CS 162 HW 4: kvfs

April 10, 2015

Contents

1 Overview

Your task in this homework is to implement a simple file system. The file system you will implement will not be general purpose like many of the file systems you have studied in class (ext2, fat32, ntfs, etc..). Instead this file system will "optimize" for lots of small files (<1KB of data). Further more we will be using the `FUSE` library to build this file system in user mode, saving you from painful kernel debugging.

This assignment will be due **11:59 pm PDT 04/27/2015**

2 Setup

```
$ sudo apt-get install autoconf
$ cd code/personal
$ git checkout master
$ git pull staff master
$ cd hw4
```

You will find a simple skeleton in `kvfs.c` and `kvfs.h`. `kvfs.c` is where you will implement the bulk of our file system operations. Currently you should be able to mount the file system and enter the folder by running:

```
$ make
$ mkdir -p /tmp/test_sandbox
$ ./kvfs /tmp/test_sandbox
$ cd /tmp/test_sandbox
$ ls
$ touch hi
$ cat hi
$ echo "dog" > hi
$ cat hi
```

All the file system operations inside the newly mounted file system turn into no-ops (until you finish this assignment).
To unmount the file system do:

```
$ fusermount -u /tmp/test_sandbox
```

3 FUSE

Before you start the assignment it is imperative that you understand FUSE. FUSE is a userspace filesystem framework. It consists of a kernel module (fuse.ko), a userspace library (libfuse) and a mount utility (fusermount). FUSE allows users to write a file system completely in user mode, but allow those user mode functions to be called through the traditional system call interface, which is the standard interface used by your familiar tools `cat`, `ls`, `touch`, etc. A file system written in usermode will not be as robust or high performance as kernel code, but it allows for extremely interesting applications and access to user data that the kernel could never have. Popular FUSE file systems include: `sshfs`, `gmailfs` and `spotifyfs`. Unfortunately we won't be implementing anything nearly as cool as those three, but this assignment will hopefully give you an appreciation for both FUSE and file systems in general.

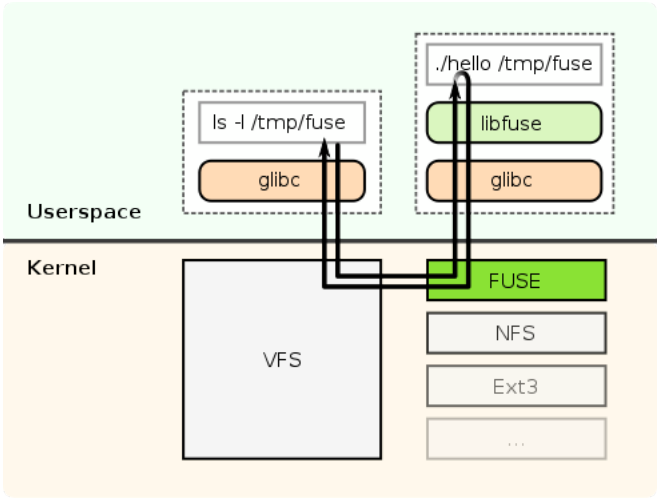

Figure 1: An illustration of how fuse allows user level file systems [4]

So lets dive into the biggest chunk inside the above diagram (`VFS`) before going into how `FUSE` works it magic.

3.1 VFS

The Virtual File System (also known as the Virtual Filesystem Switch) is the software layer in the kernel that provides the filesystem interface to userspace programs. It also provides an abstraction within the kernel which allows different filesystem implementations to coexist. VFS system calls open(2), stat(2), read(2), write(2), chmod(2) and so on are called from a process context.[3]

The virtual filesystem switch is an abstraction layer on top of a more concrete file system. The purpose of a VFS is to allow client applications to access different types of concrete file systems in a uniform way. A VFS can, for example, be used to access local and network storage devices transparently without the client application noticing the difference. It can be used to bridge the differences in Windows, Mac OS and Unix filesystems, so that applications can access files on local file systems of those types without having to know what type of file system they are accessing.

A VFS specifies an interface (or a "contract") between the kernel and a concrete file system. Therefore, it is easy to add support for new file system types to the kernel simply by fulfilling the contract. The terms

of the contract might change incompatibly from release to release, which would require that concrete file system support be recompiled, and possibly modified before recompilation, to allow it to work with a new release of the operating system; or the supplier of the operating system might make only backward-compatible changes to the contract, so that concrete file system support built for a given release of the operating system would work with future versions of the operating system. [1]

As an aside, it is also important to understand that VFS is not a special/optional kernel module that is used just for FUSE.

In fact every file system operation you ever make (in OSX and Linux), passes through the VFS switch, 2 is a pretty good illustration of the role of vfs.

