CS380L: Advanced Operating Systems Lab #3

Zeyuan Hu [1], iamzeyuanhu@utexas.edu

EID:zh4378 Spring 2019

1 Environment

Unless otherwise noted, we use a Linux server for all the experiments. The server has 4 Intel(R) Xeon(R) CPU E3-1220 v5 @ 3.00GHz processors and 16GB of memory, and runs Ubuntu 16.04.2 LTS (kernel version 4.11.0).

2 ELF

Executable and Linkable Format (ELF) is a file format for executable, object code, shared libraries, and core dump on UNIX-like systems [1]. ELF format is shown in Figure 1, which is taken from [2]. An ELF binary starts with a fixed-length *ELF header*, which holds a "road map" describing the file's organization (e.g., instruction set architecture, endianness, system ABI, entry point of the program). A *program header table*, if present, tells the system how to create a process image (i.e., execute a program). Each entry of program header table corresponds to a segment in the virtual address space. Part of the segment contents can be loaded from the executable file, which is also defined in the program header table. A *section header table* contains information describing the file's sections (e.g., `.text`, `.data`, `.rodata`, `.bss`). Some of them will be loaded into virtual address space as part of segments indicated by program header table. Each entry gives information such as section name, the section size, etc. Program header table is required for to-be-run file and optional for relocatable file; section header table is required for file used during linking and optional for others.

There are some special sections we care about in ELF. `.text` section contains the program's executable instructions; `.rodata` represents read-only data, such as ASCII string constants produced by the C compiler; `.data` holds the program's initialized data, such as global variables declared with initializers like `int x = 5;`; `.bss` section holds uninitialized data (e.g., uninitialized global variables such as `int x`;) that contribute to the program?s memory image. `.bss` immediately follows `.data` in memory. C requires that "uninitialized" global variables start with a value of zero. Thus, the system initializes the data with zeros when the program begins to run. The section occupies no file space (linker records just the address and size of the `.bss` section). The loader or the program itself must arrange to zero the `.bss` section.

[1]30 hours spent on this lab.

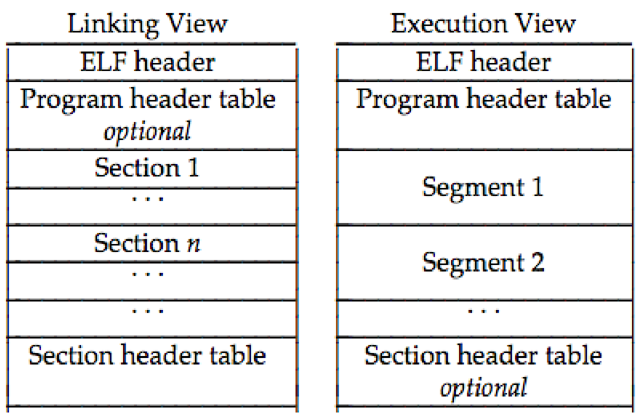


Figure 1: ELF format. There are two views of ELF binary: linking view and execution view. Linking view is used during the dynamic linking, whereas execution view is needed when load the program into memory for the program to run.

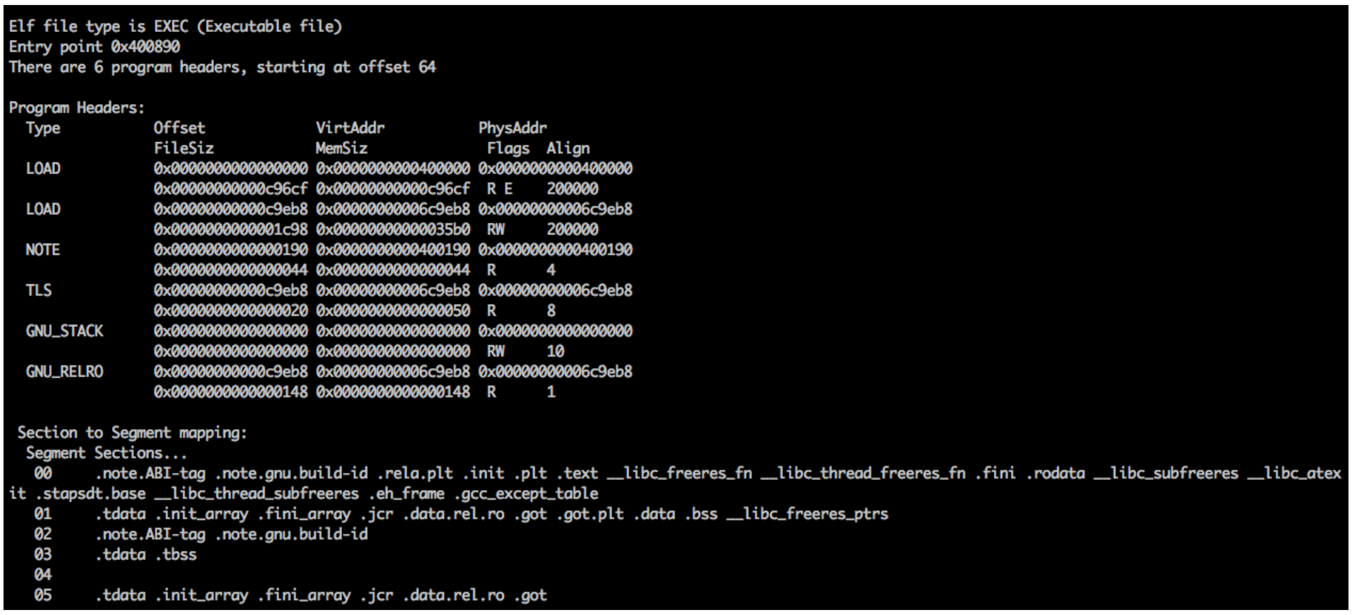
To gather better sense of ELF, we write a simple "Hello World" program (`test_helloworld.c`) shown below to study its ELF content.

```c
#include <stdio.h>

int main() {
  printf("Hello world.\n");
  return 0;
}
```

