Lab 2 Report for CS380L

Omeed Tehrani

March 5, 2024

Introduction

In this assignment, we were instructed to create a a user-level file system using FUSE. The idea behind FUSE is that it lets us develop a fully functional file-system that can be accessed by non-privileged users, is stable and has a simple API library. We can develop a file system as an executable binary that is linked to the FUSE libraries, allowing us to not worry about the internals of file systems or kernel based programming.

User-level file system: The Implementation

0.1 The Setup

First, I had to decide what machines I wanted to use to accomplish this lab. I intially went with the frame of using my personal M1 Macbook Pro as the client, and the lab machine as the server side. But, this would mean that I would need to install FUSE on my local machine, which has a lot of issues on the Apple chip. Instead, I decided I was going to use two UTCS lab machines. I navigated to `https://apps.cs.utexas.edu/unixlabstatus/`, and found the machines with the lowest number of users. I selected `cracklin-oat-bran.cs.utexas.edu` and `corn-flakes.cs.utexas.edu`. Both machines are running Ubuntu 20.04.06 LTS. The machines are both running Linux version `5.15.0-92-generic`. The machines have Intel Xeon W-1270 CPU's with a 3.40 GHz clock speed, which has 8 cores. Initially, I had dealt with permission issues, and realized it was because of the way that the lab machines were mounting file systems. This was a problem I had to solve, especially because the UTCS administration does not give our accounts `sudo` access. After some exploring, I ran the command `df -T`, which listed all the mounted file systems and their types on the system. This was important, because I needed to really understand where to mount and work, and where to experiment and evaluate (for the later part of the lab). The lines that stood out to me were the following:

```
filer5b:/home_grad/omeed nfs4 1643762816 1587276608  56486208  97% /u/omeed
tmpfs tmpfs 3256876 0 3256876 0% /run/user/39706
```

What this basically showed me is that my directory `/u/omeed` was mounted using NFS, and that there exists a temporary file system that is stored in the RAM, aka. the temporary memory. This would mean a couple of things: to work off these machines, I would need to mount inside of the `/tmp/` directory on the client side machine, and be prepared for the mount to disappear if the system somehow ended up crashing from some alternative user or rebooted. This is why I decided to code in my home directory, and then mount to the `/tmp/` folder on my machine. In the next section, I will outline the general directory structure on my client side machine to get my file system working.

0.2 The Directory Structure

For the sake of clarity, I will outline each directory I created, which machine it is on, where it is on the machine, and its contents. I will proceed in the subsequent section to explain the actual details of the implementation in the `omeed_fuse_fs.c` file.

Machine #1: omeed@cracklin-oat-bran

1. **/u/omeed/FUSE_CODE:** In this directory, there is a `omeed_fuse_fs.c` file, that contains all the contents of my file system to be compiled and mounted. There is a `test_fs_script.c` file as well, which is the provided script from the assignment page. I build on top of this later for my experimentation phase. Additionally, I have a `Makefile` as well, mainly to automate the compilation, mounting, un-mounting, and other repetitive operations that were necessary during the development of my file system. I also used it for some basic testing as I developed, such as doing a `cat` operation on `foo`, since that uses file system operations such as `open` and read. compilation,
2. **/tmp/FUSE_MOUNT:** This directory is where the file system is mounted. As explained above, the only risk is losing this mount on the system reboot, since it is stored on the RAM. But... since the main file system code is inside of my home directory, I was not very concerned about this, because I could just `make clean` and then `make` to remount.
3. **/tmp/FUSE_REAL:** This directory is where the file system points to. Any file we pull from the remote server will sit inside of this directory, since that is where the file system is pointing to. I will expand on this in the explanation of my design.

Machine #2: omeed@corn-flakes

1. **/tmp/FUSE_MACHINE_2_FILES**: This directory just contains the foo file that is transferred over through the scp network protocol into the local `/tmp/FUSE_REAL`.

