FUSE Driver Update

BY ALAN SOMERS

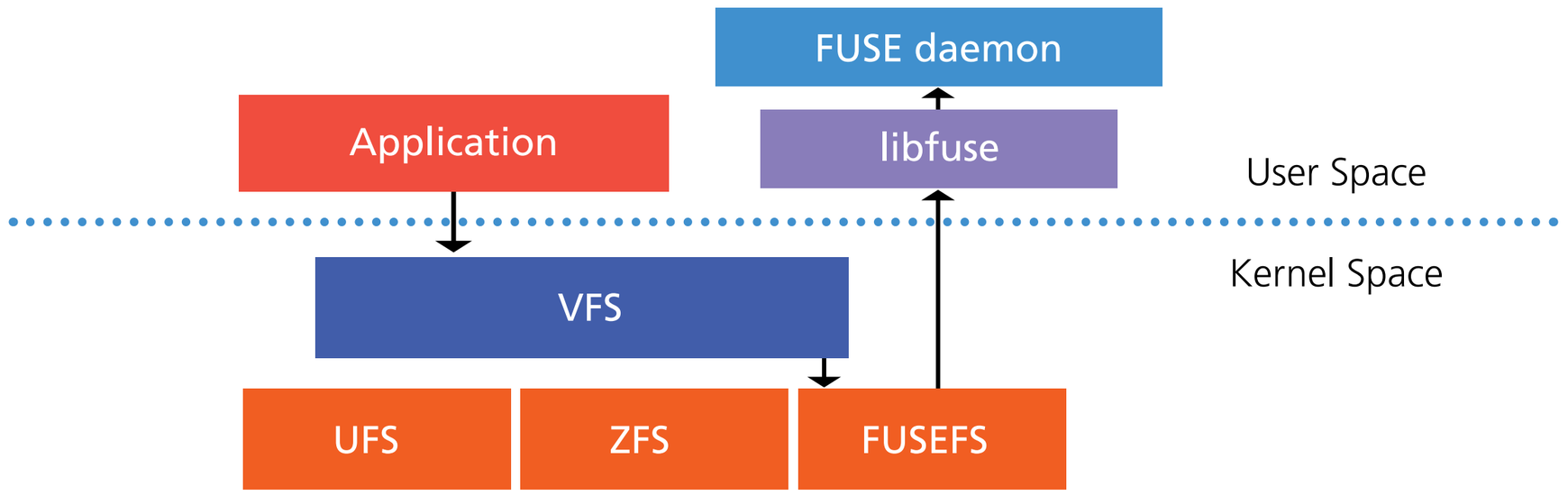
Have you ever mounted an NTFS-formatted hard drive on your FreeBSD or Linux laptop? Or have you ever connected your PTP camera to your laptop and browsed its photos through your file browser? Maybe you've used a distributed file system like MooseFS, GlusterFS, or CephFS? If you answered yes to any of those questions, then you've already used FUSE.

What Is FUSE?

FUSE (Filesystem in USErspace) is a driver and a protocol for allowing userspace processes to implement a file system which the kernel presents to other processes just like any other. Running in userspace makes file systems considerably easier to develop and debug than kernelspace file systems. That's why file systems like NTFS use FUSE. Userspace programs can also access libraries and utilities that aren't normally available in kernelspace, enabling virtual file systems like sshfs (which mounts a remote server's files over an SSH connection) and encfs (an encrypted file system that uses an ordinary file system as its backing store). Finally, FUSE provides license hygiene for kernel modules. For example, a GPLv3 kernel module cannot coexist alongside either a CDDL or a GPLv2 module; the result would not be legally redistributable. But a GPLv3 FUSE file system is not a derived work of the kernel, so it's perfectly redistributable.

FUSE was originally written for Linux in 2005, first appearing in kernel version 2.6.14. It proved so useful, however, that ports soon began to appear. Today, OSX, OpenBSD, Illumos, Minix, and of course FreeBSD also support FUSE. NetBSD does not use the FUSE protocol, but it still supports many FUSE file systems via a userspace compatibility layer. And a FUSE driver for DragonflyBSD is under development.

FUSE API Stack



FreeBSD's FUSE Port

FreeBSD's FUSE driver began life as a GSoC project in 2005 by Csaba Henk but wasn't integrated into the base system. A further GSoC project in 2011 by Ilya Putsikau finished the port and Attilio Rao merged it soon after. However, the 2011 version was still a few years behind the then-current protocol and had some unresolved bugs. In the subsequent 8 years, many

of those bugs went unaddressed and there was little maintenance or new features until now. Thanks to Foundation sponsorship, I was able to rectify this situation. I've largely rewritten the driver, updated the protocol version, fixed dozens of bugs, and added new features and performance enhancements.

