Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS

Zdeněk Bouška

HelenOS VFS-FUSE connector

Department of Distributed and Dependable Systems

Supervisor of the master thesis:	Mgr. Martin Děcký
Study programme:	Informatics
Specialization:	Software Systems

Prague 2014

I'd like to thank my supervisor Mgr. Martin Děcký for his guidance during my work on this thesis. I'd also like to thank HelenOS developers Jakub Jermář, Jiří Svoboda and Vojtěch Horký for their work on HelenOS. My thanks also go to my family who provided moral support.

Název práce: HelenOS VFS-FUSE connector

Autor: Zdeněk Bouška

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí diplomové práce: Mgr. Martin Děcký, Katedra distribuovaných a spolehlivých systémů

Abstrakt: Tato magisterská práce se zabývá implementací konektoru mezi FUSE ovladači souborových systémů a nativním VFS rozhraním v HelenOS. Práce nejprve popisuje možné způsoby řešení a možnosti, které přicházely v úvahu. Zvoleno bylo napojení na nízkoúrovňové vrstvě, které se prokázalo jako nejlepší. Práce dále popisuje skutečnou implementaci tohoto konektoru. Implementace byla úspěšná, a proto se práce detailně zaměřuje na toto plně funkční řešení na HelenOS operačním systému. Dané řešení mimo jiné umožňuje to, že téměř nejsou potřebné změny na obou spojovaných platformách - FUSE i Helenos VFS. Implementace konektoru ukazuje dva reálně používané FUSE souborové systémy exFAT a NTFS na operačním systému HelenOS.

Klíčová slova: HelenOS, VFS, FUSE

Title: HelenOS VFS-FUSE connector

Author: Zdeněk Bouška

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Martin Děcký, Department of Distributed and Dependable Systems

Abstract: This master thesis deals with the implementation of a connector between FUSE file system drivers and HelenOS native VFS interface. The thesis first describes the way of finding the best solution and the potential possibilities. The low level layer solution is described as the best one. Further the thesis describes the real implementation of the connector. As the implementation of the connector was successful the thesis then describes in detail the parts of the fully functional solution in real-life HelenOS system. With this solution in place almost no changes are necessary to be done neither in FUSE nor in Helenos VFS. The connector implementation is demonstrated on two real-life FUSE file systems exFAT and NTFS which were ported to HelenOS.

Keywords: HelenOS, VFS, FUSE

Contents

Introduction

Motivation

There are many FUSE file system drivers because FUSE makes it easier for a developer to implement a file system driver. This thesis makes it possible to use these drivers in HelenOS.

Goals

The goal of this master thesis is to design and prototype a connector between FUSE file system drivers and HelenOS native VFS interface.

The design has to be made in such a way that would minimize the amount of changes in both the HelenOS VFS and the FUSE file system drivers.

The Linux FUSE implementation should be utilized so that reuse of the code is maximized, e.g. between libfuse and the connector implementation. The next goal is to demonstrate the connector implementation functionality as a prototype on a real-life FUSE file system ported to HelenOS. The part of the goals is also the intention to compare this implementation of the FUSE interface with implementations in other operating systems.

Structure of the thesis

The first chapter (Development context) of this thesis deals with the context of the connector implementation. Deep knowledge of both parts which are being connected is necessary. This chapter describes HelenOS architecture with a view to the kernel and servers IPC communication. It also includes information about file systems in the HelenOS operating system and describes how it works there. This description also includes the block device interface. The last part of this chapter is about how FUSE file systems are developed and how FUSE architecture looks on Linux and other platforms.

The second chapter describes the analysis which is necessary to choose the right solution. All available possibilities to make the connector are described. The advantages and disadvantages are carefully considered. The more detailed view is focused on the connection layer selection, whether to choose the high level, the low level or the kernel channel layer. The problem with accessing block devices from FUSE drivers in HelenOS is also described in this chapter.

The third chapter includes details of the implementation. It shows how the results from analysis were transformed into a working connector prototype. Firstly the integration with the libfs library deals with the necessary parts such as operations mapping, storing data from mounted file systems and opened files. One section in this chapter describes the block device file system - way to access block devices from FUSE drivers. Another section lists reused source code from Linux FUSE and then other necessary changes in the HelenOS operating system that had to be done.

FUSE file system drivers, that were ported as part of this thesis, are described in the fourth chapter of the thesis. Namely exFAT, NTFS and some examples from the Linux FUSE package are described.

Chapter 1

Development context

This chapter includes the summary of the development context and the background which is necessary for understanding to this thesis. The summary includes both FUSE and HelenOS point of view.

1.1 HelenOS architecture summary

HelenOS[1] is an operating system that is based on the microkernel architecture. The development of this system started at the Faculty of Mathematics and Physics, Charles University. HelenOS is very portable and can run on several platforms - e.g. IA-32, x86-64, IA-64, PowerPC, ARM, MIPS.

HelenOS microkernel architecture provides the possibility to have a smaller kernel with less bugs. More about microkernel architecture can be found in Modern Operating Systems [5] on page 62. HelenOS can be also seen as a component system. The aim of the HelenOS's microkernel and component based design is to provide the system that can be called "smart design, simple code".

The kernel of HelenOS implements only several most important features like multitasking, virtual memory management, symmetric multiprocessing and ability for communication between processes - inter process communication (IPC). All other services are implemented as common user processes. Also file system drivers are implemented in this way.

Traditional systems distinguish very much between system and end-user applications. HelenOS architecture makes no distinction between the operating system and end-user applications. Applications that provide services to other applications are called servers.

Userspace tasks in HelenOS are separated, each of them has its own address space. Because of this fact tasks need a way to communicate with the kernel and other tasks. The kernel provides an IPC communication which is mostly asynchronous. There is also an asynchronous framework which provides layer over IPC communication. This asynchronous framework makes it easier to write

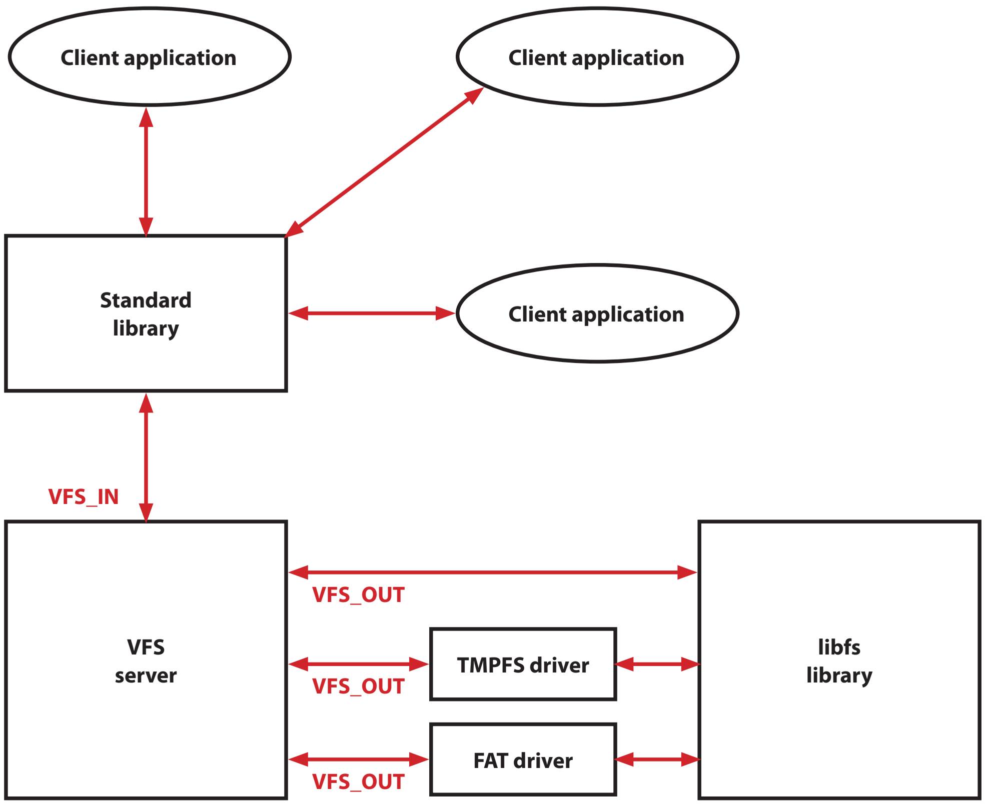


Figure 1.1: Filesystems in HelenOS

a task which communicates through IPC. Article IPC for Dummies [17] describes in detail the IPC communication and the asynchronous framework.

1.2 Filesystem in HelenOS

HelenOS file system architecture is described by Jakub Jermář in [4]. HelenOS file system architecture can be divided into three sections: Standard library, VFS server and file system driver which uses the libfs library. How this works together is the best seen in figure 1.1

1.2.1 Standard library

The standard library contains the code that transforms POSIX calls from the user task to the VFS input protocol. This protocol is understood by the entry part of the VFS server. Some calls as `opendir()`, `readdir()`, `rewinddir()` and `closedir()` are implemented by the standard library directly by calling functions `open()`, `read()`, `lseek()` and `close()`.

The standard library translates relative paths to absolute paths because the VFS server can work only with the absolute file paths. The Standard library by itself implements `getcwd()` and `chdir()` calls. The current directory is stored only in the standard library.

The standard library has no data structures and algorithms for a file system support. This means that every task that this library cannot realize by itself is given via IPC to the VFS server.

1.2.2 VFS server

The virtual file system server plays a central role in the file system support in HelenOS. This server can be divided to the input and output parts.

The input part receives calls from client tasks. If the parameter of the call is the descriptor of the file, then VFS looks in the table of opened files and finds the pointer to the structure that represents the open file. If the parameter of the call is a path, VFS performs a lookup which returns a VFS triplet. The triplet identifies the file by the global number of the file system, the global number of the device and the number of the file. Based on this triplet the VFS server tries to find the VFS node. All files are represented by these VFS nodes.

The output part of the VFS communicates with the driver of the end file system. It includes the code which calls the file system drivers using output operations. Only a lookup operation uses a path as a parameter. Other operations use VFS node as a parameter. This means: the global number of the file system, number of the device and file identification.

1.2.3 Libfs library

The libfs library implements structures and design patterns which have to be implemented by almost all file system drivers. These structures are often very similar or even the same for each file system driver. The libfs library also contains the code which registers a file system to the VFS server during the initialization.

The other fundamental role of the libfs library is connected with the functionality of the function `libfs_lookup()`. This function implements the VFS output lookup operation (VFS_OUT_LOOKUP). This operation must be implemented by every file system. The `libfs_lookup()` function does not only implement the file lookup but also manages creating and deleting files. This operation also includes the creating and the deleting links to files.

Several libfs operations must be implemented by a file system driver to ensure the functionality of `libfs_lookup()`. Those operations ”tell” the libfs library how to list a directory, how to create or delete a file in the directory tree.

1.2.4 Block device drivers in HelenOS

Each block device type (for example ATA) has its block device driver server. These servers are either part of the Device Driver Framework, or are started separately. Currently, in HelenOS only `ata_bd` is integrated in the Device Driver Framework. Other block device servers (like sata_bd) are planned to be integrated in the future.

A great example of separately started block device server is `file_bd`. This server gets a file name as an argument on the command line. This file is an image of a disk and `file_bd` creates a virtual block device over this image.

The block device driver server provides the following operations: reading from blocks, writing to blocks, getting block size and getting number of blocks on the device. There is no cache inside the block device driver servers.

File system drivers use block client library. First of all, this library offers functions to get the number of blocks and the size of one block. The main functions are `block_get` and `block_put`. These functions operate over the block device cache. The block cache uses logical blocks. One logical block can include more physical blocks.

`block_get` function locks the block in the cache and loads the block from the cache. If the block is missing in the cache it is added there before loading. It is possible either to request a block representation with valid data or to request a block without valid data (`NOREAD` flag). A block representation without valid data is useful for writing to a block when a caller does not care about previous data in the whole block. `block_get` returns the block with data in `.data` structure member variable.

`block_put` unlocks the block in the cache. A block structure contains dirty flag as `.dirty` structure member variable. If this dirty flag is set, then `block_put` writes the block data to the block device (for the write through cache type).

The block client library also includes functions to access more blocks. These functions bypass the cache and call the block device server directly.

1.3 Developing a file system with FUSE

FUSE (File System in User Space) has its origins in 1995 in GNU Hurd operating system. The concept was based on the file system driver placed not in the kernel of the system but in an userspace. This method is intended for Unix operating systems and enables to create specific file systems without changing the kernel of the system. The real FUSE development started in October 2004 as a separate project.

The FUSE file system drivers runs in the userspace. Therefore their development is as simple as the development of other userspace applications.

There are two different interfaces: A low level interface and a high level interface.

