Łukasz Faber
Krzysztof Boryczko

PORTABLE USERSPACE VIRTUAL FILESYSTEM SWITCH

Abstract *Multiple different filesystems — including disk-based, network, distributed, abstract — are an integral part of every operating system. They are usually written as kernel modules and abstracted to the user via a virtual filesystem switch.*

In this paper, we analyze the feasibility of reimplementing the virtual filesystem switch as a userspace daemon and applicability of this approach in real-life usage. Such reimplementation will require a way to virtualize processes behavior related to filesystem operations. The problem is non-trivial, as we assume limited capabilities of the VFS switch implemented in userspace. We present a layered architecture comprising of a monitoring process, the VFS abstraction and real filesystem implementations. All working in userspace. Then, we evaluate this solution in four areas: portability, feasibility, usability, and performance. Our results demonstrate possible gains in the use of a userspace-based approach with monolithic kernels, but also underline problems that are encountered in this approach.

Keywords userspace virtual filesystem, operating systems, ptrace, FUSE

1. Introduction

The idea of moving non-critical components off of an operating system from the kernel has been a recurring theme for at least thirty years. One such component is the filesystem manager. This is the part of the OS that is responsible for providing a filesystem-related interface to user applications; i.e. dispatching calls to individual filesystems, managing mounts, translating paths, and other similar operations. In this paper, we will analyse how this component can be moved to the userspace in systems with mostly monolithic and modular kernels.

A *filesystem*, in its most basic form, is a collection of files and directories, usually organised into some kind of a structure or hierarchy. When we think of a filesystem, we tend to connect it with some external storage. However, in addition to disk-based solutions, there are also many network or special filesystems (e.g. NFS, Lustre[1] or procfs).

A *virtual filesystem switch* (VFS)[2] is a kernel module responsible for providing an abstraction over individual, heterogeneous filesystems. The userland can access them with a well-defined, common interface in a nearly transparent way. To emphasise the difference between a filesystem located on, e.g., the disk and the virtual filesystem, we sometimes call the former a *underlying filesystem*.

The kernel provides a *filesystem interface* to the userland. In this paper, we are interested mostly in the operations defined by the POSIX standard.

Relationships between these three layers: *filesystem interface*, *VFS* and *filesystems* are shown in Figure 1.

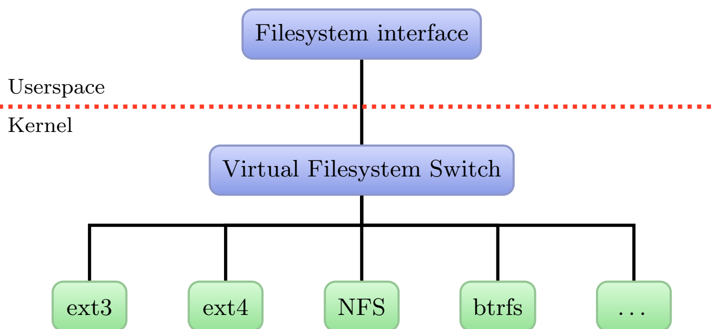

Figure 1. The filesystem implementation stack.

This paper also discusses the consequences of moving services from the kernel to the userspace. In short, the important difference between the kernel and userspace programs is the lack of rights to perform certain actions. A userspace process cannot access memory of other processes or change their parameters. Also, userspace

[1] Available at `http://wiki.lustre.org/index.php/Main_Page`.
[2] Sometimes called *virtual filesystem*.

filesystems are separated from other services. This increases security and stability of the system. Moreover, userspace filesystems do not require root capabilities to be mounted. This typically leads to easier interaction from the end-user point of view [16].

Despite the advantages described above, there are concerns regarding the performance of user-level kernel services and filesystems. However, with the increasing computing power of modern processors, this problem plays a smaller and smaller role. In fact, it was shown that it is possible to implement userspace drivers without a significant loss of performance [10].

The traditional approach to implementing filesystems has several limitations, mostly related to the fact that the implementation is done within the OS kernel. These are:

Lack of extensibility understood as difficulty in extending the functionality. It is a result of, e.g., a requirement for strict backward compatibility and compatibility with a particular kernel version.

Difficulties in development as kernel programming is more error-prone than application-level programming, errors are more fatal, utility libraries are not available and debugging is harder.

Unportability as kernel code is unportable to unrelated operating systems.

Lower security and stability as mistakes and errors made in the kernel code are much more dangerous for the end user than those in the userspace code, especially, if the code has contact with the external world (e.g. remote filesystems). This is one of the reasons for creating microkernels [6].

Kernel "bloating" i.e. monolithic kernels become too large to maintain [15]. Including in the kernel all the filesystems potentially interesting to some users could greatly increase the size of the kernel[3].

Difficulties in usage for the end-user. Classical filesystems require root privileges for mounting so new devices cannot be easily attached and made available. Moreover, it is not usually easy for the user to add a new filesystem to their system.

2. Short review of selected solutions

There are several methods that can be used to implement a userspace VFS. However, they provide very different levels of transparency for the user. Also, their convenience from a developer's point of view varies. Some require writing code very close to the operating system, whereas others may be written in any available programming language. Some of the already-existing solutions, as well as methods used in them, are discussed below.

[3]For example, the FUSE project lists currently over 150 filesystems based on it [4].

2.1. Kernel module – FUSE

Filesystem in Userspace (FUSE) is a userspace filesystem framework for Linux [3, 11]. It consists of a kernel module (`fuse.ko`), a userspace library (libfuse) and a utility for mounting filesystems. The main goal of FUSE is to enable secure mounting of filesystems for an unprivileged user in a manner transparent to applications.

