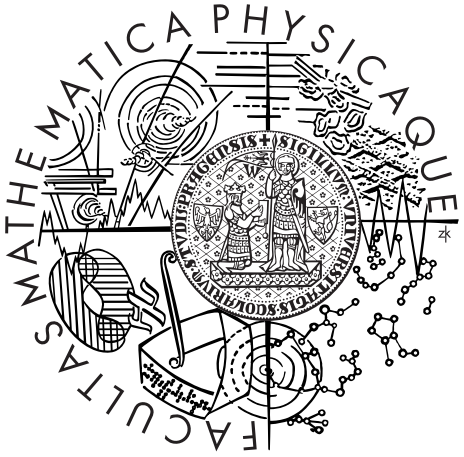
Charles University in Prague

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 2013

I'd like to thank my supervisor Mgr. Martin Dětský 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 August 2, 2013 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, 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 skutečný 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 filelesystem 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. Then the concern is focused on the real implementation of the connector. The implementation was successful, so the thesis describes in detail the parts of the fully functional solution in real-life HelenOS system. The solution enables almost no need of changes to be made in both FUSE and Helenos VFS. The connector implementation is demonstrated on a real-life FUSE filesystem Exfat ported to HelenOS.

Keywords: HelenOS, VFS, FUSE

Contents

Introduction

Goals

The goal of this master thesis is the design and prototype the connector between FUSE filesystem drivers and HelenOS native VFS interface.

The design has to be done in such a way to minimize forced changes in both the HelenOS VFS and the FUSE filesystem drivers.

Utilize the Linux FUSE implementation so that code reuse will be maximized, e.g. between libfuse and the connector implementation.

Demonstrate the connector implementation functionality as prototype on a real-life FUSE filesystem ported to HelenOS.

Compare this implementation of the FUSE interface with implementations in other operating systems.

Text organization

First chapter (Development context) of the thesis deals with context of the connector implementation. Deep knowledge of both parts to be connected is necessary. HelenOS architecture is described with the view on kernel and servers IPC communication. It also concerns on filesystem sybsystem in HelenOS operating system and describes how it works. Last part is about how FUSE filesystems are developed and how FUSE architecture looks like on Linux and other platforms.

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

Next chapter includes implementation details. It shows real results of the context and the analysis in praxis. Firstly the integration with libfs deals about necessary parts as e.g. operations mapping, mounting and opened files data. Chapter lists reused code from Linux FUSE and then other changes necessary to be made in HelenOS operating system.

Ported FUSE filesystem drivers are described in last chapter of the thesis. Namely exFAT and some examples from Linux FUSE package are described.

Chapter 1

Development context

This chapter includes brief summary development context background from both FUSE and HelenOS point of view.

1.1 HelenOS architecture summary

HelenOS[1] is new operating system based on the microkernel architecture. It started on 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.

Microkernel architecture enables that the system relies less on the quality of kernel design. The system can be also called as component system. The aim of the HelenOS system microkernel and component based design is to provide system that can be called "simple design, smart code".

The system kernel implements only several most important features like multitasking, virtual memory management, symmetric multiprocessing and so called IPC - inter process calls. All other services are implemented as common user processes. Also file system and drivers are implemented in this way.

Traditional systems are making great distinction between OS and end-user applications. HelenOS with its concept makes no distinction between them.

Userspace tasks in HelenOS are separated, each of them has its own address space. Because of that they need a way to communicate with kernel and other tasks (servers). Kernel provides a IPC communication, it is mostly asynchronous. For more simple use of IPC comunication can be used asynchronours framework. IPC comunication and Asynchronous framework is described in IPC for Dummies [14].