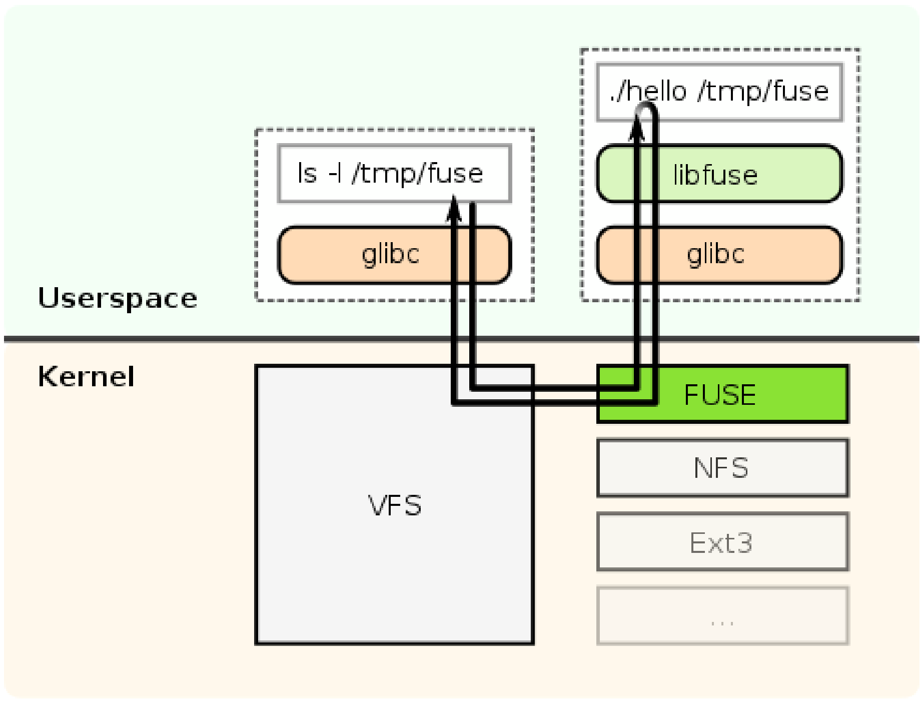


Figure 1.2: Filesystem in Userspace in Linux [8]

The high level interface identifies files by their names in all cases. For example when you want to read a file content you create a function which does this:

```
int read(const char *path, char *buf, size t size, off t offset,
struct fuse file info *fi);
```

On the other hand the low level API uses numbers to identify files. So for example when reading a directory both file names and numbers are returned. Later when reading a file, the low level driver function uses only this number for file identification purpose. In the following example ”ino” is the file number:

```
int ll_read(fuse req t req, fuse ino t ino, size t size, off t off,
struct fuse file info *fi);
```

1.4 FUSE architecture in Linux

FUSE (Filesystem in Userspace)[6] has two parts: a kernel module and an userspace library. When a call is made for example to read a file the FUSE kernel module forwards this call to an userspace driver. How does it work is best seen in the figure 1.2.

The kernel channel interface is used for exchanging messages between the userspace library and the Linux kernel. The main operations of this interface are `receive` and `send`. These messages are exchanged through a device `/dev/fuse`. The userspace library decodes these messages upon an arrival and encodes the replies before sending them back to the kernel.

When the messages are decoded by the library then an appropriate low level operation function in the low level driver is called. This function is later supposed to call a reply function with an answer. That answer is encoded and passed to the kernel through the `send` function from the kernel channel interface.

The high level interface is implemented as a library. This library is written in the same way as low level drivers are. The main purpose of the high level library is a mapping between file numbers and names.

1.5 FUSE in other operation systems

FUSE is supported in other operation systems then just in Linux. A list of them can be found on the FUSE website [9]

1.5.1 NetBSD

NetBSD has its own file system in userspace. It is called PUFFS (Pass-to-Userspace Framework File System) and its architecture is similar to FUSE on Linux.

ReFUSE [12] library was introduced in NetBSD 5.0. It linked FUSE drivers with userspace PUFFS library. It only supported FUSE High Level Drivers.

PERFUSE (PUFFS Enabled Relay to FUSE) is implementing PUFFS to FUSE kernel API bridge in NetBSD 6.0. Userspace daemon Perfused[13] translates PUFFS requests into FUSE messages. This daemon creates `/dev/fuse`, which FUSE drivers connects to. Modified version of the FUSE library from [6] is used in this case. `mount()` and `open()` of `/dev/fuse` are modified to use their variants from libperfuse [14]. Both the low and the high level interfaces are supported.

1.5.2 OS X

FUSE for OS X [10] has two parts: OS X specific in-kernel loadable file system and a userspace library based on the FUSE project [6]. The userspace library has numerous OS X specific extensions and features.[11]

1.5.3 FreeBSD

FUSE was ported to FreeBSD [15] during Google Summer of Code 2007 and 2011. It uses the userspace library from the FUSE project [6] and is currently maintained. The architecture is similar to FUSE for Linux.

1.5.4 Solaris

In Solaris only the high-level FUSE interface from version 2.7.4 is present. Solaris FUSE uses header files ported from Linux but the implementation is Solaris specific. It is 'just' a wrapper over libuvfs. UVFS is the Solaris equivalent of FUSE. UVFS uses doors calls and a pseudo file system for communication between the kernel and the userspace. [16]

Chapter 2

Analysis

This chapter includes the analysis of problems connected with the different possible solutions. At the end of each section of this chapter the final selected solution is described.

2.1 Decision whether to use a FUSE server or a library

One important decision is to choose a form how the implementation of the connector between FUSE drivers and HelenOS VFS server can be done so that it would function best.

One way to connect a specific FUSE file system driver to the VFS server is to create a FUSE server. This new server would do all the data recoding and therefore it would smooth out the differences between FUSE and HelenOS VFS. This server would forward requests and responses to and from specific FUSE file system driver servers.

Another possible solution is to create a library that would convert the FUSE driver to HelenOS file system server. Basically this newly created library would convert the FUSE driver to the HelenOS file system driver server. This solution removes the need for changes in the VFS server.

The best solution depends on the layer selection which is discussed in the next section 2.2. So the selected solution is described there.

2.2 Layer selection

It is important to choose a FUSE interface layer which would best fit to connect a FUSE driver and HelenOS's VFS. As described in the section 1.4 there are three interface layers: the kernel channel interface, the low level interface and the high level interface.

+	Code which best fits HelenOS VFS
-	No support for low level API drivers
-	File names vs. file numbers problem
-	Almost all must be written from scratch

Table 2.1: Advantages and disadvantages of connection at high level interface layer

+	Similar to VFS_OUT and libfs operations
+	No need for FUSE server
+	High level interface code from Linux FUSE library
+	Both high and low level interface drivers supported
-	Not using low level code from Linux FUSE library

Table 2.2: Advantages and disadvantages of connection at low level layer

The connection can be made in all these three layers. Every solution has its own advantages and disadvantages.

2.2.1 High level interface

The high level interface uses file names for identification. This fact means a great complication because HelenOS VFS output interface uses integer indexes to identify files.

Choosing this layer would mean rewriting all the code which is already present in the Linux FUSE library. On the other hand this new code could be more suitable for VFS output and libfs operations.

Another drawback of choosing the high level layer solution is that it doesn't support the low level interface file systems.

Solaris 1.5.4 and NetBSD 5.0 1.5.1 are using this possibility - high level interface.

The advantages and disadvantages of the connection at the high level interface layer can be seen in the table 2.1.

2.2.2 Low level interface

This interface is the most similar to HelenOS VFS output protocol. Both of them use integer indexes to represent files. The only exception is the VFS output operation lookup. Fortunately the libfs library divides this operation into several operations which are similar to the ones in the FUSE low level interface.

Because of this similarity this interface represents a good choice for creating the library which could convert the FUSE driver to the HelenOS file system server.

The advantages and disadvantages of connecting at the low level interface layer of FUSE can be seen in the table 2.2.

+	Designed for connection in this layer
+	Almost all Linux library code reusable
+	Works good with other programing languages then C
-	Encoding and decoding messages
-	FUSE server is necessary

Table 2.3: Advantages and disadvantages of connection at kernel channel interface

2.2.3 Kernel channel interface

In FUSE all the file system operations are encapsulated into the kernel channel interface messages. In order to select this layer VFS output operations need to be converted into these messages.

The best way to implement the connector while using this interface would be to use a FUSE server. Kernel channel API messages would then be sent between the FUSE server and a specific FUSE file system driver in the form of IPC messages. The implementation would be very similar to the way how the FUSE driver works in Linux from the driver point of view.

Connecting the FUSE file system driver to the VFS server based on the kernel channel interface allows using almost all the code from Linux's FUSE library.

NetBSD 6.0 1.5.1 uses a similar solution by its Perfused daemon.

The advantages and disadvantages of connecting at the kernel channel interface layer can be seen in the table 2.3.

2.2.4 Summary of the selected solution

According to the previously described analysis the low level interface layer is the most suitable solution for connecting FUSE file system driver to HelenOS VFS server. The main reason for this suitability is the great similarity of this layer to HelenOS libfs and VFS output operations. There is also no need for encoding operations into messages (as would be the case case with the kernel channel interface) or to convert file paths into file node integer indexes (as would be in case of the high level interface).

The next advantage is that the FUSE server does not need to be present in this solution and the connector can be implemented as a library. The FUSE library will use the libfs library in the same way as other file systems do. This minimizes changes in both HelenOS VFS and FUSE (library and drivers). The description of how the selected solutions work in HelenOS's file system architecture can be seen in the figure 2.1.

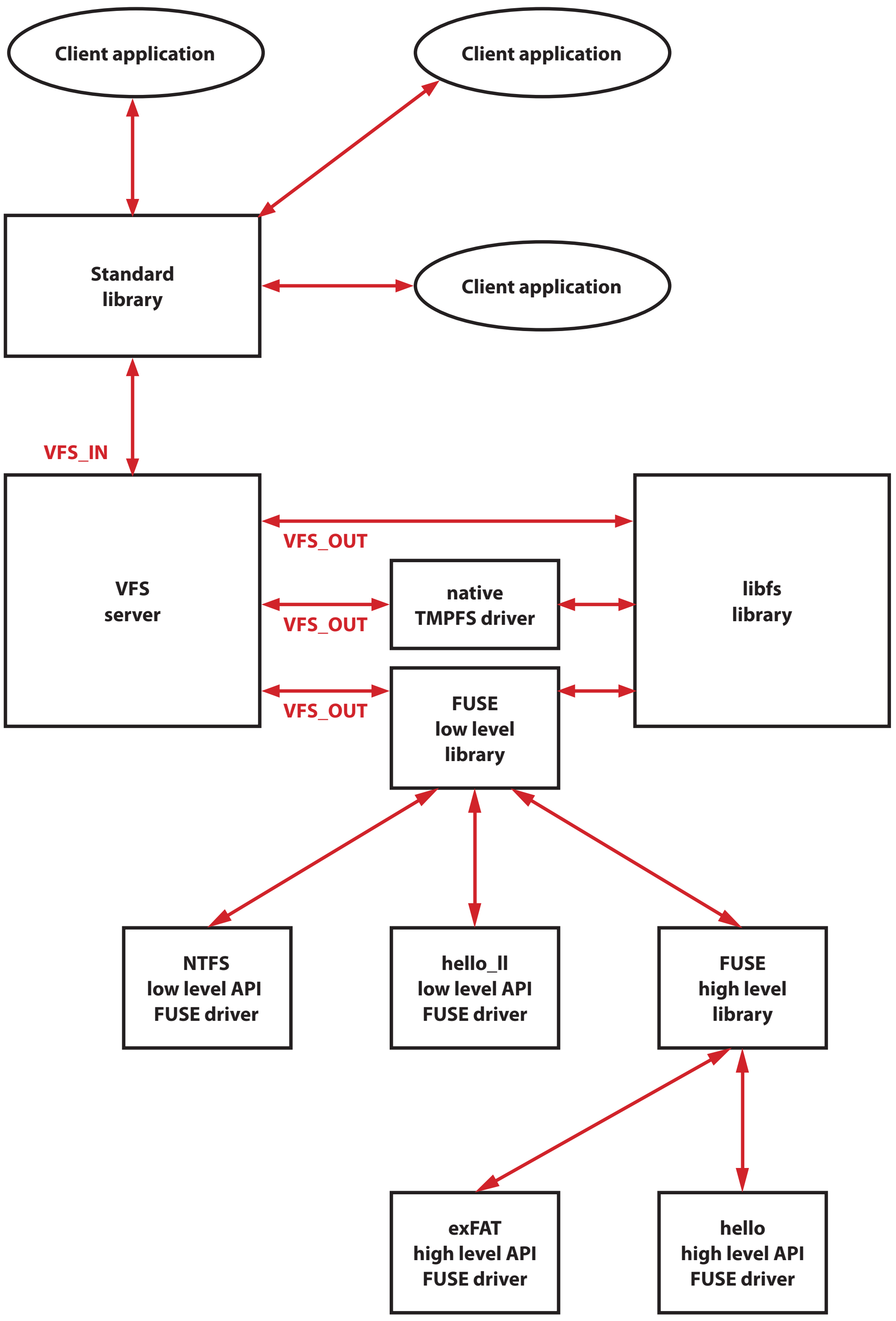


Figure 2.1: FUSE in HelenOS

2.3 Separate task for each file system driver instance

One file system server in HelenOS serves more instances of the same file system. On the other side each instance of a FUSE driver needs its own task. Fortunately this feature causes no problem since the new HelenOS file system task can be launched for each FUSE driver instance.

