ECE566 Enterprise Storage Architecture Program: Intro to FUSE

For the course project, you'll have the option of developing a novel filesystem with FUSE ("Filesystem in USErspace"). This assignment will introduce you to FUSE and walk you through some programming exercises on it.

Directions:

- This assignment will be completed **individually**.
- Format:
    - **Software deliverables:** You'll be asked to write some programs. Submit tarballs (.tgz files) and a PDF to the Canvas locker for this assignment as directed.

1 Filesystems and FUSE

First, let's discuss what a filesystem is at a high level (to be covered much deeper later in the course). Your hard drive or SSD is a dumb storage device: you just tell it which block to read or write and it goes there and does that: the interface is just the following (approximately):

- `char* read_block(int blocknum)`
- `void write_block(int blocknum, char* data)`

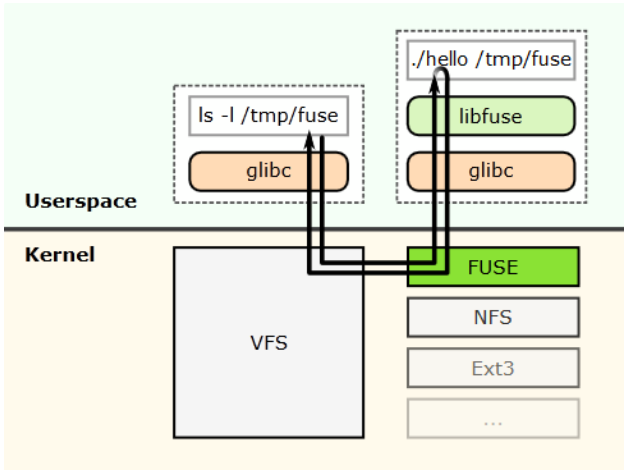
We *could* just live with that – just making a note that block 42 is my tax records and blocks 43-49 is a picture of my dog, but that gets cumbersome.

A filesystem stores information about which blocks correspond to which pieces of information (files) on the hard drive itself. It provides the hierarchical directory abstraction, the permissions abstraction, and other metadata things (timestamps, etc.). The filesystem uses the block device's `read_block`/`write_block` interface in order to provide a richer interface that includes OS primitives you're used to: `open`, `close`, `read`, `write`, `mkdir`, `delete`, etc.

The usual architecture for a modern filesystem is a module within the OS kernel. When a user program tries to do an IO call, it is passed to a handler in the OS that follows the filesystem's algorithm to perform that operation using a given block device. On modern systems, you may have multiple filesystems available ("mounted") at a time. For example, you might have your laptop's hard drive mounted (formatted on Windows as the NTFS standard) as well a USB stick (formatted in the FAT32 standard). The modern OS allows many different handlers to all have the same file IO interface – the pivot that selects among them is called the Virtual Filesystem (VFS). In this example, NTFS and FAT32 are the actual filesystems.

It is possible to write your own filesystem code as a kernel module, but writing kernel code can be troublesome and hard to debug. The Linux kernel provides an interesting facility called FUSE ("Filesystem in USErspace") that can mitigate this. In this system, kernel functions to do filesystem things are redirected back into a user program for handling – one that you will write.

This is illustrated in the figure to the right; a call to the usual `ls` command goes down to the kernel VFS, is routed back into userspace to the example hello filesystem shown, which decides how to handle the call. The answer is piped back through the kernel to the original caller. In this way, user code written in C, Python, or any other language with FUSE bindings can act as a fully-fledged filesystem. There's a performance cost for this, but for this class, that's a cost we're willing to play for the simplicity of writing user code vs. kernel code.



In this assignment, you'll go through a FUSE tutorial and write a few small example FUSE programs.

2 Get comfortable with FUSE

Let's get acquainted with FUSE using some example code. Do the following:

First, reserve an **Ubuntu Linux 22.04 VM** in the Duke Virtual Computing Manager (VCM) environment[1].

Second, update and install some important pre-reqs:

```
sudo apt update
sudo apt dist-upgrade
sudo apt install build-essential pkg-config libfuse-dev
```

Third, create a directory for this assignment (and hopefully a private git repo, but that's up to you), and within it, a subdirectory called "hellofs".

Fourth, the FUSE development package (`libfuse-dev`) comes with an example filesystem we're going to examine. Copy the "hello" filesystem into your "hellofs" directory:

```
cp /usr/share/doc/libfuse-dev/examples/hello.c .
```

[1] This is the same version of Linux deployed to your class servers, so code and build environments you set up now will be portable to your server, which may help with the course project later on.