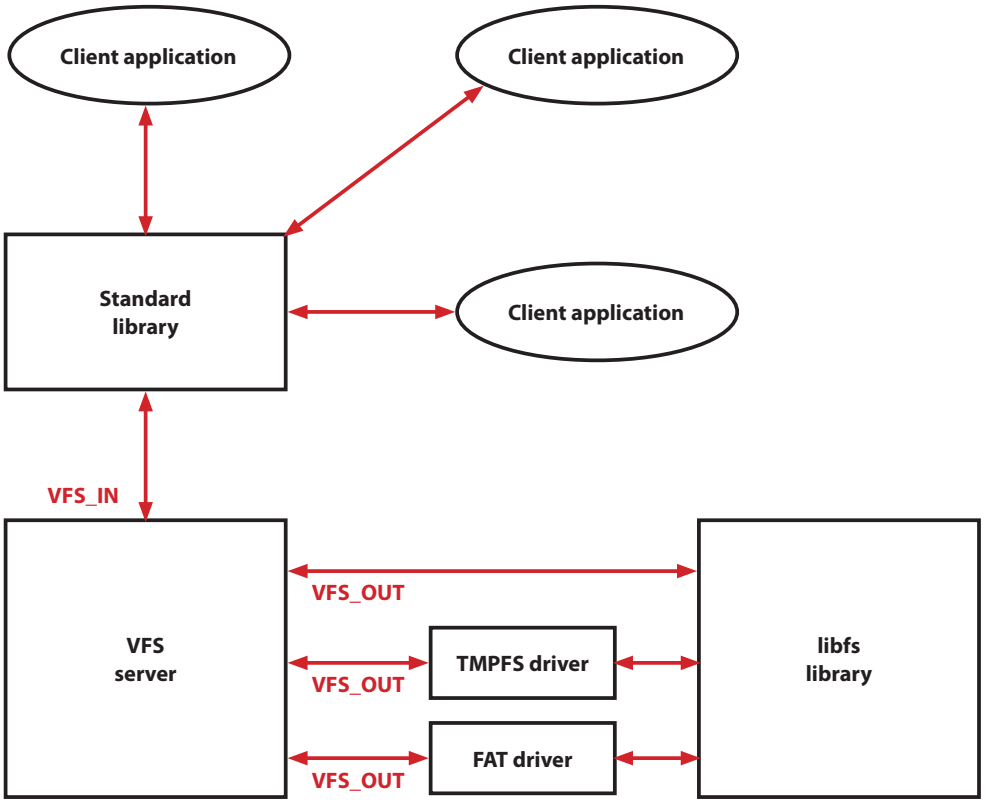


Figure 1.1: Filesystems in HelenOS

1.2 Filesystem in HelenOS

Functionalty of HelenOS filesystem subsystem describes Jakub Jermář in [4]. It can be divided to four sections: Standard library, VFS server and filesystem driver which uses libfs library. How it works together is best seen in Figure 1.1

1.2.1 Standard library

Standard library contains some code that transforms POSIX calling of user task to protocol that can be understood by entry part of VFS server. Some calls as `opendir()`, `readdir()`, `rewinddir()` and `closedir()` are implemented by standard library directly by calling functions `open()`, `read()`, `lseek()` and `close()`. Standard library also enables the use of `getcwd()` and `chdir()`. This is needed for VFS that can work only with absolute file paths. Standard library also translates relative paths to absolute ones. Standard library has no data structures and algorithms for file system support. Every task that it cannot solve is given via IPC to VFS server.

1.2.2 VFS server

VFS plays central role in the file system support in HelenOS. Logically can the functioning of VFS be divided to IN and OUT parts.

IN part receives the demands from client tasks. If the parameter of task is descriptor of the file, VFS looks in the table of opened files and finds the pointer to the structure that represents the open file. If the parameter is a path, VFS performs `vfs_lookup_internal()` and gives as a result VFS triplet. It identifies file by global number of the file system, global number of the device and number of the file. Based on this triplet VFS tries to find VFS node. All files are represented by those VFS nodes. When VFS server knows the VFS node, it can transfer it to the device driver of the file system. VFS also performs seek operations.

OUT part of the VFS then communicates with the driver of the end file system. All the needed information is in VFS node - number of the FS, number of the device and file identification.

1.2.3 libfs library

Library libfs was prepared for implementing some structures, that have to be implemented by drivers and often is very similar or the same for each end file system. Libfs contains some code by means of which file systems are registered in VFS system. The other fundamental role of libfs is function `libfs_lookup()`. This function implements output operation VFS LOOKUP. This operation must be implemented by every file system. `libfs_lookup()` doesn't only implement looking up, but performs also creating and deleting file names in directory tree. Then also creating of empty files and directories. To ensure the functionality of this feature in concrete file system there must be implemented several operations. They "tell" the libs library how to list in a directory, how to create or delete the file name in the directory tree and how to create or delete a file.