The FUSE drivers also mounts itself during the driver initialization. This can be done automatically after the start of the FUSE file system server. The FUSE file system server suspends itself just after the FUSE file system is finally unmounted.

2.4 Reading directories

There is a difference in reading directories in the HelenOS VFS output operation and the FUSE low level interface.

The HelenOS VFS output operation is performed for each file in a directory. This operation has a file offset as a parameter. This parameter represents a file order in a directory. So for example index 5 means 5th file in a directory. When HelenOS VFS is reading a directory it gives the position of the file in a directory.

The FUSE low level interface requests a byte offset when it is reading a directory. This byte offset points to a directory entity structures. It is not possible to request a specific (for example 5th) file name from a directory because the byte offset of that file name is not known.

Before the FUSE low level driver returns a directory structure it adds directory entities to the buffer using the low level interface function `fuse_add_dirent()`. It seems like this function could count positions in a directory and save offset for each file. This offset would be later used in order to read the desired file position without going from the beginning. Unfortunately this solution is not possible because there is no guarantee that the function `fuse_add_dirent()` is called with the same buffer which is then returned by the reply function `fuse_reply_buf`.

The reading of the whole directory is not enough efficient since the whole directory must be read again for each file. This can be accelerated by caching of the bytes offset of the last read file and requesting that offset in `readdir` low level operation. Another possibility is to cache next directory entries. This way the `readdir` low level operation would not be called more times than it is really necessary.

The connector prototype currently saves the directory entities buffer. This avoids calling the FUSE low level operation `readdir` more than once for a given specific offset.

2.5 Mounting FUSE file systems

The FUSE file system drivers are standalone applications. They receive a mount point path as a parameter from a command line. This behavior is different in the native HelenOS file system drivers. The native drivers are started before the mount action happens.

`mount.file_system_name` script can be used in order to mount FUSE file systems in the same way as the native HelenOS file systems (`mount` command) are being mounted. For each FUSE file system there would be a specific script. This script would start FUSE file system driver and mount it. The standard system library would then determine whether this script existed before the standard mount procedure. If the script existed, it would launch this script instead of sending the `VFS_IN_MOUNT` method.

2.6 Accessing block devices

There is a difference in accessing block devices in native HelenOS drivers and in FUSE drivers. The FUSE drivers access block devices directly as files. For example exFAT [18] uses `pread` and `pwrite` functions. Block devices are accessed through block device servers in the native HelenOS drivers.

There is a possibility to change some parts of the code of a FUSE driver; namely parts that access block devices. But this would need to be made for each file system repeatedly again and again. There are three ways of solving this problem without modifying all ported FUSE file system drivers: rewriting the POSIX functions, creating the block device file system server and adding VFS output interface support in a block device drivers. They are described in the following sections.

One of the problems here is consistency when writing data only to a part of the block and not to the whole block. It is necessary to read the rest of the block before writing it back because otherwise the rest of the data in the block would be lost.

2.6.1 POSIX functions rewrite

The native HelenOS applications do not follow the POSIX specification. But there is a POSIX library in HelenOS. This library makes it possible to run the POSIX applications. The POSIX calls are converted to the native calls in this library.

The FUSE drivers also follow the POSIX specification. One possible solution is to rewrite POSIX file functions. This means adding conditions to the POSIX library. For a certain prefix these functions would send read or write requests to the block device server instead of sending them to the VFS server.

+	Direct communication with block device driver
+	Uses block cache
-	All POSIX file functions must be overwritten
-	Block device is not a real HelenOS file
-	Is a ”hack”, not a proper solution

Table 2.4: Advantages and disadvantages of block devices access through POSIX functions rewrite

There are two advantages to this solution: direct communication with the block device driver server and the possibility to use cache from the block device library for consistency in writing.

The main disadvantage when choosing this way of accessing block devices is that all POSIX file functions need to be overwritten. Also, block devices are not real files from HelenOS point of view. It is more like a ”hack” than a clean solution.

Advantages and disadvantages are in table 2.4.

2.6.2 Block device file system server

The second possibility to access block devices is to create a special newly designed file system server which would enable the desired access to the block devices via VFS.

There are two approaches. This file system would either have a virtual file for each block device. Accessing this file would result in accessing the block device. This file system could be mounted at boot. It needs to constantly monitor the available block devices.

Another approach is to have one instance of this file system per a block device. This means a file system with only one file in it. This file represents the block device. The block device file system would then be mounted only for a device which is used by a FUSE file system. This approach offers simplicity, the block device file system accesses block devices and communicates with VFS server in the same way as any other native HelenOS file systems. This would be a big benefit for the future development because the code to access block devices from FUSE drivers is separated from the block device library and the server. Advantages and disadvantages of this approach can be seen in table 2.5.

The advantage of both of these approaches is the ability to use cache from a block device library for consistency in writing.

The disadvantage of this solution (for both approaches) is that it adds another layer between a FUSE driver and a block device driver.

Implementation is described in detail in the section 3.4.1.

+	Device looks the same as real HelenOS file
+	No changes to existing parts of HelenOS
+	Works the same as native file system drivers
+	No complications in future development
+	Uses block cache
-	Adds another layer between a FUSE driver and a block device driver

Table 2.5: Advantages and disadvantages of block devices access file system server

2.6.3 VFS output protocol support in block device drivers

The third possibility is to add VFS output interface support directly to the block device driver servers. Only some VFS output operations would be necessary for this solution (mainly `VFS_OUT_READ` and `VFS_OUT_WRITE`).

The console device drivers use a similar solution.

In this case the block device driver servers use a shared common skeleton library (`bd_srv.h`). There would be no need to add support for these VFS output operations into every single block device driver. They could be implemented directly in the common skeleton library (in `bd_srv.c`). But this library does not cover everything. More parts from current block device servers need to be integrated there.

There are differences between VFS output interface and block device interface. These differences would need to be sorted out first. There are other complications with integration with the Device Driver Framework. Main complications are connected with the shared function which handles incoming IPC calls. Even if these differences were sorted, it would create a more complicated code. And this would make future development more complicated.

The VFS output operations would be implemented by calling block manipulation functions for reading or writing blocks. But for writing to only a part of a block it does not offer data consistency. A newly designed block locking mechanism would need to be added. This is a big disadvantage comparing to other solutions of accessing block devices.

Advantages and disadvantages can be found in the table 2.4.

This solution looked promising at first before all the complications appeared to the extent that the final solution would not be efficient enough. Partially working implementation is described in the section 3.4.2.

2.6.4 Conclusion

The POSIX function rewrite is probably the easiest way to implement, though it is not the cleanest way, the block device does not appear as a file and it is hard to track all POSIX file functions.

The block device file system server is a nice clean solution. It adds another layer between the FUSE file system driver and the block device. This can be viewed as

+	Device looks same as real HelenOS file
+	Only VFS server is between FUSE driver and block device driver
-	Block device interface conflicts with VFS output interface
-	Problems with integration with the Device Driver Framework
-	No block cache
-	Synchronization problem for writing
-	Complicates future development of block device and driver framework

Table 2.6: Advantages and disadvantages of block devices access file system server

a downside or an upside. Downside is obvious - another layer makes block device access a bit slower. Upside is separating accessing block devices as files from the actual block device drivers and the HelenOS driver framework. The variant of block device file system with a new instance for each block device is more flexible and also easier to implement.

Supporting VFS output protocol in block device drivers comes with too many complications. It also makes the future development harder by mixing two different interfaces (VFS and block device) together.

From the previous analysis the conclusion is that the best solution is to create a block device file system (described in the section 2.6.2) in the approach with a single file for a block device.

The implementation is described in the section 3.4.1.

Chapter 3

Implementation

This chapter describes implementation details of the HelenOS VFS-FUSE connector. The connector is implemented as FUSE library. The source code of this library can be found in `uspace/lib/posix/fuse` in HelenOS tree.

3.1 Integration with libfs

The connector works as a library implementing libfs and VFS output operations in the same way as any other file system. As discussed in the chapter 2 there is almost no need to do any changes in the libfs library.

3.1.1 Mapping operations

The mapping between HelenOS VFS output operations (including libfs operations) and the FUSE low level operations is described in the table 3.1. From the FUSE point of view there is no difference in mapping libfs and VFS output operations.

Some libfs operations do not need to call FUSE low level operations. The following operations `is_directory`, `is_file`, `lnkcnt_get`, `size_get` only return data retrieved by the previous call of libfs operation `node_get`. `node_put` only frees data from memory. Some other operations are not necessary for the functional prototype of the connector and therefore are not implemented.

The call to FUSE driver is delayed until the first `link` operation because the file name is not known in libfs operation `create_node`.

Most of the low level interface conversion code can be found in the file `uspace/lib/posix/fuse/lib/fuse_lowlevel.c`.

HelenOS libfs operations	FUSE low level operations
root_get	getattr
node_get	getattr
node_open	opendir, open
link_node	mkdir, mknod, link
unlink_node	rmdir, unlink
match	lookup
HelenOS VFS_OUT operations	**FUSE low level operations**
mounted	getattr
unmounted	destroy
read	getattr, read, readdir
write	getattr, writebuf, write
close	release, releasedir, flush
truncate	setattr
sync	fsync, fsyncdir

Table 3.1: Operations mapping between HelenOS FS and FUSE low level interface operations

root_get, node_get

`root_get` and `node_get` libfs operations return the node representation. This node (`fs_node_t`) is later used as an argument for other libfs operations.

The `root_get` operation is implemented by the `fuse_lowlevel_root_get` function. The `node_get` operation is implemented by the `fuse_lowlevel_node_get` function.

The `fuse_lowlevel_root_get` function calls the `fuse_lowlevel_node_get` function with the root node index as an argument. The FUSE root node index is determined in the VFS output `mounted` operation. This is different from other native HelenOS file systems which use zero (`0`) as root node index. But this difference does not matter. The actual value is not used anywhere in the VFS server or the libfs library.

The `fuse_lowlevel_node_get` calls the FUSE low level `getattr` operation to read the file attributes.

The file system node (`fs_node_t`) is created from the `stat` structure which comes from the `getattr` operation. The `fuse_lowlevel_node_get_impl` function manages that. This function is also used by the `match` and `create_node` libfs operations.

mounted

The `mounted` VFS output operation is called when a file system is being mounted. The root node index, link count and size need to be set to the attributes of this operation.

The mounted operation is implemented by the fuse_lowlevel_mounted function.

The fuse_lowlevel_mounted function initializes the file system. The fuse_lowlevel_node_get function with the number one (1) as index argument loads the file system root node. The root node index, link count and size are taken from the root node.

unmounted

The unmounted VFS output operation is called when file system is being unmounted.

The unmounted operation is implemented by the fuse_lowlevel_unmounted function.

The fuse_lowlevel_unmounted function calls the destroy FUSE low level operation.

The FUSE server should terminate here because the FUSE file system server serves only one instance of the file system driver. Unfortunately the driver termination is not possible in the unmount VFS output operation because asynchronous answer (async_answer_0(rid, rc);) in the function vfs_out_unmounted in uspace/lib/fs/libfs.c needs to be called first. This answer lets the VFS server know the return code from the unmounted operation.

This thesis introduces the callback operation after_unmounted. From the driver point of view it looks like a VFS output operation. This callback operation is called at the end of the vfs_out_unmounted function. This callback operation is implemented by the fuse_lowlevel_after_unmounted function from uspace/lib/posix/fuse/lib/fuse_lowlevel.c. FUSE drivers terminate here in this function.

create_node

The create_node libfs operation creates a new node. This node is not connected or linked anywhere. The libfs library later creates a link to this node by calling the link_node operation.

The create_node operation is implemented by the fuse_lowlevel_create_node function. This function does not call any FUSE low level operation. The empty node with index 0 is returned. This index is later changed by the link_node operation.

link_node

The link_node libfs operation adds link to the node into the directory tree.

The link_node operation is implemented by the fuse_lowlevel_link_node function.

There are two cases that can happen: linking to a new node or adding new node to the node which is already linked.