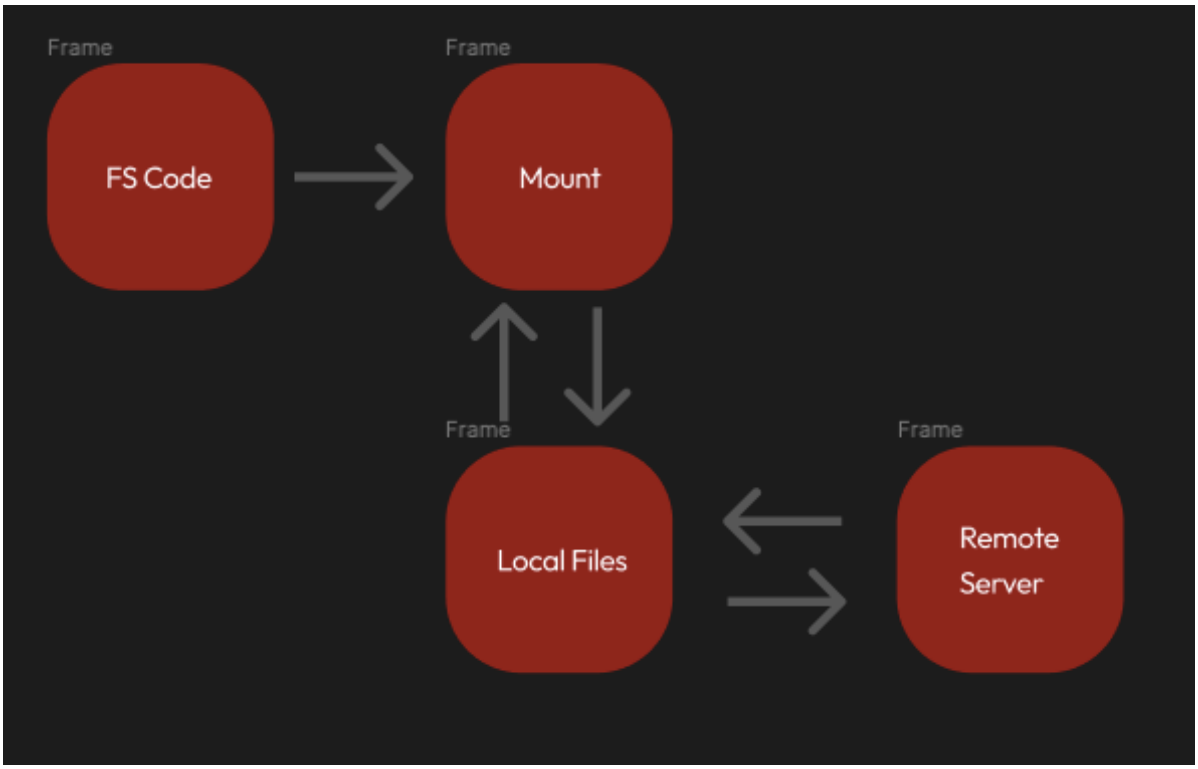
0.3 The Makefile

Just briefly, before I dig into the file system design, I wanted to display the core commands necessary to test and get my file system working:

```
To compile the FS:
    gcc -Wall omeed_fuse_fs.c ‘pkg-config fuse --cflags --libs‘ -o omeed_fuse_fs

To compile the test script:
    g++ test_fs_script.cpp -o test_fs_script

To populate a starting file that the FS is pointing to:
    touch /tmp/FUSE_REAL/foo

To mount the file system to the proper path through the fuse_main()
    ./omeed_fuse_fs /tmp/FUSE_MOUNT

To test my networked FS and verify its outputs:
    ./test_fs_script
    cat /tmp/FUSE_REAL/foo
```

0.4 My FS Design

Since I had never made a FUSE file system, it was important for me to have a baseline. I began by working off of a minimal example filesystem using high-level API from the libfuse GitHub, which had a lot of overlaps with the fusexmp.c. At a high level, my file system design is the following:



After compiling my FUSE filesystem code, I mount it to a specified directory. Once mounted, this directory mirrors all the files located in my local file directory, specifically at /tmp/FUSE_REAL. In my implementation, which I will elaborate on later, I ensure that every file in the directory my filesystem is linked to is accessible. Additionally, the remote server has the capability to write directly into this local file directory, with changes promptly reflected in the mounted directory. An important aspect to note is that, although the local file directory can contain multiple files and the filesystem displays their presence, it currently only supports operations on a specific file defined by a global variable. This setup could be evolved to accept arguments or be designed to automatically recognize and operate on any file within the directory based on user interactions.

