Universiteit Leiden
The Netherlands

# Opleiding Informatica

Linker-Directive Based File System in Userspace:

Introducing LDP_FUSE

Sjors Holtrop

Supervisors:
Alexandru Uta & Kristian Rietveld

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

04/08/2022

**Abstract**

File systems are an important part of modern day computing. Traditionally, they reside in kernel space, which raises the barrier to entry for developing new file systems. Linux' FUSE offers an interface to easily develop a file system in user space. However, FUSE is detrimental to file system performance due to the overhead it adds. This thesis introduces LDP_FUSE, a C library that allows developers to leverage `LD_PRELOAD` for creating user space file systems more easily. Filebench benchmarks show that it degrades performance only slightly (−8%) on workloads where FUSE performs terribly (−91%). Significant challenges remain in order to make this approach viable, such as adding support for memory-mapped IO.
Code available at https://github.com/sholtrop/ldpfuse.

# Contents


**1 Introduction** **1**

**2 Background** **3**

2.1 File systems . . . . . 3

2.2 FUSE . . . . . 3

2.2.1 What is FUSE . . . . . 3

2.2.2 Performance concerns . . . . . 5

2.3 LD_PRELOAD . . . . . 6

2.3.1 What is LD_PRELOAD . . . . . 6

2.3.2 An LD_PRELOAD based file system . . . . . 6

**3 The LDP_FUSE Library** **8**

3.1 Features & Design . . . . . 8

3.2 Examples . . . . . 10

3.2.1 StackFs . . . . . 10

3.2.2 Transparent cryptographic file system . . . . . 11

3.3 Shortcomings . . . . . 11

**4 Related work** **13**

**5 Experiments** **15**

**6 Results** **17**

**7 Conclusion** **19**

**References** **21**

# 1 Introduction

File systems are ubiquitous in modern day computing. They first appeared in the 1960s as a response to the growing need for per-program data storage. At the time, computers were designed to run a single program. This program could therefore presume exclusive access to the computer's disk, and store data on it in any way it saw fit [6]. However, with the advent of multi-program computers, this approach became inadequate. Multiple programs running on a single computer could interfere with one another, e.g. by overwriting each other's data. A more structured approach to storing data was needed [9]. This led to the advent of the first file systems.

A file system organizes the otherwise unorganized sequence of data a disk contains. There are multiple ways to approach this. The most popular of which makes a distinction between two entities: *Files*, which contain raw data, and *directories* (or *folders*), which contain files or (recursively) other directories. This paradigm still forms the foundation of most modern-day file systems, such as those found on Linux [8] and Windows. These operating systems were designed to be *multi-user*, in addition to being multi-program. When they first rose to popularity in the 90s, they therefore brought with them the need for *file permissions*. Because users of such systems should be able to have private data, there should be files that cannot be read by other users.

File systems manage all of this by storing *metadata* per entity: Is it a file or a directory? Who may read it, and who may write it? In addition, this metadata often includes which *blocks* contain the file's contents. In this case, the file system treats the storage space of a disk as a sequence of blocks. It then (de)allocates said blocks for a file as needed. This useful for e.g., when a file outgrows the space that the file system provisioned for it initially. It simply allocates a new block to the file, rather than copying the entire file to a new location with more space. Thus, file systems use metadata for all kinds of benefits.

For the features, ease and flexibility that file systems offer, they necessarily incur a performance cost. Storing, retrieving, parsing and checking metadata is not free. This is exacerbated by the fact that non-volatile storage, such as disks, are one of the slowest parts of a computer system. Many times slower than a processor, in fact. Good file system performance is therefore vitally important to the overall system performance. Another important aspect of file system performance is *where* file system operations happen: Inside the operating system (*kernel space*), running in a privileged CPU mode, or outside the kernel, running as a 'regular' program (*user space*). User space file systems were the norm for the *microkernels* that were common in the mid 1980s [14]. However, popular modern operating systems like Windows and Linux implement their file systems in kernel space.

The file system residing in kernel space adds overhead due to the required *context switches* when switching between user and kernel mode. Additionally, it makes developing a custom file

system harder because of the need to write a kernel module rather than a normal program. Ways to develop file systems in user space do exist. Linux' *Filesystem in Userspace* (FUSE) is the most popular, but has some performance issues. Namely, it adds too much performance overhead to frequent, small operations. More detail on this in Section 4.

In this thesis, I therefore propose a novel approach for developing file systems in user space on Linux-based operating systems. I have named this approach *LD_PRELOAD Filesystem in user space* (LDP_FUSE). It makes the following contributions to the field:

- Offer future researchers of user space file systems an ergonomic abstraction to implement them.
- Provide insight into the feasibility of this kind of file system.
- Outline challenges that must be overcome in order to make improvements to this approach.