In the first case linking to a new node means that the node has been created by the create_node operation. mkdir or mknod FUSE low level operations are called in this case. The mkdir operation is called for the case of directory, the mknod operation is called for the case of file. Both of these operations return stat structure. It is extracted by the fuse_lowlevel_reply_attr function and replaces the old temporary stat structure within the node. This also fills the new node index number in the node.

In the second case the link FUSE low level operation is called for the already linked node.

unlink_node

The unlink_node libfs operation removes the node from the directory tree.

The unlink_node operation is implemented by the fuse_lowlevel_unlink_node function.

rmdir or unlink FUSE low level operations are called in the fuse_lowlevel_unlink_node function. The rmdir operation is called for the case of a directory, the unlink operation is called for the case of a file.

match

The match libfs operation finds node by a given patch.

The match operation is implemented by the fuse_lowlevel_match function.

The fuse_lowlevel_match function calls the FUSE low level lookup operation.

The file system node (fs_node_t) is created from the stat structure which comes from the lookup operation. The fuse_lowlevel_node_get_impl function manages that.

node_open

The node_open libfs operation opens a node.

The node_open operation is implemented by the fuse_lowlevel_node_open function.

This operation is used both for files and directories.

The fuse_lowlevel_node_open_dir function is called for directories.

The fuse_lowlevel_node_open_file function is called for files.

The fuse_lowlevel_file_get_file_info function is called at the beginning of both of these functions. New file data structure (fuse_file_info) for the FUSE low level `open` and `opendir` operations is allocated by the `fuse_lowlevel_file_get_file_info` function. A pointer to this structure is also added to the `opened_file_info` hash table. This hash table allows to use this file data structure later when there is a request to read or write to this file. This is a way to allow FUSE file system to save file system specific data about opened file.

The `fuse_lowlevel_node_open_dir` function is simple. It just calls the FUSE low level `opendir` operation.

The `fuse_lowlevel_node_open_file` function is a bit complicated. HelenOS VFS server does not send opening flags (for example read-only) over the VFS output interface. Because of that the `fuse_lowlevel_node_open_file` function tries to call the FUSE low level `open` operation with read-write flag (`O_RDWR`) If opening with these flags fails this function tries a second attempt with the read-only flag (`O_RDONLY`). One must have in memory that the append file open mode could not be used here. The append file open mode is completely manged by the VFS server which stores current position in the file. File system drivers in the HelenOS always work with an absolute position in a file, so the append file open mode does not make sense in this case.

read

The `read` VFS output operation reads data from a file or from a directory.

The `read` operation is implemented by the `fuse_lowlevel_read` function.

This operation is used both for files and directories. In both of these cases the function `fuse_lowlevel_get_file_info` loads a pointer to the saved file data (`fuse_file_info`). This pointer is stored in the hash table from the `open` operation. This is a way to allow FUSE file system to use the stored file system specific data about opened file.

The `read` FUSE low level operation is called for the case of a file. The pointer to the file data (`fuse_file_info`) is given as an argument to this operation. Then the size of the data that has been read is calculated.

More complicated case is reading a directory content. The requested position in the directory to be read means reading a node name at that position. Only one node name is read. First of all it is necessary to increase the requested position in the directory by two (`2`). That skips `.` and `..` directories which HelenOS does not use while FUSE does. Then `readdir` FUSE low level operation is called as many times until desired position is found in the directory buffer from the `readdir` operation. The pointer to the file data (`fuse_file_info`) is given as an argument to this operation. The function `find_dirent` then extracts the desired directory entry from this buffer.

More detailed analysis of reading directories in the VFS-FUSE connector can be found in the section 2.4.

write

The `write` VFS output operation writes data to a file.

The `write` operation is implemented by the `fuse_lowlevel_write` function.

The function `fuse_lowlevel_get_file_info` loads a pointer to the saved file data (`fuse_file_info`). This pointer is stored in the hash table from the `open` operation. This is a way how to allow the FUSE file system to use the stored file system specific data about opened file.

There are two FUSE low level operations which write to a file: `write_buf` and `write`. The `write_buf` operation is more general, it allows writing data from several buffers.

If the FUSE low level operation `write_buf` does not exist in a given FUSE file system then the `write` operation is used. The `write` operation is sufficient in this case because only a single buffer is used in HelenOS VFS output `write` operation. The pointer to the file data (`fuse_file_info`) is given as an argument to both of these operations.

close

The `close` VFS output operation closes a file.

The `close` operation is implemented by the `fuse_lowlevel_close` function.

This operation is used both for files and directories. The function `fuse_lowlevel_get_file_info` loads a pointer to the saved file data (`fuse_file_info`). This pointer is stored in the hash table from the `open` operation. This is a way how to allow FUSE file system to use the stored file system specific data about opened file.

The file handle reference counting is handled by the VFS server. So the `close` VFS output operation is called only once, immediately after all references are dropped.

The `releasedir` FUSE low level operation is called for the case of a directory. The pointer to the file data (`fuse_file_info`) is given as an argument to this operation.

The `flush` and `release` FUSE low level operations are called for the case of a file. The pointer to the file data (`fuse_file_info`) is given as an argument to these operations.

At the end the function `fuse_lowlevel_remove_file_info` deallocates the file data (`fuse_file_info`) and removes the pointer to those data from the hash table `opened_file_info`.

truncate

The `truncate` VFS output operation truncates the file to a given size.

The `truncate` operation is implemented by the `fuse_lowlevel_truncate` function.

The `setattr` FUSE low level operation is called with the `to_set` flag set to `FUSE_SET_ATTR_SIZE`.

sync

The `sync` VFS output operation performs the synchronization of file contents.

The `sync` operation is implemented by the `fuse_lowlevel_sync` function.

This operation is used both for files and directories. In both of these cases the function `fuse_lowlevel_get_file_info` loads a pointer to the saved file data (`fuse_file_info`). This pointer is stored in the hash table from the `open` operation. This is a way to allow FUSE file system to use the stored file system specific data about opened file.

The `fsync` FUSE low level operation is called for the case of a file. The pointer to the file data (`fuse_file_info`) is given as an argument to this operation.

The `fsyncdir` FUSE low level operation is called for the case of a file. The pointer to the file data (`fuse_file_info`) is also given as an argument to this operation.

3.1.2 Reply functions from the low level interface

The implementations of FUSE low level operations use reply functions. These functions return the status of operations and they send back the actual requested data for some operations (like `read`). In the original FUSE library [6] these reply functions send messages back to the kernel.

It was necessary to create a reply structure as part of the FUSE request structure `fuse_req_t`. This request structure is passed as the first parameter in all FUSE low level operations. FUSE reply function adds data to the reply part of the request structure. These data are then extracted after the FUSE low level function call is finished. It is necessary to convert some of these data and then return them using the asynchronous framework.

In most of the cases more then one FUSE low level operation is called in one libfs or VFS output operation. This is a reason why data can not be sent to VFS server in the reply function.

Reply structure

FUSE low level reply functions use reply structure `fuse_req_t`. This structure has several members.

The main member's name is err. The err member is an integer return code which is set by the fuse_reply_err function. The err member is set to EOK by all other reply functions.

A group of members saves FUSE entry parameter (struct fuse_entry_param). The FUSE entry parameter is saved in the entry member. The entry_valid boolean member describes whether the entry member contains a valid data. The entry_used boolean member describes whether the entry was requested by the fuse_lowlevel_reply_entry function. If the entry_used equals to true then the entry is not deallocated by the fuse_lowlevel_destroy_req function. The deallocation responsibility lies on the fuse_lowlevel_reply_entry caller.

A group of members saves file attributes (struct stat). The file attributes are saved in the attr member. The attr_valid boolean member describes whether the attr member contains a valid data. The attr_used boolean member describes whether the attributes were requested by the fuse_lowlevel_reply_attr function. If the attr_used equals to true then the attr is not deallocated by the fuse_lowlevel_destroy_req function. The deallocation responsibility lies on the fuse_lowlevel_reply_attr caller.

Another group of members saves a buffer. The buffer is saved in the buf member. The buffer size is saved in the buf_size member. The buf_valid boolean member describes whether buf and buf_size members contain a valid data. The buf_used boolean member describes whether the buf were requested by the fuse_lowlevel_reply_buf function. If the buf_used equals to true then the buf is not deallocated by the fuse_lowlevel_destroy_req function. The deallocation responsibility lies on the fuse_lowlevel_reply_buf caller.

The write_count stores the size of the written data.

The *_used attributes are there in order to skip unnecessary data copying of the entry, attributes or a buffer.

fuse_reply_err

The fuse_reply_err function sets the error code to the reply structure (err member).

fuse_reply_entry

The fuse_reply_attr handles the reply with the FUSE entry parameter.

The fuse_reply_entry function sets the err member to EOK. The EOK is a constant which means that there was no error. It also fills the entry member of the reply structure with the FUSE entry parameter (struct fuse_entry_param). The entry_valid is set to true because the entry member contains valid data and the entry_used is set to false.

The attr member field is also filled with file attributes because the fuse_entry_param contains file attributes. The attr_valid is set to true because the attr member contains valid data and the attr_used is set to false.

The fuse_reply_entry function is called from lookup, mknod, mkdir and link FUSE low level operations.

fuse_reply_attr

The fuse_reply_attr handles the reply with file attributes.

The fuse_reply_attr function sets the err member to EOK. The attr member field is filled with file attributes. The attr_valid is set to true because the attr member contains valid data and the attr_used is set to false.

The fuse_reply_attr function is called from getattr and setattr FUSE low level operations.

fuse_reply_buf

The fuse_reply_buf handles a reply with a buffer.

The fuse_reply_buf function sets the err member to EOK. The buf member is filled with the buffer. The buf_size member is set to the buffer size. The buf_valid is set to true because the buf member contains valid data and the buf_used is set to false.

The fuse_reply_attr function is called from getattr and setattr FUSE low level operations.

fuse_reply_write

The fuse_reply_write handles a reply from write operations.

The fuse_reply_write function sets the err member to EOK. The number of bytes written by the operations is set to the write_count member of the reply structure.

The fuse_reply_write function is called from the write FUSE low level operation.

Other reply functions

Other reply functions are implemented only as a dummy functions which set the err member to EOK. They are not necessary for the working VFS-FUSE connector prototype.

3.1.3 Mounting

The FUSE driver is mounted automatically during the initialization to the mountpoint path specified in a command line. This is done for example for FUSE exFAT driver by this command: `fuse_exfat /bd/dev /mnt`).

The Linux FUSE library mounts a file system inside function `fuse_mount_common` in `uspace/lib/posix/fuse/lib/helper.c` file. This mounting is done too early for the HelenOS. Too early means that the FUSE driver is not yet connected to the VFS server. The mountpoint is saved into the `fuse_chan` structure and later used in the `fuse_session_loop` function in the file `uspace/lib/posix/fuse/lib/fuse_lowlevel.c`. The `mount` function is called from the `fuse_session_loop` function. This `mount` function is the same function which is called by the `mount` command from the command line. The name `fuse` with the process id as suffix is used as the name of the FUSE file system.

The FUSE server should terminate when the file system is being unmounted because the FUSE file system server serves only one instance of the file system driver. Unfortunately that is not possible because the asynchronous answer (`async_answer_0(rid, rc);`) in the function `vfs_out_unmounted` in `libfs.c` needs to be called first. This answer lets the VFS server know the return code from the `unmounted` operation.

This thesis introduces callback operation `after_unmounted`. From the driver point of view it looks like a VFS output operation. This callback operation is called at the end of the `vfs_out_unmounted` function. This callback operation is implemented by the `fuse_lowlevel_after_unmounted` function from `uspace/lib/fs/libfs.c`. FUSE drivers terminate here in this function.

3.1.4 Mounting other file systems under FUSE

Sometimes other file systems are mounted within the FUSE file system directory tree. The libfs library then needs to store some data about nodes which work as mount point for them.

The mount point data are stored in the `mp_data` member in the file system node (`fs_node_t`). The mount point node with the mount data is stored in the hash table `mounted_fs_nodes`. The `fuse_lowlevel_node_get_impl` function manages saving to this table.

The `fuse_lowlevel_node_put` function does not free the node when the mount point is active (`mp_data.mp_active`).

3.1.5 Storage for data about opened files

The FUSE low level driver allows storing some file system specific data for opened files. The structure `fuse_file_info` is used for that. These data are later used in all other FUSE low level file operations (`read`, `write`, `flush`, `release`, `fsync`).

VFS output operations are stateless and it is necessary to store these data somewhere. Opened files data are stored in a hash table similarly as the data about the mount points. These data are removed from the hash table while closing the file.

There are two functions which manage data about opened files: `fuse_lowlevel_get_file_info` and `fuse_lowlevel_remove_file_info`. First parameter of both of them is numeric file index.

