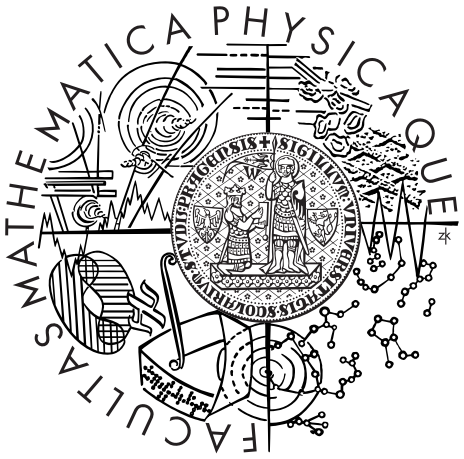
Charles University in Prague

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.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague April 11, 2014 Zdeněk Bouška

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 reálně používaný FUSE souborový systém exFAT 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 a real-life FUSE file system exFAT 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 the 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 kernel and servers IPC communication. It also includes information about file systems in the HelenOS operating system and describes how it works there. 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 will be carefully considered. The more detailed view is focused on the connection layer selection, whether to choose a high level,

low level or 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 implementation. It shows how results from analysis were transformed into a working connector prototype. Firstly the integration with libfs library deals with the necessary parts such as operations mapping, storing data from mounted file systems and opened files. This chapter 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 and some examples from the Linux FUSE package are described.

Chapter 1

Development context

This chapter includes a summary of the development context and the background which is necessary for understanding of 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 developement 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 a 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

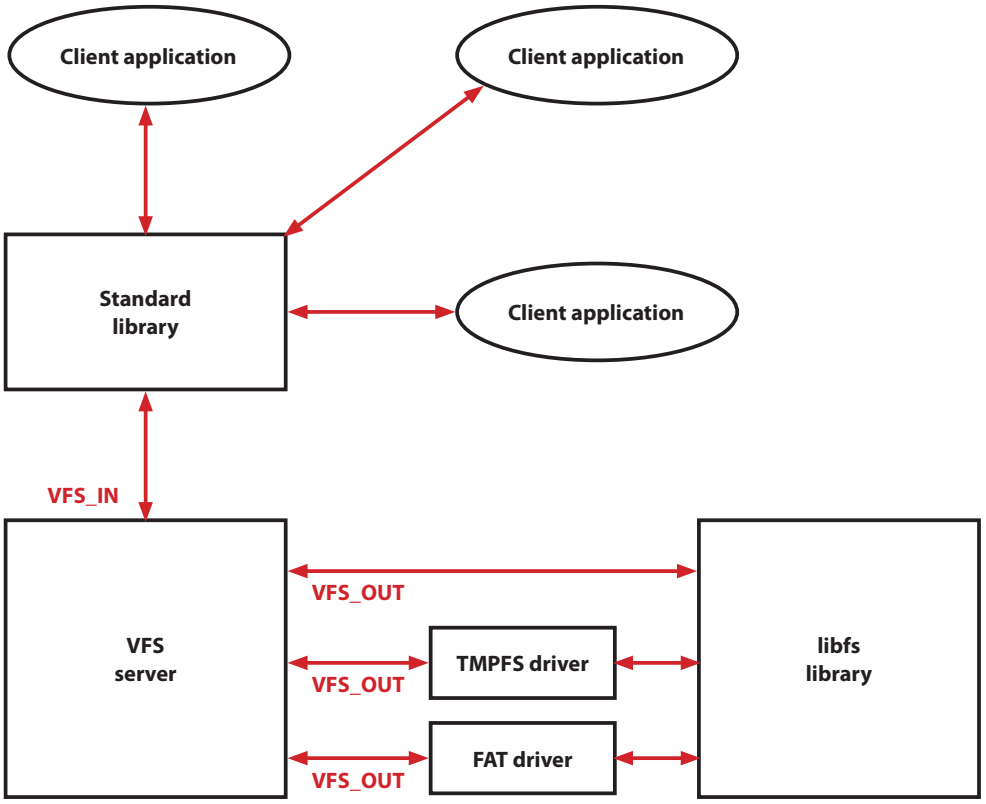


Figure 1.1: Filesystems in HelenOS

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 a 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 a 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 parameter. That 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 a 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 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.