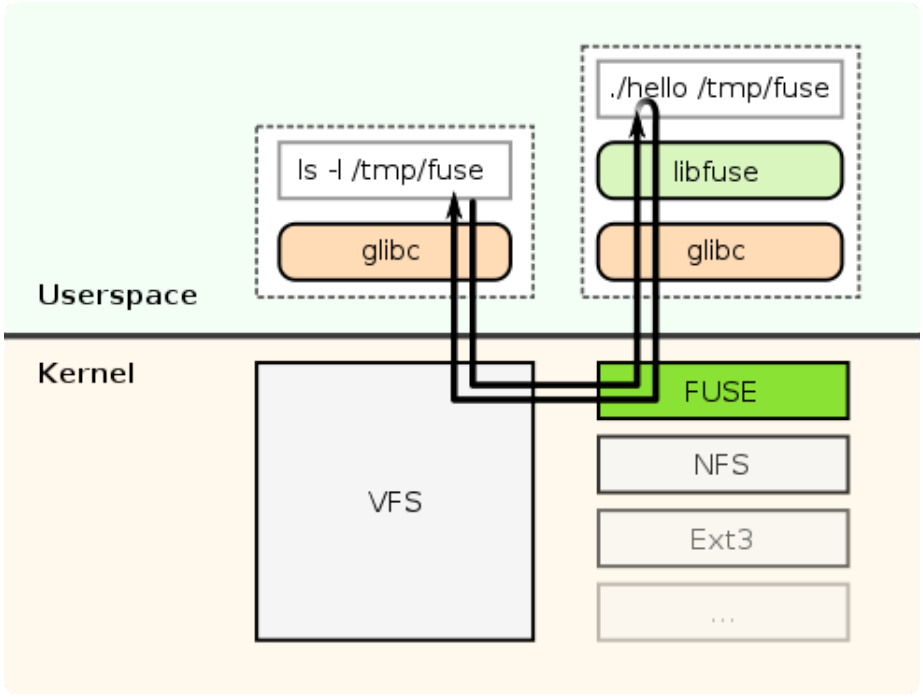


Figure 1.2: Filesystem in Userspace in Linux [6]

1.3 Developing filesystem with FUSE

FUSE filesystem drivers run in user space. Developing them is as simple as developing other userspace applications.

There are two different APIs: Low level and High level API.

High level API identifies files by their names in all cases. Do for example when you want to read file content you create function with does that:

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

Low level API uses numbers for identifing files. So for example when reading directory both file names and number are returned. Later when opening file low level driver function gets only this number for file identification purpose. "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)[5] has two parts: kernel module and userspace library. When call is made to for example read a file the FUSE kernel module forwards this to userspace driver. How does it works is best seen in Figure 1.2

For exchanging messages with kernel userspace library has kernel channel API.

Main operations are `receive`, `send`. Those messages are later decoded by low level part of userspace library. Those messages are exchanged through device `/dev/fuse`.

After decoding messages appropriate low level operation function in low level driver is called. This function later is then supposed to call reply function with an answer. That answer is encoded and passed through `send` function from kernel channel API.

High level API is implemented as a library which is written in same way as low level drivers are. Mapping between file numbers and names is the main purpose of high level library.

1.5 FUSE in other operation systems

FUSE is supported in other operation system then just Linux. List of them can be found on FUSE website [7]

1.5.1 NetBSD

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

In NetBSD 5.0 ReFUSE [10] library was introduced. It linked FUSE drivers with userspace PUFFS library. It only supported FUSE High Level Drivers.

In NetBSD 6.0 PERFUSE (PUFFS Enabled Relay to FUSE) is implementing PUFFS to FUSE kernel API bridge. Userspace daemon Perfused[11] translates PUFFS requests into FUSE messages. It creates `/dev/fuse`, which FUSE drivers connects to. Modified version of FUSE library from [5] is used. `mount()` and `open()` of `/dev/fuse` are modified to use variants from libperfuse[12]. Both low and high level APIs are supported.

1.5.2 OS X

FUSE for OS X [8] has two parts. OS X specific in-kernel loadable filesystem and userspace library based on FUSE project [5]. Userspace library has numerous OS X specific extensions and features.[9]

1.5.3 FreeBDS

During Google Summer of Code 2007 and 2011 FUSE was ported to FreeBSD https://wiki.freebsd.org/FuseFilesystem. It uses userspace library from FUSE project [5] and is currently maintained. Architecture is similar to FUSE for Linux.

