CS380L: Advanced Operating Systems Lab #2

Zeyuan Hu [1], iamzeyuanhu@utexas.edu

EID:zh4378 Spring 2019

1 Environment

We have two Linux machines for experiments: `erode-lom` and `thoothukudi-lom`. Both of them run Ubuntu 16.04 LTS. [2] Both machines are located in the same rack and connected via Ethernet.

2 User-level File System

2.1 Implementation

We leverage `libfuse` and `libssh` to implement a network file system. Filesystem in Userspace (FUSE) is a user-space file system framework. FUSE consists of a Linux kernel module and a user-level daemon. When a user application performs operations on a mounted FUSE file system, the operation will be routed to FUSE's kernel driver by VFS. The operations as requests will be maintained by a queue and user-level daemon will pick a request from the kernel queue and process the request. Daemon will write response back to kernel once it is done with processing. More information about FUSE can be seen in [1].

2.1.1 System Architecture

We implement a network file system. Like NFS, it supports a server with multiple clients. On the client side, the FUSE client program will mount a user-space file system with remote user, remote host, remote path, local cache path, and local mount point provided by the user. Remote user and remote host specify the server that our network file system want to contact. Remote path specifies the location on the server we want to store and fetch files for the clients. Our system requires user to specify a path for local cache as we serve clients' requests from local cache as much as we can. In other words, some file operations are directly served by the local cache without further contact with server. Communications between clients and the server are done via SSH protocol. Thus, there is no server-side implementation in our system.

[1] 30 hours spent on this lab.

[2] `erode-lom` with ip address: `192.168.1.120` and `thoothukudi-lom` with ip address: `192.168.1.121`. Due to the network setup, those two machines are not properly registered in the department DNS server. We use ip address instead host names to logon these two machines.

2.1.2 FUSE Calls Implemented

We implement the following operations: `getattr`, `readdir`, `create`, `open`, `read`, `write`, `fsync`, `release`, `mkdir`, `unlink`, and `rmdir` in our system. Our operation implementation is based on `libfuse` version is 2.9.4 [3] API. In `libfuse`, there are two sets of APIs: a "high-level", synchronous API, and a "low-level" asynchronous API. The key difference between "high-level" and "low-level" API is that "high-level" allows us to work with file names and path instead of inodes in synchronous fashion. Thus, for the simplicity, our implementation adopts the "high-level" API.

`getattr` operation is used to get attributes of a file or a directory. Function signature of `libfuse`'s `getattr` operation is `int getattr(const char * path, struct stat * stbuf);`. Our implementation should fill `stbuf` to contain attributes of file or directory indicated by `path`. Since `struct stat` is the same structure used in `stat` call [2], we can leverage `stat` command [3]. Specifically, in our implementation, we execute `stat` command remotely via SSH to collect attributes of the desired target (specified by `path`). The output of the `stat` command is parsed on the client side of the file system. We fill `st_size`, `st_blocks`, `st_mode`, and `st_nlink` fields of `struct stat` from the parse result. `st_uid` and `st_gid` are filled with UID and GID of the file system client process. `st_mode` is filled with `644` (i.e., `rw-r--r--`) if the target is file and `755` (i.e., `rwxr-xr-x`) if the target is directory.

`readdir` operation is used to get entries in a directory (i.e., read directory). There are two modes of operation for the implementation: whether we keep track of offset provided as one of function signature argument. If we ignore the offset, the content of whole directory will be read in a single operation. For the simplicity, we ignore the offset in our implementation. This can be troublesome if the content of directory is greater than the supplied buffer size. The key to our implementation is the implementation of filler (with type `fuse_fill_dir_t`), which is used to add directory entry to the supplied buffer. In our case, `getattr` is invoked whenever filler is called.

`create` and `open` operations correspond to `creat` and `open` system calls. libfuse's `open` operation has signature `int open(const char* path, struct fuse_file_info* fi);`. `path` specifies the file to be opened and `fi->flags` indicates the open flags (same as `flags` in `open` system call [4]). `fi->fh` represents file handle, which may be filled for future usage. In our implementation, when `open` is invoked, the target file will be downloaded from server via SCP and saved in the local cache path. Operations like `read`, `write`, and `fsync` will be served from local cache copy. The local cache copy is opened with the same flags and corresponding `fd` is saved in `fi->fh` for future use. `create` operation is similar to `open` except it will first create the file remotely if the file does not exist.

`read`, `write`, and `fsync` operations are served by local cache file copy via `fi->fh`. Same as NFSv2, we implement flush-on-close semantic for update visibility. Specifically, we do not upload the file to the server on `write` and `fsync`, instead we update server's copy when the file is closed.

