

## Contents

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>A quickstart for DIY types</b>   | <b>1</b>  |
| 1.1      | Get the sources . . . . .   | 2         |
| 1.2      | Dependencies . . . . .  | 2         |
| 1.3      | Compilation . . . . .   | 2         |
| 1.3.1    | Library (Fuse) . . . . .  | 3         |
| 1.3.2    | SshFs . . . . .   | 3         |
| 1.3.3    | Kernel module . . . . .   | 3         |
| 1.4      | Using the Fuse based ssh filesystem . . . . .   | 4         |
| 1.5      | Note on other filesystems . . . . .   | 5         |
| 1.6      | Bugs . . . . .  | 6         |
| <b>2</b> | <b>FAQ</b>  | <b>6</b>  |
| <b>3</b> | <b>Implementation notes about Fuse for FreeBSD, with special emphasis on comparing it to Linux Fuse</b> | <b>10</b> |
| 3.1      | VFS API . . . . .   | 11        |
| 3.2      | Mounting . . . . .  | 12        |
| 3.2.1    | interface . . . . .   | 12        |
| 3.2.2    | security . . . . .  | 13        |
| 3.2.3    | dealing with the “allow other” misery . . . . .   | 15        |
| 3.2.4    | anything else . . . . .   | 16        |
| 3.3      | [v]inode operations . . . . .   | 16        |
| 3.4      | Syncing . . . . .   | 18        |
| 3.5      | Messaging . . . . .   | 19        |
| 3.6      | Miscellaneous . . . . .   | 21        |

## 1 A quickstart for DIY types

Here we will describe how to install manually and try Fuse under FreeBSD. There are a lot of Fuse based filesystems – see <http://fuse.sourceforge.net/wiki/index.php/FileSystems>. While many of them can be expected to work under FreeBSD as-is or with minor tweaks, here we will concentrate on one specific filesystem, sshfs. See also <http://fuse.sourceforge.net/wiki/index.php/FileSystemsOnFreeBSD>.

For most users, it is recommended to use the ports instead of using these instructions. Follow these instructions if you want to use a development snapshot or you want to tweak something (eg., port another Fuse based filesystem to FreeBSD), and you are looking for a starting point. Otherwise just install the `sysutils/fusefs-sshfs` port.

First of all, note that currently you will be able to perform a fully functional installation only on FreeBSD systems from the RELENG\_6 or newer branch. (Userspace components can be compiled also on RELENG\_5 systems.)

## 1.1 Get the sources

Here we will describe the case when you get all code from their respective SCM repositories. You should use snapshots from not earlier than 17th Nov, 2005. The first Fuse release with FreeBSD support is 2.5.0-pre2, released ad 9th Jan, 2006.

- Get fuse and sshfs sources. First login into the Fuse Anoncvs service:

```
cvs -d:pserver:anonymous@fuse.cvs.sourceforge.net:/cvsroot/fuse login
```

(when you asked for a password, hit enter), then fetch the sources:

```
cvs -z3 -d:pserver:anonymous@fuse.cvs.sourceforge.net:/cvsroot/fuse co -P fuse
cvs -z3 -d:pserver:anonymous@fuse.cvs.sourceforge.net:/cvsroot/fuse co -P sshfs
```

- You have to have the FreeBSD module.
  - If you read this document as a part of a snapshot of the module, you already have it.
  - Else if you read it online, you can expect these instructions to work with the latest committed code. Go to <http://fuse4bsd.creo.hu> and check out the code from your preferred SCM.
  - Alternatively, you can download a release tarball from <http://fuse4bsd.creo.hu> and use the version of this document which is bundled with the release.

## 1.2 Dependencies

- sshfs depends on glib20 and pkgconfig.
- To compile Fuse and sshfs from CVS, you will need to have an up to date installation of the gnu-autoconf, gnu-automake, libtool15 and gettext ports.

## 1.3 Compilation

Each of the subsections below will assume that you are at the top of the respective source tree. The instructions below will assume you use a Bourne compatible shell.

We will do a non-privileged installation (I'd say that's easier than set up a jail). Choose your installation path and set the `FPREFIX` variable to this value (be it an absolute path).

We will assume that you have put all three source trees in the same directory, and their top directories are named `fuse`, `sshfs`, and `fuse4bsd`, respectively.

### 1.3.1 Library (Fuse)

- As CVS snapshots don't ship with pre-generated instances of autotools related files, you have to generate them. Do it with

```
ln -sf /usr/local/share/aclocal/libtool.m4 . &&
ln -sfv 'pkg_info -L 'gettext*' | grep /usr/local/share/aclocal/' . &&
(export PATH=/usr/local/gnu-autotools/bin:$PATH &&
libtoolize &&
aclocal -I . &&
autoheader &&
autoconf &&
automake -a)
```

- Install fuse with

```
ln -sf ../kernel/fuse_kernel.h include/ &&
./configure --prefix=$FPREFIX --with-pkgconfigdir=$FPREFIX/libdata/pkgconfig &&
make &&
make install
```

(The pkgconfigdir setting is not necessary, it's just for making the installation more "BSDish".)

### 1.3.2 SshFs

- Generate autotools files with

```
ln -sf /usr/local/share/aclocal/pkg.m4 . &&
(export PATH=/usr/local/gnu-autotools/bin:$PATH &&
aclocal -I . &&
autoheader &&
autoconf &&
automake -a)
```

- Type

```
PKG_CONFIG_PATH=$FPREFIX/libdata/pkgconfig/ ./configure &&
make
```

### 1.3.3 Kernel module

- Copy (or link) the `kernel/fuse_kernel.h` file of the Fuse source tree into the `fuse_module` directory.