The `fuse_lowlevel_get_file_info` function loads file data from the hash table (`opened_file_info`). Second parameter of this function determines what to do when there are no data about the file in the hash table. If second parameter is equal to `false`, then NULL is returned when there are no data about the file in the hash table. If second parameter is equal to `true`, then new file data structure (`fuse_file_info`) is allocated. The pointer to this data structure is added to the hash table and returned as return value.

The `fuse_lowlevel_remove_file_info` deallocates the file data structure and removes pointer to this file data structure from the hash table.

The `fuse_lowlevel_get_file_info` function is called at the beginning of the `open_node` libfs operation. Only in this place the second parameter is set to true, that mean allocating new file data structure.

All other FUSE low level operations (for example `read`) load file data by the `fuse_lowlevel_get_file_info`.

The `fuse_lowlevel_remove_file_info` function is called at the end of the FUSE low level `close` operation. This deallocates the file data and removes the pointer to these data from the hash table. The VFS server counts references to the opened file and calls the FUSE low level `close` operation only when there is no remaining reference to this file. So it is safe to deallocate the file data in the `close` operation.

3.1.6 Multithread support

A multithread support in the connector prototype is limited since the Pthread support in HelenOS POSIX library is also limited. The multithread access is supported only for low level layer drivers.

There is a locking system around all low level operations. The `fuse_lowlevel_single_thread_lock` function lock is called before each FUSE low level operation. The `fuse_lowlevel_single_thread_unlock` function is called after each FUSE low level operation. These functions lock and unlock the `fuse_lowlevel_single_thread_mutex` lock. This lock is ignored by these functions when the driver supports multi thread environment and starts from the multi thread loop function `fuse_session_loop_mt`. This lock gives assurance that no more than one FUSE low level operation is called at the same time.

3.1.7 File indexes

Both HelenOS VFS output operations (including libfs library operations) and FUSE low level interface operations use numeric file indexes. HelenOS operations use `fs_index_t` numeric type for file index.

Libfs operations and VFS output operations use the same file indexes as the FUSE low level interface. Their types are compatible.

3.1.8 Creating and renaming files

`mkdir` or `mknod` FUSE low level operations are not called from the libfs operation `create` in the time when a file or a directory is created. Instead of this a dummy node with the file index set to zero (`0`) is returned. The reason for this behavior is that the file name is not known at that moment. `mkdir` or `mknod` FUSE low level operations are called later when the libfs library calls the first `link` operation on this node with the file name.

There is one issue: It is not possible to rename files, when the file system does not support the `link` operation. The VFS server does not have the `VFS_OUT_RENAME` operation and instead of this just calls `link(new_name)` and then `unlink(old_name)`. This issue is not specific only to the FUSE drivers in HelenOS. It is also present in other native HelenOS file system drivers which cannot handle more than one link to a file.

This rename issue appears only in the exFAT FUSE driver. The NTFS FUSE driver is unaffected because it can handle more than one link to a file.

3.2 High level interface

The high level interface source code implements the low level interface operations. Almost all the code that implements the high level interface is reused from Linux FUSE library. There is also some code which is not used (not called from the connector library). This unused code is left there to enable easier upgrades based on newer versions of Linux FUSE library.

The most of the high level interface source code is in the `uspace/lib/posix/lib/fuse.c` file.

Pthread library

The high level interface uses pthread locks and condition variables. In order to make this work then pthread locks and condition variables are transformed to HelenOS fibril variants of locks. A fibril is a cooperatively scheduled thread which is used by the HelenOS libfs library. These fibrils handle incoming operations in the HelenOS file system servers.

include/fuse.h
include/fuse_lowlevel.h
include/fuse_compat.h
include/fuse_common_compat.h
include/fuse_kernel.h
include/fuse_lowlevel_compat.h
include/fuse_opt.h
lib/fuse_misc.h
lib/fuse_opt.c

Table 3.2: Files from Linux FUSE library with no changes

include/fuse_common.h
lib/fuse_i.h
lib/fuse.c
lib/buffer.c
lib/helper.c

Table 3.3: Files from Linux FUSE library with small changes

3.3 Reused code from Linux FUSE

The table 3.2 lists files with no changes to the upstream Linux FUSE library[6].

The table 3.3 lists files with small changes. Those changes are separated by `#ifdef __HelenOS__` in order to make it easier to update them to new versions of Linux FUSE library.

The last table 3.4 lists files with HelenOS specific code.

3.4 Block devices access

This section describes implementation of two ways of solving the access to block devices from FUSE file systems. Analysis and decision which solution to use are described in the section 2.6.

include/config.h
lib/fuse_kern_chan.c
lib/fuse_lowlevel.c
lib/fuse_mt.c
lib/fuse_session.c
lib/fuse_signals.c

Table 3.4: Files with almost all code being HelenOS specific

3.4.1 Block device file system server

This section describes implementation details of the block device file system server. The properties of this solution are described in the section 2.6.2.

This solution is part of the final code.

The block device file system has only two nodes, the root node and a file `dev` which represents the block device. Theoretically in this solution there could be only one node. Nevertheless, letting the root node be a file is not expected when mounting a file system. Also, VFS server does not expect the root node to be a file.

`block_get` and `block_put` functions are used to access a block device.

The description of some VFS output and libfs operations follows. They can all be found in `uspace/srv/fs/bdfs/bdfs_ops.c`

bdfs_mounted function

This function is called after the block device file system is mounted.

First the block device library is initialized with the `block_init` call.

The physical block size is determined by the `block_get_bsize` function call.

Afterwards the block cache is initialized with the `block_cache_init` function call. The logical block size is equal to the size of the physical block.

bdfs_unmounted function

This function is called after the block device file system is unmounted.

The `bdfs_unmounted` function calls the `block_cache_fini` and also the `block_fini` function.

bdfs_read function

VFS output read operation is the same for a file and for a directory.

First `bdfs_read` function receives the asynchronous read request and gets this requested size. This is handled by the `async_data_read_receive` function.

If the file index is `BDFS_DEVICE`, then the block size and the number of blocks are determined by functions `block_get_bsize` and `block_get_nblocks`. While using these variables the offset in the first block and its address are calculated from the requested position in the file.

As a next step this function loads the block by the `block_get` function. The requested data from the blocks are read by the `async_data_read_finalize` function from the address `block->data + offset_in_block`. Only the remaining data in the block are read. This amount of data can be smaller than the requested size.

If that is the case, the reader will repeat the reading at the beginning of the next block.

It would be possible to read all the requested data in more blocks at once but that would require allocating special buffer and copying all the data there.

Index BDFS_ROOT means reading the root directory. Only the name of the device file is read.

bdfs_write function

This function first receives the asynchronous read request and gets the requested size of the data to be written. This is handled by `async_data_write_receive` function.

The block size and the number of blocks are determined by functions `block_get_bsize` and `block_get_nblocks`. Using these variables the offset in the first block and its address are calculated from the requested write position in the file.

After that this function loads the block by the `block_get` function. The previous data in the block are read only when the write position is in the middle of the block or when the size of the data is smaller then the block size.

The data to be written to the file are written to the address `block->data + offset_in_block`. Only the remaining data in the block are overwritten, which can be less than the size of the data to be written to the file. If this is the case, the writer will repeat the writing at the beginning of the next block.

Similarly, as in reading, it would be possible to write all the requested data in more blocks at once but that would require allocating a special buffer and copying all the data there.

The `block->dirty` flag is set and data are written to the block device by the `block_put` function.

3.4.2 VFS output interface in block device drivers

This section describes the source code for the block device access solution variant which was considered in the analysis. This solution implements VFS output interface directly into block device drivers. The properties of this solution are described in the section 2.6.3. It is not a part of the final source code because other solution was chosen as a better one. A Patch against `helenos_fuse` is in `helenos_fuse_bd_alt.patch`.

The main reason to include this section is to show how complicated this solution actually is in comparison with the block device file system (described in the section 2.6.2). This solution was previously chosen as the most promising one but big complications appeared during the implementation.

VFS and block device interface differences

In this solution the block device driver needs to provide both the VFS output interface and the block device interface. The VFS output interface is for serving FUSE file system drivers and the block device interface is made up to serve the native HelenOS file system drivers.

The main difference between these interfaces is in the initialization of the connection. The native file system driver makes the first connection to the block device driver when the file system is being mounted. For a FUSE file system driver, the block device driver connects to the VFS server even before the device is being mounted.

The first complication comes with the callback connection from the block device to the native block device driver. This callback connection is established after the entry to the block device driver function, which handles incoming IPC connections. It is either necessary to determine whether the connection comes from the VFS server and then skip creating the callback connection. The second variant is to remove this callback connection entirely. Currently this callback connection is not used for anything, so removing the callback connection is possible and the patch includes it.

The second complication comes from the fact that block device drivers determine which device (identified as `service_id` variable in the sources) to use before the `bd_conn` function is called. `*srvs` parameter of the `bd_conn` function includes the device identification. This either needs to be completely changed or the block device driver needs to register each device as a new service to the VFS server. Separate registrations would lead to a complicated code. Also the VFS server does not expect multiple registrations from the same file system driver.

Some block device drivers use the Device Driver Framework. Solving the previous complications is even more complicated within the Device Driver Framework.

3.4.3 Other patch notes

The patch does not include any block caching or block synchronization during writing. The functions which handle VFS operations call directly the block device operations. This is another important complication in this solution.

The functions which handle VFS operations in the patch use the global variable `global_srv` to access block device operations. This is just a "hack" to make it work quickly during the analysis of the solutions. It would need to be changed in the final solution. This limits the block device driver to only one instance (device).

`file_bd` file system is the only file system which uses the solution in the patch. It is a file system which converts a file with a disk image to a block device. Limitation by the global variable `global_srv` does not matter here because `file_bd` has only one instance.

The patch includes changes in the block device library `bd.c`, block device driver skeleton `bd_srv.c` and changes in the `file_bd` file system.

3.5 Other necessary changes in HelenOS

During implementation it was necessary to make some changes in the HelenOS source code. Almost all of them are improvements which can be easily integrated in the HelenOS mainline without causing any harm.

3.5.1 HelenOS and POSIX return codes

It was necessary to use both POSIX and HelenOS native error codes in the file `fuse_lowlevel.c`. Definitions of POSIX error codes overwrite the native ones in POSIX programs in HelenOS.

So it was necessary to introduce other names for these native error codes.

The native error codes are now also accessible in the POSIX applications with the `NATIVE_` prefix.

3.5.2 Open error in libfs library

This thesis uncovered a bug in the lookup function in the libfs library.

Among other things the lookup function manages the opening of directories. The lookup function in libfs did not call libfs operation `node_open` when `lflag = L_OPEN | L_CREATE`. These flags represent an opening of a new file.

This bug was not found earlier because most file systems have a stateless open where the `node_open` operation does nothing. It has been unnoticed for a few years.

3.5.3 Pread and pwrite functions

The FUSE drivers use `pwrite` and `pread` functions for accessing block devices. Those functions were not implemented in HelenOS. New VFS input operations VFS_IN_PREAD and VFS_IN_PWRITE were introduced in order for these functions to work. The only difference between them and VFS_IN_READ and VFS_IN_WRITE consists of having another parameter: offset in a file.

For the simplicity of the implementation HelenOS does not read or write the whole buffer even when there is no end of file. This feature causes a problem because some FUSE drivers expect full buffer usage. For solving this problem POSIX `pwrite` and `pread` were mapped to `pwrite_all` and `pread_all` versions. These functions call the native `pwrite` and `pread` more times until the whole buffer is used.

3.5.4 POSIX prefix defines collision

During the development there were problems with collisions with POSIX definitions. This was caused by the fact that FUSE operations are implemented as structure members. Unfortunately they had sometimes the same name as POSIX functions. And POSIX functions were implemented like `#define read posix_read`. The code was requesting different inexistent structure members (instead of `read` it requested `posix_read`).

The pushing and poping of those definitions was used as a workaround. Later this was solved directly in the HelenOS mainline by Vojtěch Horky by overwriting function names at link time and this problem vanished.

3.5.5 Comparison between native and FUSE drivers on HelenOS

The connector adds another layer of code. FUSE file system drivers are therefore a bit slower than the native file system drivers on HelenOS.

The connector is adding a data copying in the `fuse_reply_buf` and `fuse_reply_data`. The reason for this is described in the section 3.1.2.

The directory reading is also not completely optimized. See section 2.4.

Another difference is connected with the access to block devices. It is necessary to mount the block device using block device file system first. But not all of the FUSE drivers use that. For example archive or network FUSE file systems does not access block devices at all.

The advantage of using all existing FUSE drivers is much greater then above mentioned disadvantages against the native file system drivers.

3.6 Development using distributed version control system

The distributed version control system and the public repository[24] were used for the development of VFS-FUSE connector. This was a great help during the development and the merge process with HelenOS mainline repository[3]. It will also help future merging of this work to the mainline repository.

