Building a Read-only Virtual File System with FUSE

A Brief Tutorial

Purpose and Overview

A virtual file system, in this context, is a file system that presents files and directories to the user that don't exist on disk. This tutorial walks through building a read-only virtual file system using FUSE.

What is FUSE?

FUSE roughly stands for "File system in userspace" but it's not a very good acronym. FUSE is a combination of a kernel module and a user-space library; the kernel module interfaces with the kernel and sends requests to the user-space program. This allows the behavior of a file system driver to live in user space.

FUSE is supported on Linux, macOS, BSD and is widely used for a variety of purposes. Common examples are encryption - where the files on disk are encrypted and a FUSE-based file system is used to decrypt them at runtime; network file systems like sshfs; and even the NTFS driver on Linux. If you mount an NTFS file system in Linux you'll see its type is listed as fuseblk, which is a FUSE driver for NTFS.

What We'll Build

In this tutorial we'll build a read-only virtual file system that contains a single file named HelloWorld.txt. The contents of that file will be "Hello, world!". That isn't fancy but it is an easy way to work through the steps necessary to build a virtual file system!

Our file system driver will be built in C.

Assumptions

This tutorial assumes you have the FUSE kernel module installed. It also assumes you know a little bit of C but you probably don't really have to. ;)

Building a Virtual File System Driver with FUSE - Step by Step

Step 1 : Registering the Driver with FUSE

When a virtual file system driver is loaded it needs to do a few things. The main thing is to call the `fuse_main` function to register the virtual file system with FUSE (and the kernel). An important part of that call is providing FUSE with a struct of function pointers that provide the actual functionality of the file system driver. The FUSE library will call these functions to perform the actions necessary to make the file system work.

FUSE also accepts a variety of options that can be specified on the command line (or defined in your code, if you'd like). It also provides a convenient macro for converting command line options to the `fuse_args` struct that must be passed to `fuse_main`.

In summary: to register a virtual file system with FUSE our code needs to do three things:

1. Define a `fuse_operations` struct
2. Define a `fuse_args` struct, probably use the FUSE_ARGS_INIT macro
3. Call `fuse_main`

Here is all the code necessary to register a (completely useless!) virtual file system with FUSE:

```
#define FUSE_USE_VERSION 31

#include <fuse.h>
#include <string.h>
#include <errno.h>
#include <stdio.h>

static const struct fuse_operations helloWorld_oper = {
};

int main(int argc, char *argv[])
{
    int ret;
    struct fuse_args args = FUSE_ARGS_INIT(argc, argv);

    ret = fuse_main(args.argc, args.argv, &helloWorld_oper, NULL);
    fuse_opt_free_args(&args);
    return ret;
}
```

Put this code into a file called helloWorld.c with your favorite text editor.

Compiling and Executing

To compile our virtual file system driver (at least on Linux) run the following command:

```
gcc -Wall helloWorld.c `pkg-config fuse3 --cflags --libs` -o helloWorld
```

Then to start the virtual file system driver run:

```
./helloWorld mnt
```

This assumes that a directory named `mnt` exists in the same directory as the helloWorld.c file and the helloWorld program that we just generated with gcc. If it doesn't you should create it now. After running the helloWorld program if you try to list that directory you'll get an error like this:

```
$ ls -al mnt
ls: cannot access 'mnt': Function not implemented
```

Which is 100% accurate, because we haven't implemented *any* functions!

Before continuing we need to unmount the (completely useless!) virtual file system we just mounted. To do so, simply execute:

```
fusermount -u mnt
```

Step 2 : The readdir function

The `readdir` function is the conceptual starting point for a file system. It is called when the operating system needs to get the contents of a directory. It needs to return both the files and directories in the given directory.

When FUSE calls our readdir function it will provide a few important arguments:

a. `const char *path` - the name of the directory in question
b. `fuse_fill_dir_t filler` - a function to use to add item
c. `void *buf` - a pointer to a structure that will hold the contents of the files

Below is a simple readdir function for our virtual file system. This does a few things:

1. Checks to make sure that the path the operating system is asking about is the only one we have - the root directory (i.e. "/").
2. Adds the entries for the current directory and the previous directory - the "." and ".." that you're used to seeing in every directory.
3. Adds an entry for our "HelloWorld.txt" file.

```
#define HELLO_WORLD_FILENAME "HelloWorld.txt"

static int helloWorld_readdir(const char *path, void *buf,
      fuse_fill_dir_t filler, off_t offset, struct fuse_file_info *fi,
      enum fuse_readdir_flags flags)
{
    if (strcmp(path, "/") != 0)
        return -ENOENT;

    filler(buf, ".", NULL, 0, 0);
    filler(buf, "..", NULL, 0, 0);
    filler(buf, HELLO_WORLD_FILENAME, NULL, 0, 0);

    return 0;
}
```

You can add this code to the top of your helloWorld.c file. Make sure it compiles but don't run anything yet - we haven't actually hooked anything new up yet.