The thesis is structured as follows. Section 2 will introduce the necessary technical background, after which I will pose my research questions.
Then, in Section 3, I will introduce my own solution of LDP_FUSE, and explain its design, inner workings and shortcomings. This is followed by a comparison with related work and similar solutions in Section 4. Section 5 contains a set of performance experiments, where file systems implemented in respectively LDP_FUSE and regular FUSE are compared with each other. The results of these experiments are discussed in Section 6. Finally, I will draw an overall conclusion in Section 7.

# 2 Background

This section provides a brief, high-level technical overview of the required background knowledge. It covers the implementation of file systems, how Linux' FUSE operates, and the `LD_PRELOAD` linker directive.

## 2.1 File systems

First, a File systems provide processes an interface for reading/writing data in a structured way, usually from/to an underlying storage device.
A file system consists of at least two layers:

- Logical file system – The application-level interface that provides simple functionalities such as reading from, writing to, creating and deleting files. The OS usually exposes these operations as syscalls.
- Physical file system – The bookkeeping data structures as well as the data itself, that are usually stored on a mass storage device.

[14]
Many operating systems, among which Linux-based ones, have a layer between these two called the *Virtual Filesystem* (VFS). The VFS is a kernel space software layer that can unify multiple underlying physical file systems into a single logical file system interface. In the case of Linux, it also consists of multiple caches (directory cache, inode cache, page cache) to boost performance [4].

Traditionally, file systems are implemented in kernel space. The kernel will have a driver that can interpret and operate on the file system's bookkeeping data structures to store and retrieve files. A major downside to this is that those who want to develop a file system will need to write a kernel module. One solution to this issue is Linux' FUSE, which is discussed in the next section.

## 2.2 FUSE

### 2.2.1 What is FUSE

*Filesystem in user space*, abbreviated as FUSE, is a Linux interface for writing file systems based on a user space program [3]. It consists of two parts:

- A kernel module (`fuse.ko`) that translates file system syscalls to requests that are sent to the user space program.

- A library (`libfuse`) for developing the user space program against. This program answers the kernel module's requests. The program may implement the file system however it wishes: E.g. on-disk, in-memory, or via a cloud-based remote service.

FUSE has become very popular for implementing user space file systems, and over 100 such file systems have been developed with it [15]. Two of the main reasons for the popularity of user space file systems (in general) include:

1. User space code is easier to port and maintain than kernel code.
2. User space code has a lower bar to entry; you can use a wider array of programming languages rather than just system ones like C.

[17]
Why is FUSE the most popular tool to accomplish this? I posit that it is because of its built-in kernel support and excellent high level interface. The following code snippet illustrates this:

```
// Function declarations, implementations
// and #include's are omitted for brevity

struct fuse_operations fuse_ops  = {
  .readdir   = custom_readdir,
  .mkdir     = custom_mkdir,
  .rmdir     = custom_rmdir,
  .getattr   = custom_getattr,
  .open      = custom_open,
  .create    = custom_create,
  .unlink    = custom_unlink,
  .read      = custom_read,
  .write     = custom_write,
  .truncate  = custom_truncate,
};

int main(int argc, char** argv) {
    // At some point...
    fuse_main(argc, argv, &fuse_ops, NULL);
}
```

As one can observe, a FUSE file system implementation can be as easy as passing a struct of function pointers. Each pointed-to function corresponds to a basic file system operation. You

implement these, and then the FUSE library `libfuse` will take care of of all the lower level details of communicating with the kernel module. In conclusion, I believe this is what makes FUSE the most popular choice for user space file systems on Linux.

### 2.2.2 Performance concerns

FUSE is known to degrade file system performance by adding overhead. To better understand this, I have visualized the path a typical read operation takes, based on Vangoor et al.'s analysis of FUSE's inner workings [16]. This is seen in Figure 1.

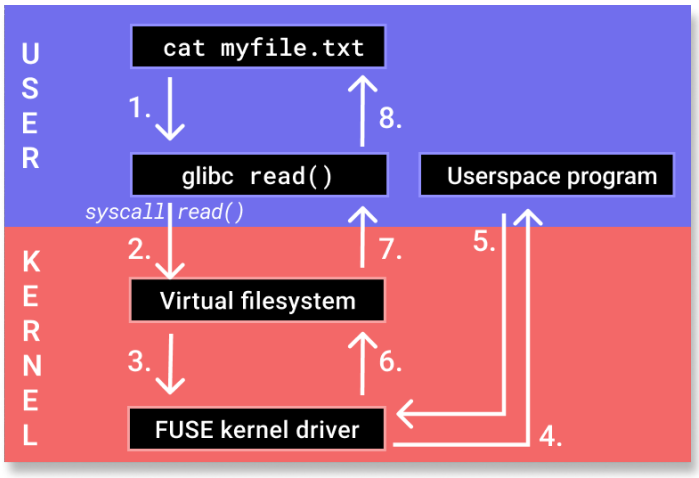


Figure 1: How FUSE handles a read call.