1.5.4 Solaris

In Solaris only the high-level FUSE API from version 2.7.4 is present. Solaris FUSE uses header files ported from Linux and the implementation is Solaris-specifc. It is 'just' a wrapper over libuvfs. UVFS is the Solaris equivalent of FUSE. UVFS uses for communication between kernel and userspace doors calls and pseudo filesystem. [13]

Chapter 2

Analysis

This chapter describes: analysis of problems and selected solutions.

2.1 FUSE server

One way to connect specific FUSE filesystem driver to VFS server is creating FUSE server. This new server would do all recoding and therefore clear all differences between FUSE and HelenOS VFS. It would then forward requests and responses to and from specific FUSE filesystem driver servers.

Another possibility is to create library which would convert FUSE driver to HelenOS filesystem server. This way there is no need for changes in VFS server.

2.2 Layer selection

It is necessary to choose API layer which would best fit for connecting FUSE driver to HelenOS's VFS. As described in 1.4 there are three API layers: kernel chan API, low level API and high level API.

In all these three layers the connection can be made. Every solution has its advantages and disadvantages.

2.2.1 High level API

High level API uses file names for identification. This is a great complication since HelenOS VFS_OUT interface uses integer indexes to identify files. Choosing this layer would mean to rewrite all code which is already present in Linux FUSE library.

One drawback of choosing high level layer solution is that it doesn't support low level API filesystems.

Solaris 1.5.4 and NetBSD 5.0 1.5.1 is using this choice.

Advantages	Disadvantages
Code which best fits HelenOS VFS	Almost all must be written from scratch
	no support for low level API drivers
	file names vs. file numbers problem

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

Advantages	Disadvantages
Similar to VFS_OUT and libfs operations	
no need for FUSE server	
high level API code from Linux FUSE library	
both high and low level API drivers supported	

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

Advantages and disadvantages of connection at high level API layer can be seen in table 2.1.

2.2.2 Low level API

This API is the most similar to Helenos VFS OUT protocol. It is even more similar to libfs library operations. This is a good choice for creating library which could convert FUSE driver to HelenOS filesystem server.

Advantages and disadvantages of connecting at low level API layer can be seen in table 2.2.

2.2.3 Kernel channel API

Connecting on Kernel channel API means using almost all code from Linux's FUSE library. Using libfs library is not possible for this case. All VFS_OUT methods needs to be converted to the format as which Linux's FUSE library recognizes. Those formats are not similar and to use this possibility to connect to FUSE library by this way specific FUSE server would be best way how to do it. Kernel channel API messages would then be send as IPC messages.

NetBSD 6.0 1.5.1 is using similar solution with Perfused daemon. Also technically it is connected a bit further in `/dev/fuse`.

Advantages and disadvantages of connecting at high level API layer can be seen in table 2.3.

Advantages	Disadvantages
Designed for connection in here	encoding and decoding messages
Almost all Linux library code reusable	FUSE server necessary

Table 2.3: Advantages and disadvantages of connection at kernel chan API

2.2.4 Summary of selected solution

Connecting at Low level API layer is most suitable for HelenOS. FUSE server is not necessary and connector can be implemented as library. It can use libfs library same as other filesystems. This minimizes changes in both HelenOS VFS and FUSE library and drivers. How will selected solution work in the whole filesystem in HelenOS can be seen in Figure 2.1.

2.3 Each instance has its own task

While one file system server in HelenOS serves more instances of concrete filesystem FUSE does not. Fortunately this is not a problem since I can run new HelenOS filesystem server for each FUSE driver instance.

Fuse drivers also mount itself on start. This can be done automatically after start of FUSE filesystem server. Once FUSE filesystem server will be unmounted it suspends itself.

2.4 Reading directories

There is a difference in reading directories in HelenOS and FUSE. When HelenOS VFS requests reading a file in directory it gives an position of file in directory. Fuse lowlevel APIs requests bytes offset in a buffer of dir entity structures. In prototype whole directory is read until it finds desired file. Since FUSE low level driver is adding directory entities by low level api function `fuse_add_dirent()` it seems like this function can count positions in directory and save offset in order to directly jump to desired file position. Unfortunately this is not possible because there is no guarantee that function `fuse_add_dirent()` is called only with the same buffer which is then returned in `fuse_reply_buf`.

Reading whole directory is not effective since for each file directory the whole directory must be read again. This can be speed up by cashing the 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 `readdir` low level operation would not be called more times than it is absolutely necessary. For the simplicity connector prototype does not implement either of these cachings.

