# Computer Systems
# Project 8
# Final Project: A FUSE file system or two

## 1 User level file systems

Our goal for this assignment is to see (and maybe experiment with) a *virtual file system (VFS)* using *FUSE*—a tool that allows us to run file system code as a regular program (a.k.a. *in userspace*). When a FUSE program is run, it creates a file system at a *mount point* provided by the user, and then makes its virtual file system appear within that mount point. All operations performed within the subdirectories and files within that mount point are passed, by the kernel, onto the FUSE program to handle.

## 2 Getting started

Perform the usual steps to create the repository and get the code...

1. **Login to the server** via `ssh`.
2. **Login to GitLab** in your browser.
3. **Start a new project:** Set the *Project name* to be `sysproj-8`.
4. **Clone the repository onto the course server:**

```
$ git clone git@gitlab.amherst.edu:YOURUSERNAME/sysproj-8.git
$ cd sysproj-8
```

5. **Download the source code:**

```
$ wget -nv -i https://bit.ly/cosc-171-24s-p8
$ ls -l
```

6. **Add/commit/push the source code to the repository:**

```
$ git add *
$ git commit -m "Starting code."
$ git push
```

# 3 Exploring the code

## 3.1 Building and using `mirrorfs`

Before bothering to look at the code, we should first try using it to see how FUSE file systems behave. The code file `mirrorfs.c` is a complete FUSE program that provides a *passthrough file system*. That is, it mirrors one part of the file system—a *storage directory* and all its contents—at another part of the file system—at the *mount point*. Every file seen in the storage directory is visible at the identical (relative) location within the mount point, and vice versa.

Seeing it in action will likely be illuminating. To do so, first compile the code:

```
$ make mirrorfs
```

Once compiled, make yourself a storage directory and a mount point, and then start the FUSE file system:

```
$ mkdir stg mnt
$ ./mirrorfs ${PWD}/stg ${PWD}/mnt
```

All that should appear is a debugging message. Note that `${PWD}` is the *environment variable* that stores the *present working directory*. Specifically, it provides the *absolute pathname*: a single, complete name that begins from the root directory (the leading `/`). FUSE needs absolute pathnames, and so we provide the storage directory and mount point in that form.

You can also check that the mount was successful:

```
$ mount | grep mirrorfs | grep YOURUSERNAME
mirrorfs on /home/YOURUSERNAME/sysproj-8/mnt type
fuse.mirrorfs (rw,nosuid,nodev,user=YOURUSERNAME)
```

Now that our file system is running, let's use it a little:

```
$ cd mnt
$ printf "Some stuff\n" > foo.txt
$ cat foo.txt
Some stuff
$ cd ../stg
$ cat foo.txt
Some stuff
```

We made a small text file in the `mnt` directory, and then verified that the file's contents are there. Then we switched to `stg` and, lo and behold, the same `foo.txt` file is there, and with the same contents! Try this trick with another file in reverse: make the file in `stg`, and then switch to `mnt` and find it there.

Go back to your project directory (that is, move out of `mnt` if you are in it), and then *unmount* our virtual file system like so:

```
$ fusermount -u ${PWD}/mnt
```

You can then use the `mount` and `grep` commands again to verify that your file system is no longer listed.

## 3.2 Groking the code (at least a little)

Here you will see some semi-gnarly C code. It's not too bad, but it contains few comments, and so it's hard to follow if you don't know what you're looking at. What matters, for this assignment, is that each of the operations that a file system must support (e.g., `READ`, `WRITE`, `OPEN`.) is handled by a corresponding FUSE function (e.g., `mirror_read()`, `mirror_write()`, `mirror_open()`). You will see that most of this source is composed of these functions.

Search, specifically, for `mirror_read()` and `mirror_write()` (they are contiguous in the source code). These functions simply call on standard file functions (i.e., `read()`, `write()`, `open()`) to open the corresponding file in the storage directory, and then perform the desired function. By *passing through* these read and write operation requests, the files within the FUSE-managed mount-point appear exactly as their counterparts in the storage directory.

## 3.3 Another example: `caesarfs`

The `mirrorfs` file system is **so** transparent in its operation that it may even be confusing. To see what is happening when a FUSE file system is used, you should examine the other fully-functioning FUSE file system included in the source code: `caesarfs.c`. If you examine the `caesar_read()` and `caesar_write()` within this file, you will see that the sequences of bytes that are read and written are modified in the process.

Specifically, `caesarfs` implements a *Caesar cipher*. Specifcally, each character of the file is *shifted* **forwards** by $k$ characters to encrypt the data upon a `WRITE` operation. If $k = 3$, then each `A` becomes a `D`, each `B` becomes a `E`, each `C` becomes an `F`, and so on. Likewise, when a `READ` operation is performed, each character is shifted **backwards** by $k$ characters, thus restoring the original data.