- Type `make`. If you want normal quantity of debug output (currently in the form of kernel messages), add `DEBUG2G=1` to `make`'s environment, if you want tons of debug output, add `DEBUG=1` to the environment.

You might want / need to make the module make use of the features enabled in your kernel config (like support for diagnostic tools like `INVARIANTS`, `WITNESS`). (You do need it, eg., if you enabled `DEBUG_LOCKS` which affects binary compatibility.) To make this happen, the simplest solution is adding a `KERNCONF=<YOUR_CONF_HERE>` parameter to your `make` arguments. This will work only if you used standard locations when you compiled the kernel. If that's not the case, you can use the `KERNCONFDIR` parameter to explicitly specify the location of the config headers. For the currently running kernel this should be the path seen in the output of `uname -v`.

Congratulations, you have all components prepared!

## 1.4 Using the Fuse based ssh filesystem

Here we will show how to setup Fuse so that non-privileged users can use it, too (although it's up to you if you want this).

As the superuser, do

```
kldload fuse_module/fuse.ko
sysctl vfs.usermount=1
```

You'll have to act as a user belonging to the `operator` group, or set device permissions appropriately (cf. 2).

Pick your favourite ssh accessible account, say, it's `foo@bar.baz`.

Go to `sshfs`' directory. First prepare the mount:

```
mkdir -p ~/fuse &&
export LD_LIBRARY_PATH=$FPREFIX/lib/
export PATH=$FPREFIX/bin:$PATH
```

Now you can activate the filesystem with:

```
./sshfs foo@bar.baz: ~/fuse
```

This will start up the filesystem daemon, who will mount the home directory of the "foo" account at `bar.baz` on `~/fuse` (using `mount_fusefs`). You can get it done vice versa, so that `mount_fusefs` will spawn the daemon:

```
mount_fusefs auto ~/fuse ./sshfs foo@bar.baz:
```

Here is also a more customized example:

```
MOUNT_FUSEFS_VERBOSE=1 \
./sshfs -o push_symlinks_in,idmap=user,reconnect -C foo@bar.baz:/ ~/fuse
```

If it failed to work for some reason, you can perform the steps of mounting the filesystem manually by yourself. Eg., first type

```
FUSE_DEV_NAME=/dev/fuse0 ./sshfs foo@bar.baz: -d
```

and then on another terminal:

```
mount_fusefs /dev/fuse0 ~/fuse
```

(This will work only if the device `/dev/fuse0` is not yet in use.)

When you have finished using the filesystem, there are several ways of taking it down. For example, you can simply

```
umount ~/fuse
```

However, for this to work, you need the filesystem daemon operating properly. Unmounting by device doesn't rely on the daemon, so the following is more robust (assuming you mounted the device `/dev/fuse0`):

```
umount /dev/fuse0
```

Or you can kill the daemon (cf. `kill(1)`). Just before she exits, she will unmount the filesystem.

For finding out these parameters of a Fuse mount, see 2.

For more details on mounting, see `mount_fusefs(8)`.

## 1.5 Note on other filesystems

The above instructions will help you to try using other Fuse based filesystems on FreeBSD, but there is one issue you should be aware of.

The Fuse library API has went through several revisions. Currently the library maintains backward compatibility on Linux, but Fuse author Miklós Szeredi decided to support FreeBSD only starting with API revision 25 (not without reason: this is the first API revision which utilizes only POSIX/SUSV3 compatible interfaces, therefore alleviates platform dependency issues filesystem authors have to face with).

For more information about the backward compatibility issue and the way of updating filesystems see the mailing list subthread starting from <http://thread.gmane.org/gmane.comp.file-systems.fuse.devel/2310> and subsequent posts. To cut the story short, your `statfs` method will have to be converted from using `struct statfs` to using `struct statvfs` (and take care about filling its `f_frsize` field). In most cases (ie., if your filesystem was using API 22) this will be enough, although it might be worth to take a look at the new features of API 25. (Note that API 23 and 24 just brought in extensions, so API 22 is a strict subset of these, with no compatibility issues.)

An overview of the API revisions of the Fuse library can be found at <http://fuse.sourceforge.net/wiki/index.php/ApiChangelog>.

Also see the following posts, with concrete examples of patches for (quick'n'dirty) API upgrade:

- Updating Fuse Perl bindings (from API 21): <http://creo.hu/pipermail/fuse4bsd-devel/2006-January/000117.html>
- Updating Fuse Python bindings (from API 21): <http://sourceforge.net/mailarchive/message.php>
- Updating Tdfs (from API 22): <http://creo.hu/pipermail/fuse4bsd-devel/2006-January/000116.html>

A more carefully coded example is provided by Fuse Python bindings (after its resurrection in May-Jun 2006). Currently it supports all APIs from 22 to 26. It's the suggested reference for getting around API compatibility problems.

## 1.6 Bugs

See the respective section of `mount_fusefs(8)`.

## 2 FAQ

*I see a new `/dev/fuseN` device is created upon each mount of a Fuse based filesystem. However, I'd like to see the existing one getting reused.*

The development version supports device reusal. If you'd like to stay away from the bleeding edge, you can force reusing an existing device by explicitly specifying it, eg.

```
mount_fusefs /dev/fuse0 /mnt/fuse foofs
```

OR

```
FUSE_DEV_NAME=/dev/fuse0 foofs /mnt/fuse
```

*The filesystem daemon has died for some reason. Can I start a new copy of it as a drop-in replacement?*

No. The filesystem hierarchy is stored in the daemon process' memory and it's neither serialized nor appears on a permanent storage. If the daemon dies, the filesystem goes away. No use of keeping it around.