1. User space utility binary `cat` is ran to read `myfile.txt`.
2. This binary invokes the syscall `read` via its glibc wrapper function.
3. The virtual file system resolves what underlying file system the call should go to. It forwards it to the FUSE kernel driver.
4. The FUSE kernel driver translates the syscall to a read request and passes it to the user space program.
5. The user space program retrieves the file's contents however it wishes, and places them in a buffer provided by the kernel module. The read request has now been answered.
6. The kernel driver marks the read request as complete.
7. The user space program is woken up, glibc's `read` function returns.

8. `cat` receives its file data and can continue execution.

Note that the read request will cross the user/kernel space barrier four times. As is well known, there is a minor overhead cost for switching from user to kernel space. In addition, `read`s and `write`s require copying memory between user and kernel space. While FUSE uses the kernel's `splice`[1] function to elide this copying, it only does so for `write`s of $\geq$1 page and `read`s of $\geq$2 pages [16]. Moreover, `splice` can not always avoid copying data. FUSE's performance degradation is a well-known and researched issue. In Section 4, I will go over this in more depth and compare several other solutions.

## 2.3 LD_PRELOAD

### 2.3.1 What is LD_PRELOAD

The `LD_PRELOAD` environment variable specifies shared libraries (`.so` files) that the linker (`ld`) will load before a program is run. Because it is loaded first, it can be used to overwrite existing symbols in a program with a custom implementation. You can therefore use it to change the implementation of C standard library (glibc) functions.

### 2.3.2 An LD_PRELOAD based file system

The standard library contains wrapper functions for most syscalls, e.g. `read`, `write`, `open`. By hooking into these and overwriting them with `LD_PRELOAD`, we can implement any functionality we wish. Figure 2 shows an example of the path a `read` function call would take under such a file system.
Note how compared to Figure 1, there are far fewer steps. In addition, the user-kernel space barrier is never crossed. This will result in no user-kernel mode switches, fewer context switches, and less memory copying [18]. All of the aforementioned are reasons why such a file system could outperform FUSE. By overwriting each file system-related function in this way, we can write an entirely custom file system. Some possibilities include:

- Stackable file systems, implemented on top of a regular one, that transparently add features such as encryption or automatic backup.
- Remote file systems, where programs are actually performing operations on data stored on a cloud service.
- High performance computing file systems.

[1] https://man7.org/linux/man-pages/man2/splice.2.html

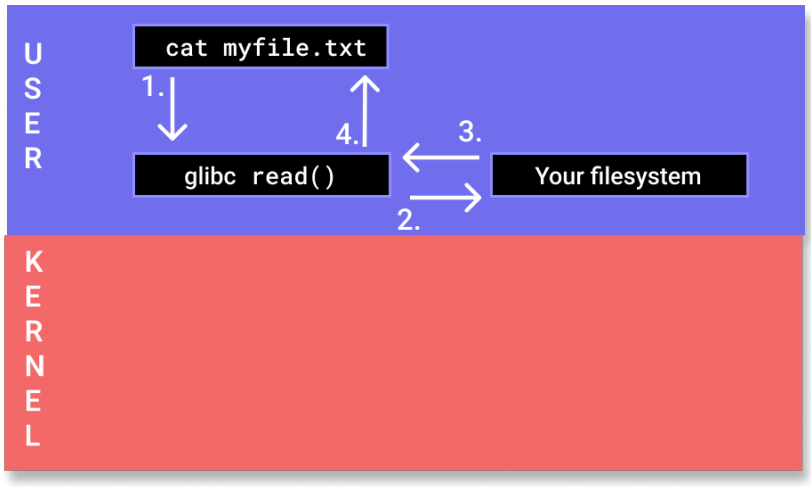


Figure 2: How an `LD_PRELOAD` based file system handles a read call.

However, doing this manually is cumbersome and involves a lot of boilerplate code. For example:

- The small differences in semantics between similar syscall wrappers (e.g. `stat` vs `lstat`).
- Passing along an IO function call to the *original* syscall wrapper in case it is not in your file system.
- To determine that a file is "not in your file system", one will need to manually implement some notion of which directory the `LD_PRELOAD`ed file system is 'mounted' under.