**Try it.** Compile the code and start the file system. Note that you must provide the *shift amount* (also known for this cipher as the *encryption key*), which specifies $k$—the shift forward and backward of each character during encryption and decryption.

```
$ make caesarfs
$ USAGE: ./caesarfs <storage directory> <mount point> <caesar shift>
$ ./caesarfs ${PWD}/stg ${PWD}/mnt 3
```

Having done that, create a file in the mounted file system, and then compare the contents of that file with at what is **actually** stored in the `stg` directory:

```
$ printf "ABCabc123" > mnt/quux.txt
$ cat mnt/quux.txt ; printf "\n"
ABCabc123
$ cat stg/quux.txt ; printf "\n"
DEFdef456
```

Finally, unmount and stop the file system:

```
$ fusermount -u mnt
```

You will now see that no files appear in `mnt`, but that `quux.txt` and any other files you create will persist in `stg` in their encrypted form.

## 3.4 Debugging this kind of code

Normally, when you run your FUSE program, it will run *in the background*—that is, it returns control of the shell, allowing you to enter more commands, all while continuing to execute. However, that means that it is no longer connected to the console's input and output; any debugging output simply won't appear. Additionally, this mode of running isn't compatible with `gdb`.

In order to help with this problem, there are options that you can append to the command line when you run your FUSE program:

- `-d` will enable *debugging output*. This is a **lot** of detailed output generated by FUSE, and is probably more than you want to see.
- `-f` will force the program to stay in the *foreground*—that is, it won't give control back to the shell. Instead, the FUSE module will sit there, waiting for things to happen. In this mode, **any print statements in your code will actually appear**. This is an essential option if you want to print debugging statements of your own.
- `-s` will limit the FUSE program to running as a *single thread*. What you need to know is that this flag is really helpful for debugging. It limits the performance of your FUSE program, but that's not important during debugging.

**Running FUSE with debugging output:** If you add `printf()` debugging statements to your code, then you should run it *single threaded and in the foreground*. However, once you run it, you will not be able to type any new commands so long as the FUSE code is running. In order to avoid this problem, you need to **create a second terminal window**, allowing you to run the FUSE program from one shell, and then typing commands into the other. Open a second terminal window, and then `ssh` to the Systems server to establish a second connection.

In the first terminal, run your FUSE program:

```
$ ./caesarfs ${PWD}/stg ${PWD}/mnt -f -s
```

In that terminal, the FUSE program will sit there, in control of the window, waiting for things to happen with files in its mount point. In the **other** terminal, you can enter commands that manipulate files. While that happens, you should see your debugging output appear in the first terminal.

When you want to stop the FUSE program from running, just click on the terminal in which it is running and type `Control-C`.

**Running FUSE in `gdb`:** You can use `gdb` on your FUSE program, but again, doing so will take over an terminalwindow, requiring you to create a secondary one into which to type commands while the FUSE program runs. You can create this other terminal and then start `gdb` on your FUSE program like this:

```
$ gdb caesarfs
[gdb startup stuff]
(gdb) b caesar_read
Breakpoint 1 at 0x1a04: file caesarfs.c, line 285.
(gdb) b caesar_write
Breakpoint 2 at 0x1b2f: file caesarfs.c, line 312.
(gdb) run $PWD/stg $PWD/mnt -f -s
Starting program: /home/yourusername/current/classes/systems/assignments/8/code/caesar
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
DEBUG: Mounting /home/yourusername/current/classes/systems/assignments/8/code/stg at /
[Detaching after fork from child process 15198]
```

`gdb` will now sit there, waiting for things to happen. If you go to your other terminal and perform operations on files within the mount point that requiring reading or writing data, the operation will trigger the breakpoint in `gdb`, then giving you back the `(gdb)` prompt, and thus the ability to step through the FUSE program and examine the data.

When you are done, you can use `Control-C` to interrupt `gdb`, and then at the prompt, use the `quit` command to get `gdb` to end and return you to the shell. At the shell prompt, you should use the `fusermount` command (see above) to completely unmount the file system.

## 4 Your assignment

**There is nothing that you are required to do for this assignment.** There is nothing that you will submit. This project is posted so that you may see and play around with FUSE modules and virtual file systems.

I do suggest that you move through the examples above (`mirrorfs` and `caesarfs`). You could then make a copy of either and modify them in an effort to change them, either in small ways (e.g., change the encryption), or in larger ways (e.g., make a *versioning file system* that keeps a copy of each file, as it existed, after each *save/write* operation.)

**This assignment isn't due, because there's nothing to do!**