1.3 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 run 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.

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.

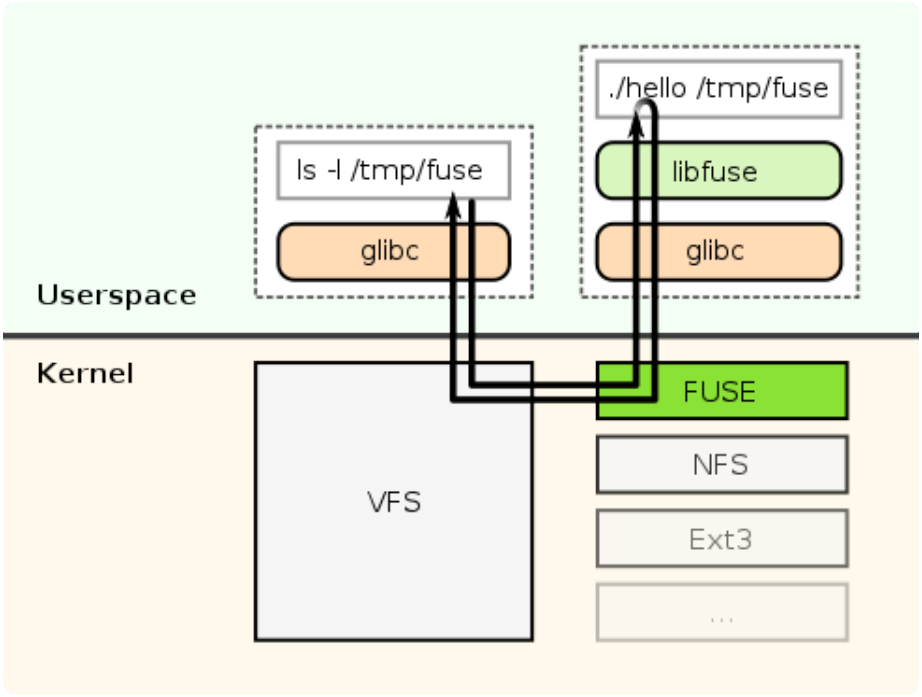


Figure 1.2: Filesystem in Userspace in Linux [8]

1.5 FUSE in other operation systems

FUSE is supported in other operation systems than 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 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 low and 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]. Userspace library has numerous OS X specific extensions and features.[11]

1.5.3 FreeBDS

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. In 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 for the connector between FUSE drivers and HelenOS VFS server.

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 library would convert the FUSE driver to the HelenOS file system driver server. This solution removes the need for changes in the VFS server.

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 section 1.4 there are three interface layers: the kernel channel inteface, the low level inteface and the high level inteface.

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

+	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

2.2.1 High level inteface

The high level inteface 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 rewritting 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 use this choice.

The advantages and disadvantages of 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 a 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 how the FUSE driver works in Linux from its point of view.

Connecting the FUSE file system driver to the VFS server based on the kernel channel interface allows using almost all 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).

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 will work in HelenOS's file system architecture can be seen in the figure 2.1.