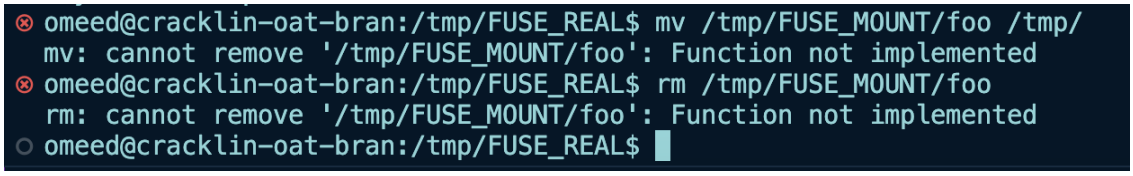
Next, I will explain my implementations of getattr, open, read, write, fsync, and release. All code can be found in my submission:

1. **Miscellaneous:** There exists some core imports such as fuse.h and other libraries that allow me to make system calls. Additionally, at the top of the file, there exists static global variables that can be changed to allow the user to operate on files besides foo. As describes above, this can eventually be evolved to operate on any file within the pointed directory based on user interactions, but for now, these global variables point directly to `foo`:

```
static const char *base_path = "/tmp/FUSE_REAL";
static const char *real_fuse_path = "/tmp/FUSE_REAL/foo";
```

```
static const char *current_file = "/foo";
```

2. **Testing Non-Implementations to Verify Working Implementaitons:** To sanity check if the implemented functions were being hit off of my file system, I attempted to perform non-implemented functions, and the result yielded exactly what I was looking for:



3. **getattr:** This function gets the attributes of a file or directory. It first checks to see if the path argument is in the root (likely where the file system is mounted), and if it is, it gives the owner read, write, and execute permissions. Next, it checks to see if the current path matches the file that we want to get statistics on, and if so, then we can safely make a lstat call to get statistics and attributes of the file. The result is then returned at the end. What is returned can include things like file user id, file group id, access time, modification time, etc.. which all basically give additional details about the file or directory, permissions, timestamps, etc. To test if these file system functionalities are working in the implementation, I ran `stat /tmp/FUSE_MOUNT/foo`, which yielded this result with no failure:

```
omeed@cracklin-oat-bran:/tmp/FUSE_REAL$ stat /tmp/FUSE_MOUNT/foo
  File: /tmp/FUSE_MOUNT/foo
  Size: 100             Blocks: 8          IO Block: 4096   regular file
Device: 43h/67d Inode: 2           Links: 1
Access: (0600/-rw-------)  Uid: (39706/   omeed)   Gid: (   20/    grad)
Access: 2024-03-04 13:51:58.079451476 -0600
Modify: 2024-03-04 13:51:57.795451734 -0600
Change: 2024-03-04 13:51:57.795451734 -0600
 Birth: -
```

4. **open:** My open function attempts to open a file specified by the global path in read only mode. If the file can not be opened, it errors out. Otherwise, it closes the file descriptor to indicate that the file exists and is accessible.

5. **read:** This function reads data from the foo file in my FUSE file-system. I start by first creating an scp command to copy the file from the remote server to the local file directory that the file system is pointing to. Next, I read size bytes of data from the file starting from offset into buf and then return the number of bytes that were read. This read is accomplished by using the pread() system call.

6. **write:** Naturally, the write is an opposite of the read, especially in the networking context. The function will write data to the file that is specified by the global `real_fuse_path()`, using the provided parameters. Since it needs to open to write, similar to the read function, I make a system call using open(), and use the pwrite() system call to write the data at the given offset. Then, using the same SCP command that I used in read, I flip the local and remote path arguments, and send the written file back to the remote server. Finally, I return the number of bytes that were written, or any errors that occured during execution.

7. **fsync:** This function checks to see if the passed in `isdatasync` parameter is true, it makes a system call to fdatasync which flushes all modified core data but does not flush modified metadata unless that metadata is needed.

8. **release:** Building off of the fuseexmp shell, this method purely just closes the file descriptor. I don't use it this way in my implementation, and invoke the close() system call directly in the functions that files are opened, but I implemented this for the sake of the request. My design does not support it.

