Großer Beleg

Porting FUSE to L4Re

Florian Pester

23. Mai 2013

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Betreuender Hochschullehrer: Prof. Dr. rer. nat. Hermann Härtig
Betreuender Mitarbeiter: Dipl.-Inf. Carsten Weinhold

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Declaration

I hereby declare that this thesis is a work of my own, and that only cited sources have been used.

Dresden, den 23. Mai 2013

Florian Pester

Contents

1. Introduction **1**

2. State of the Art **3**
2.1. FUSE on Linux 3
2.1.1. FUSE Internal Communication 4
2.2. The L4Re Virtual File System 5
2.3. Libfs 5
2.4. Communication and Access Control in L4Re 6
2.5. Related Work 6
2.5.1. FUSE 7
2.5.2. Pass-to-Userspace Framework Filesystem 7

3. Design **9**
3.1. FUSE Server parts 11

4. Implementation **13**
4.1. Example Request handling 13
4.2. FUSE Server 14
4.2.1. LibfsServer 14
4.2.2. Translator 14
4.2.3. Requests 15
4.2.4. RequestProvider 15
4.2.5. Node Caching 15
4.3. Changes to the FUSE library 16
4.4. Libfs 16
4.5. Block Device Server 17
4.6. File systems 17

5. Evaluation **19**

6. Conclusion and Further Work **25**

A. FUSE operations **27**

B. FUSE library changes **35**

C. Glossary **37**

List of Figures

2.1. The architecture of FUSE on Linux 3
2.2. The architecture of libfs 5

3.1. Using a character device server 9
3.2. Using IPC directly 10
3.3. FUSE server and library share and address space 10
3.4. The architecture of FUSE on L4Re 11
3.5. Class diagram for the FUSE Server 12

4.1. Handling an example request 13

5.1. The average speed on ext2 with one and two threads 19
5.2. The average speed of the benchmarks on different file systems 20
5.3. Throughput on the different file systems 21
5.4. Write throughput with and without copied write buffer 21
5.5. Untar with and without copied write buffer 22
5.6. Cycles spent in Server and Library in the different file systems 22

1. Introduction

Implementing a file system is hard and tedious work. Users expect file systems to be reliable, because they trust them with their data. File systems must also be fast, because users do not like to wait. Widely used operating systems, such as Linux or Windows, are still monolithic operating systems, which run file systems in kernel. This means that writing file system code results in writing kernel code.

L4Re [l4r] is a userland for the Fiasco microkernel developed at TU Dresden. It provides access to the C library, offers a C++ environment and an infrastructure for client/server programming. Unfortunately it has little file system support yet. Users of L4Re would benefit from file system support, especially if those were file systems supported by widely used operating systems, such as Linux or Windows.

Porting a file system, especially from a monolithic operating system to a microkernel, may require significant changes to the file system implementation, due to the design of the rest of the system. Owing to this complexity, file systems are seldom ported from one system to another.

Filesystem in Userspace [Szea], also called FUSE, provides the ability to run file system logic in user space under Linux and other POSIX based systems. There are implementations for many widely used file systems on top of FUSE. NTFS-3G [Sza] is an implementation of the NTFS file system and fuse-ext2 [Akc] provides support for the ext2 file system. Other file systems provide access to network file systems, sshfs [Szeb] and SMB for FUSE [Wag] being the most well known examples.

On Linux FUSE works by loading a module into the Linux kernel virtual file system and provides a user space library with an API, against which file systems are written. Once there is support for FUSE on an operating system, this OS can support every file system, that uses the API, if other dependencies, such as networking are fulfilled. This provides for easy reuse of legacy file systems, which have been implemented for FUSE and gives support for every new file system written against the FUSE API.

New features are added to the FUSE library frequently, therefore one goal of this work is to change the library as little as possible, in order to easily port FUSE updates to L4Re. Existing FUSE file systems should work with no or minor modifications. On the other hand, the FUSE kernel module is an extension to the Linux kernel and therefore highly dependent on the Linux kernel. It will be necessary to completely reimplement the kernel module.

2. State of the Art

In this chapter I will introduce the design of FUSE on Linux, as well as the design of the Virtual File System (VFS) on L4Re, and libfs, a library that provides file systems to other address spaces for L4Re. This chapter also includes a section on ports of FUSE to other systems and the Pass-to-Userspace Framework File System (puffs) [Kan07], which allows running file systems, written for NetBSD, in user space.

2.1. FUSE on Linux

The Linux version of FUSE is made up of two parts. A Linux kernel module that registers itself as a file system towards the Linux VFS and provides a means of communication towards a library, running in user space. This library communicates with the kernel module and provides an API against which file systems can be implemented.

A kernel module is necessary, because each file system request first arrives in the Linux kernel, specifically the Linux Virtual File System. The VFS provides an interface, against which file systems are written by defining a set of structures, which consist of pointers to functions. The Linux kernel calls these functions once a file system request arrives. In Linux file systems run in kernel mode, usually as kernel modules. Each FUSE request must somehow be passed back from the kernel to user space.

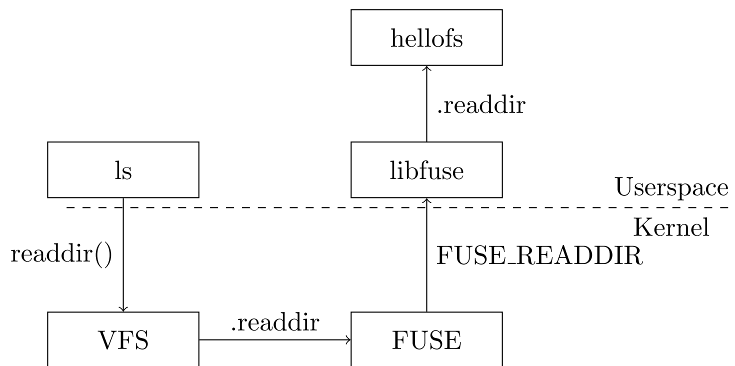


Figure 2.1.: The architecture of FUSE on Linux

To explain the architecture of FUSE on Linux I will use an example, which is illustrated by Figure 2.1. An application running in user mode wants to read the contents of a directory. The application makes a call to the `readdir()` function in

the C library, which in turn calls the kernel's `getdents()` system call. This system call is handled by the Linux kernel, which determines and calls the corresponding file system.

In the case of FUSE, the kernel module receives the calls to all FUSE file systems currently running. In order to pass the `readdir()` call to FUSE, the kernel calls the function defined by the `.readdir` function pointer. The kernel module wraps all parameters in a `FUSE_READDIR` request and signals the FUSE library, that a new request has to be handled.

Communication between the kernel module and the FUSE library is realized via a character device, which is located at `/dev/fuse`. The FUSE library reads requests from the device. For each request the library identifies the correct file or directory involved and calls the file system to handle the request.

Once the FUSE library has handled the request, a reply is written to the character device, in order to transfer it back to the kernel module. From there the reply is passed to the C library. The C library returns the reply to the application, which issued the request.

The abstraction for files and directories in FUSE is called a node. Every node in FUSE is identified by a nodeid. The FUSE library keeps nodes in a hash map, with the nodeid as key. Every request made to the FUSE library includes a nodeid, to be able to identify the node involved. The kernel module can learn about nodeids by calling `lookup` with the nodeid of the parent node. In order to bootstrap lookups, the nodeid of the root node is fixed.

2.1.1. FUSE Internal Communication

In order to make a decision on how to port FUSE to L4Re it is necessary to understand how the FUSE library and the FUSE kernel module communicate.

The FUSE kernel module issues requests towards the FUSE library. A single system call can issue multiple requests. All requests are done on nodes, which are uniquely identifiable by their nodeid. Nodes in FUSE are the abstraction for any entry inside a file system. The common case is a simple file or a directory.