2.3 Own 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 FUSE driver needs its own task. Fortunately

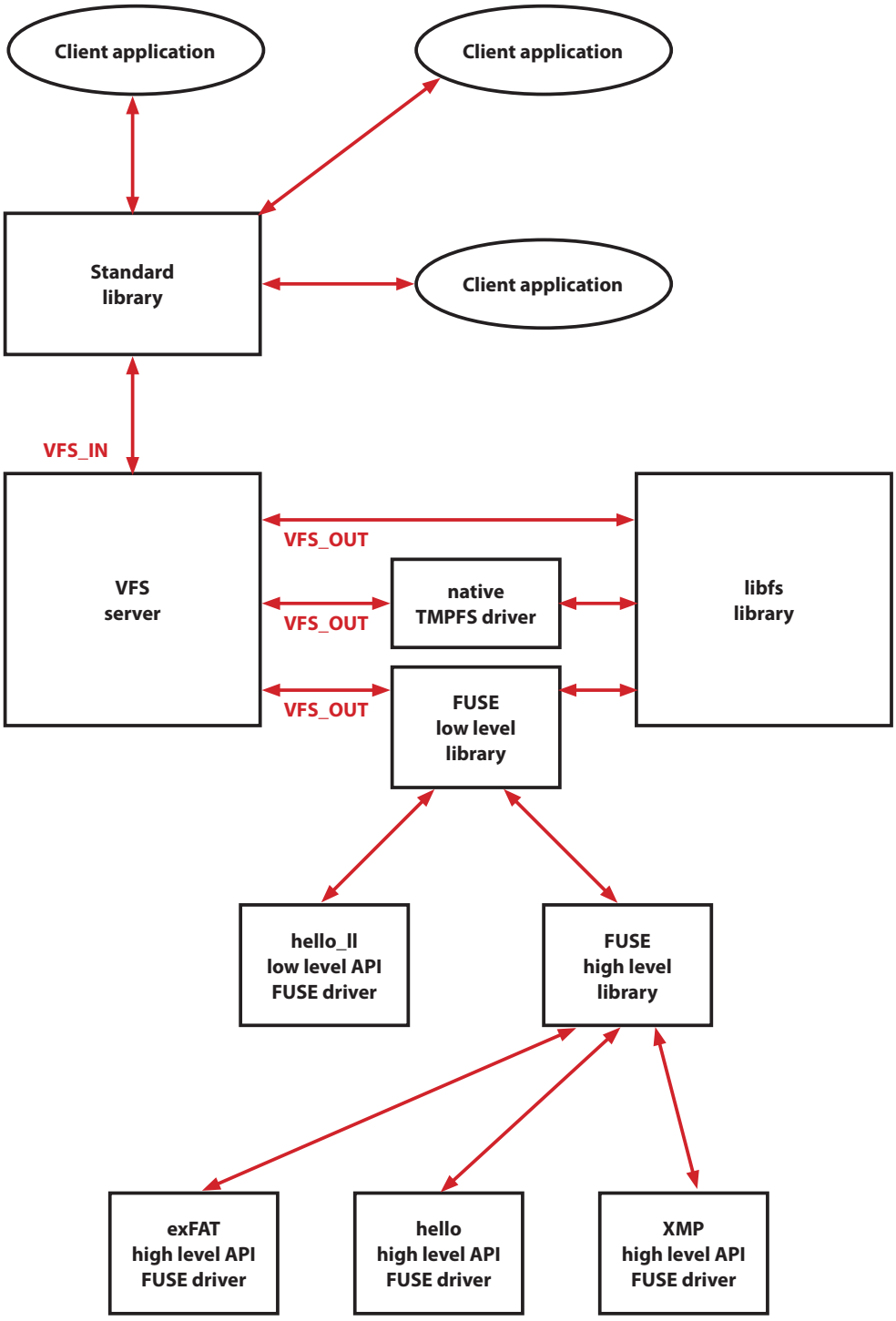


Figure 2.1: FUSE in HelenOS

this feature causes no problem since the new HelenOS file system task can be launched for each FUSE driver instance.

The FUSE drivers also mount 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 later after the FUSE file system is 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.

In the current connector implementation the whole directory is read until the desired file is found.

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 a 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 efficient since the whole directory must be read again for each file. This can be accelerated by caching 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 does not implement either of these caches.

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 behaviour 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 exists it would launch this script instead of sending `VFS_IN_MOUNT` method.

2.6 Accessing block devices

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

Of course 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 again and again. There are three possibilities how to solve this problem without modifying all ported FUSE file system drivers.

2.6.1 POSIX functions overwrite

The native HelenOS applications does not follow the POSIX specification. But there is 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 overwrite POSIX read and write functions. This means adding conditions to the POSIX library. For some prefix it would send read or write requests to block device server instead of sending them to the VFS server.

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 that would enable the access to the block devices via VFS. This file system would have a virtual file for each block device. Accessing this file would result in accessing the block device server.

2.6.3 VFS output protocol support in a block device drivers

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

The console device drivers use a similar solution.

In this case the block device driver servers use shared common skeleton library (`bd_srv.h`). There would be no need to add support of those VFS output operations into every single block device driver. They could be implemented directly in the common skeleton library (in `bd_srv.c`). The VFS output operations would be implemented by a block manipulation functions for reading or writing blocks.

2.6.4 Conclusion

The POSIX function overwrite is probably the easiest way to implement this access though it is not the cleanest way.