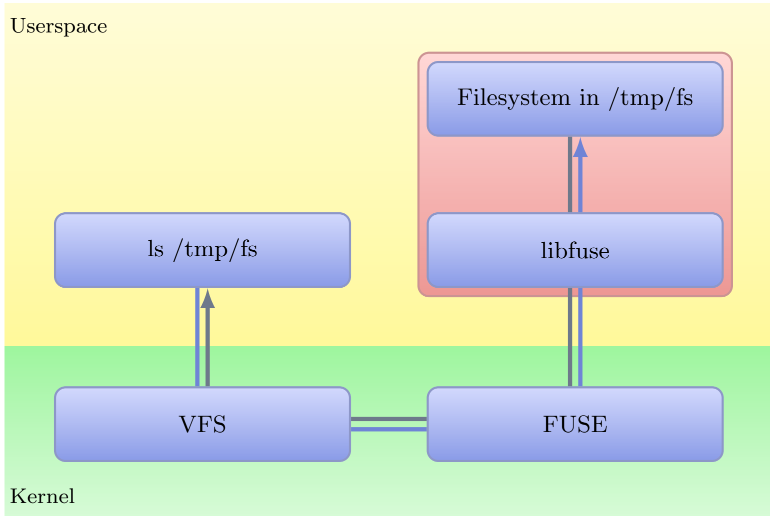

Figure 2. The architecture of FUSE. A `/tmp/fs` mountpoint is used as an example.

The kernel module acts as a "bridge" between kernel filesystem interfaces and userspace applications. It implements the standard Linux VFS interface and, therefore, acts as a real filesystem to the kernel. However, instead of performing filesystem operations on its own, it forwards them to the userspace.

The libfuse library provides an API for implementing real filesystems. It handles forwarded calls from the kernel and executes relevant functions defined by the filesystem implementation. When mounted, the FUSE filesystem exists as a daemon process in the user space.

Figure 2 shows the architecture of FUSE and the processing path for the filesystem operations. Communication between the userspace daemon and the kernel module is performed over the so-called "filesystem connection". Its lifetime is limited by the existing mounts and the exit of the daemon.

From the point of view of the filesystem creator, FUSE greatly simplifies the implementation of a new filesystem. The operations that need to be implemented by the developer are much simpler than standard filesystem operations required by the kernel, and they are usually not directly mapped to filesystem calls. The developer can use any available libraries or methods to implement its filesystem. It is a direct result of writing a userspace code. This possibility is actively used by, for example,

EncFS[4] that uses OpenSSL library to provide encryption capabilities to users. Moreover, filesystems can be implemented in languages other than C. Bindings have been created, for example, for Python[5], Java[6] or Erlang[7].

An additional benefit for the developer is a possibility to use standard debugging mechanisms intended to interact with userspace code, e.g. GNU Debugger (gdb), Valgrind, strace and others. It actually may improve the stability of the filesystem and usually makes the development process easier [2].

The other positive aspect is the portability of such filesystems. Firstly, they are usually more immune to Linux API changes because FUSE guarantees a stable API and provides protocol version negotiation during a filesystem initialisation. Secondly, FUSE itself was ported to other operating systems, e.g.: NetBSD[8] (especially [7] contains a detailed description of this port), MacOS X[9] or Hurd[10]. This means that a lot of FUSE-targeted filesystems can work on other POSIX platforms without (or with only slight) changes.

From the filesystem user's point of view, the most important feature is the possibility to do non-privileged mounts. Also, access is, by default, restricted only to this user. The support for many filesystems is also very important.

The biggest drawback of FUSE is its performance penalty as compared to more-traditional approaches. The OS needs to perform at least one additional mode switch and must send data between user- and kernelspace.

2.2. Explicit userspace virtual filesystem – GVFS

By the *explicit userspace virtual filesystem*, we understand a library that provides all common filesystem operations in the form of a public API. It implies that the program using this kind of a filesystem must explicitly call functions provided by this library. Operations provided by the library do not need to match those offered by the underlying OS. The use of a filesystem of this kind greatly reduces the burden of porting the application to other platforms, as all the filesystem operations are provided by an abstract interface. Some of the better-known virtual filesystems in this category are GVFS, KIO[11] and Commons VFS[12].

GVFS is a virtual file system for GNOME[13]. It consists of two parts [5]:

GIO — a shared library providing API for accessing the VFS,

[4] Available at `http://www.arg0.net/encfs`.
[5] Available at `https://code.google.com/p/fusepy/`.
[6] Available at `http://sourceforge.net/projects/fuse-j`.
[7] Available at `https://code.google.com/p/fuserl/`.
[8] Available at `http://www.netbsd.org/docs/puffs/`.
[9] Available at `http://fuse4x.org/`.
[10] Available at `http://www.nongnu.org/hurdextras/`.
[11] Available at `http://api.kde.org/4.x-api/kdelibs-apidocs/kio/html/`.
[12] Available at `https://commons.apache.org/vfs/index.html`.
[13] Available at `http://www.gnome.org/`.

GVFS — which provides backends, e.g. for protocols such as SSH, FTP, ObexFTP, and a daemon (gvfsd) that tracks GFVS mounts.

GVFS architecture is shown in Figure 3.

GIO provides a higher-level interface than the POSIX calls. It is designed to be "document-centric" [5]. In addition to simple file operations, it supports I/O streams, sockets, file monitoring, D-Bus integration, asynchronous I/O, etc. Some of these operations are strictly related to standard system calls. For example, `g_file_delete` is a method similar to `unlink`. Yet, most of the GIO methods provide much more complex operations, e.g. `g_file_get_parent` that returns a parent directory of a file. The GIO API is object-oriented and quite complex. For example, it has over one hundred methods operating on the `GFile` file object.