Chapter 4

Ported FUSE file systems

This chapter describes the FUSE file systems which were ported to the HelenOS as part of this thesis. The porting of real file systems proves that the work on the connector was successful and adds new features to the HelenOS operating system.

4.1 ExFAT

ExFAT[19] is a file system from Microsoft. ExFAT is optimized for flash drives.

Free exFAT file system implementation[18] was ported to HelenOS at the same time as the connector was developed. This implementation uses the FUSE high level interface.

The exFAT FUSE driver can be found in the userspace part of HelenOS sources: `uspace/srv/fs/fuse/exfat/`. It also uses the exFAT library which can be found in `uspace/lib/posix/libexfat/`.

Free exFAT file system implementation has an operation system specific section for the detection of endianness and byte swapping. During the porting of the exFAT driver it was necessary to add a specific HelenOS section to this driver for handling a detection of endianness and byte swapping.

The exFAT FUSE driver uses `pread` and `pwrite` POSIX functions (section3.5.3) to access the block device. The implementation of these functions was added to HelenOS in this thesis in order for exFAT to work. Accessing real block devices is solved by the block device file system. The block device file system is described in the section 3.4.1.

One problem still remains: It is not possible to rename files on exFAT. VFS server does not have `VFS_OUT_RENAME` operation and instead of this just calls `link(new_name)` and then `unlink(old_name)`. This means that for a moment there are at least 2 links to the file which is being renamed. However, exFAT does not support more then one link and therefore the rename operation fails. This issue is not specific for FUSE file system drivers on HelenOS. It is also present in other native HelenOS file system drivers which cannot handle more than one link to a file.

HelenOS also has a native file system driver for exFAT. The same file system was selected for porting because it is possible to test a file system image in another working implementation of the same file system. This helped the during development of the VFS-FUSE connector.

4.2 NTFS

NTFS[21] is a File System developed by Microsoft. NTFS means "New Technology File System". This file system is used for writing and reading files on hard disks of computers that are managed by Windows operating systems. NT File System was developed to serve new needs that were not possible to be maintained by the old Microsoft FAT system. As the main reason was working with very large files. The NTFS is a largely used file system mainly on personal computers and windows servers.

The NTFS-3G FUSE driver[20] was ported to HelenOS as part of this thesis. Its full name is "Third Generation Read/Write NTFS Driver". This driver is available for free under the GNU General Public Licence. It has now a stable version and has all features needed for working with HelenOS via FUSE. The NTFS-3G driver includes all necessary operations for working with files: create, read, modify, rename, move and delete. These operations can be performed on NTFS partitions.

The NTFS-3G driver comes in two variants. The first variant uses the high level FUSE interface. The second variant uses the low level FUSE interface. The low level variant was ported to HelenOS.

The NTFS-3G driver can be found in `uspace/srv/fs/fuse/ntfs/` directory.

An important feature is that the NTFS-3G FUSE driver supports hard links. By means of this it is possible to have more than one link to a file. This means that unlike by the ExFAT FUSE driver, it is possible to move files by the NTFS FUSE driver.

The NTFS-3G FUSE driver ignores the offset (`off`) argument of the FUSE low level operation. This ignoring means that it is not possible to call the `readdir` FUSE low level operation more than once with the same offset. The HelenOS VFS output `read` operation saves the directory buffer in the hash table with the information about the opened file. Saving this buffer allows calling the `readdir` FUSE low level operation only once for multiple VFS output `read` calls.

The NTFS-3G FUSE driver uses `pread` and `pwrite` POSIX functions (section 3.5.3) to access the block device. Accessing real block devices is solved by the block device file system. The block device file system is described in the section 3.4.1.

No changes were necessary to be made in the source code of the NTFS-3G FUSE driver. The only two files specifically written for HelenOS port of this driver are custom `config.h` and `Makefile` files.

4.3 Examples from FUSE package

The Linux FUSE package includes some example file systems. Some of them were ported during VFS to FUSE connector development as the first working FUSE file system drivers. They were easy enough for debugging during the beginning of the connector development.

4.3.1 Hello world in high level interface

This is the simplest example file system which demonstrates using of the FUSE high level API. It can be found in `uspace/srv/fs/fuse/hello/` directory.

4.3.2 Hello world in low level interface

This is the simplest example file system which demonstrates using of the FUSE low level API. It was the first working FUSE file system in HelenOS. It can be found in `uspace/srv/fs/fuse/hello_ll/` directory.

4.4 Estimation of difficulty to port other file systems

The main problem when porting FUSE file systems to HelenOS is that most FUSE file systems depend on some other library. This is especially the case of pseudo file systems (for example an access to archive) or network file systems (for example sshfs). It is somewhat harder to port these libraries to HelenOS because of the limited POSIX support. Multiple POSIX files and functions were necessary to be added for both the exFAT and the NTFS FUSE drivers.

Some FUSE file system drivers work as a layer over another file system. This means that they store some data in another file system. HelenOS VFS server has a problem with namespace read-write lock. This lock locks too much code and it causes deadlock in some of these file systems. The only exception is when the file in other file system is opened during a file system initialization. The solution to this problem is necessary to be done before porting them to HelenOS. There is an issue [22] about this deadlock in the HelenOS issue tracker.

There is no complication with accessing real block devices. The access is solved by the block device file system (section 3.4.1).

Some FUSE file systems need to add HelenOS specific section in them. For example for a byte swapping and a changing endian ordering.

file system	porting problems
MP3FS	VFS server deadlock, libraries
Ramfuse	VFS server deadlock, PERL
squashfuse	libraries, OS specific
CryptoFS	VFS server deadlock, libraries
LoggedFS	VFS server deadlock
SshFS	libraries
ZFS	OS specific
gitfs	VFS server deadlock, libraries

Table 4.1: Porting other FUSE file systems

The table 4.1 shows an estimated problems of the porting difficulty for some popular file systems from the FUSE file systems list ([7]).

Conclusion

The goal of this master thesis was to design and implement the connector between FUSE file system drivers and HelenOS native VFS interface. The goal of finding the solution and consequently the development of the implementation of the connector was achieved.

The important part of this work was the decision how to implement the connector. The selected decision to implement connection at low level layer has proved as a very good choice. This allowed reusing of a great portion of code from Linux FUSE implementation. Practically no changes were necessary to be made in the FUSE file system drivers. In fact almost all of those changes do not relate to FUSE but to the limited POSIX libraries in HelenOS.

The implementation of the connector at low level layer also allowed the use of the same libraries as the native HelenOS file system drivers use and therefore no changes in HelenOS VFS server were necessary to be done. This fact will make the future development of the HelenOS VFS easier because there will be no need to have FUSE in mind.

As part of this thesis the block device file system was created. It allows FUSE to access real block devices as files.

Also the FUSE variant of exFAT file system driver was ported to HelenOS. This did not add new features because HelenOS already has the native exFAT file system driver. This was intentional for help in the development.

The implementation was widened to port another file system: NTFS. The NT File System is one of the most used in working with data stored on disk drives. This adds a new practical feature to the HelenOS operating system because the HelenOS does not have the native NTFS driver.

The performance and speed was not the goal of this thesis. This is similar to the concept of the rest of the HelenOS operating system. So no speed tests were performed.

Future work

The current implementation provides a solid base for the direct use of FUSE file system drivers in HelenOS. Although several FUSE features can be developed at a deeper level and optimization. This includes for example multi-threaded drivers and the `readdir` operation.

Introducing mount scripts will enable to mount FUSE file systems in the same way as the native file systems do.

The block device file system can work more efficiently by caching the block size and the block count. Also the block device file system can support virtual blocks as an option.

Some FUSE file system drivers work as a layer over another file system. The solution to the problem with namespace read-write lock [22] is necessary to be fixed before porting them to HelenOS.

Last but not least, some work will be necessary to port other FUSE file system drivers to HelenOS.

Bibliography

[1] *HelenOS*, `http://www.helenos.org/`

[2] *HelenOS documentations*, `http://www.helenos.org/documentation`

[3] *HelenOS sources*, `http://www.helenos.org/sources`

[4] Jakub Jermář: *Implementation of file system in HelenOS operating system*, `http://www.helenos.org/doc/papers/HelenOS-EurOpen.pdf`

[5] Andrew S. Tanenbaum: *Modern Operating Systems*, 3rd edition, ISBN 0-13-813459-6978-0-13-813459-4

[6] *Filesystem in Userspace*, `http://fuse.sourceforge.net/`

[7] *File systems using FUSE*, `http://sourceforge.net/p/fuse/wiki/FileSystems/`

[8] *FUSE structure image*, `http://en.wikipedia.org/wiki/File:FUSE_structure.svg`

[9] *Operating Systems - FUSE*, `http://sourceforge.net/p/fuse/wiki/OperatingSystems/`

[10] *FUSE for OS X*, `http://osxfuse.github.io/`

[11] *FUSE for OS X FAQ*, `https://github.com/osxfuse/osxfuse/wiki/FAQ`

[12] *ReFUSE (NetBSD)*, `http://netbsd.gw.com/cgi-bin/man-cgi?refuse+3+NetBSD-6.0`

[13] *PUFFS Enabled Relay to FUSE Daemon (NetBSD)*, `http://netbsd.gw.com/cgi-bin/man-cgi?perfused+8+NetBSD-6.0`

[14] *PUFFS enabled relay to FUSE Library (NetBSD)*, `http://netbsd.gw.com/cgi-bin/man-cgi?libperfuse++NetBSD-6.0`

[15] *FreeBSD FUSE module*, `https://wiki.freebsd.org/FuseFilesystem`

[16] Jiří Svoboda: discussion about FUSE in Solaris, `http://lists.modry.cz/private/helenos-devel/2012-June/005773.html`

[17] *IPC for Dummies*, `http://trac.helenos.org/wiki/IPC`

[18] *Free exFAT file system implementationFree exFAT file system implementation*, `https://code.google.com/p/exfat/`

[19] *exFAT File System*, `http://www.microsoft.com/en-us/legal/intellectualproperty/IPLicensing/Programs/exFATFileSystem.aspx`

[20] *NTFS-3G*, `http://www.tuxera.com/community/ntfs-3g-download/`

[21] *The NTFS File System*, `http://technet.microsoft.com/en-us/library/cc976808.aspx`

[22] *VFS deadlock ticket*, `http://trac.helenos.org/ticket/480`.

[23] *QEMU machine emulator and virtualizer*, `http://qemu.org`

[24] *Development brach in Launchpad*, `https://code.launchpad.net/%7ezdenek-bouska/helenos/fuse`

List of Tables

Appendices

Appendix A

CD-ROM content

This thesis includes a CD-ROM medium on which you will find:

- **HelenOS sources** with VFS-FUSE connector inside in the tar archive called `helenos_fuse.tgz`
- **HelenOS bootable CD** image `image.iso`
- **NTFS** file system image `ntfs.img`
- **ExFAT** file system image `exfat.img`
- **README** a readme text file, reading it is recommended.
- **Qemu wrapper script** `run.sh` starts HelenOS in Qemu[23] emulator.
- **An electronic version of this thesis** in the file `thesis.pdf`.
- **Patch** with alternative access to block devices in the file `helenos_fuse_bd_alt.patch`.

Appendix B

User Documentation

As a start point of the installation copy the CD content to hard disk. This step is necessary in order to be able to mount for writing.

The easy way how to run HelenOS operating system is Qemu[23]. It emulates the whole PC and is the recommended emulator for HelenOS. You can do this by starting Qemu by the run script `run.sh`.

B.1 ExFAT

Start Qemu by the run script `run.sh`. As a hard disc parameter specify the exFAT block device image.

```
./run.sh -hda exfat.img
```

Next start the block device file system:

```
bdfs
```

Change the current directory to `loc` (this steps only enables command line completion of the ATA block device name):

```
cd /loc
```

Mount the block device file system:

```
mount bdfs /bd devices/\hw\pci0\00:01.0\ata-c1\d0
```

Now mount the FUSE exFAT driver:

```
fuse_exfat /bd/dev /mnt
```

Access the files in the mounted exFAT file system.

```
ls /mnt
```

Unmount the FUSE exFAT file system:

```
umount /mnt
```

Unmount the block device file system:

```
umount /bd
```

B.2 NTFS

Start Qemu by the run script `run.sh`. As a hard disc parameter specify the NTFS block device image.