[3]check via `fusermount -V`

Doing so removes network communication overhead between each file update, but we may face file inconsistency if there are multiple clients updating the same file competitively.

`release` operation is invoked by `libfuse` when closing a file descriptor. This is the place where we implement flush-on-close semantic. We first close `fi->fh`, which is the file descriptor the local cache copy. Then, we upload the local cache copy to the server via SCP.

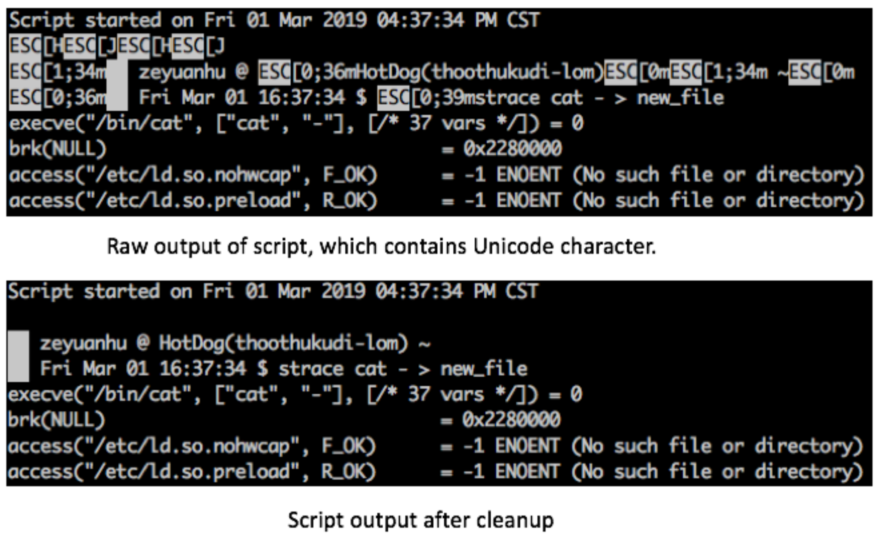
`mkdir`, `rmdir`, and `unlink` operations are performed by executing `mkdir` and `rm` command remotely via SSH.

2.1.3 Limitation

Our implementation has coarse-grained access control in the sense that all the files and directories have the same UID, GID, and permission mask. To enable a more fine-grained access control, we could use a file similar to `/etc/exports` in NFS to indicate what ip address can have what access (read, write, or both) to what directories and files. Doing so requires us to implement a server-side code as a guard to perform identity check. However, access control is orthogonal to our experiment goal and we left this feature as future work.

Incomplete file metadata also impacts how we implement the file consistency mechanism. In our implementation, we do not keep creation time, access time, and modification time of directories and files. As a result, we cannot selectively perform SCP on `open`: if we maintain file stats, we can perform stats comparison between remote copy and cached copy to see whether we need to perform expensive data transmission over the network. In addition, since we only SCP file during `open`, file can change between `open` and `read`. NFS periodically issues `getattr` to server to ensure cache consistency. Since we do not support file timestamp, we cannot issue `getattr` periodically and let `read` directly read from server when the local cache is invalidated.

Some other issue might exist regarding file consistency. For example, we implement the last-writer-wins policy: if a file is updated by the multiple users, the last one who close the file will keep its change to the file on server. However, this might be troublesome. A more sophisticated method is to automatically merge change to the file whenever possible and maintain multiple versions of files on the server with each version associated with its owner. Other versions may not be visible the user and only a specific system command issued will make those versions visible. We also allows a file can be opened multiple times. This is troublesome as later open operations can overwrite changes made by previous file descriptors. One possible fix is to only download copy file from server when there is no local cache copy or implement a copy-on-write mechanism: each open will lead to a unique version of the file and it is up to user to resolve potential conflicts.



```
Script started on Fri 01 Mar 2019 04:37:34 PM CST
ESC[HESC[JESC[HESC[J
ESC[1;34m   zeyuanhu @ ESC[0;36mHotDog(thoothukudi-lom)ESC[0mESC[1;34m ~ESC[0m
ESC[0;36m   Fri Mar 01 16:37:34 $ ESC[0;39mstrace cat - > new_file
execve("/bin/cat", ["cat", "-"], [/* 37 vars */]) = 0
brk(NULL)                               = 0x2280000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
```

Raw output of script, which contains Unicode character.

```
Script started on Fri 01 Mar 2019 04:37:34 PM CST

  zeyuanhu @ HotDog(thoothukudi-lom) ~
  Fri Mar 01 16:37:34 $ strace cat - > new_file
execve("/bin/cat", ["cat", "-"], [/* 37 vars */]) = 0
brk(NULL)                               = 0x2280000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
```

Script output after cleanup

Figure 1: `script` output before and after cleanup

2.1.4 Comparison vs. NFS

3 System Tools Exercise

3.1 strace

`script` command allows user to record terminal printout into a file [5]. Per the lab instruction, we use `strace` to trace the syscalls and signals of a target process [6]. In our case, we trace the process involving `cat`. One thing I notice is that `script` contains some unicode as shown in Figure **??**. Thus, we use the following code to clean up the output:

```
cat $FILE | perl -pe 's/\e([^\[\]]|\[.*?[a-zA-Z]|\].*?\a)//g' | col -b > $FILE-
    processed
```

The result is shown in Figure 1 [4].