On Linux requests are transfered between the FUSE library and the FUSE kernel module by reading from and writing to `/dev/fuse`. This character device is provided by the kernel module and repeatedly read by the FUSE library.

Every read from the character device transfers one request from the kernel module into the FUSE library and every write transfers a reply back to the kernel module. Each request starts with a header section, that contains information about the type of the request, the size of the request and a unique identifier for the request. Immediately following the header section are the arguments for the request, the layout of which depends on the type of the request. Different data structures abstract from the layout of the arguments in memory. An overview of all request types and the corresponding data structures can be found in Appendix A.

Like the requests, each reply starts with a header, that contains the identifier of the request it belongs to and the size of the reply. The reply's data immediately follows the header. A write to the character device transfers the reply to the kernel module.

FUSE has two types of requests. For synchronous requests the kernel module waits for an answer and processes it immediately. On asynchronous requests the kernel module assigns a callback function, which the kernel module calls, once a reply to the request arrives. When the kernel module issued the asynchronous request, the module can now process the next request until the reply arrives.

2.2. The L4Re Virtual File System

In L4Re every application has its own file system tree. By default a mount can only be used by the application, that created the mount. For FUSE, this is a problem, because, every FUSE file system runs as an application. Therefore I need some method to provide a file system to another application.

The Virtual File System on L4Re is written in C++. There are classes that abstract from a file system, from a file and from a directory. The base `File_system` class provides an interface to implement a L4Re file system. `Be_file_system` inherits from `File_system` and takes care of file system registration with the VFS.

POSIX directories are abstracted by the `Directory` class, while the different types of POSIX files are abstracted by `Generic_file`, `Regular_file` and `Special_file`. The `File` class inherits from all of these and provides the basic interface for files and directories. The `Be_file` class extends the `File` class and provides function stubs for the POSIX interface on a file or directory.

2.3. Libfs

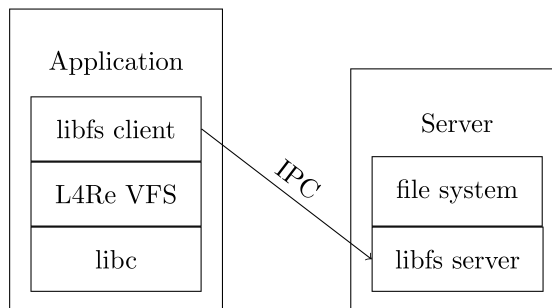


Figure 2.2.: The architecture of libfs

Libfs is a library that provides a method to access a file system server in L4Re. There is a client plugin that provides a backend to the C library and acts as a file system registered towards the L4Re VFS. This client forwards all calls to a L4Re Server. This architecture is similar to the Network File System (NFS), where a client module forwards calls to a server, which issues calls to another file system.

The server runs on the same machine, in another address space and dispatches all calls to an underlying file system. Communication between the two parts is done with Inter-Process Communication (IPC). Figure 2.2 offers a graphical representation.

2.4. Communication and Access Control in L4Re

L4Re is a capability based system. This means, that every subject – usually a running program – must present a capability in order to perform certain tasks on objects, such as files or servers.

Generally speaking a capability works like a key to a door. If a subject presents a valid capability to the door the system allows the subject to go through the door and perform a task.

Inter-process communication (IPC) on L4Re is done through IPC gates. In order to perform an IPC to another address space a program must access an IPC gate. Accessing a gate requires a capability to that gate, this way communication between address spaces in L4Re is controlled.

For asynchronous communication L4Re provides `Irq` objects. An `Irq` object provides a `trigger()` and a `receive()` operation and is shared between two communication partners. Internally for each call to `trigger()` a counter is increased, while for each `receive()` the counter is decreased. If the counter goes below 0 in a `receive()` operation, the corresponding thread yields until `trigger()` is called again. As IPC gates, `Irq` objects are secured by capabilities.

2.5. Related Work

In 1993, Userfs [Fit93] was one of the early user space file systems for Linux. Userfs worked with Linux kernels up to 2.0 but was abandoned thereafter. The original intention was to create a prototyping testbed for file systems, so they could eventually be accepted into the Linux kernel. Userfs exposed the raw kernel VFS interface for that reason. The downside of this approach was, that the interface constantly changed with the kernel file system interface.

Arla [WD98] is a free implementation of the Andrew File System (AFS), which aims to be a replacement for NFS and is therefore aimed at network file systems. Similar to FUSE, Arla uses a kernel module, that provides a character device to a user space daemon. The same kernel module is used by the Coda file system [SKK+90].

The authors of the paper "A toolkit for user-level file systems" [Maz01] use a different approach by utilizing NFS loopback servers to communicate with the user space file system. As other approaches before, the toolkit relies on a kernel module, that gets the system calls from the kernel. This kernel module then transfers these calls back to user space via standard UDP networking calls.

The NFS loopback servers described suffer from a few complications, the worst being that any blocking operation can result in deadlock. The authors show, that file

system development is greatly simplified compared to standard in-kernel file systems. Unfortunately no comparison to character device based user level file systems is made.

Ptrace can also be used to develop user level file systems [SWSZ07]. This approach extends the `ptrace()` system call to monitor a user space file system. The monitor allows for rapid file system prototyping by offering a high level interface.

Another approach on simplifying file system development is taken with FiST [ZN00]. FiST is a high level language designed to implement stackable file systems. The file systems will run in kernel mode as loadable kernel modules, with the benefits and drawbacks in-kernel file systems bring with them.

All of the above where implemented on Linux or other POSIX system, mostly as research prototypes. While some might allow for slimmer or faster file system implementations, none offer as many file systems as FUSE does.

On Windows, Dokan [dok] provides the ability to develop user space file systems. It is described as being similar to FUSE, but not compatible with it.

2.5.1. FUSE

The original Filesystem in Userspace was written for the Linux kernel. The kernel module part was introduced into the Linux kernel with kernel version 2.6.14.

There are two ports for MacOS, the first and older one being MacFUSE [Maca], which is now discontinued. According to the Website the "in-kernel file system is specific to Mac OS X and is not based on Linux FUSE" [Macb]. It is not a goal of MacFUSE to be compatible with the original FUSE implementation, this means most file systems are not compatible without modifications.

The newer implementation is FUSE4X [FUSd], which is still under active development. FUSE4X has been forked off MacFUSE. One of the goals of the implementation is to make it more compatible with the original FUSE implementation in order to make porting FUSE file systems from Linux to MacOS easier.

There have also been efforts to port FUSE to Solaris [fusc] and FreeBSD [fusb], but these have been discontinued [fusc]. On Windows, the FUSE API is provided by Crossmeta FUSE [cro], which is used to provide access to the Linux file systems ext2/3/4 and ReiserFS on Windows.

There is also an implementation for Minix [min11] based on the Pass-to-Userspace Framework Filesystem and ReFUSE, which I will explain in more detail in the next Section.

2.5.2. Pass-to-Userspace Framework Filesystem

NetBSD introduced the Pass-to-Userspace Framework Filesystem (*puffs*), to implement file systems in user space [Kan07]. The goal of *puffs* is to make testing file systems safer and easier by running them in user space during testing. In order to provide the ability to run file systems in kernel mode for performance critical application, *puffs*'s interface is kept close to the interface of NetBSD's VFS.

Although the communication protocol is different from FUSE, *puffs* uses a character device to transfer calls from the kernel to user space, like FUSE.

Libpuffs, the *puffs* user level library is single threaded. One goal of *puffs* is not to require threading for a non-blocking file system. The solution is a continuation framework, provided with the library. A file system must provide explicit scheduling points, for example when expensive I/O operations are executed.