*But what if I use a daemon which is "displaying" a stateless filesystem or organizes serialization and permanent storage by itself?*

Well, if really that's the case, the daemon should fork a "gatekeeper co-daemon" which keeps the fuse device open and spawns the new daemons on

demand. As long as the device is open, the filesystem won't be considered as "doomed".

Bug the authors of permanency-capable filesystems if you have such needs or implement it by yourself.

*What filesystems? As far as I know, sshfs is the only useable Fuse based "real" filesystem under FreeBSD...*

This is just the humble present. Other filesystems should be easy to port to FreeBSD (just a matter of a compilation with proper flags and includes). If a binding for an interpreted languages gets ported, probably a whole bunch of filesystems will be available immediately.

Feel free to contribute. See also <http://fuse.sourceforge.net/wiki/index.php/FileSystemsOnFreeBS>

*The daemon has died. Now I can't get rid of the Fuse mount: when I try to unmount the filesystem, the `umount` command hangs.*

Try unmounting by device name. That is, if you mounted the filesystem as

```
mount_fusefs /dev/fuse0 /mnt/fuse foofs
```

then do

```
umount /dev/fuse0
```

(instead of `umount /mnt/fuse`) – this works without asking for assistance from the daemon. If you refused with *Device busy*, then try a forced unmount (add the `-f` flag). If unmount still hangs, that means that there is a process which is using the filesystem and is also hanging. Search & destroy such processes and try again.

*But do I not need to fear of a forced unmount? Won't my system panic upon doing so?*

It definitely should not. However, if it still does, that's a bug. Please drop then a mail to the fuse4bsd-devel mailing list with the appropriate description of the case.

*How can I find out the name of the Fuse device used by my filesystem and the pid of the daemon?*

If you mount with an explicit device name like

```
mount_fusefs /dev/fuse0 /mnt/fuse foofs
```

then of course you will know the device name (although your mount attempt will be refused if `/dev/fuse0` is already in use).

In case of "auto" mounts you'll be informed about the device if you mount with `-v` like:

```
mount_fusefs -v auto /mnt/fuse foofs
```

OR

```
MOUNT_FUSEFS_VERBOSE=1 foofs /mnt/fuse
```

When the mount is in place, the `mount(8)` utility will give you information how devices correspond to mount paths.

If you know the device of your mount, do an

```
fstat /dev/fuse*
```

to see the users of the fuse devices. See also `fstat(1)`.

*Now the contrary. I unmounted a Fuse based filesystem, and the unmount has completed cleanly. However, the daemon is still kicking and is occupying the respective Fuse device, and I have to **kill -9** it.*

You either use an old version of the Fuse library / `fuse4bsd` or the filesystem in question doesn't make use of the respective standard library routine.

Try upgrading to Fuse  $\geq$  2.5.0-pre2 and `fuse4bsd`  $\geq$  0.3.0-pre1.

*Yet another weirdness upon unmount. The daemon did exit, but uhh... It wasn't a nice death. It dumped core or displayed a strange message like **Thread 8090600 has exited with leftover thread-specific data after 4 destructor iterations***

You use an old version of the Fuse library. Try upgrading to Fuse  $\geq$  2.5.0-pre2.

*So what is this hype about Fuse and non-privileged mounts?*

Fuse is designed so that anyone can use it without risking the consistency of the system. That is, a Fuse daemon process is not needed to be trusted. Whatever the daemon is doing, no worse outcome should this have (beyond the daemon's privileges) than a non-functional mount.

*How can I enable non-privileged mounts?*

Use the standard FreeBSD toolset for this purpose. That is, set the `vfs.usermount` sysctl to 1. Then find out whom should be able to mount Fuse based filesystems and set the `devfs(8)` rules for `/dev/fuse*` appropriately. Eg., letting all users to mount Fuse devices would look like

```
devfs ruleset 10
devfs rule add path 'fuse*' mode 666
```

(of course, you can `chmod/chown` individual Fuse devices, too). According to the default permissions, members of the `operator` group are those who can set up Fuse based filesystems. (Ie., only they can do *primary mounts*. Others still can use an already mounted Fuse filesystem via *secondary mounts*, if the filesystem doesn't prohibit this. See 2.)

*Does this support of non-privileged mounts mean that I should let everyone mount Fuse filesystems?*

No. Your mileage may vary. Noone will say you are a fool if you find non-privileged mounting inconsistent with your system policy. And regarding Fuse, there might be bugs in it, as in any software. Allowing a user to mount Fuse based filesystems will make those bugs exploitable for that user.

However, it looks not too much dangerous to allow non privileged mounting of Fuse filesystems.

*I do want to make it possible for some users on my system to mount Fuse filesystems, yet I don't want to allow them to mount without restrictions. Is there any way to achieve this?*

Yes. Keep the `vfs.usermount` sysctl on 0, and set the appropriate access policy on the usage of `mount_fusefs` as root by `sudo(8)`.

Note that you *should not* put `mount_fusefs` into your “sudoers” file as is, as that command is capable of executing an arbitrary command. There are various ways of disabling this feature. Eg., passing a `-S` flag to `mount_fusefs` will cause it to reject spawning daemons, or specifying a certain daemon (by an absolute pathname) will make it run that specific daemon and nothing else.

In general, you are suggested to wrap `mount_fusefs` in a script. The command line parsing routine of `mount_fusefs` desinged so that it makes scripting easy.