Note that this path also includes HTML documentation for FUSE. To make it easier to browse, this very documentation has been mirrored within the course site here.

The hello.c code does include the command to compile it, but to avoid typing this long command repeatedly, create the following Makefile:

```
# Makefile for FUSE projects in C
all: hello

clean:
	rm -f hello

# note: the pkg-config automatically generates the appropriate
# compiler flags for use with fuse
hello: hello.c
	gcc -Wall $^ `pkg-config fuse --cflags --libs` -o $@
```

If you're not familiar with make and Makefiles, take a quick detour to learn the basics to understand the above. With this file, you can now compile the hello filesystem simply by typing:

```
make
```

At this point, please watch **this introductory video I've prepared**. Summary of key takeaways:

- Mountpoints are directories to which we attach the root of another filesystem, such as one provided by a FUSE project.
- You can list current mounts with `mount`, possibly filtering with `grep`.
- Can mount the hello filesystem with "`./hello mountpoint`"; will background and go silent.
- Can unmount with "`fusermount -u mountpoint`".
- Can mount the hello filesystem with "`./hello -d mountpoint`" for interactive debug mode (recommended during development).
- Walked the hello code, including the FUSE `getattr` operation and its relationship to the UNIX `stat` system call (including fields of `struct stat`), as well as Linux error numbers (errno).

If you want an alternative introduction to FUSE, see the tutorial linked from section 0 of this document.

3 Creating the `hellotime` filesystem

Create a variant of the hello filesystem called `hellotime.c`:

```
cp hello.c hellotime.c
```

Further, edit the `Makefile` so it builds both (i.e., add "`hellotime`" to the "`all`" target, and add a new recipe to build "`hellotime`" from "`hellotime.c`" by copying the existing "`hello`" recipe).

You must modify `hellotime.c` to add the following features:

1. A new directory "`time`" is introduced under the root directory.
2. A new file "`now.txt`" is present within the "`time`" directory.
3. The "`now.txt`" will, when read, contain a 20-byte timestamp showing the current local time.
4. The file size of "`now.txt`" will be 20.
5. The modification time (`mtime`) of "`now.txt`" will be the current time when checked via the "`ls -l`" or "`stat`" commands.
6. The UID owner of "`now.txt`" will be the current user checking the file.

Tips:

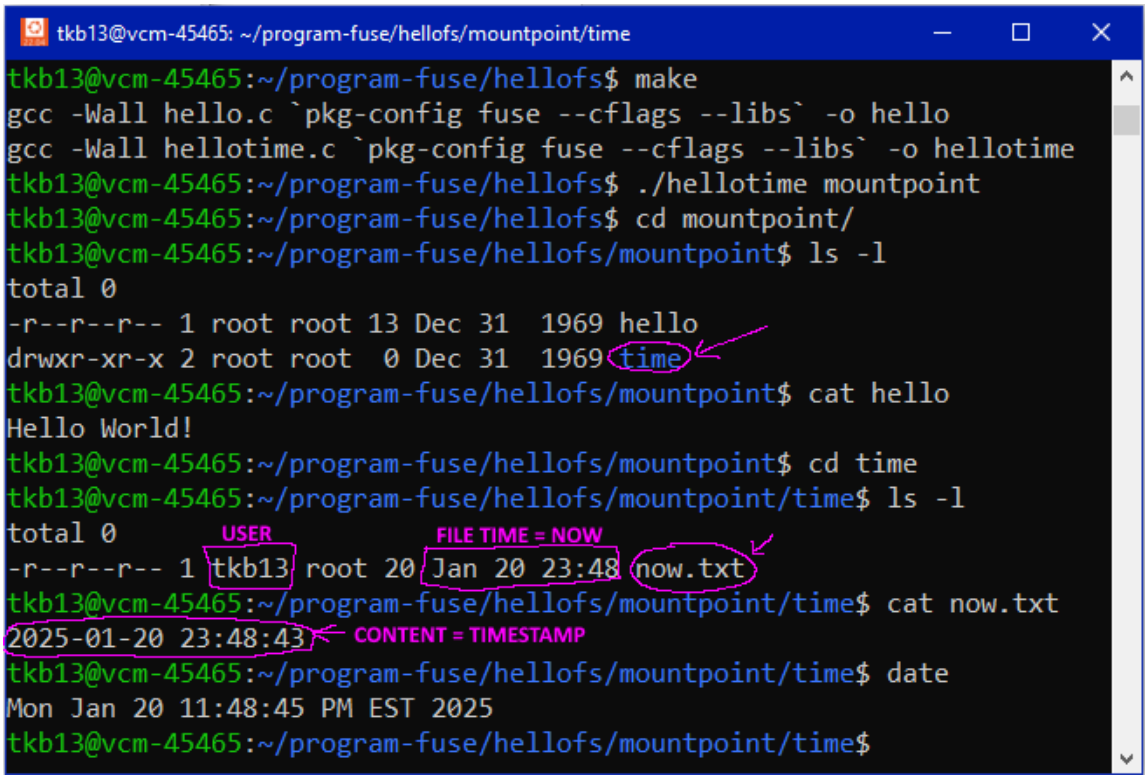
- Here is code to generate the exact timestamp we're looking to see inside "now.txt":

```
const int TIMESTAMP_LEN = 20;
/**
 * Write the current local time to the given character buffer,
 * which must be at least TIMESTAMP_LEN+1 bytes long.
 * The timestamp itself (excluding null terminator) will be
 * exactly TIMESTAMP_LEN bytes long.
 */
void timestamp(char *output) {
    // Get the current time
    time_t raw_time;
    struct tm *time_info;
    time(&raw_time);
    time_info = localtime(&raw_time);

    // Format the timestamp
    strftime(output, TIMESTAMP_LEN+1,
       "%Y-%m-%d %H:%M:%S\n", time_info);
}
```