LDP_FUSE is a library that seeks to abstract these problems away, so developers can focus on the actual implementation of the file system itself.
My main research questions for this thesis are:

1. Can an `LD_PRELOAD`-based file system, like LDP_FUSE, be a viable alternative to FUSE?
2. How is the performance of LDP_FUSE compared to FUSE?

# 3 The LDP_FUSE Library

The code for LDP_FUSE is available at: https://github.com/sholtrop/ldpfuse

## 3.1 Features & Design

LDP_FUSE is a single, header-only C library. This is similar to the extremely popular Boost library, and is done for ergonomic reasons. However, it does require explicitly linking `libdl`, e.g. by specifying `-ldl`.
To save users of the library from duplicating code for very similar system calls (e.g. `stat`, `fstat`, `lstat`, and `fstatat`), LDP_FUSE will translate and redirect many such system calls to its most generic version. The user therefore only has to implement that single generic function.
For example:

```
open(const char* pathname, int flags, mode_t mode)
```

is equal to

```
openat(AT_FDCWD, pathname, flags, mode)
```

where `AT_FDCWD` is a constant indicating the `pathname` is relative to the current working directory, thereby having the same semantics as `open`.
LDP_FUSE will overwrite the original glibc symbols for nearly all file system related functions. An overview of which LDP_FUSE function matches which syscall is given in Table 1.

| Syscalls | LDP_FUSE function | Notes |
|---|---|---|
| access | access | |
| faccessat | | |
| euidaccess | | |
| eaccess | | |
| mkdir | mkdir | |
| close | close | |
| close_range | | Flags argument is ignored.<br>Will call close once on each file descriptor in the range. |
| opendir | opendir | |
| creat | open | |
| open | | |
| openat | | |
| truncate | truncate | |
| chmod | chmod | |
| chown | chown | |
| symlink | symlink | |
| rename | rename | |
| link | link | |
| rmdir | rmdir | |
| unlink | unlink | |
| mknod | mknod | |
| mknodat | | |
| readlink | readlink | |
| write | write | |
| pwrite | | |
| lstat | stat | |
| fstat | | |
| stat | | |
| fstatat | | |
| read | read | |
| pread | | |
| getxattr | getxattr | |
| lgetxattr | | WILL follow symbolic links, contrary to the default lgetxattr which interrogates the link itself. |
| fgetxattr | | |

Table 1: List of Linux syscalls that LDP_FUSE overwrites, and any changed semantics.

However users may still want to access the original functions, e.g. when developing a stacked file system on top of an underlying one. This is why every LDP_FUSE function receives a function pointer to the original one as its last argument.

LDP_FUSE knows which directory it is 'mounted' under through an environment variable specifying a path. It intercepts every I/O syscall, checks whether the provided path name or file descriptor is in this 'mounted' path, and if not, simply passes it to the respective original function. This crucial because under Linux a file may also refer to e.g. a block device or a

stream, rather than a 'regular' file. Problems will arise if a user tries to read a stream under LDP_FUSE, as it will translate `read` calls into `pread`, the latter of which does not work on streams. This is avoided by not mounting any such files under LDP_FUSE.

To keep track of which file descriptors are and are not in the file system, LDP_FUSE has its own *open file descriptor table* (OFDT) per process, in which it tracks results of `open` calls. It also uses this OFDT to track open file offsets. This is necessary for a number of reasons:

- Calls like `read` are redirected to `pread`. The latter does not increase the open file's offset in any underlying file system, whereas the former does. LDP_FUSE reimplements these semantics itself.
- Reading `/proc/pid/self/fd` on every call to figure out if the file descriptor is in the file system is slow.
- In the case of no underlying file system, this saves the library user the effort of having to implement this themselves.

As mentioned, each process has its own OFDT. This has some obvious disadvantages, among which no builtin interprocess read/write locking. This is discussed in detail in Section 3.3.

LDP_FUSE has a conditional compilation flag `LDP_FUSE_THREAD_SAFE`. Normally, a multithreaded application concurrently accessing the global OFDT will lead to race conditions. With this conditional compilation flag enabled, the OFDT is made thread safe by adding a `pthreads` reader-writers lock to every entry in the OFDT, used to synchronize access. A user of the LDP_FUSE library will therefore have to know whether the program(s) using it are multithreaded, and should specify this flag if they are.