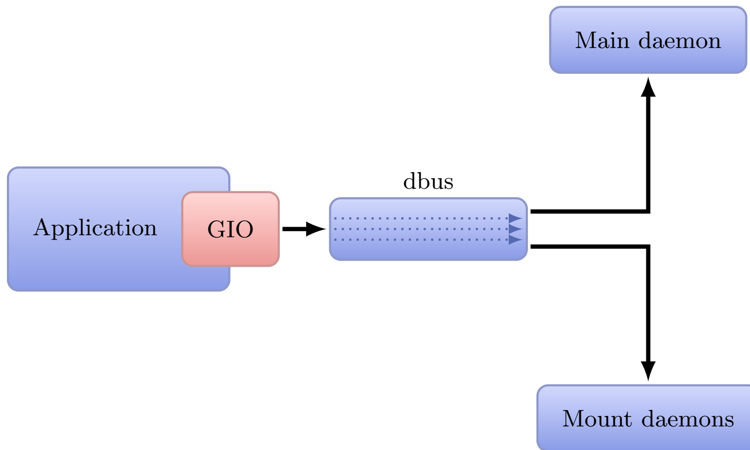

Figure 3. The architecture of GVFS.

The virtual system runs wholly in OS daemons [8]. There is one main daemon that keeps track of all user mounts and a separate daemon for every mount. An application does not have to link to, or even know about, the existence of backend-specific libraries. The communication is performed over dbus connections. Unfortunately, this approach is completely opaque to the user. There is no easy way to use files from a filesystem of this kind, with applications that were not written to explicitly exploit its functionality.

2.3. Overriding dynamically loaded libraries – libtrash

Another way to create a userspace filesystem is to replace the standard library of the employed programming language. Typically, it will be the standard C library. The replacement is done by altering the work of the dynamic loader. In Linux, it is usually done by modifying the `LD_PRELOAD` variable, as illustrated in Listing 1 [14].

This works because the called function is searched in linked libraries in the order they were loaded. Chains of the function name resolution are shown in Figure 4.

Listing 1 An example usage of LD_PRELOAD.

```
LD_PRELOAD="/path/to/replacement.so:$LD_PRELOAD"
    ./program_to_run
```

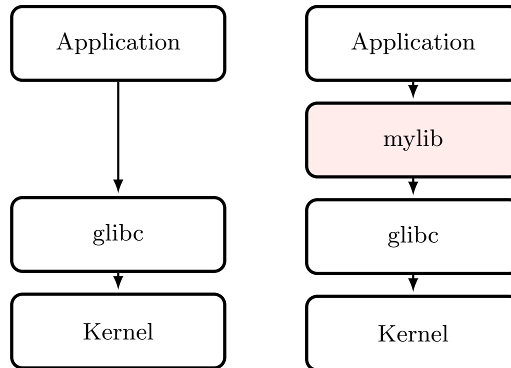

Figure 4. On the left: a call chain without library preloading. On the right: a call chain in an application executed with `LD_PRELOAD=./libmylib.so`.

In order to stub a call to, e.g. the `open` syscall, the library has to define a function with the same exact name as the stubbed function.

One example of `LD_PRELOAD` usage is libtrash[14]. It is a shared library that can be preloaded, and that transparently implements the trash functionality for Linux. It intercepts *potentially destructive* calls from the standard C library and backs up removed or modified data.

2.4. `ptrace` system call – Goanna

The `ptrace` is a common Unix system call[15] used for process tracing. It lets the calling process trace the execution of another process. `ptrace` is available in several Unix and Unix-like systems, including: Linux, FreeBSD, NetBSD, Mac OS X [1] and UnixWare [17]. However, its implementations provide very different sets of capabilities. The most limited implementation is the one available in Mac OS X. It offers little more than single-stepping. The Linux implementation, on the other hand, provides a very wide set of functions: e.g. reading and modification of process data, an emulation of syscalls, inheriting tracing through `fork`, etc. `ptrace`-based approaches to the filesystem implementation are not new. Frameworks for rapid filesystem development were created, although, with moderate success. One example is Goanna, presented in 2007 [18].

[14]Available at `http://pages.stern.nyu.edu/~marriaga/software/libtrash/`.
[15]Commonly abbreviated as *syscall*.

Goanna was created mainly for prototyping of experimental filesystems. The authors of Goanna focus on reducing the effort of a filesystem developer during the prototyping and debugging of their code. In order to do so, they have built a `ptrace`-based monitoring infrastructure. The architecture of Goanna is simple. It consists of a single userspace monitor process that dispatches system calls to multiple filesystems. The fact that Goanna has one narrow task to perform makes it possible for its authors to freely use all the facilities provided by the Linux kernel and even extend the kernel itself.

Our approach, although similar in nature to Goanna, is focused on creating a more end-user-centered and portable solution. In the prototype, we intentionally focus on creating an architecture that would require minimal effort for porting to other operating systems and, at the same time, would simplify the interface for filesystem developers. Therefore, we try to avoid using platform-specific features. This is in contrast to Goanna, where, for example, modifications to Linux `ptrace` implementation are suggested in order to improve execution speed. Another main difference to Goanna is an approach towards implementation architecture. For better portability, our system introduces three strictly-separated layers, while Goanna's monitor is joined with the specific filesystem that is using it. Overall, Goanna is much more lightweight than our solution due to its specialization as a prototyping framework.

2.5. Summary

Table 1 summarizes the comparison given in this section. Categories shown therein are related to the issues pointed out in the description of the discussed technological solutions. *Overhead* is related to additional computational work when using a given solution; *portability* relates to how easily a given solution may be reimplemented on another OS; *transparency* says how well it integrates with standard OS tools and how transparent it is to the user; *limitations* enumerates special limitations of a given solution.