*I run mounted a Fuse daemon using my own unprivileged account, and now my wife is in tears because she is prohibited from using the filesystem (which works finely for me). What's happening here?*

If your wife used the filesystem, all her I/O activity directed at the filesystem would be seen and controlled by your daemon process. This shouldn't happen without your wife's agreement, so to stay on the safe side, she will be bluntly refused from using your filesystem.

The “agreement” above is not just a metaphor. She can effectively “sign” such an agreement by doing a so-called *secondary mount* of your filesystem. First she has to find out or be told about device your filesystem is using, say it's `/dev/fuse0`. Then she can do the secondary mount like

```
mount_fusefs /dev/fuse0 /home/kate/lesliefs
```

From that on, she will be able to use the filesystem (either via her private mount point or via the original) while the secondary mount is in place. The secondary mount will appear with `/dev/fuse0#1` as device name and it should be unmounted via device name like

```
umount /dev/fuse0#1
```

(`umount /home/kate/lesliefs` would be redirected to the primary mount like any query on the secondary mount!)

*I want to mount a Fuse daemon such that the mount would be systemwide useable, but I want the daemon running on a low privilege level (like as user `nobody`). Will then all of my users have to do their own secondary mount before they can use the filesystem?*

Not necessarily. Just use the `allow_other` mount option (available only for root).

### 3 Implementation notes about Fuse for FreeBSD, with special emphasis on comparing it to Linux Fuse

Here I summarize the differences between the original Fuse implementation for Linux, and Fuse for FreeBSD, with references to the differences of OSes themselves, where it's necessary.

First let's see what's implemented by the FreeBSD Fuse module. Perhaps the most platform-independent way of expressing the functionality is to list the utilized Fuse operations and say: the OS functionality provided is what these operations can be used for. So, we (*Fuse for FreeBSD*) utilize these:

- FUSE\_LOOKUP
- FUSE\_FORGET
- FUSE\_GETATTR
- FUSE\_SETATTR
- FUSE\_READLINK
- FUSE\_SYMLINK
- FUSE\_MKNOD
- FUSE\_MKDIR
- FUSE\_UNLINK
- FUSE\_RMDIR
- FUSE\_RENAME
- FUSE\_LINK
- FUSE\_OPEN
- FUSE\_READ
- FUSE\_WRITE
- FUSE\_STATFS
- FUSE\_RELEASE
- FUSE\_FSYNC
- FUSE\_INIT

- FUSE\_OPENDIR
- FUSE\_READDIR
- FUSE\_RELEASEDIR
- FUSE\_FSYNCDIR
- FUSE\_ACCESS
- FUSE\_CREATE

Ie., we don't use the following ones:

- FUSE\_SETXATTR
- FUSE\_GETXATTR
- FUSE\_LISTXATTR
- FUSE\_REMOVEXATTR
- FUSE\_FLUSH

However, some nuances are left hidden by this rudimentary description. We do provide `mmap`, and give kernel level support to the `allow_other`, `direct_io`, `kernel_cache` options. We don't support attribute and name caching (“`d_cache`” in Linux). We neither have specific large file support.

Support for all the above mentioned goodies can be added in due time, maybe except for (the functionality behind) `FUSE_FLUSH` (more on this later). That is, these differences are related to the level of maturity of the two implementations, and as such, these are the less interesting ones. Let's see those differences of the modules which stem from the differences of the two OS and/or preferences of the implementors.

### 3.1 VFS API

This is the one of the greatest impact. Linux VFS operations are inherently (struct) file based, BSD VFS operations are inherently vnode based (BSD vnodes correspond to Linux inodes). That is, all file-related VFS operations take a (struct) file parameter in Linux.

In BSD, in some cases you can put your hands on (struct) files, in some cases not. The API encourages and helps you to not use files (if the nature of the fs permits, these can be completely avoided), and a consequently file based design is simply impossible (as I could experience when I tried to make a “naive” port). You are given a subtle system of structures and op vectors which let you do many things with files if you insist on to do so, but which gently shields you from seeing those ugly files in your everyday life. (The

vnode centered nature is common to all BSD flavours; how comprehensive is that optional access to files is flavour dependent, but as I said, never complete.)

This becomes manifest in different ways. The most visible of these is

## 3.2 Mounting

### 3.2.1 interface

There are many factors which result in a different user interface for the two OS.

In Linux, there is one Fuse device, and when a Fuse daemon is started, it calls a helper utility which opens the Fuse device for the daemon and performs the mount syscall “atomically”. Hence all mount parameters, including mount point, has to be passed to the daemon.

In FreeBSD, there are several Fuse devices. When a Fuse daemon is started, it attaches itself to one – either completely on its own or using an already existing file descriptor, but it doesn’t call any helper program and doesn’t do anything mount related. Mounting is done by an external utility. This means that there must exist a global namespace in which the mounter can specify the daemon to mount. As one Fuse device can serve no more than one daemon, it’s easy: the device name is used to specify the subject of the mount (what a novelty).

In Linux, this couldn’t work out this way (given the design chosen by Linux Fuse): there is one Fuse device only, and a fail-safe identification of the subject of the mount is possible only within the scope of one process.

In FreeBSD, altering from the Linux way of atomic mounting is not a design choice, it’s a must. Unlike Linux, BSDs use a different type of op vector devices and for regular files (the device access and vnode API is not unified). The device access API is not only different but almost completely (struct) file unaware. So we can’t distinguish between different openers of a given device.

The traditional solution would be working with a static set of fuse devices (nnpfs, the multiplatform userspace vfs framework of Arla has chosen this solution in its BSD implementations). However, FreeBSD has a peculiarity which lets us implement Fuse as dynamically as it’s in Linux: a wisely designed in-kernel device filesystem.