## 3.2 Examples

### 3.2.1 StackFs

An implementation of Vangoor et. al's StackFs [16] is very simple in LDP_FUSE. The following snippet illustrates how one would write a passthrough function for the `read` operation:

```
ssize_t stackfs_read(int fd, void *buf, size_t count, off_t offset,
                     orig_pread_t pread_fn) {
  return pread_fn(fd, buf, count, offset);
}
```

In other words, call the function given by the last argument, passing in all the other arguments as-is.

### 3.2.2 Transparent cryptographic file system

You can make more interesting stacked file systems by writing custom logic before and after calling the original function. The next snippet is an example of a transparently encrypted file system:

```
#define KEY 0b11110000

ssize_t encrypted_read(int fd, void *buf, size_t count, off_t offset,
                       orig_pread_t read_fn) {

  ssize_t read_bytes = read_fn(fd, buf, count, offset);
  for (size_t i = 0; i < read_bytes; i++) {
    ((unsigned char *)buf)[i] ^= KEY;
  }
  return read_bytes;
}

ssize_t encrypted_write(int fd, const void *buf, size_t count, off_t offset,
                        orig_pwrite_t write_fn) {

  for (size_t i = 0; i < count; i++) {
    ((unsigned char *)buf)[i] ^= KEY;
  }

  return write_fn(fd, buf, count, offset);
}
```

Every byte is `XOR`'ed with a key on `write`, and `XOR`'ed again on `read`. This is, needless to say, not cryptographically secure, but purely for demonstration purposes. It shows that user-defined logic is easily included in an LDP_FUSE file system. Moreover, it is possible to not use the original `read` and `write` functions at all, and instead opt for e.g. in-memory or network-based storage.

## 3.3 Shortcomings

This approach comes with several inherent drawbacks. I have enumerated them below.

- Memory-mapped I/O (e.g. `mmap`) can not be used with LDP_FUSE. The `mmap` function simply returns a pointer that the program may then read/write freely to perform file

system operations. It is not possible to intercept this and translate them to LDP_FUSE calls. FUSE does not have this problem due to its kernel space module. These memory-mapped file system operations are intercepted by the VFS and are simply passed along to the kernel module, as if they were regular `read`s and `write`s.

- Linux does not allow `LD_PRELOAD` in secure-execution mode, e.g. for `setuid` binaries. This is because of the security implications of `LD_PRELOAD`: It would be unsafe to allow modification to a program running with elevated privileges. As a result, LDP_FUSE will not work at all for `setuid` binaries.

- Certain syscalls, such as `openat2`[2], do not have a glibc wrapper. As such, LDP_FUSE cannot intercept it. The same applies to any programs that use `syscall`[3] to invoke a filesystem-related syscall, or use inlined syscalls.

- Statically linked executables do not use a dynamically linked glibc. This means `LD_PRELOAD` does not work. As a consequence, LDP_FUSE will not work either.

Additionally, the current design of LDP_FUSE presumes the existence of an underlying file system to manage shared/exclusive locks of files. Although nothing prevents a user from implementing this on their own, it is not ergonomic. Adding this to LDP_FUSE would be a good first point of improvement. A possible implementation could be a single, user space OFDT in shared memory. This OFDT would be shared among all processes running with LDP_FUSE, thereby being able to manage concurrent access through read/write locks. Although, this would replicate a large part of the kernel's own OFDT, and no doubt add performance overhead.

To summarize: LDP_FUSE is not a drop-in replacement. If one wants to use an LDP_FUSE file system, they will need detailed knowledge of how programs will access it. Finding a solution to (some of) the shortcomings listed above will be crucial for making `LD_PRELOAD`-based file systems feasible. This answer research question 2, posed in Section 2.3.2: Can an `LD_PRELOAD`-based file system, like LDP_FUSE, be a viable alternative to FUSE? To which the answer is no. In its current state, there are too many scenarios in which it cannot replace FUSE.

[2] https://man7.org/linux/man-pages/man2/openat2.2.html
[3] https://man7.org/linux/man-pages/man2/syscall.2.html

# 4 Related work

To better understand how LDP_FUSE fits into the field, I will go over several related works. This section starts with papers that have analyzed FUSE's performance in more depth, and have identified workloads under which FUSE performs the worst. Followed by this is a comparison between several existing solutions, which I will compare to LDP_FUSE.

In their 2010 paper, Rajgarhia and Gehani performed several experiments on FUSE [13]. They note that its performance is comparable to a regular, in-kernel file system under workloads that feature large and sustained I/O, such as copying a large file. However, workloads that feature many small operations noticeably suffer from the overhead that FUSE adds, which caused a severe performance degradation.