Table 1

Comparison of the discussed VFS solutions.

Solution	Overhead	Portability	Transparency	Limitations
FUSE	medium	low	very high	requires module in the kernel
Explicit UVFS	small	high	none	no interaction with standard OA tools
Overriding DLL	variable, typically small	depends on the underlying library	high	easily circumvented; does not work with statically compiled applications
`ptrace`	very high	high	high	

3. `ptrace`-based portable userspace VFS

This section describes our approach to the implementation of the userspace virtual filesystem. It discusses both the high-level conceptual view of the system and low-level implementation details. We start with the description of the logical architecture and then move to the details of the system layers and solutions used within them.

3.1. High-level architecture

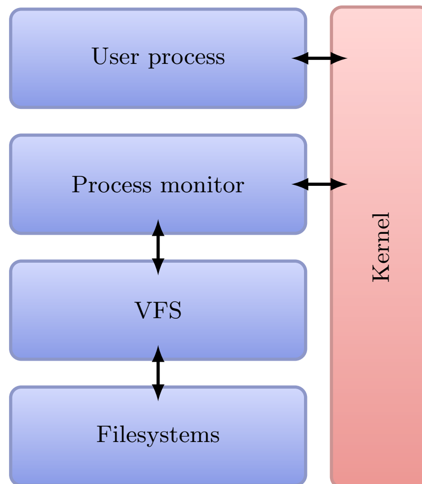

Figure 5. The logical architecture of the proposed system.

The conceptual view of the architecture is shown in Figure 5. Four logical layers can be defined:

1. the user process layer in which the executed user process is being run,
2. the monitor layer, which controls the user process and emulates its system calls,
3. the virtual filesystem switch layer that takes care of presenting a common interface, for all supported filesystems,
4. the filesystems layer that consists of all loaded and mounted filesystems and performs the actual processing of requests.

Additional requirements for these layers are:

1. the only layer that can depend on an underlying OS is the monitor layer,
2. the VFS and filesystems layers are global in the operating system,
3. programs see changes introduced to the filesystem by other processes, as we emulate the real VFS,
4. user applications have in their disposition a wholly-emulated filesystem, so there is no direct interaction with the host filesystem interface,
5. the system has to be able to work with multiple user processes,

6. the VFS and monitor layers should not have any assumptions about the communication between them; in particular, no external communication might be needed in the single process case,
7. there is no interaction between non-adjacent layers.

3.1.1. Monitor layer

The main role of the monitor layer is to intercept all system calls of the user process, choose these that should be virtualized and rerouted to the VFS layer. The process monitor is called a *tracer* and its children (i.e. observed processes) are called *tracees*. Tasks for this layer include:

1. starting and monitoring the running user process,
2. following forks, clones and replacements (`execve`) of this process,
3. analyzing, restarting and aborting selected system calls and signal deliveries.

3.1.2. Virtual filesystem switch layer

The virtual file system layer is an equivalent of the virtual system interface of the OS kernel. It implements all operations exposed to the userspace by system calls. It also defines an interface for a concrete filesystem to implement. Tasks for this layer include:

1. process-related:
   (a) tracing data about processes (e.g. current working directory),
   (b) keeping information about open files,
   (c) keeping information about mapped memory regions;
2. filesystems-related:
   (a) handling mounting of filesystems and keeping the filesystem tree,
   (b) translation of process-relative data to mountpoint-relative data (e.g. translation of an absolute path of a file to its equivalent relative to the root directory of the mountpoint),
   (c) simplifying the interface of filesystems.

3.1.3. Interface for filesystems

Implemented filesystems may provide only a subset of operations that are available to the user process. Moreover, not all of these operations have equal semantics. The reason behind this is that a lot of operations can be generalized and, thus, simplified. For example, `chown` function may exist in three versions:

- `chown` which receives a path and follows symlinks,
- `lchown` which it receives a path and *does not* follow symlinks,
- `fchown` which it receives a file descriptor and follows symlinks.

A VFS can perform converting a file descriptor to a path or dereferencing a symlink on behalf of the target filesystem. Therefore, the underlying filesystem needs to provide only the `lchown` operation. Additionally, there are functions that can be completely handled within the VFS, e.g. `chdir` syscall.

3.2. Implementation details

Figure 6 shows what the physical architecture of the prototype looks like. There are three group of processes shown:

- user processes, i.e. applications started by the user,
- process monitors, i.e. monitors attached to the user processes,
- one global VFS daemon.

There is an $n : m$ relation between the number of user processes and monitors. The number of monitors is always lower than the number of user processes. This is due to the fact that, when a user process forks, there is no new monitor created. The monitor is based on the `ptrace` syscall and is implemented as a simple state machine.

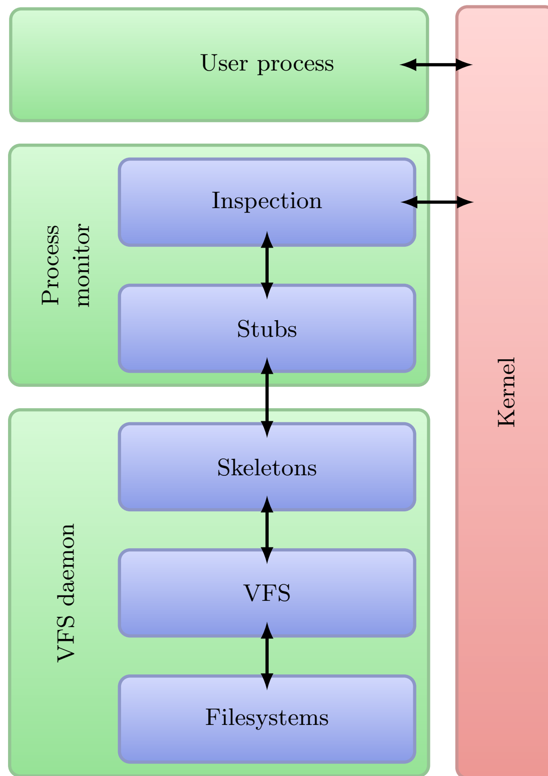

Figure 6. Physical implementation of the proposed system.