2.5 Mounting FUSE filesystems

Fuse filesystem drivers are standalone applications. They gets mountpoints on command lines. In order for it to be possible to mount FUSE filesystems same as native HelenOS filesystems `mount.fsname` script can be used. For this to work as system command library would look up for mount script which would run that script instead of sending `VFS_IN_MOUNT` method. FUSE library would takes care of it.

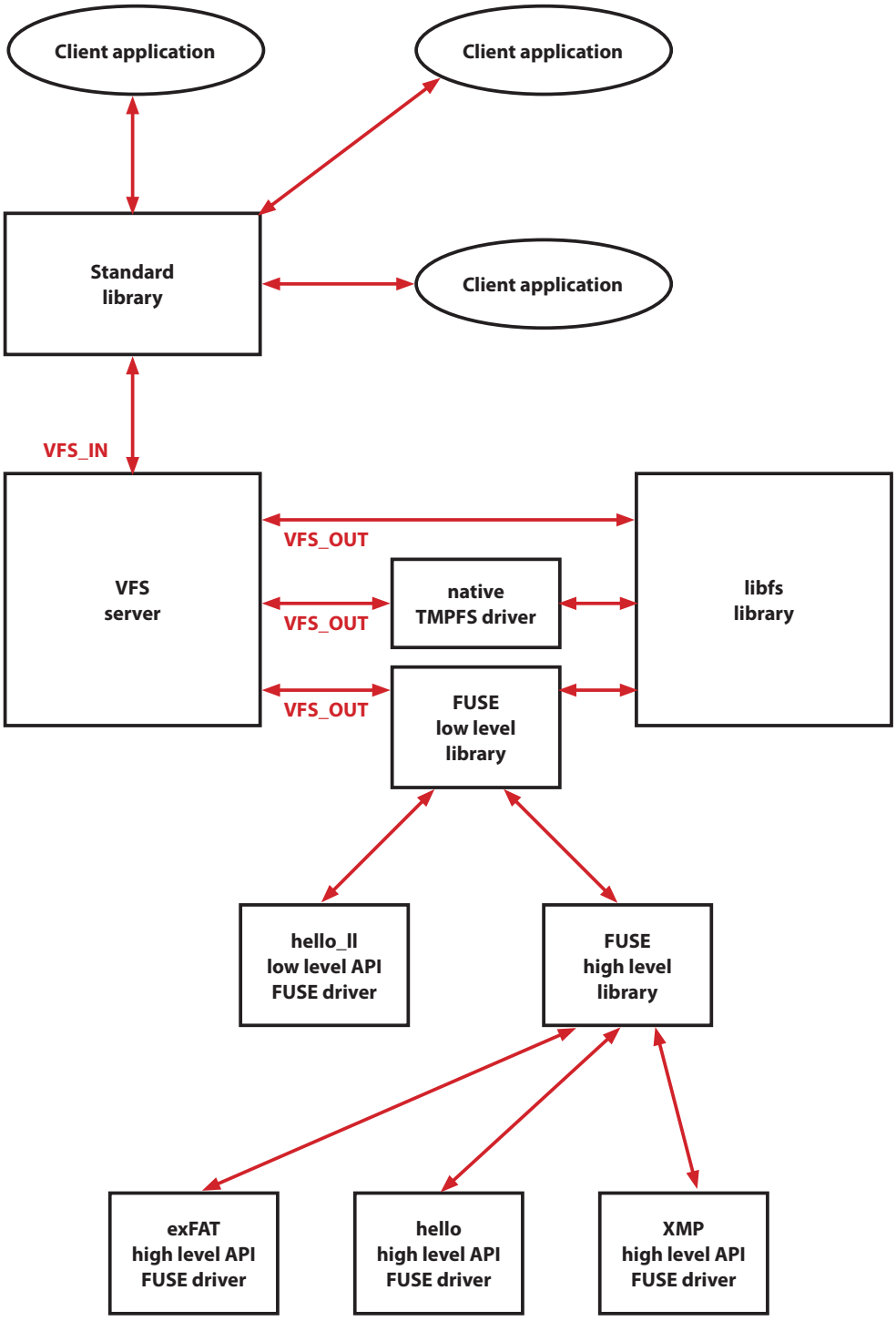


Figure 2.1: FUSE in HelenOS

2.6 Accessing block devices

There is a difference in accessing block devices in HelenOS and FUSE applications. FUSE drivers access block devices directly as files. For example exFAT [15] uses pread function. In HelenOS block devices are accessed through block device servers. There is of course possibility to overwrite parts of FUSE driver which access block devices. But this would need to be done for each filesystem again and again. There are three possibilities how to solve this problem without modifiing all ported FUSE filesystem drivers.