We compile the program via `gcc -static -O0 -g -std=c11 -o test_helloworld test_helloworld .c -lpthread` and then inspect its ELF content via `readelf -l test_helloworld`. The printout of the `readelf` is shown in Figure 2. We see there are six segments with five kinds of `p_type`: `PT_LOAD`, `PT_NOTE`, `PT_TLS`, `PT_GNU_STACK`, and `PT_GNU_RELRO`. `PT_LOAD` indicates loadable segments; `PT_NOTE` specifies the location and size of auxiliary information; `PT_TLS` specifies the thread-local storage template [3]. `PT_GNU_STACK` and `PT_GNU_RELRO` are linux-specific `p_type` values [4]: `PT_GNU_STACK` indicates whether stack is executable and `PT_GNU_RELRO` specifies the location and size of a segment which may be made read-only after relocation shave been processed. As we will see in the following section, linux `execve` implementation mainly concerns about program header table entry with type `PT_LOAD`.

`readelf -l` also gives section to segment mapping. As we can see, multiple sections can be contained within the same segment. For example, the first segment contains sections like `.init` (contains the process initialization code, which is executed before calling the main program entry point), `.text`, `.rodata` and the second segment contains sections like `.data`, `.bss`. Note the second



```
Elf file type is EXEC (Executable file)
Entry point 0x400890
There are 6 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x00000000000c96cf 0x00000000000c96cf  R E    200000
  LOAD           0x00000000000c9eb8 0x00000000006c9eb8 0x00000000006c9eb8
                 0x0000000000001c98 0x00000000000035b0  RW     200000
  NOTE           0x0000000000000190 0x0000000000400190 0x0000000000400190
                 0x0000000000000044 0x0000000000000044  R      4
  TLS            0x00000000000c9eb8 0x00000000006c9eb8 0x00000000006c9eb8
                 0x0000000000000020 0x0000000000000050  R      8
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     10
  GNU_RELRO      0x00000000000c9eb8 0x00000000006c9eb8 0x00000000006c9eb8
                 0x0000000000000148 0x0000000000000148  R      1

 Section to Segment mapping:
  Segment Sections...
   00     .note.ABI-tag .note.gnu.build-id .rela.plt .init .plt .text __libc_freeres_fn __libc_thread_freeres_fn .fini .rodata __libc_subfreeres __libc_atex
it .stapsdt.base __libc_thread_subfreeres .eh_frame .gcc_except_table
   01     .tdata .init_array .fini_array .jcr .data.rel.ro .got .got.plt .data .bss __libc_freeres_ptrs
   02     .note.ABI-tag .note.gnu.build-id
   03     .tdata .tbss
   04
   05     .tdata .init_array .fini_array .jcr .data.rel.ro .got
```

Figure 2: `readelf -l` printout of the `test_helloworld` executable.

segment also contains `.init_array` and `.fini_array`, which are pointers to functions that will be executed when program starts and ends respectively [5]. In addition, we see the memory layout starts from address `0x400000`, which is configured by linker. We need to change this address in some way for our loader program.