0.5 Arguing the Design

0.5.1 Benefits

The FUSE filesystem presents notable advantages by operating entirely within user space, eliminating the need for any kernel modifications. This feature alone significantly simplifies the development process, making FUSE an appealing choice for filesystem experimentation and deployment. The ability to implement and modify filesystems without altering the kernel not only enhances security by minimizing the system's exposure to potentially unstable code but also allows for quick modifications and extensions during development. Another huge benefit of this file system is its capability to interact with remote servers for read and write operations. This flexibility extends the file system's utility beyond local storage, enabling seamless access to files across diverse networks. In this implementation, the system is not restricted to the utexas network, suggesting that, with further development, it could be adapted to fetch data from virtually any remote location. Such an extension would dramatically increase its applicability, making it a versatile tool for accessing and managing data across different environments. The design choice to operate with files in the computer's RAM can lead to significant performance improvements. By leveraging the speed of RAM compared to traditional disk-based storage, the file system can achieve faster data retrieval and write times. This approach optimizes the performance of file operations, potentially making the system more responsive and efficient for users. There's also a reduced likelihood of kernel panics, the worst case will be just general input / output errors. Also it is usable by non-privileged users!

0.5.2 Limitations and Issues

There are several disadvantages with this implementation as well. A widely known fact is that FUSE file systems can be slower, especially if you want multiple users to use the same file system at the same time. Additionally, it can be borderline impossible to guarantee consistency, mainly due to the nature of FUSE. For example, we have a huge lack of control over caching mechanisms, and how we fetch data remotely is limited to what we can do in user space. Additionally, there is a lot of context switching that likely occurs between user space and kernel space. Also another huge downside is that we are on a Unix system. As I discussed before, because everything outside of `/tmp/` is NFS, this means that on a system crash or reboot, I can lose the entire mounted file system, and additionally, multiple clients might access the same data and lead to errors. My implementation is also just generally incomplete. There are a HUGE amount of functions that aren't necessarily required for a file system to work, but to have the diverse range of functionality that a privileged or non-privileged user would want to use, I would need to implement those. Additionally, this file system is designed to scope in on a specific file foo. This is not how file systems work in the real world. Eventually, you'd want it to be able to reflect and handle all files in the directory that the file system is pointing to, and that likely comes from either adding extensive flagging options when you run the command, or.. just being able to take in the argument that the user inputted into the terminal and doing the necessary operations on that particular input and not the direct path specified.

0.5.3 Consistency Model

In conducting further research, I found that the NFS file system adheres to an `open-close consistency` model. This model ensures that when a client closes a file, any subsequent open operation on that file by another client will access the most current version of the file. In contrast, the FUSE file system I implemented follows more of a `eventual consistency` model. This discrepancy arises from the reliance on SCP (Secure Copy Protocol) for file operations. Due to SCP's operational nature, modifications made to a file are not immediately propagated and are reflected upon the completion of the actual command. This delay in synchronization can lead to potential write errors and conflicts, as changes might not be instantaneously visible.

User-level file system: The Evaluation

0.6 Experiment Setup and Environment

Network Topology: My network topology is communication between two UTCS lab machines on the utexas network. When I work remotely, I simply just SSH using the Cisco VPN.

To establish an NFS experimentation, it's important to recognize that the UTCS lab machine already utilizes NFS4 for its home directories. This setup simplifies the process significantly. By creating the foo file on the remote lab machine, this file automatically becomes accessible from any client machine connected to the same NFS. This is due to the nature of NFS, where files in the networked file system appear local to all machines in the network. Therefore, there's no need for direct cross-machine communication to access the foo file I create.

With this setup, executing a script located in the home directory that interacts with foo becomes straightforward. This approach can be particularly useful for conducting timing experiments.

Standardized Environment: I would argue that the experimental environment is quite standardized. The hardware is consistent between both machines, meaning that for all tests, I am using the same hardware. The servers hosting the file systems are also the same across machines, which I verified by doing `systemctl list-units --type=service --state=running`. Across both machines, the linux kernel version and operating system are also the same.