In contrast to FUSE the *puffs* interface uses pathnames just for operations that require a pathname component. To enable file systems to operate on paths, rather than on nodes, *puffs* provides a mount flag, with which all operations include a pathname component. Additionally, file systems can provide their own routines for pathname building, if they use a different naming scheme, than *puffs* does.

To keep interoperability with FUSE, there is a FUSE reimplementation on top of libpuffs, called ReFUSE [KC07]. ReFUSE works as a translation layer between the *puffs* user space library and the FUSE file systems, this means it translates requests it receives from the *puffs* kernel module into the FUSE request format and does the reverse for replies from the FUSE file system.

ReFUSE is source code compatible with FUSE and supports all revisions of the FUSE protocol.

3. Design

A FUSE port for L4Re should keep the changes to the FUSE library minimal, in order to be able to easily update the FUSE library. Adapting the FUSE library to be compatible with the L4Re Virtual File System directly is therefore only an option if the changes are small and self-contained, because these changes have to be reapplied for each new version of FUSE. For larger changes, this will result in a high maintenance effort.

It would be desirable to change neither the library, nor the VFS, or at least keep changes to an absolute minimum. This can be solved by using the adapter pattern [GHJV94]. In this design the adapter receives calls from the VFS, translates them into FUSE requests and transmits them to the FUSE library. Replies from the library would need to be translated back for the VFS.

The FUSE kernel module for Linux is written specially for the Linux kernel. It mainly acts as an adapter between the Linux VFS and the FUSE library. Reusing the kernel module would require a large number of structures and functions from the Linux kernel. The kernel module would still be closely tied towards the Linux VFS and communication towards other address spaces in L4Re would be an issue. If the kernel module should be used, an adapter from the L4Re VFS towards the Linux VFS would still be necessary. Therefore I decided to reimplement my own Server part, tied directly to L4Re.

Figure 3.1 shows a design that does not require any change to the FUSE library at all. As on Linux, the FUSE library runs in the same address space as the file system. Communication works exactly the same way as on Linux, which means the FUSE library reads from and writes to a character device. This design requires an L4Re server, that provides a character device with `open()`, `read()` and `write()` calls, towards the FUSE library. A file system server, which receives calls from clients and translates them into FUSE requests is also necessary.

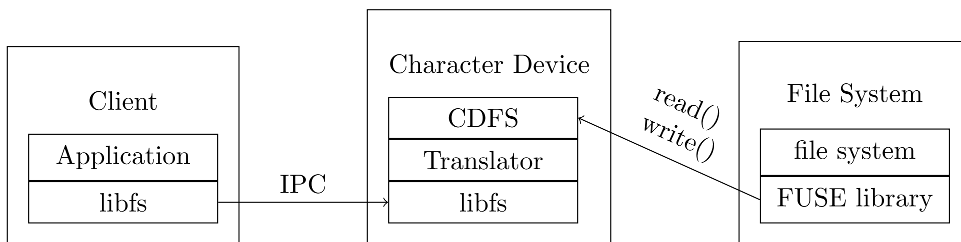


Figure 3.1.: Using a character device server

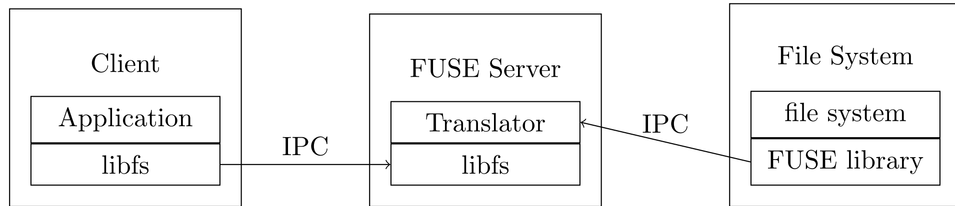


Figure 3.2.: Using IPC directly

The FUSE library offers the possibility to change the communication channel via function pointers. Changes to the communication channel are self-contained and therefore acceptable. Communication can be changed to use Inter-Process Communication (IPC) between library and server directly. This will result in the design shown in Figure 3.2. The drawback of this design is that it still needs two IPC calls, and therefore four address space switches for each request.

On Linux the main reason why library and kernel module do not share an address space is that one is part of the kernel – running in kernel mode – while the other one is not. The L4Re Virtual File System already runs in user mode, therefore both parts of FUSE will run in user mode. Consequently they can share an address space. When the FUSE server and the FUSE library share an address space, the two modules can communicate with shared memory. This design, depicted in Figure 3.3 needs only one IPC per request and only two address space switches.

As a consequence of this design, each file system has its own FUSE server, making the servers smaller, because there is no need for the server to manage different FUSE file systems.

Finally every application that makes a request to the Server has a capability on that Server. This allows for per file system access control by utilizing L4Re's capability system.

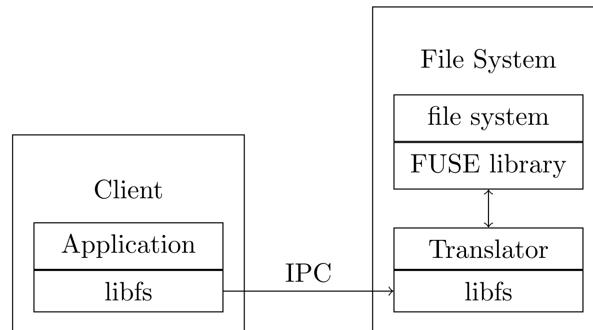


Figure 3.3.: FUSE server and library share and address space

The FUSE library and the libfs server, explained in Section 2.3, both implement a server loop. The libfs server loop waits for IPC calls from clients, processes the IPC

calls and waits again once processing is finished. The FUSE library reads requests from the kernel module, processes the requests and writes a reply back. If the FUSE library and server share an address space the two loops will block each other.

There are two ways to solve this problem. It is possible to put the FUSE library in a separate thread and exercise its server loop the same way Linux does. It will be necessary to provide the two threads with a method to wake each other up, so the server can wake up the library whenever there is a request and the library can wake up the server whenever there is a reply. This way no change to the FUSE library's server loop will be necessary, but the thread ping-pong possibly introduces significant overhead.

A single threaded version will need one more patch to the FUSE library, but have less overhead. I built both versions to be able to measure the overhead the second thread introduces and if the performance benefit is worth the maintenance effort the additional patch requires.

3.1. FUSE Server parts

File system calls from clients will arrive at the FUSE Server via IPC, through libfs, which I explained in Section 2.3. I call this part `LibfsServer`. A second part, which I call `Translator`, translates all calls from libfs into FUSE requests. The `RequestProvider` is responsible to provide requests to the FUSE library. Figure 3.4 shows a call graph, while Figure 3.5 shows a UML class diagram.

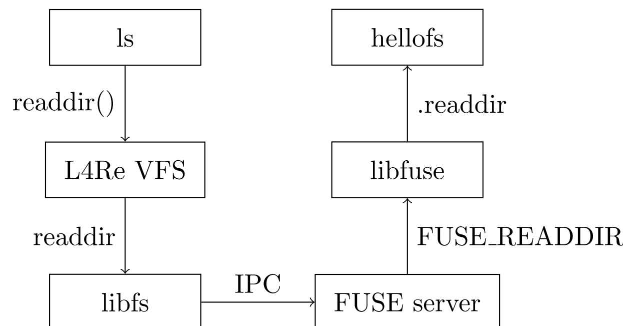


Figure 3.4.: The architecture of FUSE on L4Re

The FUSE library uses nodeids to identify nodes in the file system. The VFS uses either path names or file handles to identify files. The FUSE server will need a translation from one to the other. The FUSE library can handle this translation with a `FUSE_LOOKUP` request. Every `FUSE_LOOKUP` request transfers the name of the node being looked up and the nodeid of that node's parent node. Among other data, like the node's size, the FUSE library replies with the appropriate nodeid.