FreeBSD devfs provides a mechanism to register event handlers for certain device name patterns, by which new devices can be spawned on the fly and system call handling can be delegated. We rely on this mechanism for providing a dedicated Fuse device to each Fuse daemon.

At this point, one could spot one definite advantage of the Linux way of doing the mount: it’s comfortable. That the state of Fuse daemons is explicitly reflected in the devfs namespace might seem to be an elegant con-

cept, yet performing the device squatting and the mount syscall via separate commands will get tedious pretty fast.

To get over this, the mount utility serves as an umbrella frontend to all these mechanisms, and running `mount_fusefs auto mp daemon args` will open a Fuse device, start the daemon with the given args and makes it chew on the device, and finally mount the daemon on mp.

A more recent enhancement of Fuse on FreeBSD is adding support also for the Linux style interface. That is, you can mount Fuse daemons with the `daemon args mp` syntax as well, in which case the daemon will invoke `mount_fusefs` by herself. (Note: this of course won't work if you want to do mounting on a different privilege level than that of the daemon).

It's also a comfortable thing in Linux that when you run `mount(8)` to see the list of mounted filesystems, what you will find in the "mounted device" (first) column for Fuse filesystems, is a compact custom description of the filesystem. Under FreeBSD you will see simply the device name as is, which might feel crude related to the Linux listing. Yet I don't plan implementing an "fs-summary-as-device" hack under FreeBSD: that would mean loss of information. Even if showing the device name is not so friendly, that is the exact unique identifier of the mount. What I might happen to do is to write (or to wait for one being written) a nifty shell script which gathers Fuse mount related informations and presents it in a human-loveable way – you can find out everything you want via `mount(8)`, `ps(1)` and `fstat(1)`.

### 3.2.2 security

Fuse is a dedicatedly Promethean filesystem: it aims to the bring the power of interaction via a custom filesystem interface to ordinary users. Practically, this boils down to doing customized non-privileged mounts. In Linux, ordinary users are usually allowed to do mounts via the `user(s)` option of `fstab`. This is a fairly static mechanism, so to be able to do the customized non-privileged mounts as it's required, Linux Fuse rolls his own: the above mentioned device opener/mounter utility is written in a way so that it can safely bear the `suid` bit, and then the appropriate permission handling logic is stuffed into this utility, too.

In FreeBSD no heroic action is needed. No `setuid` mounting is needed – unlike Linux, there are no `user(s)` option in `fstab`, and `mount(8)` itself is not `setuid`.

Non-privileged mounts are handled via a system-wide policy: if the `vfs.usermount sysctl` is 1, users can mount over mount points owned by them, if it's 0, only root is allowed to mount. Unlike as it's with the `user(s)` `fstab` options, Fuse mounts fit into this frame nicely, so there is no need to implement special methods for non-privileged mounts.

Yet there might be administrators who find this control too much rudimentary... What do such people do? Most likely, they set `vfs.usermount`

to 0, and use their homebrew privilege delegation policies; by all chance, they implement it via `sudo(8)`. However, `mount_fusefs` can run an arbitrary other command, so can't be called from `sudoers` without care. Nevertheless, the commandline interface is designed in a way such that it is easy to ban daemon spawning (eg., `mount_fusefs -S` will not be willing to start a daemon).

Moreover, besides the already mentioned above mentioned `vfs.usermount` `sysctl`, FreeBSD has an other access control mechanism for mounting. This comes into play if the filesystem is backed by a device. In that case, only those can mount the filesystem who have read/write access to the device to be mounted (or read access for a read only mount).

This latter mechanism can be used with Fuse, too: despite its somewhat synthetic character, Fuse is a device backed filesystem. There is though one subtle difference between Fuse and traditional device (disk) backed filesystems in this respect: with traditional filesystems, permissions of the device are used also for providing access control for the device file as such, which is a valid entity on its own and can be used for performing raw I/O on the appropriate hardware.

On the contrary, Fuse devices has no use without being mounted (the kernel is not willing to interact with a reader/writer of the device file until the VFS layer pushes messages onto it). Hence permission settings of Fuse devices are to be directly interpreted as permissions for mounting Fuse filesystems. So this is the tool by which a fine-grained control on mounting Fuse filesystems can be set up.

By default, Fuse devices can be used by members of the operator group (that's used for controlling access to, eg., usb devices). One can set permissions of fuse devives directly, by `chmod`, or generally, via `devfs(8)` rules.

Let's elaborate a bit more on this "naturally useable" BSD mount access control. This also makes Fuse more exposed to attacks under FreeBSD than it is under Linux: in Linux, non privileged mountability of Fuse based filesystems don't open up further privileged tasks. In FreeBSD, mounting and unmounting will be available more generally if the respective permissive move ("`sysctl vfs.usermount=1`") has been done. (With the help of `sudo`, one can setup an access control scheme which is similar to that of Linux, yet we are to give full support for the system provided facilities.)

As we said, device permissions can fall into the role of mount permissions, thus there are limits to the freedom provided by `vfs.usermount=1`, but this happens only with device based filesystems. The null filesystem (providing functionality similar to "bind mounting" in Linux) is one example for a deviceless filesystem. A user can create a deadlock by null mounting a directory of a Fuse filesystem over another directory, if the Fuse filesystem requires this other directory during its operation. And users can freely unmount their filesystems, including forced unmounts, which can easily lead to panics. (Note that while Linux tends to stay on the safe side and refuse

forced unmounts too, if the filesystem is busy, FreeBSD tends to go forward and perform the forced unmount and occasionally panic.) So in FreeBSD, we will have to cope with these, too, if we want to claim that mounting of untrusted daemons is safe.