0.7 Experiment #0: Running the given test program

Based on where I put the script, which was in my FUSE_CODE directory, I modified the `open` to point to `/tmp/FUSE_MOUNT/foo`. This is because I want to direct all file operations through the defined logic in my FUSE file. After running it, this is the output that I got:

```
g++ test_fs_script.cpp -o test_fs_script
touch /tmp/FUSE_REAL/foo
./hello /tmp/FUSE_MOUNT
./test_fs_script
Wrote 100, then 100 bytes
cat /tmp/FUSE_REAL/foo
        QAq0@`d~@U@@@@@@D@@@D@@sd~@U@R@@@sd~@U qd~@Ub@Domeed@cracklin-oat-bran:~/FUSE_CODE$ cd
omeed@cracklin-oat-bran:~$ scp ~/FUSE_CODE/
```

This output tells me that the initial content was written to the buffer, and the rest of the file is non-printable characters. 81 is the ASCII value for an upper case Q, `AqQ` are all just characters, and 9 is likely some non-visible value. This tells me that it worked! Now it's time for some actual experiments, to mainly see how my file system performs in comparison to the NFS file system.

0.8 Experiment #1: Run the test program for 100 iterations

The test program that was given to us opens the file for writing, opens the file for reading, writes, closes, reads, closes, etc. I took this script, and wrapped it into a loop of 100 iterations with a timer. I

ran it once for NFS and once for my file system. The difference was actually quite significant.

```
My FS:
Repeated write-read 100 times
Total time: 33.803 seconds

NFS:
Repeated write-read 100 times
Total time: 0.052 seconds
```

0.9 Experiment #2: Iterative Opening

This experiment is extremely simple. It simply opens the file repeatedly without performing any read or write operations over the network. It spams open operations on the specified file, and does the same timing mechanism as the previous experiment. The results were quite surprising. There was generally a lot of variability, but the difference was quite minimal:

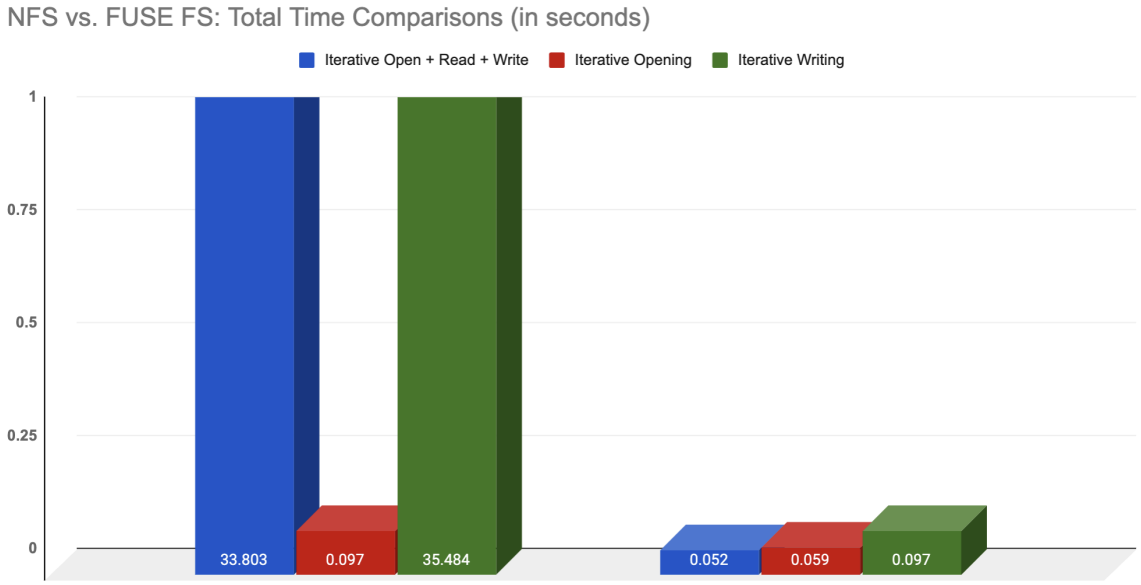
```
My FS:
Opened the file 10000 times
Total time: 0.097 seconds

NFS:
Opened the file 10000 times
Total time: 0.059 seconds
```

0.10 Experiment #3: Iterative Writing

This experiment is also quite simple. Instead of repeatedly opening the file, it performs multiple write operations for 200 iterations. On each iteration, it writes "Hello World" and then closes the file. The time differences are shown below:

```
My FS:
Wrote to the file 200 times
Total time: 35.484 seconds

NFS:
Wrote to the file 200 times
Total time: 0.097 seconds
```