Entries in a directory are abstracted by a `Node` class. Every `Node` has a name, a path and a nodeid, which is the same information the FUSE library keeps. The FUSE

server must know nodeids, because they are used to identify a node in communication between library and server. If the `Node` is currently open it also has a file handle. A `Node` can be either a `File` or a `Directory`.

Directories differ from files in their behavior for `open()`, `close()` and `read()` and contain other files and directories. The other behavior is the same, which is why both use `Node` as a common base class and the `Directory` class knows about all `Node`s the corresponding directory contains. This part of the design applies the Composite pattern [GHJV94].

If a path has multiple components, the FUSE server must lookup up each of these components individually. This results in lots of `FUSE_LOOKUP` requests for files, that reside deep in the file system. To minimize the amount of `FUSE_LOOKUP` requests, I use a node cache.

If the FUSE server asks the `NodeCache` object for a file with an unknown nodeid, the `NodeCache` issues a lookup request on that path and creates nodes for each directory it enters in the cache. Once the `NodeCache` looks up the requested file, the `NodeCache` also adds the file to its cache. This way the server only needs to look up a path on first use. The server removes unused files from the cache, to avoid unnecessary memory consumption.

A `RequestContainer` class stores the requests to the FUSE library. This class handles setting the correct opcodes and stores the data transmitted to the FUSE library and back. The actual header and data that get transmitted use the same structures that the FUSE library and the FUSE kernel module use in Linux.

There is a special class for each type of request FUSE distinguishes, each of these inherits from the base `RequestContainer` class and overwrites methods that add data to the request or get replies from it.

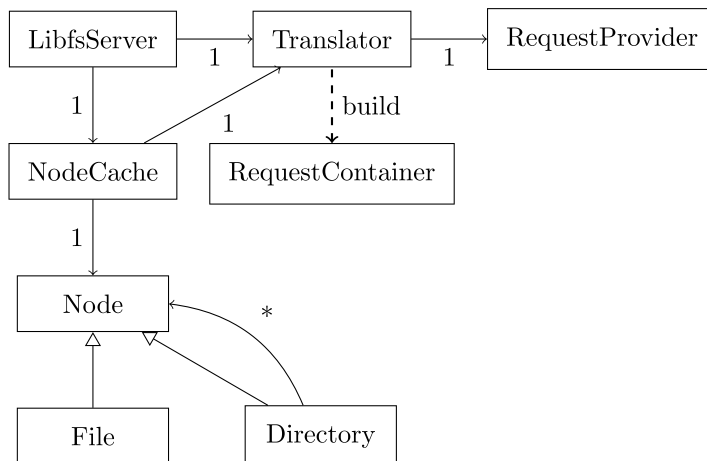


Figure 3.5.: Class diagram for the FUSE Server

4. Implementation

4.1. Example Request handling

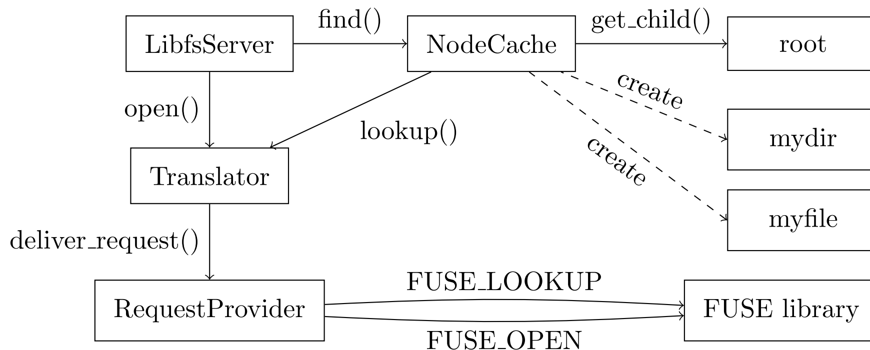


Figure 4.1.: Handling an example request

In this Section I will explain how the system handles requests before it passes them on to the FUSE library and how the system processes replies that the FUSE library returns. As an example, illustrated by Figure 4.1, I will use an `open()` call for the file with the pathname *mydir/myfile*. At the time the client makes the call, the `NodeCache` is empty.

Once the request reaches the FUSE Server via libfs, the `LibfsServer` calls the `find_ent()` function on the `NodeCache` to get the `File` object for *mydir/myfile*.

For an empty cache, the only node that exists in the node cache is the root directory. In order to get the nodeid of the *mydir* directory, the `NodeCache` object calls the `get_child()` function on the root `Directory` object. If the object is already in the node cache, this will return the `Directory` object for mydir. In this case it is not yet in the cache, so the function returns `NULL`.

In order to get the directory into the cache, the `NodeCache` calls the `lookup` function on the `Translator`, which creates a lookup request and calls the `deliver_request()` method on the `RequestProvider`. This will deliver a synchronous request to the FUSE library.

Once the FUSE library handled the request, the `Translator` passes the result back to the `NodeCache`, which creates the appropriate cache entry for the *mydir* directory.

This process is repeated for the *myfile* file. After the library finished the lookup on the file, the `NodeCache` creates a cache entry for the file and returns a `File` object for

myfile to the `LibfsServer`.

To translate the `open()` request for the FUSE library, the `LibfsServer` calls the `open()` method on the `Translator` object and relays the open options set by the client application and the `File` object received from the `NodeCache`. The `Translator` creates an open request and passes it to the `RequestProvider`, which in turn passes the request on to the FUSE library.

When the library has finished processing the request, the `RequestProvider` copies the results into the open request, then the `Translator` checks for errors and passes the result back to the `LibfsServer`. If there are no errors, the `LibfsServer` attaches the file handle, assigned by the FUSE library, to the cache entry for the file and returns the file handle back to the client.

Some file systems, like NTFS-3G, do not set a file handle for an open file. The L4Re VFS expects a file handle as a return value from an open call. As a workaround for these file systems, the FUSE server returns the nodeid of the file as a file handle.

4.2. FUSE Server

I implemented all parts and abstractions mentioned in Chapter 3 as C++ classes. For all methods the FUSE library needs to access, C wrapper functions are provided.

4.2.1. LibfsServer

The `LibfsServer` class provides an Inter-Process-Communication interface for clients. Every request to a FUSE file system arrives in this class. It invokes the `NodeCache` to look for nodes or creates new nodes itself and passes nodes and options to the `Translator` object.

Most file systems assign a file handle to every file that has been opened at least once. This file handle is a unique number, that the VFS can use to reference a file in subsequent calls. If a file is opened multiple times it gets the same file handle from the file system. When the file is closed, the Server needs to check if it has been opened multiple times, before it can be removed from the cache. To allow this every file is reference counted.

For symlinks the fuse-ext2 implementation returns the same file handle as for the file the symlink points to. Therefore the `LibfsServer` corrects the reference count for the original file if a symlink is opened. Files and links to files are referenced by different names. If a file is removed it is safe to remove the entry to that file from the cache.

4.2.2. Translator

For every call made, the `Translator` creates a new request, assigns a unique identifier and adds the arguments to the request. The `Translator` passes all requests to an instance of the `RequestProvider` class, which sends it to the FUSE library. Replies are translated back to a format libfs understands by the `Translator`.

In order to be able to work with different kernel modules, the FUSE library on Linux can handle different versions of the protocol the FUSE library and kernel module use to communicate. There is also a number of options, that allow for larger write buffers or asynchronous reads. The `Translator` is responsible for negotiating these options with the FUSE library after the FUSE server has been started. On L4Re reads are done synchronous and buffer sizes are set to the page size.

4.2.3. Requests

Requests get a unique identifier, an opcode and the appropriate nodeid from the `Translator` class on creation. This data is saved in an instance of the `RequestContainer` class. After the request has been created, the `Translator` class attaches the request header, as a `fuse_in_header` and the appropriate data to the request.