Careful code review slowly leads us toward a state in which this claim can be maintained. To add, Linux Fuse doesn't seem to rely on its more protected situation: Linux Fuse was immune to those crashing schemes that were used to be possible to summon by non-privileged users in FreeBSD (in Linux, these were attempted as root, of course).

### 3.2.3 dealing with the “allow other” misery

This is related to security, too, but deserves an dedicated subchapter.

The problem is that the Fuse daemon sees I/O activity of the users of the filesystem driven by her. As usually no privileges are needed for using Fuse, we can't guarantee that the daemon belongs to a trusted user account. Moreover, mounts are transparent for normal filesystem related activities, so the user of a filesystem might be unaware of the fact that a given path belongs to a Fuse mount.

How is this handled in Linux? By default a user is denied from usage of a Fuse mount unless the daemon is “more privileged” than the user (that is, the daemon's credentials allow her trace the user's processes).

There is a mount option, `allow_other`, which removes this limitation. Of course, if anyone could use this option, that would pretty much defeat the very purpose of its existence. So by default, only root can use this. However, the final decision is made by the `setuid` dispatcher; and his decision is based upon settings in the respective config file `/etc/fuse.conf`.

The problem with this approach is that it's pretty hard for the root to make sure that no user of the system would mind an “everyone can `allow_other`” policy.

And in FreeBSD, we couldn't even follow this approach, as we don't have a `setuid` dispatcher for deciding about the fate of `allow_other` attempts.

So, what do we do about it in FreeBSD?

The basic setup is the same as in Linux: there is an `allow_other` mount option, useable only by root. But we don't make exceptions: `allow_other` can be used only by root, period.

Yet we have our own ways of being not too draconian. We have an explicit global unique userspace identifier of daemons in work.

This allows the introduction of shared daemons. When the first (primary) mount of a daemon has been completed, other users can do secondary mounts of the same daemon. A secondary mount works like a symlink – it forwards all requests to the primary mount. So it is a very lightweight mechanism.

And what's most important, doing a secondary mount can be viewed as signing an *agreement of traceability*. The secondary mounter can be expected to have the necessary knowledge about the primary mount to which she or he joins. Hence while she or he possess the secondary mount, s/he will be allowed to use the filesystem – either via its primary or secondary mounts.

### 3.2.4 anything else

Not specific to Fuse, but it's a great revelation that one doesn't need to fight with the dreaded `/etc/mstab` file under FreeBSD. Even under Linux, traditional file system authors don't get involved with such activities, but due to its alternative mounting mechanisms, Linux Fuse does have to take special care about `mtab` maintenance.

## 3.3 [vi]node operations

The design of the Fuse “rpc” system shows the traits of the file centric nature of the Linux VFS. In Linux, upon each open system call, a `FUSE_OPEN` message is sent to the daemon, who then performs whatever opening means to her, and sends back a file handle identifier to the kernel, which then gets attached to the file structure open is done for. In the sequel, when this file structure is used for I/O, the kernel will use the file handle identifier as a reference by which the daemon can find the file's userspace counterpart. So, not only inodes and the daemon's file (node) structures are kept in sync, but entities representing open files, too. (Though this is somewhat relaxed by the fact that the daemon can use the same file handle to serve I/O request for multiple (struct) files.)

Keeping `vnodes` and daemon's file (node) structures in sync happens in FreeBSD, too – it's a central concept of Fuse, this could be hardly avoided. But concerning files... It's not possible to maintain such a close correspondence between in-kernel file structures and the daemon's file handles in FreeBSD. Luckily for us, this is neither necessary (even if the `rpc` system suggests having this correspondence).

This is so because fuse filehandles are not stateful, unlike file descriptors/streams/structures. That is, if we are in the userspace, and we open a file (node) three times, then it's pretty much makes a difference which file descriptor/stream is used in a given read operation (assuming other parameters of the read call are fixed): each of them stores a file offset during its lifetime, and the read performed on one is understood to start from that given offset (which is then advanced according to the number of bytes read).

There is no offset kept with Fuse filehandles: the `FUSE_READ` operation requires an explicit offset parameter and the filehandle should serve data according to it, regardless of its exact identity. As process id and credentials are also given with a `FUSE_READ`, these parameters might better match,

but apart from that, it should be the indifferent which filehandle is used. (“Highly synthetic” filesystem daemons might opt to serve data in a filehandle specific manner, but that’s an extremity.)

So far, it’s been explained why is it possible to alter from the “strict correspondence between file structures and daemon’s filehandles” model. Now let’s see: why is this necessary?

There is the so-called “strategy” vnode method, which is used to transfer data between the “storage” (the daemon in our case) and the vmio buffers; this is the engine of buffered I/O in BSD. It takes only two parameters: the vnode we operate on, and the buffer object we read into from or write to the storage (to read or to write: this info is kept with the buffer). With Fuse, what are we to say to the daemon, when the strategy is invoked? We need a “key”, a suitable filehandle identifier to perform the I/O request – where to get one?

The situation is easy when reading or writing regular files: these operations can easily be arranged in a way that they will be file aware. When they arrive to the point where the strategy needs to be called, they simply don’t call the “official” strategy... but an internal version, which happens to take a filehandle parameter.

On the contrary, the readdir operation is not (struct) file aware (file awareness can be forced by a dirty hack, but we better refrain from going that way). Even “worse”, when doing an internal mmap, which happens eg. when executing a file, there is no file structure involved at all, and strategy is invoked by the vmio system, with no chance to smuggle in a file parameter.

In these cases, the `FUSE_OPEN` message is sent by the strategy itself, and thus it gets its vehicle for the I/O.