Vangoor et al. in 2017 came to a similar conclusion [17]. They performed experiments with a stackable filesystem (*Stackfs*) implemented on top of FUSE. Stackfs merely passes along FUSE requests to an underlying file system, thereby isolating FUSE's added overhead. Most workloads suffered less than a 5% penalty in execution speed. However, an 83% penalty was incurred when randomly reading 4KB chunks from a large file with 32 threads, on an SSD. The same workload on an HDD suffered less relative performance loss because the storage device itself is slower, therefore hiding some of FUSE's overhead.

From this we can conclude that FUSE's performance degradation is a well-known fact. What solutions to this problem exist currently? For certain workloads, mainly those in the cloud, one possible solution would be *unikernels*. Unikernels are applications where a specialized subset of operating system functionalities (e.g. networking) have been compiled together with the application logic, and live in the application's address space [5]. Shared resources, such as the network and file system, would be implemented as a shared server in user space [11]. This would improve performance by removing the need for a mode switch between user and kernel space. However, unikernels are not always feasible. They are suited to systems with a single user, as multiple users dramatically increase overhead due to the need for (file system) authentication logic [12]. A more generally adoptable solution is the *Direct-FUSE* framework [18]. Direct-FUSE seeks to improve performance by moving a part of FUSE's functionality into user space. It builds on top of the libsysio [10] framework. Libsysio provides a way to overwrite glibc syscall wrappers, similar to LDP_FUSE. Another similarity is that libsysio implements a complete user space VFS, which LDP_FUSE currently only does partially with its per-process OFDT. Direct-FUSE itself offers a way to port FUSE systems to libsysio. In addition, it supports running multiple file system backends at the same time. It can then be either dynamically linked to an application, or `LD_PRELOAD`ed. It differs in being a lower level framework, rather than a higher level library. One must write a driver that interfaces with Direct-FUSE in order to use an existing FUSE file system. Depending on the complexity of the file system, this could be cumbersome.

Other efforts include an extension to FUSE, rather than a complete replacement. One such solution is *ExtFUSE* [7]. ExtFUSE leverages the eBPF framework [1], an in-kernel interpreter that provides a safe execution sandbox for user-defined code that requires lower level kernel access. Using this, ExtFUSE brings the possibility for users to add *extensions*: In-kernel handlers for file system requests. ExtFUSE then allows these to handle file system requests, also referred to as the *fast path*. When more complex logic is needed, ExtFUSE will forward the file system request to the user space FUSE daemon, as normal. This is the *slow path*. This split allows for gradual adaptation of an existing FUSE codebase to ExtFUSE. Authors note that rewriting FUSE code to ExtFUSE is not difficult, but not trivial either: Several hundreds lines of code changed to adapt an existing FUSE file system. LDP_FUSE, on the other hand, has no "slow path" that requires entering kernel space. Additionally, adapting an existing FUSE codebase (provided it is written using the high level API) is easier, though gradual adaptation is not possible. Other than this, ExtFUSE also supports immediately passing through simple requests (read/write) to an underlying file system. This is useful for stackable file systems that add only a thin layer of functionality on top of an existing one.

Interestingly, this last part is now a regular part of FUSE itself. Read/write passthrough support has been added to FUSE in Linux kernel `v5.11-rc5` [2]. From then on, FUSE will support read/write requests going directly to the underlying file system rather than to the user space daemon. On an open/create operation, FUSE will decide whether the file can be accessed in passthrough mode. This is only possible for reads and writes. A version of FUSE with passthrough mode has not been included in the experiments in this thesis, but is an interesting avenue for future research.

We can conclude using `LD_PRELOAD` for developing file systems is not a completely new idea. However, as discussed in Section 3, the novelty of LDP_FUSE is that it is a library to allow others to write file systems on top of `LD_PRELOAD` more easily. An assessment of LDP_FUSE's performance is available in the next section.

# 5 Experiments

In this section, I will explain the experiment setup that I used to assess LDP_FUSE's performance. To measure performance, I used Filebench[4] (version 1.5-alpha3). Filebench allows one to simulate a file system workload by specifying a set of I/O operations, e.g. `open`, `delete`, `readwholefile`. It then reports on the overall bandwidth that was measured. This means we can compare the overall throughput of LDP_FUSE against native and FUSE respectively. In addition, it will give insight into which workloads degrade performance the most. In all cases, a stackable (passthrough) file system is used. This is to measure purely the added overhead of the file system itself.