- The stat `mtime` field uses "epoch time" (seconds since Jan 1, 1970 UTC), and the current epoch time is found by the `time()` function.
- The UID of the current user making a request can be found via the `fuse_get_context()` function.

The screenshot below shows correct operation of `hellotime` and demonstrates all its features:



Submit a file called **`hellotime.tgz`** with your code. It should be compiled with "`make`". The executable produced should be called `hellotime`, and it should have the same calling syntax as `hello`.

4 Creating the `twofs` filesystem

Start a new FUSE program from scratch called **`twofs`**. This program will be much more like a traditional filesystem: it will take a block device (or a file which we'll treat as a block device) as its first argument, then a mountpoint. The program will translate IO calls to the filesystem to read/write calls on this block device.

However, it will do so in a really, really simple way.

The filesystem always contains just two files, and they're always called `file1` and `file2`. The metadata for these files is static: owned by root, created on Jan 1 1970 at midnight UTC, read/write permissions for everyone (0666).

`file1` represents the first half of the block device; `file2` represents the second half. More formally, if the size of the block device is N, `file1` represents bytes [0,N/2) and `file2` represents bytes [N/2,N), where the division shown is integer division. The size metadata for the files should reflect this.

This means that your filesystem will have no metadata, which will make the job much easier.

Attempts to read and write to `file1` and `file2` should work accordingly, reading from or updating the block device as appropriate. Attempts to write past the end of the files should fail (though a read or write that is only partially out of bounds should succeed but be cut off). Attempts to truncate (set file size to zero) or otherwise resize the file should be silently ignored. Attempts to do anything involving other files or any directories should fail (with error code ENOENT, a UNIX error code).

The program should have the calling syntax as follows:

```
twofs <blockdevice> <mountpoint>
```

Below is an example interaction with a twofs filesystem. In it, we create a filesystem image to act as our block device (a 2 kB image file), create a mountpoint directory, mount a twofs filesystem, and interact with the files we see. Typed commands and prompts are shown in **blue bold**.

```
$ dd if=/dev/zero of=filesystem_image bs=1k count=2
2+0 records in
2+0 records out
2048 bytes (2.0 kB, 2.0 KiB) copied, 0.002556 s, 801 kB/s
$ mkdir mountpoint
$ ./twofs filesystem_image mountpoint
$ cd mountpoint
$ ls -l
total 8
-rw-rw-rw- 1 root root 1024 Jan 01 1970 file1
-rw-rw-rw- 1 root root 1024 Jan 01 1970 file2
$ echo hi > file1
$ cat file1
hi
$ hexdump -C file1
00000000  68 69 0a 00 00 00 00 00  00 00 00 00 00 00 00 00  |hi..............|
00000010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000400
$ echo > otherfile
-bash: otherfile: No such file or directory
$ ls -l
total 8
-rw-rw-rw- 1 root root 1024 Jan 01 1970 file1
-rw-rw-rw- 1 root root 1024 Jan 01 1970 file2
```

When done, submit a file called **`twofs.tgz`** with your code. It should be compiled with "`make`".