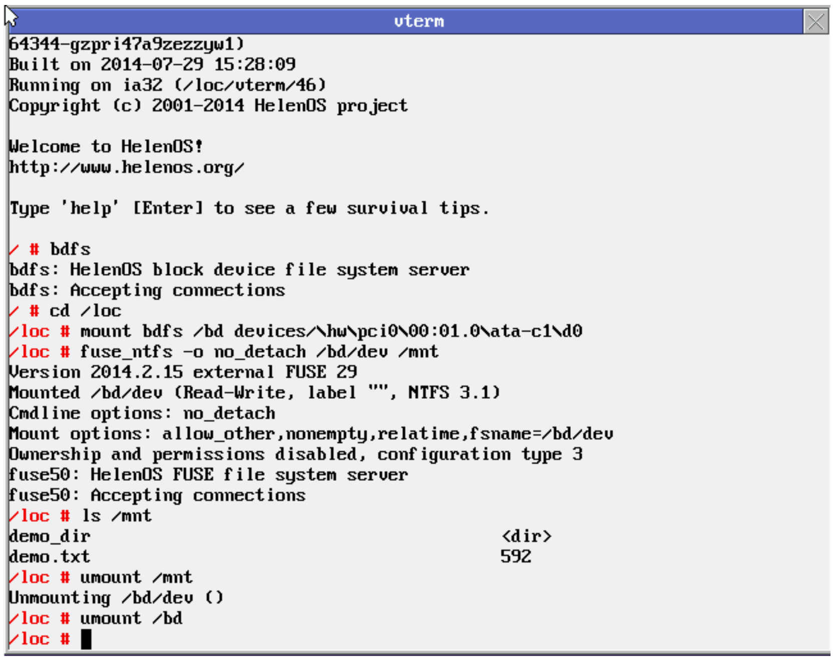
```
./run.sh -hda ntfs.img
```

Next start the block device file system:

```
bdfs
```

Change the current directory to `loc` (this steps only enables command line completion of the ATA block device name):

```
cd /loc
```



```
64344-gzpri47a9zezzyw1)
Built on 2014-07-29 15:28:09
Running on ia32 (/loc/vterm/46)
Copyright (c) 2001-2014 HelenOS project

Welcome to HelenOS!
http://www.helenos.org/

Type 'help' [Enter] to see a few survival tips.

/ # bdfs
bdfs: HelenOS block device file system server
bdfs: Accepting connections
/ # cd /loc
/loc # mount bdfs /bd devices/\hw\pci0\00:01.0\ata-c1\d0
/loc # fuse_ntfs -o no_detach /bd/dev /mnt
Version 2014.2.15 external FUSE 29
Mounted /bd/dev (Read-Write, label "", NTFS 3.1)
Cmdline options: no_detach
Mount options: allow_other,nonempty,relatime,fsname=/bd/dev
Ownership and permissions disabled, configuration type 3
fuse50: HelenOS FUSE file system server
fuse50: Accepting connections
/loc # ls /mnt
demo_dir                                        <dir>
demo.txt                                        592
/loc # umount /mnt
Unmounting /bd/dev ()
/loc # umount /bd
/loc # █
```

Figure B.1: Screenshot of FUSE NTFS file system usage

Mount the block device file system:

```
mount bdfs /bd devices/\hw\pci0\00:01.0\ata-c1\d0
```

Now mount the FUSE NTFS driver:
(`no_detach` option just avoids warning by unsupported fork() POSIX function)

```
fuse_ntfs -o no_detach /bd/dev /mnt
```

Access the files in the mounted exFAT file system.

```
ls /mnt
```

Unmount the FUSE NTFS file system:

```
umount /mnt
```

Unmount the block device file system:

```
umount /bd
```

You can see this at screenshot in the figure B.1.

B.3 Compiling from sources

The Linux operating system is recommended for compiling. First you need to unpack the sources:

```
tar -zxvf helenos_fuse.tgz
```

and then move to the newly created directory

```
cd helenos_fuse
```

Next install the development toolchain. Specific versions of the compiler and binutils are necessary to compile HelenOS. The toolchain has dependencies. Most of them are listed when you run the toolchain. In order to save time install these dependencies first. The toolchain will be installed to the directory specified by the `CROSS_PREFIX` environment variable. If the variable is not defined, `/usr/local/cross` will be used by default.

```
./tools/toolchain.sh ia32
```

After that run

```
make
```

and in the configurator select

```
--- Load preconfigured defaults ...
```

```
ia32
```

```
Done
```

For cleaning after the compilation you can use

```
make clean
```

or for cleaning the configuration as well

```
make distclean
```

Appendix C

List of files

This appendix lists the files which were added to or modified in the HelenOS source code during the work on this thesis.

The list is divided by the features they provide.

C.1 FUSE high and low level library

New files

```
uspace/lib/posix/fuse/include/config.h
uspace/lib/posix/fuse/lib/fuse_kern_chan.c
uspace/lib/posix/fuse/lib/fuse_lowlevel.c
uspace/lib/posix/fuse/lib/fuse_mt.c
uspace/lib/posix/fuse/lib/fuse_session.c
uspace/lib/posix/fuse/lib/fuse_signals.c
uspace/lib/posix/fuse/lib/mount_dummy.c
uspace/lib/posix/fuse/Makefile
uspace/lib/posix/fuse/README
```

Modified files

```
uspace/lib/fs/libfs.c
uspace/lib/fs/libfs.h
uspace/Makefile
uspace/Makefile.common
```

Files taken from the Linux FUSE library and modified

```
uspace/lib/posix/fuse/include/fuse_common.h
uspace/lib/posix/fuse/lib/buffer.c
```

```
uspace/lib/posix/fuse/lib/fuse.c
uspace/lib/posix/fuse/lib/fuse_i.h
uspace/lib/posix/fuse/lib/helper.c
```

Files taken from the Linux FUSE library

```
uspace/lib/posix/fuse/include/fuse_common_compat.h
uspace/lib/posix/fuse/include/fuse_common.h
uspace/lib/posix/fuse/include/fuse.h
uspace/lib/posix/fuse/include/fuse_kernel.h
uspace/lib/posix/fuse/include/fuse_lowlevel_compat.h
uspace/lib/posix/fuse/include/fuse_lowlevel.h
uspace/lib/posix/fuse/include/fuse_opt.h
uspace/lib/posix/fuse/lib/fuse_misc.h
uspace/lib/posix/fuse/lib/fuse_opt.c
```

C.2 New VFS server input operations (pread and pwrite)

Modified files

```
uspace/app/trace/trace.c
uspace/lib/c/generic/vfs/vfs.c
uspace/lib/c/include/ipc/vfs.h
uspace/lib/c/include/unistd.h
uspace/srv/vfs/vfs.c
uspace/srv/vfs/vfs.h
uspace/srv/vfs/vfs_ops.c
```

C.3 Block device file system

New files

```
uspace/srv/fs/bdfs/bdfs.c
uspace/srv/fs/bdfs/bdfs.h
uspace/srv/fs/bdfs/bdfs_ops.c
uspace/srv/fs/bdfs/Makefile
```

Modified files

```
.bzrignore
bootuspace/Makefile.common
uspace/Makefile
```

C.4 New features of the POSIX library

New files

```
uspace/lib/posix/include/posix/endian.h
uspace/lib/posix/include/posix/grp.h
uspace/lib/posix/include/posix/poll.h
uspace/lib/posix/include/posix/sys/ioctl.h
uspace/lib/posix/include/posix/sys/mount.h
uspace/lib/posix/include/posix/sys/param.h
uspace/lib/posix/include/posix/sys/statvfs.h
uspace/lib/posix/include/posix/sys/uio.h
uspace/lib/posix/include/posix/syslog.h
uspace/lib/posix/include/posix/utime.h
uspace/lib/posix/source/grp.c
uspace/lib/posix/source/poll.c
uspace/lib/posix/source/sys/ioctl.c
uspace/lib/posix/source/sys/types.c
uspace/lib/posix/source/syslog.c
```

Modified files

```
uspace/lib/c/generic/dlfcn.c
uspace/lib/c/include/dlfcn.h
uspace/lib/posix/include/posix/pthread.h
uspace/lib/posix/include/posix/stdlib.h
uspace/lib/posix/include/posix/string.h
uspace/lib/posix/include/posix/sys/stat.h
uspace/lib/posix/include/posix/sys/types.h
uspace/lib/posix/include/posix/time.h
uspace/lib/posix/include/posix/unistd.h
uspace/lib/posix/include/posix/errno.h
uspace/lib/posix/include/posix/fcntl.h
uspace/lib/posix/Makefile
uspace/lib/posix/source/fcntl.c
uspace/lib/posix/source/pthread/condvar.c
uspace/lib/posix/source/pthread/keys.c
uspace/lib/posix/source/pthread/mutex.c
```

```
uspace/lib/posix/source/pthread/threads.c
uspace/lib/posix/source/stdlib.c
uspace/lib/posix/source/unistd.c
```

C.5 ExFAT, NTFS and other FUSE file system drivers and libraries

New files

```
uspace/lib/posix/libexfat/Makefile
uspace/srv/fs/fuse/exfat/Makefile
uspace/srv/fs/fuse/hello/Makefile
uspace/srv/fs/fuse/hello_ll/Makefile
uspace/srv/fs/fuse/Makefile.common
uspace/srv/fs/fuse/null/Makefile
uspace/srv/fs/fuse/xmp/Makefile
```

Modified files

```
.bzrignore
bootuspace/Makefile.common
uspace/lib/c/arch/sparc64/Makefile.common
uspace/Makefile
uspace/Makefile.common
```

Files taken from the original driver and modified

```
uspace/lib/posix/libexfat/byteorder.h
uspace/lib/posix/libexfat/mount.c
uspace/lib/posix/Makefile
```