3 `execve` Implementation

To write our own loader program, we first need to understand how `execve` system call is implemented in Linux kernel [2]. Essentially, `execve` performs the heavylifting work of loading a program into memory and start the execution of program as a process in Linux. The main logic of `execve` is implemented in `do_execveat_common` in `fs/exec.c`. Inside `do_execveat_common`, we see the critical function call stack looks like below:

```
do_execveat_common
|- exec_binprm
   |- search_binary_handler
      |- load_binary
```

For binary with ELF format, `load_elf_binary` is registered with `load_binary` in `fs/binfmt_elf.c`. Thus, we locate the core function `load_elf_binary`, which is responsible to load binary with ELF format in the Linux kernel. Further study of the function reveals that the following critical steps are performed in order to load the program into memory and start execution:

[2]we study the source code of Linux 4.8.12

- Read the ELF header and perform some simple consistency checks. For example, the function checks whether the binary file is executable file (`ET_EXEC`) or shared object file (`ET_DYN`).
- Read the program header table by calling `load_elf_phdrs`.
- Walk through the entries of the program header table and perform specific actions based on program header table entry type (e.g., `p_type`). Note that `e_phnum` holds the number of entries in the program header table. Each entry of the table corresponds to a memory segment in the binary file. Kernel is only interested in three types of program header entries during the walk through of the program header table:
  - `PT_INTERP` entry, which identifies the run-time linker needed to assemble the complete program [6].
  - `PT_GNU_STACK` entry, which determines whether the program's stack should be executable [7].
  - `PT_LOPROC ... PT_HIPROC`, which are values reserved for processor-specific semantics [6].
- Once all the program header table entries have been processed, the function performs some checks based on `p_type` value it obtained from the previous step (e.g., check for interpreter, check for specific processor architecture).
- The function now is ready to set up the new program. The very first step is to call `flush_old_exec` , which clears up state in the kernel that refers to the previous program. Some minor setups (e.g., set up program's personality [8]) are performed immediately after.
- `setup_new_exec` is called to set up the kernel's internal state for the new program and new credentials for this executable is installed via calling `install_exec_creds`.
- `setup_arg_pages` is invoked to set up kernel's memory tracking structures (e.g., stack `vm_area_struct`). This step is part of the goal to set up virtual memory for the new program.
- The function now traverse the program header table again and look for entries with type `PT_LOAD`, which indicates loadable segments (i.e., areas of the new program's running memory). The entries contain code and data sections that come from the executable file and the size of a `BSS` section. For each `PT_LOAD` entry, the function maps it into the process' address space via `elf_map` call and sets up the new program's memory layout accordingly.
- `set_brk` is invoked to set up zero-filled pages that correspond to the program's BSS [9] segment.
- `create_elf_tables` is called to set up the rest of the new program's stack. Basically, the function puts `argc`, `argv`, `envp`, and auxiliary vectors. LWN article [7] provides an illustration

of the content of the stack set up by the kernel. In addition, Figure 3.11 of System V ABI for `x86_64` manual [10] provides what initial process stack looks like.

- `start_thread` is invoked to start the execution of the new program.

4 User-space Loader

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 [11].

4.0.1 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.

4.0.2 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

[3] check via `fusermount -V`

struct stat is the same structure used in stat call [?], we can leverage stat command [?]. 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 [?]). 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. 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.

4.0.3 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.

4.1 Evaluation

In this section, we compare our network file system with NFS under three workloads.

4.1.1 Experiment Setup

For our network file system, we use `thoothukudi-lom` as client and `erode-lom` as server. On the client side, our system is mounted under `/tmp/barfs` and the local cache is under `/tmp/barfs_cache` . For NFS, to make a fair comparison, we also use memory as storage location for NFS server and we require the NFS to reply requests only after changes have been committed to stable storage (`sync`) [?]. Our `/etc/exports` looks like below:

```
/tmp/nfs *(rw,sync,no_subtree_check,no_root_squash)
```

On the client side, we mount our NFS client under `/tmp/nfs` as following:

```
sudo mount -t nfs -o sync 192.168.1.120:/tmp/nfs /tmp/nfs
```

As pointed out earlier, `192.168.1.120` refers to `thoothukudi-lom`.