I used FUSE version 3.11, running under Linux kernel version 5.13.0-52-generic. The FUSE file system used is the example passthrough file system found in the `libfuse` GitHub repository[5]. As the authors have noted, the performance of this FUSE file system is "terrible". It is possible to develop a more performant passthrough file system using FUSE's lower level API, as Vangoor has done [16]. However, I think the higher level FUSE API is a more fair comparison. It is what most developers would reach for first when developing a file system, and it is closer to how one would use LDP_FUSE.
The LDP_FUSE file system I used was the simple passthrough system, implemented as described in Section 3.2.1.

The workloads that Filebench simulated were as follows:

- *file-sq-re-4KB-1th-1f* – Single thread that opens and reads a single 60GB file sequentially in 4KB chunks.
- *file-sq-re-4KB-32th-32f* – 32 threads that open and read 32 2GB files sequentially in 4KB chunks.
- *file-cr-wr-4KB-1th-4Mf* – Single thread that creates 4 million 4KB files, then overwrites each file fully
- *file-cr-wr-4KB-32th-4Mf* – 32 threads that create 4 million 4KB files, then overwrite each file fully
- *web-server-100th* – 100 threads that simulate a web server workload: Repeatedly open, fully read, close 1.250.000 16KB files.
- *file-server-50th* – 50 threads that simulate a file server workload: Repeatedly create, write, read, close, delete and stat 200.000 128KB files.

[4] https://github.com/filebench/filebench
[5] https://github.com/libfuse/libfuse

The experiment results are the average bandwidth of all the operations, obtained after running the workload five times. The server that ran the experiments had the following hardware: AMD Ryzen 7 3700X CPU, 32GB of DDR4 2133MHz RAM, Samsung 980 PRO M.2 SSD. In the case of LDP_FUSE, a thread-safe version of the library is used when the workload features >1 threads. The thread-safe version adds additional overhead because it uses a readers-writers lock for every entry in the OFDT, as mentioned in Section 3.

# 6 Results

This section contains the results of the experiments described in Section 5. In addition, I will elaborate on the findings and give my reasoning

Table 2 shows the results of the experiments described in the previous section. Each cell lists the average bandwidth of all the performed file system operations. For LDP_FUSE and FUSE, it also notes the relative performance loss when compared to native. On every

| | | **File system** | | |
|---|---|---|---|---|
| | | *Native* | *LDP_FUSE* | *FUSE* |
| **Workload** | *file-sq-re-4KB-1th-1f* | 659 MB/s | 562 MB/s (**-15%**) | 119 MB/s (**-82%**) |
| | *file-sq-re-4KB-32th-32f* | 4937 MB/s | 3363 MB/s (**-32%**) | 722 MB/s (**-86%**) |
| | *file-cr-wr-4KB-1th-4Mf* | 76 MB/s | 70 MB/s (**-8%**) | 6MB/s (**-91%**) |
| | *file-cr-wr-4KB-32th-4Mf* | 99 MB/s | 91 MB/s (**-9%**) | 21 MB/s (**-79%**) |
| | *web-server-100th* | 1360 MB/s | 1194 MB/s (**-12%**) | 108 MB/s (**-92%**) |
| | *file-server-50th* | 2713 MB/s | 2079 MB/s (**-23%**) | 437 MB/s (**-84%**) |

Table 2: Results of running six different workloads under three different file systems. For details on workloads, see Section 5

workload, the naïve passthrough file system in LDP_FUSE outperforms the same file system in FUSE significantly. LDP_FUSE performs worst on workloads with a high number of threads. This is due to the overhead added by the necessary locking per open file descriptor.
Interestingly, FUSE's performance is far worse than the worst case of 83% degradation discussed in Section 4. This is likely due to the SSD used in the current experiment being faster, which makes the overhead even more noticeable. Coming back to research question 2 (Introduced in Section 2.3.2) – How is the performance of LDP_FUSE compared to FUSE? – these results indicate that LDP_FUSE outperforms FUSE by a wide margin.

Although LDP_FUSE's performance is still noticeably worse than native on some workloads, for example *file-sq-re-4KB-32th-32f* (Sequential read of 32 files by 32 threads in 4KB chunks). The overhead added by the `pthread` read-lock and the updating of the file offset in LDP_FUSE's OFT is especially pronounced here. This is due to it being a very fast benchmark on native. Comparing this to *file-sq-re-4KB-1th-1f* (Sequential read of 1 file by 1 thread in

4KB chunks) – which runs without any mutex locking – we see that the overhead is far less noticeable. In short, LDP_FUSE outperforms regular FUSE on every workload, but is still has noticeable performance loss as compared to native.