3.2 lsof

`lsof` lists all open files [7]. `lsof | grep /dev` shows all the open devices used by user-space programs. On our machine, we have the following opened devices:

- `/dev/null`: null device
- `/dev/pts/*` and `/dev/tty`: terminal devices
- `/dev/urandom`: kernel random number source device

[4]raw output and cleanup output comes with the report as `session_record` and `session_record-processed` respectively

- `/dev/ptmx`: a character file to create a pseudoterminal master

4 Network Tools

`ifconfig` command lists all the network interfaces the machine is using to communicate externally. On our machine, interface for Ethernet is `eno1`. We can find IP address, gateway address, and subnet mask from the output.

`tcpdump` command can dump traffic on a network interface. We use the `tcpdump` output provided by the lab instruction to answer the questions below.

a. Are DHCP messages sent over UDP or TCP?

We use `tcpdump -nn -r tcpdump.out.1 | grep -i dhcp` to filter out the DHCP messages from the dump. `-nn` ensures that we can see the actual port number instead of the port name. One line of the output is:

```
10:19:24.525962 IP 0.0.0.0.68 > 255.255.255.255.67: BOOTP/DHCP, Request from
    a8:20:66:3b:66:51, length 300
```

The first field shows the time that the packet was traveling. The second field shows the source host address and port, followed by the destination host address and port. The third field shows the protocl the packet was using. From DHCP [8], we know DHCP messages sent over UDP. As shown by the printout, the messages are sent between port 68 (client) and port 67 (server).

b. What is the link-layer (e.g., Ethernet) address of your host? (Feel free to obscure the last couple bytes for privacy's sake)

We use the same `tcpdump` command as above with extra `-e` option to show linke-layer header. The following printout contains DHCP messages for acquiring IP address:

```
10:19:24.525962 a8:20:66:3b:66:51 > ff:ff:ff:ff:ff:ff, ethertype IPv4 (0x0800
    ), length 342: 0.0.0.0.68 > 255.255.255.255.67: BOOTP/DHCP, Request from
    a8:20:66:3b:66:51, length 300
10:19:24.566258 00:21:9b:fb:61:0c > a8:20:66:3b:66:51, ethertype IPv4 (0x0800
    ), length 342: 128.83.158.2.67 > 128.83.158.160.68: BOOTP/DHCP, Reply,
    length 30
```

From the printout we can see that the link-layer address (MAC address) of the host is `a8:20:66:3b:66:51`.

c. What is the IP address of your DHCP server?

From the printout above, we can see the IP address of DHCP server is `128.83.158.2` and the new IP address acquired from DHCP server is `128.83.158.160`.

d. What is the purpose of the DHCP release message?

DHCP release message is used to release IP address.

e. Does the DHCP server issue an acknowledgment of receipt of the client's DHCP request?

DHCP server does not issue an acknowledgment of receipt of the client?s release message.

f. What would happen if the client's DHCP release message is lost?

If DHCP release message is lost, the DHCP server has to wait for the lease to timeout before assigning it to other clients.

References

[1] B. K. R. Vangoor, V. Tarasov, and E. Zadok, "To {FUSE} or not to {FUSE}: Performance of user-space file systems," in *15th {USENIX} Conference on File and Storage Technologies ({FAST} 17)*, pp. 59–72, 2017.

[2] "stat(2) - linux man page." `https://linux.die.net/man/2/stat`.

[3] "stat(1) - linux man page." `https://linux.die.net/man/1/stat`.

[4] "open(2) - linux man page." `http://man7.org/linux/man-pages/man2/open.2.html`, 2018.

[5] "script(1) - linux man page." `http://man7.org/linux/man-pages/man1/script.1.html`.

[6] "strace(1) - linux man page." `https://linux.die.net/man/1/strace`.

[7] "lsof(8) - linux man page." `http://man7.org/linux/man-pages/man8/lsof.8.html`.

[8] "Dynamic host configuration protocol." `https://en.wikipedia.org/wiki/Dynamic_Host_Configuration_Protocol`, 2018.