3.3. System call inspection

The system call inspection is performed in the monitor layer. Its role is to intercept all system calls, obtain information about them, decide whether a syscall is related to the filesystem, and collect all data from the tracee. The monitor layer is dependent

on the operating system and, therefore, its structure is also related to OS. For Linux, the following units are defined:

1. `ptrace`-based monitor,
2. methods for obtaining data from the memory and registers of the tracee,
3. list of all syscalls with metadata, including number, name, number of parameters, an action to be performed, and a pointer to the stub,
4. implementation of stubs for the RPC service [19].

Inspection is performed using the `ptrace` function. After every stop, the tracee is restarted with `PTRACE_SYSCALL` call that forces it to stop at both the entry and the exit from a syscall. When the tracee stops, a reason for stopping is inspected. The stop may occur for many reasons, so we need to be able to differentiate between them. The tracer waits for a child to stop using the `waitpid` function. After returning from this call, the status is inspected to see what action should be performed. If the child stopped due to a system call, its state is `TRST_RUNNING`. Then, the routine related to syscall inspection is executed. First, the process registers are obtained using the `ptrace` call `PTRACE_GETREGS`. Second, the type of the syscall is checked. Three types are recognized:

1. filesystem-related syscall that needs straightforward stubbing and forwarding,
2. memory-mapping related syscall that needs to be examined before forwarding,
3. other syscall that is ignored and executed directly by the kernel.

In the first case, following actions are performed:

1. data about the syscall is saved in the child structure,
2. the state of the child is changed to `TRST_IN_SYSCALL`,
3. the syscall is canceled by overwriting the syscall number to -1,
4. a VFS stub is called and the return value is saved in the child structure,
5. the child is restarted,
6. if the child stops *just before* returning from the cancelled syscall, the return value from the VFS is written to the registers,
7. the child is restarted again.

Handling of memory-mapping operations is described in Section 3.4. For all other system calls, the monitor restarts their execution without performing any actions.

3.3.1. Syscall error handling (`errno`)

Sometimes a syscall needs to notify the caller about an error that occurred during its execution. From the point of view of a user application, such information is passed through the `errno` variable and a predefined return code[16]. However, there is no special interface on the kernel side — an error is returned the same way as correct values. Accordingly, the process monitor returns the error to the application by setting the `RAX` register and the standard C library perform actions necessary to store the error value in an appropriate place.

[16] Usually -1.

3.4. Memory-mapping operations

Memory mapping presents a more-complex challenge than normal filesystem operations. First, all memory maps must be created by the OS kernel as it needs to know which memory regions are in use. Second, some mappings may be created without any associated file. Third, the tracer has no way to control what happens when the user process access mapped memory.

All memory mappings that are backed by the filesystem are replaced with anonymous ones. Access to these anonymous mappings is forbidden to the process, so the tracer will be notified by the kernel when the process wants to read from a mapped memory page. When this happens, memory is filled with the data from the file. If the mapping was created with write access, the modified data is written on call to `msync` or `munmap`.

On a more-technical level, handling of memory-mapped filesystem operations involve:

1. when `mmap` is called with a file descriptor or without `MMAP_ANONYMOUS`, it is converted to an anonymous mapping with the `PROT_NONE` access;
2. the original data about mapping is sent to the VFS layer;
3. when the tracee tries to access the memory page with `PROT_NONE` access, `SIGSEGV` is generated and delivered to the tracer;
4. the tracer consults with the VFS whether the address that generated the page fault is located in any mapped region; if so, it receives appropriate data from the backing file and puts it into this memory region;
5. the memory is unlocked by an injected `mprotect` call;
6. the tracee is restarted.

Of course, all operations are validated against the original values: if the original access did not have `PROT_READ` permission, it would not be allowed. Writes performed by the user are noticed in the same way as reading from the region, i.e. by being notified with the `SIGSEGV` signal. It is also important for the implementation to allow splitting regions and changing their properties, as these are very common operations.

3.5. Communication between monitors and VFS daemon

Communication between the process monitor layer and VFS daemon is performed using a remote procedure call interface from the protobuf-c[17] library. The RPC model was preferred over other models (e.g. shared memory). It is simple and it fits well with the way both layers operate. Most of the time, the process monitor forwards the intercepted calls to the VFS without any modification or additional actions.

For simplicity, calls in the prototype from the monitor layer to the VFS daemon, are synchronous and blocking. However, in actual OS deployment, they can be asynchronous and non-blocking. This is due to the fact that one monitor will usually be

[17] Available at `https://code.google.com/p/protobuf-c/`.

tracing more than one process. With asynchronous processing, the monitor can start a call to the VFS daemon and then serve another stopped process instead of waiting for the call to finish.

3.6. VFS layer

The VFS layer, as shown in Figure 5, is mapped into a daemonised process with a running RPC service that dispatches calls to the skeletons, which later forward them to the functions implementing VFS logic. These functions handle such issues as process identification, input sanitization, etc. In the very end, calls are forwarded to the underlying filesystems.

The main logic of VFS is implemented as a single shared library. It is linked with the daemon module that is able to load the configuration (e.g. initial mounts), which starts the protobuf-c service. The library containing the RPC skeletons is also linked to the program.