# 7 Conclusion

In this thesis, I have explained the importance of file systems and why it is attractive to run them in user space instead of in the kernel. Namely, because it benefits performance by avoiding user-kernel mode switching, and it is easier to develop user space programs. Then I showed that while Linux' FUSE is a good tool to develop such file systems, it suffers from bad performance especially on workloads with small, frequent IO.

The LDP_FUSE library, introduced in this thesis, leverages the `LD_PRELOAD` linker directive to develop user space file systems instead. By overriding glibc syscall wrappers for IO operations, we can insert our own logic through which we implement a file system. Benchmarks performed by Filebench show promising results: LDP_FUSE incurs a $-32\%$ performance penalty in the worst case (versus FUSE's $-92\%$ worst case), and only $-8\%$ in the best case.

However, there are significant shortcomings and challenges that need to be overcome to make this method viable. Most importantly, LDP_FUSE cannot intercept file system operations performed via memory mapped IO. Other than this, not all IO syscalls can be intercepted either, as some do not have a glibc wrapper. Lastly, binaries that use secure-execution mode (e.g. `setuid`) do not work with LDP_FUSE. Other than these shortcomings, LDP_FUSE currently lacks an interprocess read-write locking mechanism for files. All of these points are an interesting avenue for further research.

# References


[1] eBPF. IO Visor Project. [Online]. Available: https://www.iovisor.org/technology/ebpf

[2] Fuse: Add support for passthrough read/write [LWN.net]. [Online]. Available: https://lwn.net/Articles/843873/

[3] FUSE documentation. LibFuse Documentation. [Online]. Available: http://libfuse.github.io/doxygen/

[4] Overview of the Linux Virtual File System — The Linux Kernel documentation. [Online]. Available: https://www.kernel.org/doc/html/latest/filesystems/vfs.html

[5] Unikernels - Rethinking Cloud Infrastructure. [Online]. Available: http://unikernel.org/

[6] A. Ashcraft and M. Satran. The Evolution of File Systems - Win32 apps. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/stg/the-evolution-of-file-systems

[7] A. Bijlani and U. Ramachandran, "Extension Framework for File Systems in User space," p. 15.

[8] R. Card and S. Tweedie. Design and Implementation of the Second Extended Filesystem. [Online]. Available: https://www.semanticscholar.org/paper/Design-and-Implementation-of-the-Second-Extended-Card-Tweedie/35a700439f28d5efaf5fa037db4fab27c27d0f82

[9] R. C. Daley and P. G. Neumann, "A general-purpose file system for secondary storage," in *Proceedings of the November 30–December 1, 1965, Fall Joint Computer Conference, Part I*, ser. AFIPS '65 (Fall, Part I). Association for Computing Machinery, pp. 213–229. [Online]. Available: https://doi.org/10.1145/1463891.1463915

[10] G. K. Lockwood, "Glennklockwood/libsysio." [Online]. Available: https://github.com/glennklockwood/libsysio

[11] M. S. on Sep 14 and 2017. Leave your OS at home: The rise of library operating systems. SIGARCH. [Online]. Available: https://www.sigarch.org/leave-your-os-at-home-the-rise-of-library-operating-systems/

[12] R. Pavlicek, *Unikernels*. O'Reilly Media, Inc. [Online]. Available: https://learning.oreilly.com/library/view/unikernels/9781492042815/

[13] A. Rajgarhia and A. Gehani, "Performance and extension of user space file systems," in *Proceedings of the 2010 ACM Symposium on Applied Computing - SAC '10*. ACM Press, p. 206. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1774088.1774130

[14] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*. John Wiley & Sons.

[15] V. Tarasov, A. Gupta, K. Sourav, S. Trehan, and E. Zadok, "Terra Incognita: On the Practicality of User-Space File Systems," p. 5.

[16] B. K. R. Vangoor, P. Agarwal, M. Mathew, A. Ramachandran, S. Sivaraman, V. Tarasov, and E. Zadok, "Performance and Resource Utilization of FUSE User-Space File Systems," vol. 15, no. 2, pp. 1–49. [Online]. Available: https://dl.acm.org/doi/10.1145/3310148

[17] B. K. R. Vangoor, V. Tarasov, and E. Zadok, "To FUSE or Not to FUSE: Performance of User-Space File Systems," p. 15.

[18] Y. Zhu, T. Wang, K. Mohror, A. Moody, K. Sato, M. Khan, and W. Yu, "Direct-FUSE: Removing the Middleman for High-Performance FUSE File System Support," in *Proceedings of the 8th International Workshop on Runtime and Operating Systems for Supercomputers*. ACM, pp. 1–8. [Online]. Available: https://dl.acm.org/doi/10.1145/3217189.3217195