But what to do with the filehandle when the strategy has done its job?

Releasing it immediately (that is, strategy releases it before return) is pretty unefficient: the file should be re-opened at each turn of a lengthy read-in.

Just simply forgetting about it and polluting the daemon with worn-out filehandles is neither a good idea. Some kind of resource management should be used for these “unbound” filehandles.

As the first step of that, a list of filehandles is maintained by each vnode, and when strategy needs one, first looks for a suitable one in the list of existing ones, and asks for a new one only if it finds none. This still doesn’t seem to considerably lower the frequency of requests for new filehandles: the criterion for “suitable” is to have the same credentials, mode and pid recorded with the filehandle, thus new processes need to get new filehandles. And filehandles are still kept around ad infinitum. Some kind of garbage collection is needed. The question then: what event should trigger gc and on what thread should gc run?

There is a neat built-in gc mechanism: it’s invoked when vnodes become unused. The intended effect of this method is disassociating the vnode from

its file (node), and putting it back to the pool of free vnodes. We don't do that, as then we would lose the number of lookups (which is needed for Fuse to operate correctly). Yet it's a pretty fine time to gc unbound filehandles.

But if a file is kept open for a longer while, that will block this mechanism: the respective vnode remains in use during this time. Then the filehandles created by subsequent internal (fileless) opens will persist. (Imagine that one opens `/mnt/fuse/bin/ls` as a regular file, and keeps it open. In this case each execution of this file will create a new unbound filehandle which won't go away upon the termination of the process.)

So one more gc entry point is needed. Using the open handler for this purpose is good enough. This will stop the proliferation of unbound filehandles in the above scenario, yet won't occur with an unbearable frequency.

### 3.4 Syncing

As far as I'm concerned, both the Linux and FreeBSD modules pass over written data to the daemon immediately, so all writes are synchronous in the traditional (vmio buffers vs. storage) sense.

Yet it doesn't mean that syncing type operations wouldn't make sense with Fuse: there might be a second layer of cached data, maintained by the daemon in the userspace (who might have its own background storage, too). So these operations do make sense, but they should be passed on to the daemon. The Fuse rpc kit does have those operations which are devoted to these purposes: `FUSE_FSYNC` and `FUSE_FLUSH`, and they correspond to the respective Linux syscall handlers. According to Linux semantics, their functionality differs in the way of relating to file data and meta data (I never remember exactly how).

In FreeBSD, there is just one syscall of this type, `fsync`. It is both for synchronizing file data and meta data. For both type of data, two kind of operation mode is available: the one where the caller waits for completing the operation, and the one where operation is backgrounded. The actual combination of these modes is determined by the mount flags of the filesystem.

So, to summarize, both OS' syncing methods can relate to file data and meta data in various subtle ways, but in rather different ways.

Making up our mind in respect of flush is easy: we don't have such a syscall under FreeBSD, hence we just never use `FUSE_FLUSH`. An alternative is merging the use of the library's flush and `fsync` functionality in FreeBSD Fuse's `fsync` handling. This could be considered upon seeing a Fuse filesystem which implements the flush and `fsync` userspace callbacks in an essentially different (yet BSD [POSIX] compatible) way, but this moment of enlightenment is yet to come.

When trying to implement `fsync` for Fuse, once again we bump into the basic difference: Linux `fsync` (flush) is file based, FreeBSD `fsync` is vnode

based. Here I can imagine that file basedness has a significance to the userspace: eg., it is possible that sshfs runs different sftp connection threads for different filehandles, and syncing the data stuffed into one connection doesn't imply syncing the data of the other (I don't know whether it goes this way or not actually, but it's enough to see that this is a realistic scenario).

So what we do during the fsync vnode op is that we walk over the list of filehandles and send a FUSE\_FSYNC for each. Unlike Linux Fuse (and regardless of the mount flags) we don't wait for the answer: sending the FUSE\_FSYNC messages and waiting for the answer one after the other would be too much pain (and we can't [yet] send/wait for many messages once, in a batch).

### 3.5 Messaging

Here I give a brief comparison of the ways of implementing messaging between kernel and userspace in Linux and in FreeBSD.

Before anything else: I don't claim superiority of either solutions over the other. I implemented my solution from scratch, without understanding the respective parts of the Linux code, and without having a clear vision how this will be used by the VFS. (This latter implies that I tried to make the design as general as possible. On one hand, this is good; on the other hand, it means it doesn't contain any Fuse-specific tuning.)

Now, *post festam* I took the effort of peeking at Linux Fuse's messaging code, and I feel able to make this comparison. Nothing is carved into stone, I might make up my mind and bend my code closer to that of Linux Fuse. Or I might make it even more different.

Terminology: *up* will mean *from kernel to userspace*, *down* will mean... you can guess. ("In" and "out" are too relativistic to my taste.) The basic vehicle of messaging is called a *request* in Linux, a *ticket* in FreeBSD.

The basic mode of operation is similar.

- There is a pool of requests/tickets.
- Syscall handler wants to get data from daemon. Takes a request/ticket from the pool, fills in its fields, and inserts into upgoing queue. If buffered I/O is being done, the backing pages/buffers are attached to the request/ticket, too. Handler alerts device's read method and falls asleep, waiting for answer.
- The device's read method pushes up the message to the daemon.
- The daemon does whatever she should do with it, and sends back an answer. The device's write method grabs the answer and finds out its requester and wakes that up. If buffered read is being done, appropriate parts of the answer are handled differently, and copied into attached page/buffer.

- Syscall handler woken up, processes answer, drops requests/ticket, returns.