3.6.1. RPC skeletons

The role of the skeletons is to receive data from the RPC service, allocate additional memory needed for *out* parameters and call the proper method in the VFS. The memory allocation is performed here due to the requirement that the VFS must work with any possible communication layer. Furthermore, overhead may be reduced when memory allocation is performed in the monitor process.

3.6.2. Filesystem operations

Filesystem operations that the VFS provides have interfaces designed after their standardized equivalents (i.e. POSIX operations they implement). They are executed to sanitize input values and provide the underlying with data that it understands. Usually, these operations take parameters very similar to their POSIX counterparts plus the `pid` parameter which carry the PID of the process executing the syscall. Listing 2 shows, as an example, the signature of the `truncate` system call.

This function needs to: *a*) obtain the real path using the information about the process, *b*) obtain the mountpoint relevant to the path (i.e. the filesystem in which this path is located), *c*) obtain the path relative to the root of the mountpoint, and *d*) call the real filesystem method with the appropriate path.

Listing 2 The signature of the `truncate` function.

```
int vfs_truncate(const char *path, off_t length, pid_t pid);
```

3.6.3. Process Identification and Related Information

The userspace VFS cannot rely on kernel to keep an updated information about user processes. Additionally, in most cases, such data cannot be stored by kernel as it

is incompatible with its view of the world. At least the following parameters of the process needs to be stored in VFS for the minimal working prototype:

1. its current working directory,
2. its open files,
3. mapped memory regions.

Adding other classes of system calls may require storage of other data in the VFS. Some of this data may require specialized structures, both to make them easier to work with and to speed up operations involving them. The open files are stored in the array with integer keys. These keys are file descriptors passed to user processes. For the memory regions, balanced binary trees are used, as implemented in the GLib library. Their properties allow for an easy look-up of the memory region that contains a specified address. This is especially useful when the memory region is being split in two by the user process.

3.6.4. Mountpoints

The VFS needs to store all mounted filesystems along with their mountpoints. It also must be able to locate the last mountpoints for the path to be looked up. The mountpoints could be stored in a simple list, but then it would be hardly usable. Instead, a partial directory tree is constructed and stored in a n-ary tree with directories as nodes. It is *partial* because it contains only nodes required for storing mountpoints and does not cache all directories. When the VFS needs to locate an appropriate mountpoint for a path it iterates over each level of the tree, looking for matching pathname prefixes.

3.6.5. Filesystem loading and mounting

Filesystems are implemented as dynamically loaded shared libraries. They are stored in a hash map with their names being the keys. This allows for their reloading, replacing, and even running different versions of the same filesystem at the same time. Mounting is performed by attaching the data about a mountpoint (e.g. its options) to the aforementioned partial directory tree.

3.7. Filesystems layer

Underlying filesystems should be stateless, i.e. not bound to a single mountpoint. This way, each filesystem may be mounted many times, and every mountpoint will use the same instance of it. Data that is required for correct operation should be passed by the VFS.

The created filesystem must conform to certain rules:

1. it must define structure that describes its available operations; this is basically a list of pointers to functions;
2. it must provide an `init` function and register itself in the VFS; in its first parameter is a filesystem name and the second parameter is a structure that defines its operations;
3. a negative returned value from an operation should be a correct `errno` error code.

4. Evaluation

Both the theoretical and practical aspects of the proposal described in previous sections were validated in four areas outlined before(above?):

1. efficiency of executing common operations in typical scenarios,
2. usefulness of the system from the filesystem developer point of view,
3. portability to other operating systems inheriting from the Unix philosophy and based on POSIX,
4. feasibility.

4.1. Efficiency

This part is carried out strictly for the sake of completeness as a comparison to other solutions. There is no doubt that the fully-userspace approach (which needs to emulate the kernel) will perform worse than non purely-userspace implementations. However, our goal is to see how large performance penalty is involved.

4.1.1. Methodology

A simple benchmark was created without employing external, off-the-shelf testing utilities. This is, in part, due to the fact that the benchmark used for the prototype needs to have all its filesystem calls virtualized. Additionally, using our own solution allows us to test non-transparent virtual filesystem switches such as GVFS.

Two sets of tests were performed; the first involved many short FS operations, whereas the second was based on a few long operations. The first set of tests was based on the skeleton with calls to short, fast functions, like: `[f]stat`, `chdir`, `mkdir`, etc. The *simple* test involved only the skeleton, whereas the *writes* test added one-megabyte writes to opened files, and the *reads* test added similar reads. In these tests, we counted the number of executions of the whole operation within ten seconds. The second set of tests measured the execution time of a single long-running operation a number of times. The *write* test wrote a 500 MiB file of random data, and the *read* test read a 500 MiB file. The *read with seek* test performed one hundred 50 MiB reads from random offsets in a file. The position was set using `lseek`. The last test — *lstat* — involved running a single function (`lstat`) 100,000 times to compute the overhead introduced by the prototype for a single, short call. This test was not run with GVFS as it has no exactly-equivalent operation.

In all test cases, the time was measured using `clock_gettime` with the `CLOCK_MONOTONIC` clock. All tests were run on the `tmpfs`-backed filesystem, to reduce potential effects of disk I/O. The tests were run on the bare kernel VFS, a FUSE-based transparent filesystem, GVFS, and our prototype-based transparent filesystem. All tests were executed on a system equipped with an AMD Phenom II X6 1055T processor and 8 GiB of RAM, 667 MHz memory clock rate, using Debian GNU/Linux with 3.0 kernel.