5 Introducing the `bbfs` filesystem

To give you exposure to many more possible system calls handled by FUSE, we'll be looking at a bit of an odd filesystem, the Big Brother Filesystem by Dr. Joseph Pfeiffer of New Mexico State University. This filesystem accepts a directory name and a mountpoint, then makes it appear as though all of the content from the directory is also present in the mountpoint. It does so by passing each FUSE request it receives to the corresponding "real" system call for the corresponding location in the underlying directory. As a result, it serves to show us what kernel system calls correlate to each FUSE request handler.

I've set up a variant of his code which has a simplified build environment based on Makefiles like we've been using: **this tweaked version is available here**. The only difference from Dr. Pfeiffer's original is that you don't need to run a "configure" script like in the original code; running **`make`** will be sufficient.

To help you get oriented, Dr. Pfeiffer provides a tutorial walkthrough here.

Build the "`bbfs`" example using my version of the code and run it.

Once you have `bbfs` working, do some experiments. Use it to mirror an empty directory, then use common tools to create, read, modify, and delete files in the `bbfs` mountpoint, observing the `bbfs` log as you go. Research the calls that are executed in the FUSE documentation and system manpages (you may also need the UNIX error code list). Become comfortable with how filesystem calls work. Ask the instructor for clarification on anything you don't understand.

6 Benchmarking `bbfs`

One additional useful property of bbfs is that it will give us a good way to measure how much slower using FUSE is, since it just acts as an extra layer on top of a normal filesystem. To be clear, running your filesystem in userspace the way FUSE does is certain to add overhead, and bbfs also includes logging code. Let's compare the underlying filesystem to FUSE using the benchmark **iozone**. Install it as follows:

```
sudo apt install iozone3
```

IOzone has a lot of options and features, see the documentation here.

The key settings are:

- **Modes:** Which tests will be run? These include 0=write/rewrite, 1=read/re-read, 2=random-read/write, and many more. In automatic mode, it defaults to all possible tests.
- **File size:** How big of a file will we operate on? Among other things, this influences what caches the file fits in (CPU caches, physical disk cache, OS buffer cache in RAM). In automatic mode, defaults to a range in 64k to 512M.
- **Record size:** How big each IO is, like the "bs" option in dd. Larger IOs are usually better, up to a point. In automatic mode, defaults to a range in 4k to 16M.

Let's use automatic mode (`-a`), but constrain it to do only a basic read/write test (`-i 0 -i 1`). Let's only use a file size of exactly 4MB (`-s 4M`). We'll let it do all the record sizes. We'll save output in Excel format to the home directory (`-b FILENAME`).

To make a temp directory and run the test on the normal filesystem:

```
mkdir /tmp/test
cd /tmp/test
touch iozone.tmp
iozone -a -i 0 -i 1 -w -s 4M -b ~/iozone-real.xls
```

To make a mountpoint and run the test through the FUSE bbfs[2]:

```
mkdir ~/mountpoint
cd (path to your bbfs binary)
./bbfs /tmp/test ~/mountpoint
cd ~/mountpoint
touch iozone.tmp
iozone -a -i 0 -i 1 -w -s 4M -b ~/iozone-fuse.xls
```

Now, let's do those tests again, but this time have iozone open the file in "SYNC" mode, so that the OS does writethrough caching instead of writeback caching. You can do this by adding the `-o` option to iozone.

[2] NOTE: One quirk of IOzone is that it wants to create the test file with zero permissions to prevent other apps interfering; this works on the native filesystem, but a quirk in FUSE makes this not work there. To get around this, we pre-create the test file with normal permissions, and include the "`-w`" flag to iozone to prevent it deleting this test file. This explains the `touch` command and "`-w`" flag we used.

Prepare a line graph showing real versus FUSE performance for the write test for all record sizes with SYNC both off and on; i.e. have lines for real+nosync, fuse+nosync, real+sync, fuse+sync. If your real+nosync line dominates the graph, note the magnitude of the difference, then remove it so you can see the other trends. Set the plot title and axes up appropriately so the plot speaks for itself. An example of such a plot is shown below (though on different hardware than yours). Include this plot in your submission as a PDF file called **`iozone-result.pdf`**. The lesson here is to observe (a) the strong effects of caching and (b) the overhead of FUSE.