For small requests the reply data is copied from the request into a new buffer, because the FUSE library `free()`s all buffers before the Server has a chance to handle the replies. For large reads, this is slows down performance too much and replies are not copied into another buffer. The buffer used for read is the same buffer, the `LibfsServer` uses to communicate with clients. I made sure, that the buffer is not touched by the FUSE library. This way, copying large reads unnecessarily is avoided.

4.2.4. RequestProvider

The communication channel between FUSE library and server is abstracted by the `RequestProvider` class.

The `struct fuse_in_header` and all data for every request is copied from the request to a consecutive memory block, when the request is transfered to the FUSE library. The library expects the data in this format and writes answers the same way.

For large writes avoiding this copy results in a performance benefit. Copying the data can be avoided if the libfs client writes its data into the libfs buffer with an offset, so the `Translator` can put the `fuse_in_header` and the `fuse_write_in` before the write data received from the client. I implemented a version, where this is done to measure how big the performance benefit will be.

Pending requests are sent to the FUSE library in a first in, first out manner. A request that has been transfered to the library, but is not yet finished is kept in a map, where the request's id is used as a key. This is used by the `RequestProvider` to find the request, once a reply arrives. Now the `RequestProvider` attaches the reply's header and data to the request.

4.2.5. Node Caching

The `File` and `Directory` classes inherit from the `Node` class and implement the `is_dir()` function. The `Directory` class also has a standard C++ map, that contains all `Nodes` in this directory. The key for this map is the name of the `Node`. This way all nodes form a tree, which can be traversed to find every node in the cache. The time of the traversal depends on the depth of the path.

If a `Node` is opened by at least one client it is saved in a standard C++ map, kept by the `NodeCache` with the file handle as key. For calls that are done on file handles instead of paths, the correct `Node` can be looked up directly in this map.

4.3. Changes to the FUSE library

According to the design discussed in Chapter 3, I changed the communication channel between the FUSE library and the FUSE server.

In order to deliver a request, the `RequestProvider` serializes the request and calls the library to process the request. For the two threaded version, this means calling `trigger()` on an `Irq` object, to wake up the library thread and then calling `receive()` on a different `Irq` object to wait for the library's reply. In the single threaded version, the FUSE server calls the FUSE library's `fuse_ll_process()` function directly. This function unpacks every request and calls the appropriate handlers in the FUSE library.

The single threaded version does not use the FUSE loop to read requests or write answers. Instead I patched the `fuse_loop()` function to call the L4Re server loop for the `LibfsServer` object. The patch is small and measurements in Chapter 5 show that it improves read throughput significantly.

Usually the FUSE library writes the reply data for a `read()` request into a buffer, that the library allocates itself. In order to provide the data to the client application, the FUSE server needs to copy that data into a read/write buffer used to communicate with the libfs client. This copy operation is not necessary on L4Re, because the FUSE server and library run in the same address space. However, the operation slows read performance down significantly. I patched the FUSE library to use a buffer provided by libfs instead.

In the two threaded version, it is better to copy small amounts of data from the library to the client, because then the FUSE library can start cleaning its data structures, while the FUSE server is still using its copy of the data for a reply to the client. In this case the FUSE server thread and the FUSE library thread can run in parallel. If the data is not copied, the FUSE library has to wait for the FUSE server to finish using the data, before the library can clean up. Once the library has cleaned up, the server can issue a reply to the client. This effectively adds two extra thread switches.

On Linux the FUSE file system calls the `fuse_mount_sys()` function to set up the communication between the FUSE library and the FUSE kernel module via `/dev/fuse`, the function is expected to return a file descriptor for the character device, which is not used any more. On L4Re the function just returns an arbitrary number.

4.4. Libfs

The version of libfs that existed in L4Re did not support the creation of hard or soft links. Support to read those links was also missing.

In order to fully support file systems, that have those features, I implemented a patch for libfs that adds the three functions `symlink()`, `link()` and `readlink()`.

This patch was straight-forward.

Similarly as the reply for a `read()` request, the data for a `write()` request is copied from a buffer managed by Libfs into a buffer managed by the FUSE library. The copy operation is done by the FUSE server, which also writes the `fuse_in_header` into the buffer, before the data. The main problem is, that there is no space for the header in the buffer with the data. In order to measure writes without copying the data I patched libfs to write the data into the buffer with an offset.

4.5. Block Device Server

There is no block device server on L4Re yet. This means that it was not possible to test the code on real disks. For now, to test the functionality of a specific file system, one needs to create a file formatted with that file systems in another OS and load that file into L4Re's romfs. Before the FUSE Server is started, this file is copied from romfs to tmpfs to make it writable. The ext2 file system can also create a formatted file in tmpfs automatically. Of course writes are not permanent with these methods, because everything is stored in the system's main memory.

In order to be used by legacy file systems the Block Device Server would need to implement the API, the file systems already use. For the ext2 file system this is defined by a structure that provides function pointers to open I/O channels, set block sizes and read and write blocks. The architecture could be similar to libfs, with a client plugin, that is attached to the file system and a server, that provides the API via IPC. Caching and asynchronous block handling would benefit the performance of the system.

4.6. File systems

File systems written for FUSE must implement the FUSE API or the FUSE lowlevel API. It should be possible to port these file systems with no or only minor modifications, as long as other dependencies are met.

I was able to compile and use an implementation of the exFAT [fusa] file system, a FUSE port of ext2 [Akc] and the NTFS-3G file system [Sza] on the L4Re FUSE server.

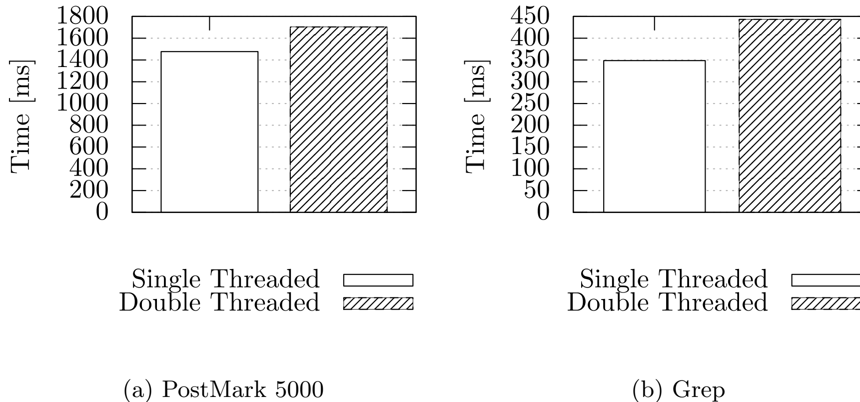
5. Evaluation

In order to evaluate the performance of the FUSE server on L4Re, I did several measurements on L4Re and Linux by replaying traces of an untar operation of a maildir archive and a grep through the files created by the untar, as well as a trace of the PostMark file system benchmark [Kat97].

The traces were recorded and played back with the benchmarking module used for the Virtual Private File System [WH08]. This module consists of two parts. A `TraceRecorder` is used to record the output of an `strace` of a program and transform this output into C code, that exercises calls to the C library and the POSIX interface. A `TracePlayer` runs the code on Linux and L4Re and measures the time taken to exercise the trace.

On Linux, it was necessary to do all measurements on a ramdisk to create comparable results between L4Re - which has no block device server - and Linux. On both systems the ramdisk contained a fixed size file, which was formatted with the measured file system. This file was mounted using the FUSE file system implementation. On Linux ramfs was used as backing storage, while on L4Re I used tmpfs. This method constrained the measurements to benchmarks which use very little memory, because the whole benchmark had to fit in main memory.

The Linux system was a fresh Debian testing installation, with Linux kernel version 3.2 and version 2.8.7 of the FUSE library. For L4Re revision 42131 checked out on December 12, 2012 was used. All measurements were done on an AMD Phenom 9550 CPU with 2200 MHz and 2 GiB RAM.