4.1.2. Results

The results from the first and the second set of tests are shown in Table 2 and Table 3 respectively. They are consistent. The kernel outperformed all of userspace-based approaches. FUSE was worse than GVFS, which could be explained by additional communication between a userspace daemon and the kernel. The prototype was slowest, as its operation involved even more kernel–userspace interactions than in FUSE. It only performed better with big-write operations. Moreover, the *read with seek* test has shown that the performance of the proposed prototype is closely related to the number of operations. With a small amount of long calls, it performs reasonably well, while for many short operations, its efficiency decreases. It is a direct consequence of the communication overhead. The *lstat* test shows that, for a short operation, the prototype performs over 1200 times worse than the native version that uses kernel without any additional layers. Very good results of GVFS may suggest that it optimizes local operations and does not route them through the separate daemon.

Table 2
A comparison of the average execution time in milliseconds. The results were obtained by executing specific tests for 10 seconds and counting the number of runs.

Test	Kernel	FUSE	GVFS	Prototype
simple	0.4072	34.7222	16.5017	175.4386
writes	48.5437	1428.5714	71.9424	1666.6667
reads	34.6021	416.6667	54.0541	1666.6667

Table 3
A comparison of execution times (in seconds) for the tests of long-running operations.

Test		Kernel	FUSE	GVFS	Prototype
write	Average	0.5831	8.1545	0.6674	5.2596
	Std. dev.	0.1037	0.2004	0.0597	0.2122
read	Average	0.4322	2.6334	0.4229	4.7311
	Std. dev.	0.0145	0.0164	0.0154	0.0476
read with seek	Average	2.4026	4.0808	2.2852	44.353
	Std. dev.	0.0592	0.0324	0.0422	0.4655
lstat	Average	0.0384	0.0414	—	48.409
	Std. dev.	0.0027	0.0034	—	0.2576

4.2. Usefulness

As mentioned in earlier sections of this work, some of the important goals that userspace development should fulfill (compared to the in-kernel modules) are eas-

ier programming and higher code quality. The proposed solution seems to fulfill these requirements. Individual filesystems are created as separate shared libraries that are loaded on-demand by the VFS during the first mount operation involving its filesystem. This implies that the shared libraries can be separately tested and debugged. They can also be easily replaced without recompiling the rest of the proposed prototype. Additionally, with some work, the filesystems implementations could even be replaced in runtime.

Userspace implementations are lightweight. The sample, transparent `localfs` takes around 240 lines of code in C (with logging). This is because of providing a strict, simple interface to implemented filesystems. Some operations are handled entirely by the VFS. Closely-related operations (e.g. `lchown`, `chown`, `chown`) require only one function in the filesystem, as the VFS can implement all of them with one specific filesystem function; for example, by executing `readlink` before `chown`.

4.3. Portability

As a test for the requirement of easy portability to other Unix-like platforms, the prototype was reimplemented in FreeBSD 9.0. The reimplementation involved only the process monitor (which was previously planned and necessary). Although the version of `ptrace` provided by FreeBSD offers virtually the same functionality as its Linux equivalent, it has some important differences regarding its usage and semantics of its operations and parameters. Nevertheless, the requirement of portability was fulfilled.

The current implementation is prone to difficulties related to non-standard filesystem operations. In other words, operating systems evolve separately and introduce new functionalities that, although generally similar in behavior, have very different mechanics and interfaces. These differences create a real problem for the portable implementation. It is not a problem of the monitor layer, as the implementation is scalable and the list of system calls is always system-dependent. The problem lies in the VFS layer that needs to generalize and abstract a common, sensible functionality.

Let us consider an example of asynchronous I/O. Traditionally, there were two mechanisms available in Unix: `select(2)` and `poll(2)` [12]. Nowadays, some platforms offer their improved equivalents. For example, Linux uses `epoll(4)` and FreeBSD has `kqueue(2)` [9, 13]. They have completely different interfaces, although both use file descriptors as a base for listening to events. We cannot just account for all possibilities in the VFS layer, because it would become system-dependent and the code would become complicated. We need to generalize the solution and provide stubs that would translate it correctly in the monitor layer for all supported operating systems. There is, however, one unfortunate consequence of such a solution: we can sensibly support only common and easily abstracted functionalities. Everything that cannot be done in the monitor layer will need to be handled in the VFS layer, which is discouraged given our requirements.

4.4. Feasibility

Evaluating feasibility involves the discussion of whether all aspects of the VFS can be implemented within the frames of the proposed solution. There are several problems in the solution and its implementation. The level of complexity is high. Specifically?, the process monitoring using `ptrace` involves making many workarounds for special cases that arise in the system calls (e.g. `mmap` handling). The correct implementation of the VFS that conforms to standards and common functionalities of all supported platforms is not easy either. Usually, there are many border cases that need to be detected (e.g. incorrect arguments, non-standard extensions) and handled in a sensible way. It is also worth reiterating that the VFS layer needs to collect all data about processes that might be required to virtualize their filesystem-related functionality. Finally, if we would like to ensure that the process is not aware that it is using a virtual filesystem, we would need to modify the behavior of many other elements of the operating system (for example the `/proc` filesystem).

Other related problems are *mixed* system calls. By *mixed*, we mean system calls that do not always need to be virtualized. There are calls, like the aforementioned `select`, that uses many file descriptors. Some of them may be strictly kernel-backed (e.g. sockets, pipes), and some may be virtualized by our system. Such a situation implies that every call to `select` must be analyzed, and a decision on how to handle these mixed arguments must be made. We can:

- fully reimplement the function, in which case, it would need to be handled by the monitor as it is not a VFS function, and also requires handling of events from the kernel,
- let the user process use a kernel version of the function, but trace its execution. Also, the kernel should be notified of changes in "our" file descriptors via dummy ones.

Such analysis and solutions could possibly be applied to other similar system calls.