Step 3 : The readdir function

The `getattr` function is called when the OS needs the attributes of a directory or file. It is responsible for telling the OS things such as the object's type, permissions, size, etc. This function gets called a LOT.

When FUSE calls our getattr function it will provide three arguments:

a. `const char *path` - the name of the file or directory in question
b. `struct stat *stbuf` - a stat struct in which to put the information about the file or directory
c. `struct fuse_file_info *fi` - you can ignore this for now.

Below is a simple getattr function for our virtual file system. This does two main things:

1. If the path the operating system is asking about is the root directory (i.e. "/") it tells FUSE that:
    a. it is a directory (S_IFDIR)
    b. it's permissions are 755, and
    c. it has two hard links
2. If the path the operating system is asking about is our (completely fake!) HelloWorld.txt file it tells FUSE that:
    a. It is a regular file (S_IFREG)
    b. Its permissions are 444
    c. It has one hard link

Add this code to your helloWorld.c file below the `readdir` function we added earlier. Make sure it compiles but don't run anything yet - we (still) haven't actually wired anything together yet.

```
#define HELLO_WORLD_CONTENTS "Hello, world!\n"

static int helloWorld_getattr(const char *path, struct stat *stbuf,
             struct fuse_file_info *fi)
{
    int res = 0;

    memset(stbuf, 0, sizeof(struct stat));
    if (strcmp(path, "/") == 0) {
        stbuf->st_mode = S_IFDIR | 0755;
        stbuf->st_nlink = 2;
    } else if (strcmp(path+1, HELLO_WORLD_FILENAME) == 0) {
        stbuf->st_mode = S_IFREG | 0444;
        stbuf->st_nlink = 1;
        stbuf->st_size = strlen(HELLO_WORLD_CONTENTS);
    } else
        res = -ENOENT;

    return res;
}
```

Step 3.5 : Hooking up the readdir and getattr functions

Now that we have defined `readdir` and `getattr` function we need to tell FUSE they exist so it can call them. To do that we need to add them to the `fuse_operations` struct of function pointers that is passed to `fuse_main`. Change the `helloWord_oper` struct we defined earlier to look like this:

```
static const struct fuse_operations helloWorld_oper = {
    .readdir    = helloWorld_readdir,
    .getattr    = helloWorld_getattr,
};
```

Now compile your updated driver with `gcc` and mount your virtual file system by running the `helloWorld` command. If you list the mnt directory, you should now see something like this:

```
$ ls -al mnt
total 4
drwxr-xr-x 2 root root    0 Dec 31  1969 .
```

```
drwxrwxr-x 3 tom  tom  4096 Nov 30 20:56 ..
-r--r--r-- 1 root root   14 Dec 31  1969 HelloWorld.txt
```

Congratulations! You've created a virtual file system containing a file! Don't try to read that file yet though. And don't forget to unmount your virtual file system (with `fusermount -u mnt`) before continuing!

Step 4 : The open function

So we have a virtual file system with a directory and a file. Just having a file isn't much good though - we need to be able to see the contents of that file! In order to do that we need to implement an `open` function.

When FUSE calls our open function it will provide two arguments:

a. `const char *path` - the name of the file to be opened
b. `struct fuse_file_info *fi` - you can ignore this for now.

Below is a simple open function for our virtual file system. This does two simple things:

1. Make sure that the name of the file being opened is our HelloWorld.txt file (because if it's not, there's nothing we can do… it's the only file in our file system!)
2. Make sure the file is being opened read-only.

```
static int helloWorld_open(const char *path, struct fuse_file_info *fi)
{
    if (strcmp(path+1, HELLO_WORLD_FILENAME) != 0)
        return -ENOENT;

    if ((fi->flags & O_ACCMODE) != O_RDONLY)
        return -EACCES;

    return 0;
}
```

Add this code to your helloWorld.c file below the `getattr` function we added earlier. Make sure it compiles but don't run anything yet. (Can you guess why?)

Step 5 : The read function

The last piece of the puzzle for our virtual file system driver is the `read` function. This goes hand-in-hand with the `open` function - first you open a file, then you read from it. They are separate steps but it doesn't make much sense to do one without the other… which is why we haven't tried any more testing yet!