The block device file system server is a nice clean solution. The downside is that it adds another layer between the FUSE file system driver and the block device. It is also probably the hardest way for the implementation.

Supporting VFS output protocol in block device drivers is clean and the most efficient way how to solve this problem.

This problem is not the main topic of this thesis so it is not implemented in the connector prototype.

Chapter 3

Implementation

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

3.1 Integration with libfs

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

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_OUT 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 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 `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
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 lowlevel interface

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 neccessary to create 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 this data and then return them using the asynchronous framework.

In some cases more then one FUSE low level operation is called in one libfs or VFS output operation. This is another reason why data can not be send to VFS in the reply function.

3.1.3 Mounting

The FUSE driver is mounted automatically during the initialization to the mountpoint path specified in a command line.

The Linux FUSE library mounts a filesystem in function `fuse_mount_common` in `helper.c` file. This mounting is done too early in the HelenOS. The mountpoint is saved into `fuse_chan` structure and later used in `fuse_session_loop` function in the file `fuse_lowlevel.c`.

The driver termination after unmounting is not implemented in the connector prototype.

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 function as mount point for them.

These data are stored in a hash table.

3.1.5 Storage for data about opened files

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

It is necessary to store these data somewhere. Opened files data are stored in a hash table similarly as the data about the mountpoints. This data are removed from the hash table when the file is closed.

3.1.6 Multithread support

A multithread support in the connector prototype is limited since the Pthread support in HelenOS POSIX is also limited. The multithread access is supported only for low level layer drivers. There are locks around all low level operations. These locks are ignored when the driver starts from the multithread function `fuse_session_loop_mt`.

3.1.7 File indexes

Both HelenOS VFS output operations (including libfs library operations) and FUSE low level interface operations use integer file indexes.

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

3.1.8 Creating and renaming files

`mkdir` or `mknod` FUSE low level operations are not called from the libfs operation `create` when a file or a directory is created. Instead of this a dummy node with empty file index 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 libfs calls the first `link` operation on this node.

There is one problematic issue: It is not possible to rename files, when the file system does not support link operation. The VFS server does not have `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

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

HelenOS. It is also present in other native HelenOS file system drivers which cannot 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 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). It 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 `lib/fuse.c` file.

3.2.1 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.

3.3 Reused code from Linux FUSE

The table 3.2 lists files with no changes to 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.

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

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 Other necessary changes in HelenOS

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

3.4.1 HelenOS and POSIX return codes

It was necessary to use both POSIX and HelenOS native error codes in the file `fuse_lowlevel.c`. Definions 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.

3.4.2 Opendir error in libfs library

This thesis uncovered a bug in lookup function in libfs library. Among other things lookup function manages opening of directories. The lookup function in libfs did not call libfs operation `node_open` when `lflag = L_OPEN | L_CREATE`. These flags represents an opening of a new file. This bug was not found earlier because most file systems has a stateless open where the `node_open` operation does nothing. It got unnoticed for a few years.

3.4.3 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.

3.4.4 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 structure members.

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.

3.4.5 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 section 3.1.2.

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

Another difference is connected with the access to block devices. Currently FUSE driver can access only disk images. This will change by implementing one of the solutions which are proposed in section 2.6.

3.5 Development using distributed version control system

A distributed version control system and a public repository[22] 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.

4.1 ExFAT

ExFAT[19] is a file system from Microsoft. ExFAT is optimized for flash drives. It uses FUSE high level API.

Free exFAT file system implementation[18] was ported to HelenOS at the same time as the connector was developed.

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 operation system specific section for 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.

ExFAT uses pread and pwrite POSIX functions (section3.4.3) to access block device. The implementation of these functions was added to HelenOS in this thesis in order for ExFAT to work. This allows mounting a file system image. Another work is necessary to be done for the mounting of the real block devices. Possible solutions are described in 2.6.

There still remains one problem: 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 during development of VFS-FUSE connector.

4.2 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.

4.2.1 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/`

4.2.2 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/`

4.3 Estimation of difficulty to port other file systems

The main problem when porting FUSE file systems to HelenOS is that the most of the 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.

Some FUSE file system drivers work as a layer over another file system. Thit means that they store some data in other 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 [20] about this deadlock in the HelenOS issue tracker.