2.6.1 POSIX function overwrite

This possibility sets conditions for calling in POSIX library. For some prefix it makes reading or writing into or from block device server (instead of VFS server).

2.6.2 Block device file system server

The second possibility is to create special filesystem server that will enable the access to block devices via VFS.

2.6.3 Block device drivers also supports VFS_OUT protocol

One more possibility is that VFS interface (methods like `VFS_OUT_READ` and `VFS_OUT_WRITE`) will support specific block device drivers directly. Console device drivers uses are using same way.

Support of those VFS methods will not be needed to be implemented by each block driver itself, but they could be implemented directly in a common skeleton library `bd_srv.c`. This will need to translate them to method calling from `bd_ops_t`.

2.6.4 Conclusion

While POSIX function overwrite is probably the easiest to implement it is not the cleanest way. Block device file system server is nice clean solution but it adds another in between of FUSE filesystem driver and block device. It is also probably the hardest way for implementation. Supporting VFS_OUT protocol in block device drivers is clean and eficent way how to solve this problem. This problem is not a direct theme of this thesis so it is not implemented in the prototype.

Chapter 3

Implementation

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

3.1 Integration with libfs

As discussed in Analysis 2. There is no need for changes in libfs. Connector works as library implementing libfs and VFS_OUT operations same as any other filesystem.

3.1.1 Mapping operations

Mapping between HelenOS VFS_OUT nad libfs operations is described in table 3.1. From the FUSE point of view there is no difference between libfs and VFS_OUT operations.

There are also libfs operations which don't need calling FUSE low level operation method. Following operations `is_directory`, `is_file`, `lnkcnt_get`, `size_get` returns data retrieved by previous call of libfs operation `node_get`. Others like `node_put` only frees data from memory or are not necessary for working prototype. Since file name is not known in libfs operation `create_node`, calling is delayed until first `link` operation.

Most of the low level API conversion code is in `lib/fuse_lowlevel.c`.

3.1.2 Reply functions from low level API

Fuse low level function implementations uses reply functions. In FUSE Package [5] they send messages back. When implementing libfs and VFS out functions it is necessary to potentially convert that data and then return them by async framework after calling FUSE low level function.

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 API

In order to make this work reply structure was created as part of FUSE request structure `fuse_req_t`. It is first parameter of all fuse low level operations. Fuse reply function adds data to reply part of request structure. This data are then extracted after FUSE low level function call is finished.

Another reason why data can't be send to VFS in reply function is that more then one FUSE low level operation is called in one libfs or VFS_OUT operation.

3.1.3 Mounting

In FUSE mountpoint is specified when driver is starting. After starting FUSE driver it is mounted automatically from mountpoint in command line. Exit after umounting is not implemented in connector prototype.

3.1.4 Mounting other filesystems under FUSE

In order for it to be possible to mount other filesystems under FUSE libfs needs to store information about nodes which function as mountpoints. Connector library uses hash table to store this information.

3.1.5 Saving fuse file info for opened files

It is necessary to store data for opened files. This information is then passed to FUSE low level operation functions. Similarly as information about mountpoints opened files data are stored in hash table.

3.1.6 Multithread support

Multithread support in the connector prototype is limited since Pthread support in HelenOS POSIX is limited. Only low level layer supports mutithread access. If driver starts multithread `fuse_session_loop_mt` then single thread lock around any low level operation is ignored.

3.1.7 File numbers

Since both `VFS_OUT` interface, libfs library and FUSE low level API uses file numbers I can easily return file numbers from FUSE to libfs respective `VFS_OUT` operations.

3.1.8 Creating and renaming files

When file or directory is created `mkdir` or `mknod` FUSE low level operations are not called from libfs operation `create`, dummy node with empty file index is returned. The reason is that file name is not known at that moment. `mkdir` or `mknod` FUSE low level operations are called later when libfs calls first `link` operation on this node.