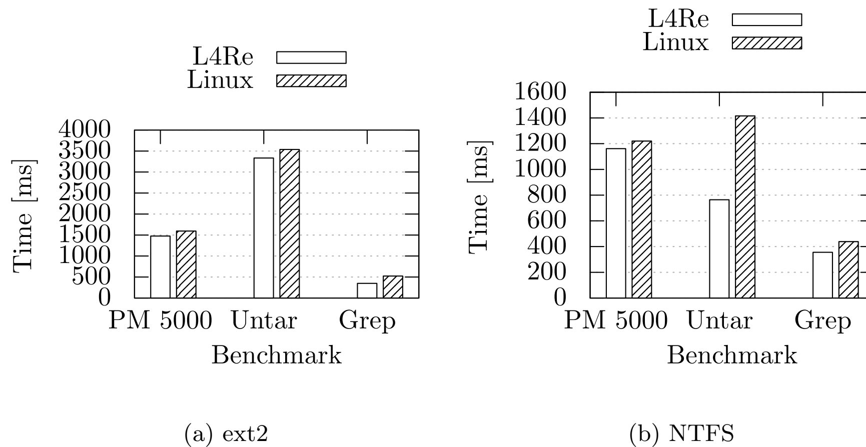
(a) PostMark 5000

(b) Grep

Figure 5.1.: The average speed on ext2 with one and two threads

First I benchmarked the PostMark trace on the two versions of the L4Re FUSE server described in Chapter 4. Figure 5.1 shows that the single threaded version is faster for the PostMark 5000 trace as well as the grep trace. For the PostMark 5000 trace, the single threaded version is 13% faster on average, while the grep trace is 22% faster. If the reply from a `read()` call is not copied between the FUSE library and the FUSE server, two extra thread switches are necessary, as explained in Chapter 4. Therefore the grep trace, which issues mainly `read()` calls, benefits greater from the single threaded version. Because of these results the benchmarks for Linux are compared to the single threaded version.

I did the benchmarks on multiple file systems, the first being a FUSE implementation of the ext2 file system [Akc].



(a) ext2

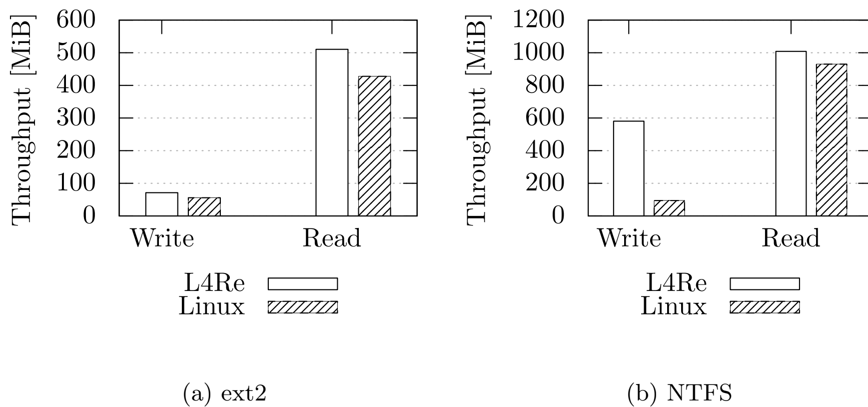
(b) NTFS

Figure 5.2.: The average speed of the benchmarks on different file systems

Figure 5.2a shows the average speed of the benchmarks on the different platforms. The L4Re version is a little bit faster for all benchmarks. FUSE needs two address space switches on Linux and on L4Re. One from the client application into the file system and another switch to go back to the client. Linux needs an additional kernel entry to perform reads and writes on the backing storage. For this reason, the benchmarks should be repeated once there is a block device server for L4Re.

I also did the benchmarks on the widely used NTFS-3G file system [Sza]. The NTFS-3G file system is faster than ext2 on both operating systems, as shown in Figure 5.2b. The untar trace is a lot slower on Linux, than it is on L4Re. The main reason is that, NTFS needs to write the file itself and the journal for each `write()` call. Each of these writes to the backing storage needs a kernel entry and exit on Linux, while this is not needed on L4Re. With a block device server this will likely change on L4Re.

Figure 5.3 shows throughput for the different file systems. Throughput was measured with the trace of a copy of a 175MiB video file. For the write throughput calls to `open()` and `close()` the file, as well as `write()` of the copy operation were traced. Likewise for the read throughput calls to `open()` and `close()` and `read()` were traced. For

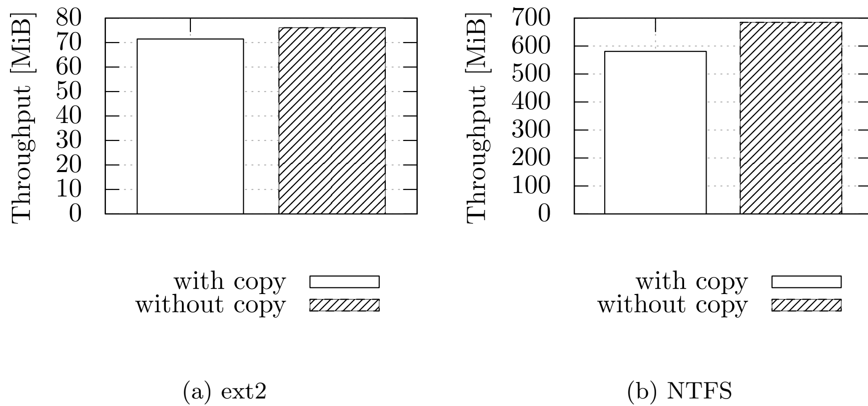


(a) ext2 (b) NTFS

Figure 5.3.: Throughput on the different file systems

NTFS-3G write throughput is significantly higher on L4Re than on Linux. This is consistent with the results of the untar benchmark taken earlier, which mainly writes data.

On L4Re the FUSE server copies all write data from the buffer used by libfs into the buffer used by FUSE. In order to further improve on write performance, I modified libfs, to leave 4 KiB offset on write calls in the buffer. This way the FUSE server can fit the `fuse_in_header` and the `fuse_write_in` before the write data without the need to copy it for the FUSE library.

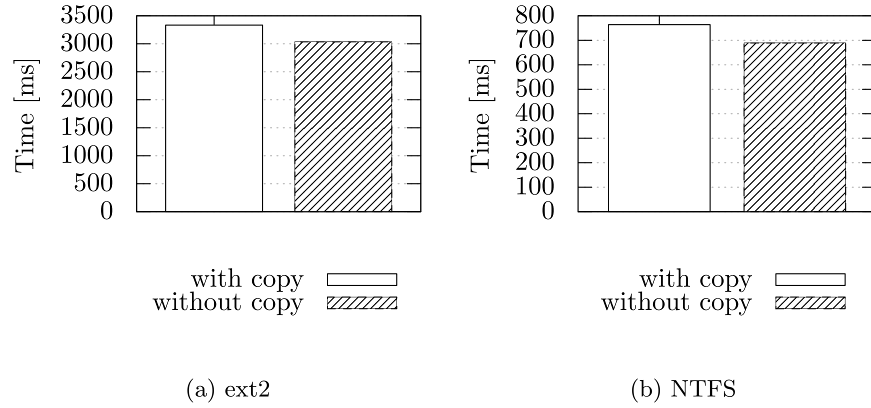


(a) ext2 (b) NTFS

Figure 5.4.: Write throughput with and without copied write buffer

I measured the throughput again with the both file systems and compared the results, which can be seen in Figure 5.4. On ext2 throughput increases about 5 MiB per second or 7%. On NTFS throughput increases about 100 MiB/s or 17%. For

the untar trace, both file systems gain about 10% in the copy-free version, visible in Figure 5.5. The trace used to measure throughput opens one file and reads the whole file with a 32 KiB buffer and closes the file in the end. The untar trace uses smaller buffers, opens and closes multiple files and does a `stat()` call for each file.