When FUSE calls our read function it will provide four important arguments:

a. `const char *path` - the name of the file to be opened
b. `char *buf` - the buffer into which to write the file contents
c. size_t size - how many bytes to write into `buf` (at most)
d. `off_t offset` - where within the file to start reading

Below is a simple read function for our virtual file system. This does two things:

1. Make sure that the name of the file being read is our HelloWorld.txt file.
2. Copied the requested number of bytes (at most) into the output buffer, starting from the given offset.

```
static int helloWorld_read(const char *path, char *buf, size_t size,
      off_t offset, struct fuse_file_info *fi)
{
    size_t len;

    if(strcmp(path+1, HELLO_WORLD_FILENAME) != 0)
        return -ENOENT;

    len = strlen(HELLO_WORLD_CONTENTS);
    if (offset < len) {
        if (offset + size > len)
            size = len - offset;
        memcpy(buf, HELLO_WORLD_CONTENTS + offset, size);
    } else
        size = 0;

    return size;
}
```

That's it! Our file system driver is now capable of providing the contents of our HelloWorld.txt file to programs. Add this code to your helloWorld.c file below the `open` function we added earlier. Then add references to our `open` and `read` functions into our `fuse_operations` struct, like so:

```
static const struct fuse_operations helloWorld_oper = {
    .readdir    = helloWorld_readdir,
    .getattr    = helloWorld_getattr,
    .open       = helloWorld_open,
    .read       = helloWorld_read,
};
```

Now compile and run the helloWorld program as before. Now you can view the contents of the HelloWorld.txt file with your editor of choice or simply use `cat` to view it quickly:

```
$ cat mnt/HelloWorld.txt
Hello, world!
```

Congratulations! You've created a functioning, read-only virtual file system with FUSE!

Next Steps

The virtual file system we have created isn't very useful but it is a good starting point for building a truly functional file system driver with FUSE. If you want (or need) to build a truly useful file system, here are some good next steps.

Logging

Logging is a key part of building (and using, and supporting!) a real file system driver. Logging will help you understand what your driver is doing and why, especially at the times it isn't doing what you expect.

Here's an easy way to handle logging in your file system driver. Open a file in your program's `main` function and then pass that file handle to `fuse_main` in the "private data" parameter. Then in the readdir, getattr, and other functions you can call the `fuse_get_context` function to access it. For example, add this line to your `main` function:

```
FILE * logFile = fopen("/tmp/hello.log", "a");
```

And then change your `fuse_main` call to:

```
ret = fuse_main(args.argc, args.argv, &helloWorld_oper, logFile);
```

Lastly, add the following to your helloWorld_readdir function:

```
FILE *logFile = (FILE*)(fuse_get_context()->private_data);

if (logFile != NULL)
    fprintf(logFile, "readdir: %s\n", path);
```

Now, each time the readdir function is called, an entry will be written to /tmp/hello.log. You should add diagnostic logging like this to all the functions in your virtual file system driver - you'll be glad you did!

NOTE: You may need to unmount your virtual file system before you see any lines appear in the log file.

Clean Shutdown

Of course, now that we have opened a log file, we need to close it. To do this we can add a `destroy` function. FUSE will call this function when the virtual file system is unmounted.

A simple destroy function that closes the log file we just added could look like this:

```
static void helloWorld_destroy(void *private_data)
{
    if (private_data != NULL)
    {
        fclose((FILE*)private_data);
    }
}
```

In order for FUSE to call it we need to add it to our fuse_operations struct, like so:

```
      .destroy    = helloWorld_destroy,
```

Now, when our virtual file system is unmounted the log file will be correctly closed.

Command Line Options

FUSE includes support for a number of options that can be specified on the command line when your virtual file system driver is launched. To see a list, run

```
./helloWorld -h
```

It's easy to add support for your own custom options using the `fuse_opt_parse` function as well.

Providing Real Content in Files

The one file in our sample virtual file system (HelloWorld.txt) contains static text. What if we wanted it to contain real content, for example, data generated at run-time or retrieved over the network? How would we manage multiple read calls against that data, especially if it were large?

FUSE provides a convenient solution to this problem - the real file system! The `struct fuse_file_info` that is passed to the `open` and `read` functions contains an `fh` element. This is a standard file handle that you can use with the system's open, read, and close methods. So, if we want to generate content dynamically and make it easy to retrieve when the operating system requests it, we can:

1. Create and write a file on the regular file system in our `open` method
2. Use the system's open method to open a file handle to that file

3. Store that file handle in fi->fh in our `open` method
4. Use the system's read method to read from that file in our `read` method!

For example, in our open method, if we do:

```
char pathToRealFile[MAX_PATH];

// generate/write data to pathToRealFile

fd = open(pathToRealFile, fi->flags);

// error checking here

fi->fh = fd;
```

Then in our `read` method, we can simply do:

```
int r = pread(fi->fh, buf, size, offset);
```

To retrieve the requested bytes!

Of course we'll also need changes in our destroy function to close the file handle and remove the file from the real file system.

Acknowledgements

The hello sample included with libfuse was instrumental in helping me figure out how to build a file system driver with FUSE and is the basis for the sample code provided in this tutorial.