There is one issue it is not possible to rename files if filesystem does not support link operation. VFS server don't have out `VFS_OUT_RENAME` operation and instead just calls `link(new_name)` and then `unlink(old_name)`. This issue is not FUSE specific, it is also in other native HelenOS filesystem drivers which cannot handle more than one link to file.

3.2 High level API

High level API code is implements Low level API operations. Almost all code from that implements high level API is reused from Linux FUSE library. There is also code which is not in used (not called from connector library)

Most of the code high level API code is in `lib/fuse.c` file.

3.2.1 Pthread library

High level API is using pthread locks and condition variables. In order to make this work pthread locks and condition variables are transformed to HelenOS fibril variants.

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

Table 3.2 lists files with no changes to upstream Linux FUSE library[5].

Table 3.3 lists files with small changes. Those changes are separated by `#idef __HelenOS__` to make it easier to update them to new versions of Linux FUSE library.

Last table 3.4 lists files with HelenOS specific code.

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 changes necessary in Helenos

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

3.4.1 HelenOS and POSIX return codes

In `fuse_lowlevel.c` it is necessary to use both POSIX and HelenOS native error codes. Since in POSIX programs POSIX error code defines overwrites native ones it was necessary to introduce other name for them. Native error codes are now also accessible in POSIX programs with `NATIVE_` prefix.

3.4.2 opendir error in libfs

This thesis uncovered bug in libfs lookup. It didn't call `ops->node_open` when `lflag = L_OPEN | L_CREATE`. This bug wasn't found earlier because most filesystems has stateless open and therefore it got unnoticed for few years.

3.4.3 pread and pwrite

FUSE drivers uses `pwrite` and `pread` for accessing block devices. Those functions were not implemented in HelenOS. In order for them to work new VFS_IN operations were introduced VFS_IN_PREAD and VFS_IN_PWRITE. Only diffrence between them and VFS_IN_READ and VFS_IN_WRITE is another parameter: offset in file. Also for simplicity of implementation HelenOS don't reads or writes the whole buffer even when there is no end of file. FUSE driver don't calculate with this possibility. For solving this problem POSIX `pwrite` and `pread` are mapped to `pwrite_all` and `pread_all` versions which in this cases calls native `pwrite` and `pread` more times.

3.4.4 POSIX prefix defines collision

Early during development there were problem with collisions with POSIX defines. FUSE operations are implemented as structure members. Unfortunatelly they sometimes had same name as POSIX functions and POSIX functions were implemented like `#define read posix_read`. As a workaround pushing and poping defines was used. Later this has been solved in HelenOS mainline by Vaclav Horky by overwriting function names at link time so this problem vanished.

3.5 Development using Mercurial

Bazzar public repository[18] for development of VFS to FUSE connector was used. This helped during development and merging with HelenOS mainline repository[3]. This will also help future merging of this work to mainline repository.

Chapter 4

Ported FUSE filesystems

This chapter describes FUSE filesystems which were ported to HelenOS.

4.1 exFAT

ExFAT[15] is filesystem from Microsoft. It is optimized for Flash drives. It uses FUSE high level API.

ExFAT fuse driver can be found in uspace `uspace/srv/fs/fuse/exfat/`. It also uses exFAT library which can be found in `uspace/lib/posix/libexfat/`

During porting It was necessary to add a specific HelenOS section for handling detection of endianness and byteswaping.

There is one issue it is not possible to rename files. VFS don't have `VFS_OUT_RENAME` operation and instead just calls `link(new_name)` and then unlink(old_name). However Exfat doesn't supports more then one link and therefore this fails. This issue is not FUSE specific, it is also in other native HelenOS filesystem drivers which cannot handle more than one link to file.

HelenOS also has a native filesystem driver for exFAT. Same filesystem was selected a way to test against another working implementation of same filesystem helped during development of VFS to FUSE connector.

4.2 Examples from FUSE package

FUSE package includes some example filesystems.

4.2.1 Hello world in high level interface

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

Conclusion

The goal of this master thesis was the design and implement the connector between FUSE filesystem drivers and HelenOS native VFS interface. The goal of finding the solution and develop the implementation of the connector was achieved.

Important part of this work was the decision how to implement the connector. Starting decision to implement connection at low level layer has proven as a very good choice. This enabled to reuse a lot of code from Linux FUSE implementation. Practically no changes were necessary to be made in FUSE filesystem drivers. In fact almost all of those changes do not relate to FUSE but to limited POSIX libraries in HelenOS.