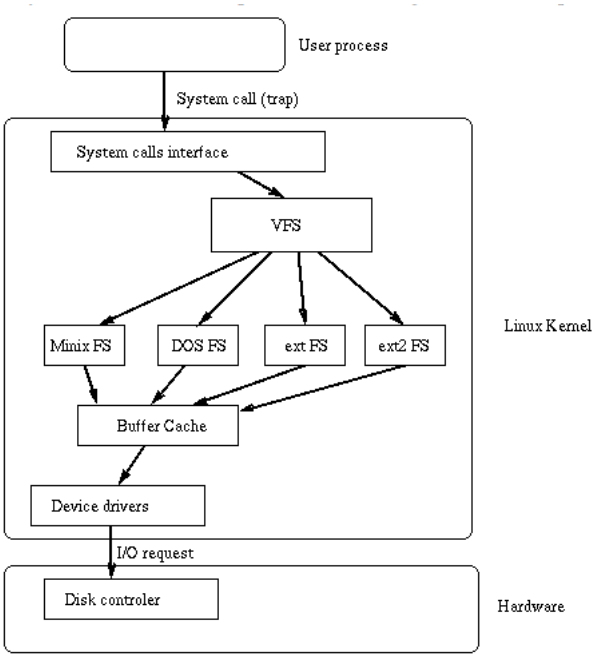

Figure 2: VFS is a very general module used for every file system [2]

3.2 fuse.ko

In the kernel fuse acts as a regular file system like the ones in Figure 2 (ext2, dos, etc..). All the system calls in kernel/file.c, kernel/inode.c, and kernel/dir.c delegate requests to the `libfuse` based user process via message passing interface.

3.3 libfuse

`libfuse` is a user level daemon, that allows the user to define a `struct fuse_operations`. Each field in the struct is function pointer that works very similarly to the well known UNIX file system operations. All methods are optional, but some are essential for a useful filesystem (e.g. getattr).

When your user mode program calls `fuse_main()` (lib/helper.c), `fuse_main()` parses the arguments passed to your user mode program, then calls `fuse_mount()` (lib/mount.c). `fuse_mount()` creates a UNIX domain socket pair, then forks and execs fusermount (util/fusermount.c) passing it one end of the socket in the `FUSE_COMMFD_ENV` environment variable. `fusermount` (util/fusermount.c) makes sure that the fuse module is loaded. fusermount then opens /dev/fuse and send the file handle over a UNIX domain socket back to `fuse_mount()`.

3.4 documentation

The fuse doxygen is an excellent resource to learn more about fuse. And will be an invaluable resource for this assignment.

4 kvfs

With background about FUSE out of the way, we can get to the details of our assignment. For this assignment you will write a very simple file system with constant file size, and a flat directory-less structure.

4.1 struct fnode

Instead of storing the metadata and data for a file separately, kvfs will use a `struct fnode` to store both the name of the file and all the data for the file. The data for the file will be a `DATA_SIZE` long `char` array, stored in the field `data`. The name of the file will be a `NAME_SIZE` long `char` array, store in the field `name`. There will also be an integer flag `magic` to check for valid/corrupt `fnodes`. Each file in `kvfs` will be represented by exactly one `struct fnode`.

4.2 struct kvfs

The file system itself will be represented by `struct kvfs`. This struct will have 2 fields, a size field denoting the number of active `fnodes` in your file system, and a size zero array of `struct fnode`.

4.3 .superblock

Your file system will delegate all reads and writes to a file `.superblock` that exists in the same directory as the mountpoint [1] . The file `.superblock` will act as a "disk image" of sorts, and will be of constant size as defined by the macro `SUPERBLOCK_SIZE`. If it doesn't exist, this file will be automatically generated and inflated to `SUPERBLOCK_SIZE` on mount. The `.superblock` file must persist between mounts, and copying this file to another location will effectively clone the filesystem to that location. One copy of the file `.superblock` will be a shared disk image for every mounted `kvfs` instance in the directory of that `.superblock` file. Moving the `.superblock` file should have the appropriate effect to `kvfs` instances mounted in the destination directory.

[1] A mount point is a directory (typically an empty one) in the currently accessible filesystem on which an additional filesystem is mounted (i.e., logically attached).

4.4 tasks

For this assignment the following shell commands must be appropriately functional inside your mounted file system.

1. `touch`
2. `cat`
3. `>` (output redirection to a file)
4. `<` (input redirection from a file)
5. `mv`
6. `rm`
7. `cp`

Part of this assignment will be to figure out which file systems calls are necessary for the above shell commands. The `strace` utility and the fuse mount flags `-d,-s,-f` will be extremely useful (note that `>` and `<` are not programs, but shell builtins). The `getattr` function is necessary for any sort of file system functionality, so we have implemented some of it for you. You may need to edit it further for proper filesystem functionality. We also gave you `kvfs_truncate` (which is nonobviously necessary for file creation).

5 Testing

In order to discourage the bad practice of ADD (autograder driven development). We gave you a simple testing framework. The framework currently has tests for the provided mount and superblock inflating code. Hopefully you can model these tests to write tests for other file system functions.

6 Mounting

One final point about mounting, mounting will only work if you mount the filesystem to a non shared folder. So if you have shared your homework repo to your host operating system, then the mounting code will not work. Instead mount to a directory in the vm itself (like inside `/tmp` or `/home`).

7 Autograder & Submission

Push your code to the autograder branch `ag/hw4` on github to test your code

```
git add -u .
git commit
git push personal master
git checkout -b ag/hw4
git push personal ag/hw4
```

Push the final code release to the branch `release/hw4` on github to submit

```
git add -u
git commit
git push personal master
git checkout -b release/hw4
git push personal release/hw4
```

References

[1] Virtual file system. http://en.wikipedia.org/wiki/Virtual_file_system. Accessed: 2015-04-08.

[2] Remy Card. Design and implementation of the second extended filesystem. http://e2fsprogs.sourceforge.net/ext2intro.html. Accessed: 2015-04-08.

[3] Richard Gooch. Overview of the linux virtual file system. https://www.kernel.org/doc/Documentation/filesystems/vfs.txt, 1999. Accessed: 2015-04-08.

[4] Terje Oseberg. Fuse api documentation. http://fuse.sourceforge.net/doxygen/. Accessed: 2015-04-08.