5. Conclusions and future work

This paper began with a discussion of traditional UNIX filesystems: the solutions they use and the problems they have. We followed by showing what advantages userspace filesystems have over the traditional approach. We analyzed the most widely-known implementation, i.e. FUSE, and later, examined other possible ways for implementing a userspace filesystem. We chose a low-level solution based on the `ptrace` function and system calls emulation and implemented the prototype using this approach. We validated several aspects of our solution: efficiency, usefulness and portability. On the basis of this validation, we discussed existing and potential problems with the chosen approach.

We have been able to fulfill all the set requirements. The initial prototype that supports most common-system calls have been built successfully, with portable and flexible architecture. Porting it to FreeBSD involved converting only the process

monitor. The VFS layer has been implemented in a way that reduces the burden to filesystem creators.

Although the implementation was successful, it is clear that such a solution introduces very large overhead due to process tracing, inter-process communication, and mode switches. There is also an issue with the complexity of the VFS layer, which needs to take care of many aspects of the system normally reserved for the kernel: keeping process information, memory mappings, etc.

When looking at the shortcomings described in earlier sections, it is easy to see what is still needed:

1. adding virtualisation of remaining syscalls on all supported platforms,
2. adding support for more POSIX operating systems,
3. improving efficiency by using more sophisticated techniques in the process monitor or in the VFS; for example, by leveraging better access to the process memory, possibly with the `/proc` filesystem, or by introducing caches similar to those kept by the kernel,
4. solving the issue with handling different implementations of similar functionalities in various OS (e.g. asynchronous calls).

There are also many issues and improvements to consider that were not mentioned before. For example, implementing a way for easier VFS debugging by providing information filesystem for it, similar to `/proc`, and testing the VFS with the POSIX compliance tests like the ones used by the NTFS-3G project.[18]

References

[1] *Apple Darwin/OS-X manual.*

[2] *Linux Programmer's Manual, section 2.* https://www.kernel.org/doc/man-pages/online/pages/man2/select.2.html.

[3] *Linux Programmer's Manual, section 4.* https://www.kernel.org/doc/man-pages/online/pages/man4/epoll.4.html.

[4] *Linux Programmer's Manual, section 8.* https://www.kernel.org/doc/man-pages/online/pages/man8/ld.so.8.html.

[5] *Filesystem in Userspace*, 2011. http://fuse.sourceforge.net/.

[6] *FUSE Wiki — File systems using FUSE*, 2011. http://sourceforge.net/apps/mediawiki/fuse/index.php?title=FileSystems.

[7] *Linux Kernel Documentation*, 2012. http://lxr.linux.no/linux+v3.2.6/Documentation/filesystems/fuse.txt.

[8] Corbet J., Rubini A., and Kroah-Hartman G.: *Linux Device Drivers, Third Edition.* O'Reilly Media, 2005. https://lwn.net/Kernel/LDD3/.

[9] The GNOME Project: *GIO Reference Manual*, 2011. http://developer.gnome.org/gio/stable/.

[18] Available at http://ntfs-3g.git.sourceforge.net/git/gitweb.cgi?p=ntfs-3g/pjd-fstest.

[10] Herder J., Bos H., and Tanenbaum A.: *A lightweight method for building reliable operating systems despite unreliable device drivers*. Tech. rep., Vrije Universiteit, Amsterdam, The Netherlands, 2006.

[11] Kantee A. and Crooks A.: *ReFUSE: Userspace FUSE Reimplementation Using puffs*. In: *Proc. of the 6th European BSD Conference (EuroBSDCon)*. 2007.

[12] Larsson A.: *gvfs status report*. http://mail.gnome.org/archives/gtk-devel-list/2007-February/msg00062.html.

[13] Lemon J.: *Kqueue - A Generic and Scalable Event Notification Facility*. In: *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pp. 141–153. USENIX Association, 2001. ISBN 1880446103. http://dl.acm.org/citation.cfm?id=647054.715764.

[14] Leslie B., Chubb P., Fitzroy-Dale N., Götz S., Gray C., Macpherson L., Potts D., Shen Y.T., Elphinstone K., and Heiser G.: *User-Level Device Drivers: Achieved Performance*. *Journal of Computer Science and Technology*, vol. 20, pp. 654–664, 2005. ISSN 1000-9000. URL http://dx.doi.org/10.1007/s11390-005-0654-4.

[15] Modine A.: *Linus calls Linux 'bloated and huge'*. http://www.theregister.co.uk/2009/09/22/linus_torvalds_linux_bloated_huge/.

[16] Rajgarhia A. and Gehani A.: *Performance and extension of user space file systems*. In: *Proceedings of the 2010 ACM Symposium on Applied Computing*, pp. 206–213. ACM, 2010.

[17] The SCO Group, Inc.: *SCO UnixWare 7 manual, section 2*. http://uw714doc.sco.com/en/man/html.2/ptrace.2.html.

[18] Spillane R.P., Wright C.P., Sivathanu G., and Zadok E.: *Rapid file system development using ptrace*. In: *Proceedings of the 2007 workshop on Experimental computer science*, ExpCS '07. ACM, New York, NY, USA, 2007. ISBN 978-1-59593-751-3. http://doi.acm.org/10.1145/1281700.1281722.

[19] Thurlow R.: *RFC 5531 RPC: Remote Procedure Call Protocol Specification: Version 2*, 2009.

Affiliations

Łukasz Faber
AGH University of Science and Technology, Krakow, faber@agh.edu.pl

Krzysztof Boryczko
AGH University of Science and Technology, Krakow, boryczko@agh.edu.pl

Received: 10.01.2013
Revised: 06.02.2013
Accepted: 22.02.2013