The implementation at low level layer also enabled the use of same libraries as native drivers use and therefore no changes in HelenOS VFS were necessary. This fact will make future development of the HelenOS VFS easier. Consequent work will have no need to care about FUSE so future development will not need to have always FUSE in mind.

FUSE variant of exFAT filesystem driver was ported to HelenOS. Since HelenOS already has native exFAT filesystem driver. This didn't add new features but this was intentional for help in development.

Similar to the rest of HelenOS performance and speed was not the goal.

Future work

Current implementation provides a solid base for direct use of FUSE filesystem drivers in HelenOS. Although several FUSE features can be developed in deeper level and optimization. This includes for example mutithreaded drivers and readdir method.

Introducing mount scripts will enable mount in the same way as native file systems does. FUSE drivers should also terminate themselves after being unmounted.

In order to port FUSE filesystem driver which works as layer over another filesystem solution to the problem with namespace read-write lock [16] is neccesary.

Ability to read block devices as discussed in 2.6 will be also important.

The last but not least work to be done will be porting other FUSE filesystem 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 filesystem in HelenOS operating system*, http://www.helenos.org/doc/papers/HelenOS-EurOpen.pdf

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

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

[7] *Operating Systems - FUSE*, http://sourceforge.net/apps/mediawiki/fuse/index.php?title=OperatingSystems

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

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

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

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

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

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

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

[15] *exFAT*, https://code.google.com/p/exfat/

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

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

[18] *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** 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[17] emulator.
- **An electronic version of this thesis** in the file `thesis.pdf`.

Appendix B

User Documentation

Easier way how to try this work is Qemu[17]. It emulates whole PC and is the recommended emulator for HelenOS. You can do that 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 Figure B.1.

B.1 Compiling from sources

For compiling is recommended to use Linux. First you need to unpack the sources:

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

and then move to newly created directory

```
cd helenos_fuse
```

Next install development toolchain. For compiling HelenOS specific versions of compiler and binutils are necessary. Toolchain has dependencies. Most of them is listed when you run it. In order to save time install this 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

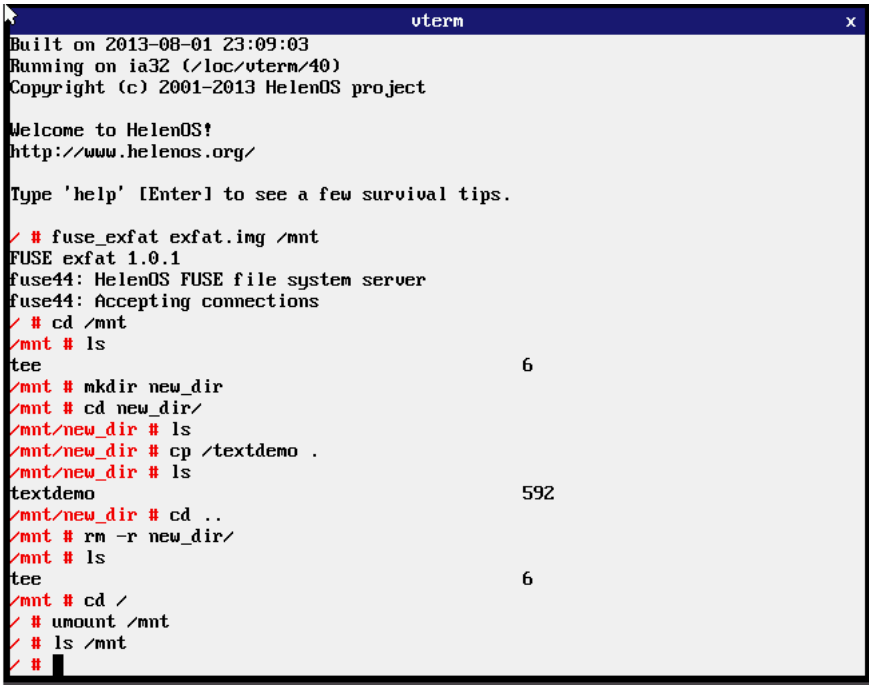


Figure B.1: Screenshot of FUSE exFAT filesystem usage

used by default.

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

After that run

```
make
```

and in configurator select

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

For clearing after compilation you can use

```
make clean
```

or for clearing config as well

```
make distclean
```