Files taken from the original driver

```
uspace/lib/posix/libexfat/cluster.c
uspace/lib/posix/libexfat/exfatfs.h
uspace/lib/posix/libexfat/exfat.h
uspace/lib/posix/libexfat/io.c
uspace/lib/posix/libexfat/log.c
uspace/lib/posix/libexfat/lookup.c
uspace/lib/posix/libexfat/node.c
```

```
uspace/lib/posix/libexfat/README
uspace/lib/posix/libexfat/time.c
uspace/lib/posix/libexfat/utf.c
uspace/lib/posix/libexfat/utils.c
uspace/lib/posix/libexfat/version.h
uspace/srv/fs/fuse/exfat/main.c
uspace/srv/fs/fuse/exfat/README
uspace/srv/fs/fuse/hello/hello.c
uspace/srv/fs/fuse/hello_ll/hello_ll.c
uspace/srv/fs/fuse/ntfs/aclocal.m4
uspace/srv/fs/fuse/ntfs/AUTHORS
uspace/srv/fs/fuse/ntfs/autogen.sh
uspace/srv/fs/fuse/ntfs/ChangeLog
uspace/srv/fs/fuse/ntfs/compile
uspace/srv/fs/fuse/ntfs/config.guess
uspace/srv/fs/fuse/ntfs/config.h
uspace/srv/fs/fuse/ntfs/config.h.in
uspace/srv/fs/fuse/ntfs/config.sub
uspace/srv/fs/fuse/ntfs/configure
uspace/srv/fs/fuse/ntfs/configure.ac
uspace/srv/fs/fuse/ntfs/COPYING
uspace/srv/fs/fuse/ntfs/COPYING.LIB
uspace/srv/fs/fuse/ntfs/CREDITS
uspace/srv/fs/fuse/ntfs/depcomp
uspace/srv/fs/fuse/ntfs/include/fuse-lite/fuse_common.h
uspace/srv/fs/fuse/ntfs/include/fuse-lite/fuse.h
uspace/srv/fs/fuse/ntfs/include/fuse-lite/fuse_kernel.h
uspace/srv/fs/fuse/ntfs/include/fuse-lite/fuse_lowlevel_compat.h
uspace/srv/fs/fuse/ntfs/include/fuse-lite/fuse_lowlevel.h
uspace/srv/fs/fuse/ntfs/include/fuse-lite/fuse_opt.h
uspace/srv/fs/fuse/ntfs/include/fuse-lite/Makefile.am
uspace/srv/fs/fuse/ntfs/include/fuse-lite/Makefile.in
uspace/srv/fs/fuse/ntfs/include/Makefile.am
uspace/srv/fs/fuse/ntfs/include/Makefile.in
uspace/srv/fs/fuse/ntfs/include/ntfs-3g/acls.h
uspace/srv/fs/fuse/ntfs/include/ntfs-3g/attrib.h
uspace/srv/fs/fuse/ntfs/include/ntfs-3g/attrlist.h
uspace/srv/fs/fuse/ntfs/include/ntfs-3g/bitmap.h
uspace/srv/fs/fuse/ntfs/include/ntfs-3g/bootsect.h
uspace/srv/fs/fuse/ntfs/include/ntfs-3g/cache.h
uspace/srv/fs/fuse/ntfs/include/ntfs-3g/collate.h
uspace/srv/fs/fuse/ntfs/include/ntfs-3g/compat.h
uspace/srv/fs/fuse/ntfs/include/ntfs-3g/compress.h
uspace/srv/fs/fuse/ntfs/include/ntfs-3g/debug.h
uspace/srv/fs/fuse/ntfs/include/ntfs-3g/device.h
uspace/srv/fs/fuse/ntfs/include/ntfs-3g/device_io.h
uspace/srv/fs/fuse/ntfs/include/ntfs-3g/dir.h
uspace/srv/fs/fuse/ntfs/include/ntfs-3g/efs.h
```

```
uspace/srv/fs/fuse/ntfs/include/ntfs-3g/endians.h
uspace/srv/fs/fuse/ntfs/include/ntfs-3g/index.h
uspace/srv/fs/fuse/ntfs/include/ntfs-3g/inode.h
uspace/srv/fs/fuse/ntfs/include/ntfs-3g/layout.h
uspace/srv/fs/fuse/ntfs/include/ntfs-3g/lcnalloc.h
uspace/srv/fs/fuse/ntfs/include/ntfs-3g/logfile.h
uspace/srv/fs/fuse/ntfs/include/ntfs-3g/logging.h
uspace/srv/fs/fuse/ntfs/include/ntfs-3g/Makefile.am
uspace/srv/fs/fuse/ntfs/include/ntfs-3g/Makefile.in
uspace/srv/fs/fuse/ntfs/include/ntfs-3g/mft.h
uspace/srv/fs/fuse/ntfs/include/ntfs-3g/misc.h
uspace/srv/fs/fuse/ntfs/include/ntfs-3g/mst.h
uspace/srv/fs/fuse/ntfs/include/ntfs-3g/ntfstime.h
uspace/srv/fs/fuse/ntfs/include/ntfs-3g/object_id.h
uspace/srv/fs/fuse/ntfs/include/ntfs-3g/param.h
uspace/srv/fs/fuse/ntfs/include/ntfs-3g/realpath.h
uspace/srv/fs/fuse/ntfs/include/ntfs-3g/reparse.h
uspace/srv/fs/fuse/ntfs/include/ntfs-3g/runlist.h
uspace/srv/fs/fuse/ntfs/include/ntfs-3g/security.h
uspace/srv/fs/fuse/ntfs/include/ntfs-3g/support.h
uspace/srv/fs/fuse/ntfs/include/ntfs-3g/types.h
uspace/srv/fs/fuse/ntfs/include/ntfs-3g/unistr.h
uspace/srv/fs/fuse/ntfs/include/ntfs-3g/volume.h
uspace/srv/fs/fuse/ntfs/include/ntfs-3g/xattrs.h
uspace/srv/fs/fuse/ntfs/INSTALL
uspace/srv/fs/fuse/ntfs/install-sh
uspace/srv/fs/fuse/ntfs/libfuse-lite/fuse.c
uspace/srv/fs/fuse/ntfs/libfuse-lite/fuse_i.h
uspace/srv/fs/fuse/ntfs/libfuse-lite/fuse_kern_chan.c
uspace/srv/fs/fuse/ntfs/libfuse-lite/fuse_loop.c
uspace/srv/fs/fuse/ntfs/libfuse-lite/fuse_lowlevel.c
uspace/srv/fs/fuse/ntfs/libfuse-lite/fuse_misc.h
uspace/srv/fs/fuse/ntfs/libfuse-lite/fuse_opt.c
uspace/srv/fs/fuse/ntfs/libfuse-lite/fusermount.c
uspace/srv/fs/fuse/ntfs/libfuse-lite/fuse_session.c
uspace/srv/fs/fuse/ntfs/libfuse-lite/fuse_signals.c
uspace/srv/fs/fuse/ntfs/libfuse-lite/helper.c
uspace/srv/fs/fuse/ntfs/libfuse-lite/Makefile.am
uspace/srv/fs/fuse/ntfs/libfuse-lite/Makefile.in
uspace/srv/fs/fuse/ntfs/libfuse-lite/mount.c
uspace/srv/fs/fuse/ntfs/libfuse-lite/mount_util.c
uspace/srv/fs/fuse/ntfs/libfuse-lite/mount_util.h
uspace/srv/fs/fuse/ntfs/libntfs-3g/acls.c
uspace/srv/fs/fuse/ntfs/libntfs-3g/attrib.c
uspace/srv/fs/fuse/ntfs/libntfs-3g/attrlist.c
uspace/srv/fs/fuse/ntfs/libntfs-3g/bitmap.c
uspace/srv/fs/fuse/ntfs/libntfs-3g/bootsect.c
uspace/srv/fs/fuse/ntfs/libntfs-3g/cache.c
```

```
uspace/srv/fs/fuse/ntfs/libntfs-3g/collate.c
uspace/srv/fs/fuse/ntfs/libntfs-3g/compat.c
uspace/srv/fs/fuse/ntfs/libntfs-3g/compress.c
uspace/srv/fs/fuse/ntfs/libntfs-3g/debug.c
uspace/srv/fs/fuse/ntfs/libntfs-3g/device.c
uspace/srv/fs/fuse/ntfs/libntfs-3g/dir.c
uspace/srv/fs/fuse/ntfs/libntfs-3g/efs.c
uspace/srv/fs/fuse/ntfs/libntfs-3g/index.c
uspace/srv/fs/fuse/ntfs/libntfs-3g/inode.c
uspace/srv/fs/fuse/ntfs/libntfs-3g/lcnalloc.c
uspace/srv/fs/fuse/ntfs/libntfs-3g/libntfs-3g.pc.in
uspace/srv/fs/fuse/ntfs/libntfs-3g/libntfs-3g.script.so.in
uspace/srv/fs/fuse/ntfs/libntfs-3g/logfile.c
uspace/srv/fs/fuse/ntfs/libntfs-3g/logging.c
uspace/srv/fs/fuse/ntfs/libntfs-3g/Makefile.am
uspace/srv/fs/fuse/ntfs/libntfs-3g/Makefile.in
uspace/srv/fs/fuse/ntfs/libntfs-3g/mft.c
uspace/srv/fs/fuse/ntfs/libntfs-3g/misc.c
uspace/srv/fs/fuse/ntfs/libntfs-3g/mst.c
uspace/srv/fs/fuse/ntfs/libntfs-3g/object_id.c
uspace/srv/fs/fuse/ntfs/libntfs-3g/realpath.c
uspace/srv/fs/fuse/ntfs/libntfs-3g/reparse.c
uspace/srv/fs/fuse/ntfs/libntfs-3g/runlist.c
uspace/srv/fs/fuse/ntfs/libntfs-3g/security.c
uspace/srv/fs/fuse/ntfs/libntfs-3g/unistr.c
uspace/srv/fs/fuse/ntfs/libntfs-3g/unix_io.c
uspace/srv/fs/fuse/ntfs/libntfs-3g/volume.c
uspace/srv/fs/fuse/ntfs/libntfs-3g/win32_io.c
uspace/srv/fs/fuse/ntfs/libntfs-3g/xattrs.c
uspace/srv/fs/fuse/ntfs/ltmain.sh
uspace/srv/fs/fuse/ntfs/m4/libtool.m4
uspace/srv/fs/fuse/ntfs/m4/lt obsolete.m4
uspace/srv/fs/fuse/ntfs/m4/ltoptions.m4
uspace/srv/fs/fuse/ntfs/m4/ltsugar.m4
uspace/srv/fs/fuse/ntfs/m4/ltversion.m4
uspace/srv/fs/fuse/ntfs/Makefile
uspace/srv/fs/fuse/ntfs/Makefile.am
uspace/srv/fs/fuse/ntfs/Makefile.in
uspace/srv/fs/fuse/ntfs/missing
uspace/srv/fs/fuse/ntfs/NEWS
uspace/srv/fs/fuse/ntfs/ntfsprogs/attrdef.c
uspace/srv/fs/fuse/ntfs/ntfsprogs/attrdef.h
uspace/srv/fs/fuse/ntfs/ntfsprogs/boot.c
uspace/srv/fs/fuse/ntfs/ntfsprogs/boot.h
uspace/srv/fs/fuse/ntfs/ntfsprogs/cluster.c
uspace/srv/fs/fuse/ntfs/ntfsprogs/cluster.h
uspace/srv/fs/fuse/ntfs/ntfsprogs/list.h
uspace/srv/fs/fuse/ntfs/ntfsprogs/Makefile.am
```

```
uspace/srv/fs/fuse/ntfs/ntfsprogs/Makefile.in
uspace/srv/fs/fuse/ntfs/ntfsprogs/mkntfs.8.in
uspace/srv/fs/fuse/ntfs/ntfsprogs/mkntfs.c
uspace/srv/fs/fuse/ntfs/ntfsprogs/ntfscat.8.in
uspace/srv/fs/fuse/ntfs/ntfsprogs/ntfscat.c
uspace/srv/fs/fuse/ntfs/ntfsprogs/ntfscat.h
uspace/srv/fs/fuse/ntfs/ntfsprogs/ntfsck.c
uspace/srv/fs/fuse/ntfs/ntfsprogs/ntfsclone.8.in
uspace/srv/fs/fuse/ntfs/ntfsprogs/ntfsclone.c
uspace/srv/fs/fuse/ntfs/ntfsprogs/ntfscluster.8.in
uspace/srv/fs/fuse/ntfs/ntfsprogs/ntfscluster.c
uspace/srv/fs/fuse/ntfs/ntfsprogs/ntfscluster.h
uspace/srv/fs/fuse/ntfs/ntfsprogs/ntfscmp.8.in
uspace/srv/fs/fuse/ntfs/ntfsprogs/ntfscmp.c
uspace/srv/fs/fuse/ntfs/ntfsprogs/ntfscp.8.in
uspace/srv/fs/fuse/ntfs/ntfsprogs/ntfscp.c
uspace/srv/fs/fuse/ntfs/ntfsprogs/ntfsdecrypt.c
uspace/srv/fs/fuse/ntfs/ntfsprogs/ntfsdump_logfile.c
uspace/srv/fs/fuse/ntfs/ntfsprogs/ntfsfix.8.in
uspace/srv/fs/fuse/ntfs/ntfsprogs/ntfsfix.c
uspace/srv/fs/fuse/ntfs/ntfsprogs/ntfsinfo.8.in
uspace/srv/fs/fuse/ntfs/ntfsprogs/ntfsinfo.c
uspace/srv/fs/fuse/ntfs/ntfsprogs/ntfslabel.8.in
uspace/srv/fs/fuse/ntfs/ntfsprogs/ntfslabel.c
uspace/srv/fs/fuse/ntfs/ntfsprogs/ntfsls.8.in
uspace/srv/fs/fuse/ntfs/ntfsprogs/ntfsls.c
uspace/srv/fs/fuse/ntfs/ntfsprogs/ntfsmftalloc.c
uspace/srv/fs/fuse/ntfs/ntfsprogs/ntfsmove.c
uspace/srv/fs/fuse/ntfs/ntfsprogs/ntfsmove.h
uspace/srv/fs/fuse/ntfs/ntfsprogs/ntfsprogs.8.in
uspace/srv/fs/fuse/ntfs/ntfsprogs/ntfsresize.8.in
uspace/srv/fs/fuse/ntfs/ntfsprogs/ntfsresize.c
uspace/srv/fs/fuse/ntfs/ntfsprogs/ntfstruncate.c
uspace/srv/fs/fuse/ntfs/ntfsprogs/ntfsundelete.8.in
uspace/srv/fs/fuse/ntfs/ntfsprogs/ntfsundelete.c
uspace/srv/fs/fuse/ntfs/ntfsprogs/ntfsundelete.h
uspace/srv/fs/fuse/ntfs/ntfsprogs/ntfswipe.c
uspace/srv/fs/fuse/ntfs/ntfsprogs/ntfswipe.h
uspace/srv/fs/fuse/ntfs/ntfsprogs/sd.c
uspace/srv/fs/fuse/ntfs/ntfsprogs/sd.h
uspace/srv/fs/fuse/ntfs/ntfsprogs/utils.c
uspace/srv/fs/fuse/ntfs/ntfsprogs/utils.h
uspace/srv/fs/fuse/ntfs/README
uspace/srv/fs/fuse/ntfs/src/lowntfs-3g.c
uspace/srv/fs/fuse/ntfs/src/Makefile.am
uspace/srv/fs/fuse/ntfs/src/Makefile.in
uspace/srv/fs/fuse/ntfs/src/ntfs-3g.8.in
uspace/srv/fs/fuse/ntfs/src/ntfs-3g.c
```

```
uspace/srv/fs/fuse/ntfs/src/ntfs-3g_common.c
uspace/srv/fs/fuse/ntfs/src/ntfs-3g_common.h
uspace/srv/fs/fuse/ntfs/src/ntfs-3g.probe.8.in
uspace/srv/fs/fuse/ntfs/src/ntfs-3g.probe.c
uspace/srv/fs/fuse/ntfs/src/ntfs-3g.secaudit.8.in
uspace/srv/fs/fuse/ntfs/src/ntfs-3g.usermap.8.in
uspace/srv/fs/fuse/null/null.c
uspace/srv/fs/fuse/xmp/fusexmp.c
```