Differences:

- In Linux, one Fuse mount works with a fixed number of preallocated requests (with some exceptions, when new ones are created), in FreeBSD, tickets are created on demand.
- Unlike Linux, messaging facilities are unconditionally available in FreeBSD. That is, there are no forced delays depending on the intensity of filesystem usage. (This is not because of a *forced-delays-are-evil* policy; it's rather due to a *better-to-do-nothing-than-to-do-something-without-understanding-it* standpoint.)
- In Linux, the buffers hosting the fields of a request are allocated on the stack (that is, these fields are pointers to structures held in variables of syscall handlers), in FreeBSD, they are allocated dynamically (they are not freed when the ticket gets dropped, they are kept, reused, and reallocated on demand).
- In Linux, the unique field of the request is filled with a really unique value upon being taken out of the pool (ie., number of take-out). In FreeBSD, unique values are owned by the ticket itself (it's not changed during the ticket's lifetime), so unique values give information about the number of messaging sessions going in parallel (there is a secondary field for each ticket which stores the number of take-outs that ticket went through, but that's rarely used).
- Messaging API: in Linux, the fields of requests are Fuse specific (eg., there are fields named `inode`, and `inode2`, as file operations take one or two inode). This means that syscall handlers usually can fill these fields in a straightforward way (`req->inode = inode;`).

In FreeBSD, there are just raw message and answer buffers attached to a given ticket. Syscall handlers use variables of pointers of the required structs, and frontend methods for tickets set them to an appropriate value (to the appropriate point in the ticket's appropriate buffer). In some of the more complex cases this means a bit of manual pointer arithmetic; for those of the complex patterns which occur repeatedly (`mknod/creat/link`), further, specific frontend methods are used (to note, in Linux, too). In general, I didn't feel that this approach yields too much tedious repetition when setting up a ticket.

Interrupt handling: in Linux, when a syscall is interrupted, the corresponding request is "backgrounded". It's put into another queue, and when

the daemon (unaware of the interrupt) sends its reply, then its get dropped silently.

And by-and-large, the same happens in FreeBSD – just as the special case of a more general mechanism.

Tickets have a callback field, which can hold an arbitrary function (of the given type), or can be NULL. When the device write method finds the ticket to which a given answer should be passed, then invokes the callback on the incoming data (so that’s what “passing” means), provided it’s not NULL. If the callback is not NULL, then the device write method expects the handler doing the necessary resource management by the handler; but if it’s NULL, the device write method takes up this role and does what it can do – drops the ticket, etc.

In most cases we use the so-called standard one, which does what’s described above: fetch the answer and wake up syscall handler, but we also use NULL and custom cleanup functions for backgrounded requests.

NULL is used when we don’t want to wait for the answer (in case of doing a `RELEASE`, and in FreeBSD, for `FSYNC` too), and to handle interruption. If the syscall is interrupted (that is, if it returns from sleep with an error), then it locks the answer queue and replaces the callback with NULL, and thus the device write routine will aptly discard the answer. Well, there is no guarantee of a race win: it’s possible that the device write routine has already taken out the ticket from the answer queue. In that case, it will be passed to the standard handler. That’s not a problem, we can make him notice the interruption, and then he will drop the answer rather than waking up anyone. <sup>1</sup> The only difference is I/O: the standard handler copies in data nevertheless, while this is skipped if the device write routine finds a NULL callback.

### 3.6 Miscellaneous

Now you can ask: what do I think, which VFS design is the better? Vnode centric BSD or file centric Linux?

Frankly: I don’t know. While it sounds reasonable to not to keep something always on the surface if it’s rarely used directly, I have never tried to get something useful out of the Linux VFS, so I don’t know how does it feel to make a filesystem under Linux.

What I do have an opinion about is the FreeBSD VFS.

Concerning the, so to say, “OO design” it features, the hierarchy of objects and their method kit: I can say it’s a carefully crafted masterpiece. There are subsystems which remain completely hidden by default, but they are there for your disposal if you need ever need to tweak it. This yields an interface which is both high-level and highly flexible.

---

<sup>1</sup>Daemons are treated as female, for the sake of correctness. Let the standard handler be male.

On the other hand, when we come to implement the functionality required by a given method (syscall handler), we can see that having a broad range of supported filesystems give Linux an edge. In FreeBSD, one has the feeling that the way syscall handlers are utilized by the OS has too deep ties with UFS, and that when NFS entered the scene, it was bastardized and hacked on until it worked, but the need of doing low-level legwork and re-implement general functionality again and again by each users of the API has not been thoroughly and conceptionally eliminated.

Some entries from my factbook:

- In Linux, you have a nifty `generic_file_read()` function which you can just plug in as the read handler, and which does data transfer between user space and the buffer cache in a generic way. In BSD you have to implement read completely by yourself. (For some VFS functionality there are useable defaults in BSD, too: these include syncing, getting/putting pages from/to the storage, and advisory file locking.)
- In BSD, you have to take care about many small details by yourself, eg.: *“shouldn’t we bail out here because we are mounted read only?”*; check whether a directory is tried to be moved into a subdirectory of itself when doing a rename; check whether vnodes are from the same filesystem when creating hard links, and so on. Sometimes it’s trivial to do these (just shouldn’t be forgotten about), sometimes not so much...
- The BSD codebase encourages “code reuse by means of copy and paste”, which is not a particularly good thing. Just run through UFS code, and grep all the other fs code for comments seen there... This is a sign of not properly abstracted generic functionality.
- In BSD, some of the arguments of the `readdir` handler are there only for the sake of NFS: you have to process them appropriately if you want your filesystem to be NFS exportable. You can ignore them if that’s not a concern.