption
[2] A parser/lexer generator
[3] Visual Studio plugin from JetBrains
[4] Coverage plugin from JetBrains

6. IMPLEMENTATION

All writing and reading from the disk are done with `BinaryWriters` and `BinaryReaders`. Through our own extension class we added a simple overload on the seek method which provided us a convenient way to access a certain address. A simple example is shown in listing 1.

```
1  public static void Seek(this BinaryWriter
       writer, VfsDisk disk, int address)
2  {
3      if (writer != null) if (disk != null)
           writer.Seek(address *
           disk.BlockSize, SeekOrigin.Begin);
4  }
```

Listing 1: A simple Seek extension

To ease the loading and creation of disks, the `DiskProperties` class was introduced. This class provides an easy way to load all necessary information from a disk as described in chapter 2.1.
To be able to easy change the layout of the disk or a file, the `FileOffset` class was introduced. It stores all offsets to the certain fields which need to be read.

6.1 Compression

Compression is done within the host filesystem. A `GZipStream` is used to compress the original file. The original file is opened in a `FileStream` and is then copied into a `GZipStream` which copies the source file into a temporary file with a `.gz` ending. This compressed file is then imported into the filesystem and later deleted.

6.2 Encryption

Encryption is based on the rijndael algorithm which provides AES encryption. In listing 2 the decryption is shown. On line 4 a new instance of the `RijndaelManaged` class with keySize 256 which corresponds to AES-256 is instantiated. To encrypt a file only line 12 is different and the try/-catch block is not necessary. To encrypt, one uses a `algorithm.CreateEncryptor()` instead of `algorithm.CreateDecryptor()`.

```
1  var input = new FileStream(inputPath,
       FileMode.Open, FileAccess.Read);
2  var output = new FileStream(outputPath,
       FileMode.OpenOrCreate,
       FileAccess.Write);
3
4  var algorithm = new RijndaelManaged {
       KeySize = 256, BlockSize = 128 };
5  var key = new
       Rfc2898DeriveBytes(password,
       Encoding.ASCII.GetBytes(Salt));
6
7  algorithm.Key =
       key.GetBytes(algorithm.KeySize / 8);
8  algorithm.IV =
       key.GetBytes(algorithm.BlockSize / 8);
9
10 try
11 {
12     using (var decryptedStream = new
           CryptoStream(output,
           algorithm.CreateDecryptor(),
           CryptoStreamMode.Write))
13     {
14         CopyStream(input,
               decryptedStream);
15         File.Delete(inputPath);
16         File.Move(outputPath, inputPath);
17     }
18 }
19 catch (CryptographicException)
20 {
21     Console.Error("Please supply a
           correct password");
22 }
```

Listing 2: AES decryption

7. COMMAND DESCRIPTION

- ls: lists all files in the current directory. Optional parameters are `-f` and `-d` to only list files or directories respectively
- cd: changes the working directory. It accepts either a relative path or an absolute path. As an alternative two dots `..` can be used to navigate to the parent directory
- createdisk or cdisk: creates a disk and requires a `-s` parameter which denotes the size of the created disk. The allowed units are `kb, mb, gb, tb, KB, MB, GB, TB`. As optional parameters we can specify a password for encryption with the `-pw` parameter. Further we can specify the block size, the name and the path where the disk should be created with the parameters `-b, -n, -p`. If they are not specified default values are used. For encryption we do not use encryption if not specified, for the block size we use a default value of 2048 bytes, for the name we use a automatically generated one and for the path we use the current executing directory
- loaddisk or ldisk: loads a disk for further usage. It accepts as a first argument either the path to a virtual disk or directly an identifier if the disk is in the execution directory. as a last argument one can pass a `-pw` parameter to specify a password for decryption
- removedisk or rmdisk: removes a disk from the host filesystem and takes either a path or a disk name as an argument. To accept a disk name, the disk must be in the same directory as the executable.
- listdisks or ldisks: lists all disks in the execution directory or if specified with the `-p` paramtere in the specified path
- copy or cp: copies the file from the first argument into the directory passed by the second argument. Each argument can either be a relative or an absolute path
- mkdir: Creates a directory either in the current directory when a relative path is provided, or the directory specified by the absolute path. If a directory does not exist in the provided path, it is automatically created.
- mk or touch: Creates a file in the given path either specified by an absolute or relative path
- remove or rm: removes a file or a directory specified by an absolute or relative path
- move or mv: Moves a file from a source path to a destination path, where every path can either be an absolute or a relative path
- rename or rn: renames a file or a directory specified in the first argument

- import or im: imports a file or a directory, specified in the first argument, from the host file system to a path specified in the second argument
- export or ex: exports a file or a directory, specified in the first argument, from a path on the disk to a path, specified in the second argument, on the host filesystem
- free and occ: display the free or the occupied space on the current disk respectively
- defrag or df: defragmentates the disk and shrinks it if space is available
- exit or quit: exits the application and savely unloads all disks