File systems which access block devices can now only mount a file system image. One of the solutions that can be found in section 2.6 is neccessary in order to change this limitation.

file system	porting problems
MP3FS	VFS server deadlock, libraries
Ramfuse	VFS server deadlock, PERL
squashfuse	block devices solution, libraries, OS specific
CryptoFS	VFS server deadlock, libraries
LoggedFS	VFS server deadlock
SshFS	libraries
ZFS	block devices solution, OS specific
NTFS-3G	block devices solution, OS specific
gitfs	VFS server deadlock, libraries

Table 4.1: Porting other FUSE file systems

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

The table 4.1 shows an estimation of the porting difficulty for some popular file systems from [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 are using 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.

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 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 in a deeper level and optimization. This includes for example mutithreaded 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. FUSE drivers should also terminate themselves after being unmounted.

Some FUSE file system drivers work as a layer over another file system. The solution to the problem with namespace read-write lock [20] is neccesary to be

done before porting them to HelenOS.

The ability to read block devices as discussed in section 2.6 will be also an important issue in the future.

The last but not least work to be done will be porting 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/apps/mediawiki/fuse/index.php?title=FileSystems

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

[9] *Operating Systems - FUSE*, http://sourceforge.net/apps/mediawiki/fuse/index.php?title=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] *VFS deadlock ticket*, http://trac.helenos.org/ticket/480.

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

[22] *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`
- **README** a readme text file, reading it is recommended.
- **Qemu wrapper script** `run.sh` starts HelenOS in Qemu[21] emulator.
- **An electronic version of this thesis** in the file `thesis.pdf`.

Appendix B

User Documentation

Easier way how to run HelenOS operating system is Qemu[21]. It emulates the whole PC and is the recommended emulator for HelenOS. You can do this by starting Qemu with run script:

```
./run.sh
```

In order to mount exFAT image with FUSE exFAT driver run

```
fuse_exfat exfat.img /mnt
```

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

B.1 Compiling from sources

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 it. 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,

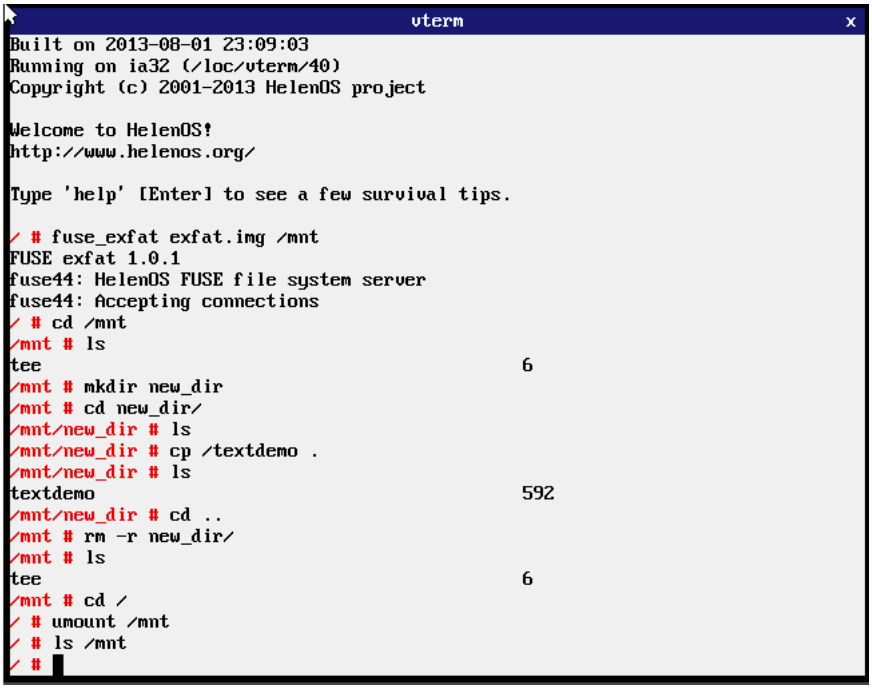


Figure B.1: Screenshot of FUSE exFAT file system usage

`/usr/local/cross` will be used by default.

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

After that run

```
make
```

and in configurator select

```
--- Load preconfigured defaults ...
ia32
Done
```

For cleaning after compilation you can use

```
make clean
```

or for cleaning config as well

```
make distclean
```