(a) ext2 (b) NTFS

Figure 5.5.: Untar with and without copied write buffer

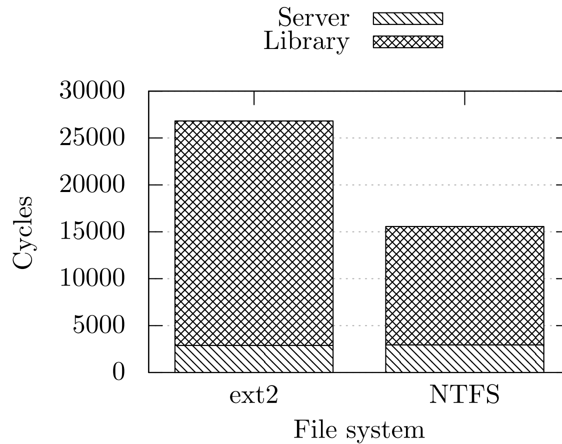


Figure 5.6.: Cycles spent in Server and Library in the different file systems

The last measurement taken, was the cycle count for a `stat()` call. Each `stat()` call takes a total of 122650 cycles on average, with a standard deviation of 2080 cycles. This was measured with a trace that does only `stat()` calls. The cycle count at entry into the FUSE server was measured for each call to `stat()`. Therefore the 122650 cycles are round trip time, they include IPC calls, libc code and VFS code. The implementation of the L4Re VFS internally issues a call to `open()`, `fstat()` and `close()` in order to

perform a `stat()` call. The cycles spent for the `open()` and `close()` calls are also included in the cycle count.

I also measured the cycles spent on just the `fstat()` call. A graphical representation for the ext2 file system and NTFS is shown in Figure 5.6. The red box at the bottom shows the amount of cycles spent in the FUSE server, on average 2941 cycles (with a standard deviation of 53) are spent inside the FUSE server. The time spent in the FUSE library varies greatly between the ext2 and the NTFS file system. For the NTFS implementation the average is 12624 cycles, while the ext2 file system takes almost twice as many cycles.

6. Conclusion and Further Work

I have implemented a FUSE server for L4Re with only minor modifications to the FUSE library. To avert the FUSE loop I changed four lines in `fuse.c`. Another 5 lines are changed so the result of a `read()` call does not have to be copied from the library to the server. A `diff` of these changes can be found in Appendix B. These changes will need to be reapplied to new versions of the FUSE library. The changes to the communication channel only redefine function pointers or change files that are operating system specific, like `mount.c`.

With slightly less than 2000 lines of code, the FUSE server for L4Re is about one third of the size of the Linux kernel module. There are two reasons for the smaller size. One, the L4Re server lacks support for asynchronous I/O. The other reason is that on L4Re communication with the FUSE library is much cleaner. There is no need to put requests into lists and wait for the FUSE library to read requests. A request is transfered to the FUSE library by simply calling the library's request processing function.

With the FUSE server I was able to run the fuse-ext2 and the NTFS-3G file systems successfully. Once there is a block device server for L4Re, it should be possible to run the file systems without any modifications. Porting other FUSE based file systems should be an easy task, as long as other requirements are present on L4Re.

For benchmarks done with a single client, the L4Re FUSE server performs faster than the Linux version of FUSE. However these measurements are not really comparable, because there is no block device server on L4Re and FUSE on Linux has more kernel entries and exits in order to utilize the backing storage than L4Re.

The FUSE server I built runs single-threaded, with the FUSE library running single threaded as well. For high file system loads with multiple clients it will improve system speed if there is a multi-threaded version of the Server. Implementing this efficiently requires a thread pool, in which one thread waits for IPC calls. Once an IPC call arrives this thread begins work and the next thread starts waiting for IPC calls. This way there is no bottleneck. The performance of this is greatly dependend on how fast IPC gates can be rebinded.

Implementing `write()` and other calls in a way, that the data will not need to be copied into a buffer for the FUSE library can improve performance by around 10%. To implement this in a generic way requires a change to the libfs buffer. The libfs client needs to get the capability to the buffer with an offset of one page. This way the FUSE server can write the header for the FUSE library directly before the data it gets from the client.

Access control towards the FUSE server is very coarse grained right now. If a client application has a capability for the server's IPC gate, it is granted read and write access to the file system. In order to provide fine grained access control, the FUSE

server needs to grant individual capabilities for each client. This can be done on a per-file basis.

A. FUSE operations

Lookup

Requests of this type are used to get information, like the nodeid, about a node. The nodeid in the `fuse_in_header` must be set to the nodeid of the parent node.

- *Opcode:* `FUSE_LOOKUP`
- *Input:* The name of the node
- *Reply:* `fuse_entry_out`

Forget

Remove nodes from the FUSE library's node cache.

- *Opcode:* `FUSE_FORGET` to remove a single node and `FUSE_BATCH_FORGET` to remove multiple nodes
- *Input:* `fuse_forget_in` and one `fuse_forget_link` for each node
- *Reply:* none

Getattr

Get the attributes of a node.

- *Opcode:* `FUSE_GETATTR`
- *Input:* `fuse_getattr_in`
- *Reply:* `fuse_attr_out`

Setattr

Set the attributes of a node. Also returns the attributes of the node.

- *Opcode:* `FUSE_SETATTR`
- *Input:* `fuse_setattr_in`
- *Reply:* `fuse_attr_out`

Readlink

Reads the contents of a link.

- *Opcode:* `FUSE_READLINK`
- *Input:* none
- *Reply:* filename the link points to

Symlink

Creates a symlink. The nodeid in the `fuse_in_header` must be set to the nodeid of the parent node.

- *Opcode:* `FUSE_SYMLINK`
- *Input:* filename of the target file, filename of the link
- *Reply:* `fuse_entry_out`

Mknod

Create a new node in the file system. The nodeid in the `fuse_in_header` must be set to the nodeid of the parent node.

- *Opcode:* `FUSE_MKNOD`
- *Input:* `fuse_mknod_in`, filename of the new node
- *Reply:* `fuse_entry_out`

Mkdir

Create a new directory. The nodeid in the `fuse_in_header` must be set to the nodeid of the parent node.

- *Opcode:* `FUSE_MKDIR`
- *Input:* `fuse_mkdir_in`, name of the new directory
- *Reply:* `fuse_entry_out`

Unlink

Remove the link to a node, if the last link is removed, the node itself is removed, as well. The nodeid in the `fuse_in_header` must be set to the nodeid of the parent node.

- *Opcode:* `FUSE_UNLINK`
- *Input:* filename of the node
- *Reply:* none

Rmdir

Remove a directory.

- *Opcode:* `FUSE_RMDIR`
- *Input:* name of the directory
- *Reply:* none

Rename

Rename a node.

- *Opcode:* `FUSE_RENAME`
- *Input:* `fuse_rename_in`, old name of the node, new name of the node
- *Reply:* none

Link

Add a hard link to a node.

- *Opcode:* `FUSE_LINK`
- *Input:* `fuse_link_in`, name of the link
- *Reply:* `fuse_entry_out`

Open

Open a node.

- *Opcode:* `FUSE_OPEN` for files, `FUSE_OPENDIR` for directories
- *Input:* `fuse_open_in`
- *Reply:* `fuse_open_out`

Read

Read data from a file.

- *Opcode:* `FUSE_READ`
- *Input:* `fuse_read_in`
- *Reply:* the data read

Readdir

Read the contents of a directory.