Using FUSE File Systems

Using FUSE file systems is fantastically easy—that's the whole point of FUSE. Once mounted, they can be accessed just like a normal file system. The mount command varies between different file systems. For example, to mount an ext2 file system:

```
sudo pkg install -y fusefs-lkl e2fsprogs
truncate -s 1g /tmp/ext2.img
mkfs.ext2 /tmp/ext2.img
mkdir /tmp/mnt
lklfuse -o type=ext2 /tmp/ext2.img /tmp/mnt
```

Notice the lackof `sudo` in the second command. That's intentional. To enable this behavior, set the `vfs.usermount` sysctl to 1. FUSE daemons can run unprivileged. When they do, the mountpoint is only accessible by the user running the daemon. That's to prevent the daemon's user from spying on the I/O of other users. In fact, many FUSE daemons eschew any explicit permission checks in this mode, allowing the mounting user to do virtually whatever he wants with the file system, like this:

```
install -m 755 -o root -g wheel -d /tmp/mnt/bin
install -m 755 -o root -g wheel /bin/sh /tmp/mnt/bin/sh
```

The unprivileged user can create files owned by root! At first glance, that looks like a whopping security hole. But it's actually ok, since other users can't access that file system at all. An astute and paranoid reader might ask whether the mounting user can set create a SUID file and elevate his privileges that way. Rest assured—he can't. Unprivileged mounts automatically get `nosuid` set. All that's really happening is that the user is changing /tmp/ext2.img, which he owns. This feature is very cool. For example, you can use it to create a complete bootable image, such as for an embedded system.

Of course, that's only one use case. More traditional mounts are possible. For example, if a certain file system weren't available in the running kernel, you could mount a FUSE implementation as root with the `allow_other` and `default_permissions` options. That way it would be available to all users, and would function just like any other filesystem:

```
umount /tmp/mnt
sudo lklfuse -o type=ext2,allow_other,default_permissions \
/tmp/ext2.img /tmp/mnt
```

Developing FUSE File Systems

Compared to an in-kernel file system, developing file systems for FUSE is much easier. And not only is it easier to write a FUSE file system, it's also very easy to write it portably. Most FUSE file systems need few to no changes in order to run on several operating systems.

One of the benefits of programming in userland is the program is not limited to C. Indeed,

FUSE bindings are available for Perl, Python, Rust, Javascript, Java, Ruby, Nim, C#, Go, and probably others too.

Starting a FUSE daemon is slightly complicated: the daemon first opens `/dev/fuse`, then calls `nmount` with that file descriptor as one of the arguments. Then it begins to read FUSE requests from that same file descriptor and write the responses back. However, developers rarely need to worry about those details, because libfuse takes care of it. Instead, whether writing in C or another language, the developer generally just has to define callbacks for each supported FUSE operation. Then, the library takes care of all the plumbing. For example, in Python the crucial code for a "Hello World" example is just 37 lines. (For the full example, see https://github.com/libfuse/python-fuse/blob/master/example/hello.py).

```
class HelloFS(Fuse):

    def getattr(self, path):
        st = MyStat()
        if path == '/':
            st.st_mode = stat.S_IFDIR | 0o755
            st.st_nlink = 2
        elif path == hello_path:
            st.st_mode = stat.S_IFREG | 0o444
            st.st_nlink = 1
            st.st_size = len(hello_str)
        else:
            return -errno.ENOENT
        return st

    def readdir(self, path, offset):
        for r in  '.', '..', hello_path[1:]:
            yield fuse.Direntry(r)

    def open(self, path, flags):
        if path != hello_path:
            return -errno.ENOENT
        accmode = os.O_RDONLY | os.O_WRONLY | os.O_RDWR
        if (flags & accmode) != os.O_RDONLY:
            return -errno.EACCES

    def read(self, path, size, offset):
        if path != hello_path:
            return -errno.ENOENT
        slen = len(hello_str)
        if offset < slen:
            if offset + size > slen:
                size = slen - offset
            buf = hello_str[offset:offset+size]
        else:
            buf = b''
        return buf
```

The security model of FUSE may come as a surprise: by default, the daemon is responsible for authorizing all operations. That does place an extra burden on the FUSE file system developer. However, the usual behavior can be achieved by always using the `default_permissions` mount option. The upside is that the FUSE file system can use exotic authorization strategies, like bespoke ACL formats. This feature is also very useful for networked file systems that do authorization on the server, rather than on the clients.

New Features

FreeBSD 13's new fusefs(5) driver adds several new features, all of which should be immediately useable by existing FUSE file systems. Here are the most interesting:

- Kernel-side permissions checks (`-o default_permissions`) is now fully implemented.
- `mknod(2)`, `pipe(2)`, and `socket(2)` are now supported, so it's possible to create any type of file on a fusefs file system.
- Server-side support for `fcntl(2)` advisory locks has been added. Previously, fcntl locks were always implemented in-kernel, but that was insufficient for network file systems that do distributed locking.
- When mounted with `-o intr`, and if the server supports it, fusefs mounts are now fully interruptible. That means that a signal can interrupt an operation that's blocked waiting for a response from the server. It's similar to NFS's mount option of the same name.
- A fusefs mountpoint can now be exported over NFS.
- The kernel will now cache file names and attributes if the server allows it. The server can also asynchronously evict part of the kernel's cache. The kernel can also cache reads and writes, if permitted by the server. Finally, it will read ahead when a process appears to be reading sequentially. These features will all improve performance with no application or configuration changes required.

ALAN SOMERS has been a FreeBSD committer since 2013. In 2019, he rewrote the fusefs driver under contract from the FreeBSD Foundation. Currently Alan works for Axcient on their FreeBSD storage servers.