0.11 Chart and Discussion



There were a lot of interesting results from my experiments. To first expand on the chart, this chart shows the total time of each file system, separated by experiments. The left bunch is my FUSE file system, the right bunch is the NFS file system. As you can see, it very clearly outperformed my simple file system. This boils down to a lot of key points, but my main postulation is because of the way that I am handling networking. Let me explain. For iterative open + read + write, The NFS file system was approximately 650x more fast than my FUSE file system. This makes a lot of sense, FUSE operates in user space, which has naturally more overhead for operations, and using SCP for file operations introduces a HUGE amount of overhead when you compare to NFS which uses smart networking, caching and optimization techniques. For the iterative opening experiment, the clearest explanation for this is that there is no networking involved for the open command. One limitation of this is that it assumes there is a file in the directory it is pointing to. That is a large limitation of this file system. Likely a mitigation of this issue is to check what files exist on the remote machine and if they exist on the local machine, and if they don't, then just pull all those files into the locally pointed directory to mirror for your file system. Since this is a very bare bones file system, I did not implement that functionality. Finally, for the iterative writing experiment, since it writes to the file and then sends it back over the network to the remote client using SCP, we are again bottle necked by the networking implementation. Although this is a "working" networked file system, my system needs to be heavily optimized for networking.

System tools: strace

The file is inside of my submitted folder. From my understanding, the purpose here was to use strace to monitor all the system calls that are made to write the input "hi mom" into the `new_file`. It was really interesting to see how many system calls were actually being made in the background! These included exec, brk, arch_prctl, access, mmap, close, etc. It seems like these are mainly a lot of necessary low level steps required across linux programs and tasks to ensure a robust execution. In the future, I definitely want to explore running this on more complicated tasks.

LSOF Command

For this portion, I ran the `lsof | grep /dev` command. These are portions of my output:

```
    omeed@corn-flakes:~$ lsof | grep /dev
systemd    3822072  omeed    0r      CHR                1,3      0t0          5 /dev/null
pulseaudi 3822079  omeed    0r      CHR                1,3      0t0          5 /dev/null
 script    3822292 omeed    1u      CHR              136,1      0t0          4 /dev/pts/1
script    3822356  omeed    3u      CHR                5,2      0t0         87 /dev/ptmx
```

There are a couple of pretty core devices here:

1. **/dev/pts/(number)**: So when I ran `tty` in my terminal, it showed me that mine was `/dev/pts/4`. This basically tells me that my pseudoterminal number is 4. Essentially, all the /dev/pts devices allow processes to communicate with the terminal emulator or a remote client, so it is interactive to the user on the machine. These are the "slaves" in this paradigm. This is a vital component of allowing us to have terminal interactions.
2. `/dev/ptmx` is also very important, which is basically a character file that is used to create a pseudoterminal master and slave pair. It basically facilitates the terminal connections.
3. **/dev/null**, a virtual device allowing any data that is written to it, to vanish. Allows us to discard things like standard output.

1 System tools: network

For this portion, we had to answer 6 questions. I was unable to do some commands on the lab machine, since it required sudo access, so for answering the questions, I temporarily requested a cloud lab machine.

1. **Are DHCP messages sent over UDP or TCP?:** DHCP uses UDP as its transport protocol.
2. **What is the link-layer (e.g., Ethernet) address of your host?:** After running `ip link`, I found that it is `14:18:77:26:67:c1`.
3. **What is the IP address of your DHCP server?:**

```
omeed26@node0:~$ sudo grep DHCP /var/log/syslog
Mar  5 12:04:06 localhost systemd-networkd[3097]:
eno1: DHCPv4 address 155.98.36.81/22 via 155.98.36.1
Mar  5 12:04:06 localhost systemd-networkd[3097]:
eno1: DHCPv6 lease lost
Mar  5 12:04:06 localhost systemd-networkd[13799]:
eno1: DHCPv4 address 155.98.36.81/22 via 155.98.36.1
```

   This tells me that the IP address on the DHCP server on cloud lab is 155.98.36.1.

4. **What is the purpose of the DHCP release message?** It is to release the IP address back to the server.

5. **Does the DHCP server issue an acknowledgment of receipt of the client's DHCP request?** No it does not.

6. **What would happen if the client's DHCP release message is lost?:** If the client's DHCP release message is lost, the client gives up the IP address, yet the server will wait until the client's lease on the address runs out before attempting to reallocate that address.