- *Opcode:* `FUSE_READDIR`
- *Input:* `fuse_read_in`
- *Reply:* One `fuse_dirent` for each node in the directory

Write

Write data into a file.

- *Opcode:* `FUSE_WRITE`
- *Input:* `fuse_write_in`, data to write
- *Reply:* `fuse_write_out`

Statfs

Return the attributes of the file system.

- *Opcode:* `FUSE_STATFS`
- *Input:* none
- *Reply:* `fuse_statfs_out`

Release

Close a file or directory.

- *Opcode:* `FUSE_RELEASE` and `FUSE_RELEASEDIR` for directories
- *Input:* `fuse_release_in`
- *Reply:* none

Fsync

Write file and directory buffers to disk.

- *Opcode:* `FUSE_FSYNC` and `FUSE_FSYNCDIR` for directories
- *Input:* `fuse_fsync_in`
- *Reply:* none

Setxattr

Set extended attributes.

- *Opcode:* `FUSE_SETXATTR`
- *Input:* `fuse_setxattr_in`
- *Reply:* none

Getxattr

Get extended attributes.

- *Opcode:* `FUSE_GETXATTR`
- *Input:* `fuse_getxattr_in`, name of the attribute
- *Reply:* `fuse_getxattr_out`

Listxattr

List extended attributes.

- *Opcode:* `FUSE_LISTXATTR`
- *Input:* `fuse_getxattr_in`
- *Reply:* `fuse_getxattr_out`

Removexattr

Remove an extended attribute.

- *Opcode:* `FUSE_REMOVEXATTR`
- *Input:* name of the attribute
- *Reply:* none

Flush

Clean up data buffers of file.

- *Opcode:* `FUSE_FLUSH`
- *Input:* `fuse_flush_in`
- *Reply:* none

Init

Initialize connection between kernel module and library. Negotiate protocol and set options. This must be the first request transfered from kernel module to library.

- *Opcode:* `FUSE_INIT`
- *Input:* `fuse_init_in`
- *Reply:* `fuse_init_out`

Getlk

Tests for a POSIX file lock.

- *Opcode:* `FUSE_GETLK`
- *Input:* `fuse_lk_in`
- *Reply:* `fuse_lk_out`

Setlk

Acquire, modify or release a POSIX file lock.

- *Opcode:* `FUSE_SETLK` or `FUSE_SETLKW`
- *Input:* `fuse_lk_in`
- *Reply:* none

Access

Get file access permissions.

- *Opcode:* `FUSE_ACCESS`
- *Input:* `fuse_access_in`
- *Reply:* none

Create

Atomic create and open.

- *Opcode:* `FUSE_CREATE`
- *Input:* `fuse_create_in` and name of the new file
- *Reply:* `fuse_entry_out` and `fuse_open_out`

Interrupt

Transfers an interrupt request to the library.

- *Opcode:* `FUSE_INTERRUPT`
- *Input:* `fuse_interrupt_in`
- *Reply:* none

Bmap

Map block index in a file to block index in the block device.

- *Opcode:* `FUSE_BMAP`
- *Input:* `fuse_bmap_in`
- *Reply:* `fuse_bmap_out`

Destroy

Clean up the file system on file system exit.

- *Opcode:* `FUSE_DESTROY`
- *Input:* none
- *Reply:* none

Poll

Poll if node is ready for I/O.

- *Opcode:* `FUSE_POLL`
- *Input:* `fuse_poll_in`
- *Reply:* `fuse_poll_out`

B. FUSE library changes

```
> #include <l4/fuse/wrapper.h>
>
2565c2563,2567
>       buf = srv_rw_buf();
---
<       buf = (char *) malloc(size);
<       if (buf == NULL) {
<               reply_err(req, -ENOMEM);
<               return;
<       }
2581a2584
<       free(buf);
3462,3463c3465,3468
>       assert(f->se != NULL);
>       return server_loop(f->se);
---
<       if (f)
<               return fuse_session_loop(f->se);
<       else
<               return -1;
```

C. Glossary

CUSE Character Device in Userspace

FUSE Filesystem in Userspace

L4Re L4 Runtime Environment

NFS Network File System

PM PostMark

puffs Pass-to-Userspace Framework Filesystem

VFS Virtual File System

Bibliography

[Akc] Alper Akcan. fuse-ext2. Website. Available online at http://sourceforge.net/projects/fuse-ext2/; visited on May 23, 2013.

[cro] Website. Available online at http://www.crossmeta.org/redmine/news/4; visited on May 23, 2013.

[dok] Website. Available online at http://dokan-dev.net/en/; visited on May 23, 2013.

[Fit93] Jeremy Fitzhardinge. Userfs. Website, 1993. Available online at http://www.goop.org/~jeremy/userfs; visited on May 23, 2013.

[fusa] fuse-exfat. Website. Available online at http://code.google.com/p/exfat/; visited on May 23, 2013.

[fusb] Fuse filesystem on freebsd wiki. Website. Available online at https://wiki.freebsd.org/FuseFilesystem; visited on May 23, 2013.

[fusc] Fuse on solaris. Website. Available online at http://hub.opensolaris.org/bin/view/Project+fuse/WebHome; visited on May 23, 2013.

[FUSd] Fuse4x. Website. Available online at http://fuse4x.github.com/; visited on May 23, 2013.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman, 10 1994.

[Kan07] Anti Kantee. puffs - pass-to-userspace framework file system. 2007.

[Kat97] Jeffrey Katcher. Postmark: a new file system benchmark. Network Appliance Tech Report TR3022, October 1997.

[KC07] Anti Kantee and Alistair Crooks. Refuse: Userspace fuse reimplementation using puffs. 2007.

[l4r] L4re. Website. Available online at http://os.inf.tu-dresden.de/L4Re/ visited on May 23, 2013.

[Maca] Macfuse. Website. Available online at http://code.google.com/p/macfuse/; visited on May 23, 2013.

[Macb] Macfuse-faq. Website. Available online at `http://code.google.com/p/macfuse/wiki/FAQ`; visited on 2012-12-20.

[Maz01] David Mazières. A toolkit for user-level file systems. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 261–274, Berkeley, CA, USA, 2001. USENIX Association.

[min11] Fuse - minix. Website, 2011. Available online at `http://wiki.minix3.org/en/SummerOfCode2011/Fuse`; visited on May 23, 2013.

[SKK+90] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, David, and C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39:447–459, 1990.

[SWSZ07] Richard P. Spillane, Charles P. Wright, Gopalan Sivathanu, and Erez Zadok. Rapid file system development using ptrace. In *Proceedings of the 2007 workshop on Experimental computer science*, ExpCS '07, New York, NY, USA, 2007. ACM.

[Sza] Szakacsits Szabolcs. Ntfs-3g. Website. Available online at `http://www.tuxera.com/community/ntfs-3g-download/`; visited on May 23, 2013.

[Szea] Miklos Szeredi. Filesystem in userspace. Website. Available online at `http://fuse.sourceforge.net/`; visited on May 23, 2013.

[Szeb] Miklos Szeredi. sshfs. Website. Available online at `http://fuse.sourceforge.net/sshfs.html`; visited on May 23, 2013.

[Wag] Vincent Wagelaar. Smb for fuse. Website. Available online at `http://www.ricardis.tudelft.nl/~vincent/fusesmb/`; visited on May 23, 2013.

[WD98] Assar Westerlund and Johan Danielsson. Arla-a free afs client. In *In Proceedings of the 1998 USENIX, Freenix track*, page USENIX., 1998.

[WH08] Carsten Weinhold and Hermann Härtig. Vpfs: building a virtual private file system with a small trusted computing base. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, pages 81–93, New York, NY, USA, 2008. ACM.

[ZN00] Erez Zadok and Jason Nieh. Fist: a language for stackable file systems. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '00, pages 5–5, Berkeley, CA, USA, 2000. USENIX Association.