4.1.2 Workloads and Results

We use three workloads to compare our network file system with NFS. We measure the total amount of time spent on file creation (`creation`), file open (`open`), file write (`write`), and file close (`close`). The detailed description of workloads and corresponding results follow:

1. Random writes. We create a 100MB file with random bytes in it. Then, we perform 10^5 4KB random writes to the created file. Note that by random writes, we mean the offset of the file is chosen randomly when write. The performance of our network file system and NFS is shown in Figure

As shown in the figure, we observe our networked file system spent more time when open the file (3 seconds vs. almost instant in NFS). This is expected as we download file from remote each time we open a file. In contrast, NFS seems to only create local copy on open when there is no modification to the file yet. There is a dramatic performance difference when come to write. Compared to our network file system, NFS spent around 440 seconds to finish writes. This difference is reasonable as we require NFS to synchronously update the local change with the remote server whereas ours only performs write into local copy. Our network file system also performs well on close due to the synchronization work need to be done by NFS.

2. Sequential writes. In this workload, we first create an empty file. Then we we repeatedly append 4KB data to file (i.e., sequential writes) until the file size reaches 500MB. The performance of two systems is shown in Figure 3.

Since we create an empty file, there is no noticeable difference between NFS and our network file system on `open`. In addition, since the major performance bottleneck `release` is called after `close`, our network file system `close` is done almost immediately. For NFS close case, since the write is already synchronized to the server, there is no much work need to be done. The major difference between two systems under this workload is the `write` part. NFS takes less time finish writing than our networked file system (4660 seconds vs. 4966 seconds). The major overhead from our network file system is the FUSE framework which cross kernel boundary multiple times whereas NFS sits in the kernel space. However, the gap is offset by our design: `fsync` after each write forces NFS to synchronize with the remote server. However, in our system, we only synchronize to the remote on `close`. This is a tradeoff we play when we design our system: we trade file availability with the performance. Unlike NFS, when `fsync` is called, our system only synchronizes writes to local disk storage. In other words, without closing file, there is only one copy sitting on the client side. If client is crashed and un-recoverable, the file is gone. However, for NFS, the file is still safe as it already synchronizes with the server. Another motivation for our design of `fsync` is to reduce conflict in

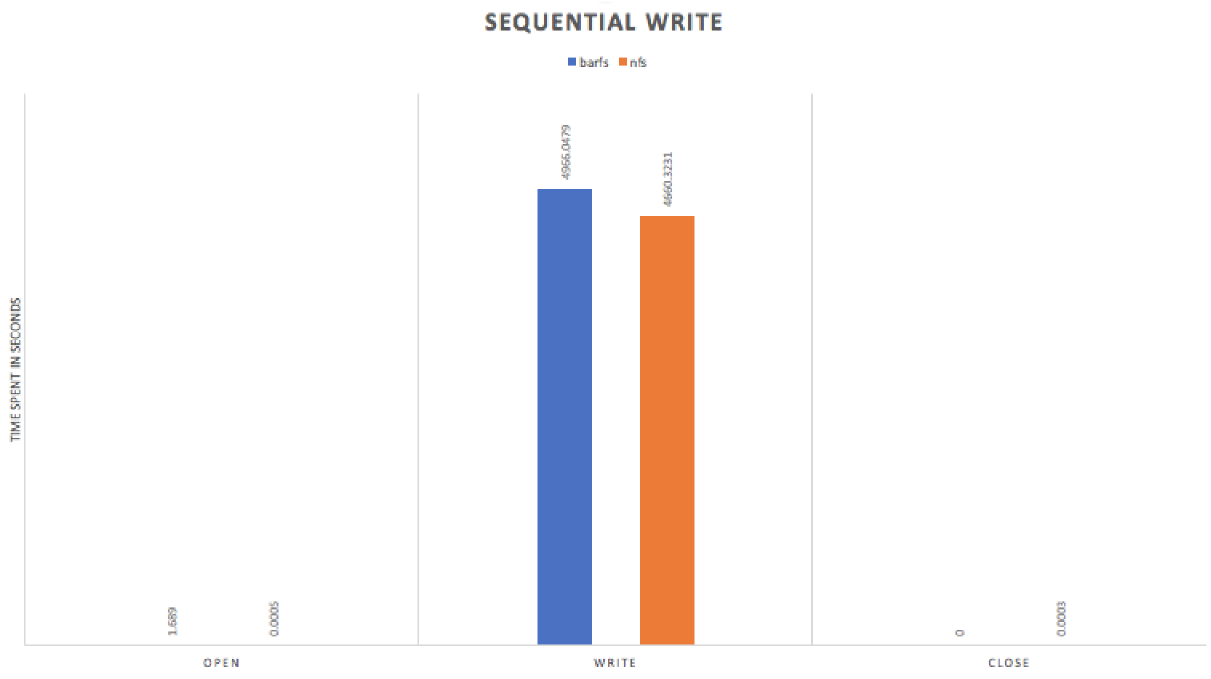


Figure 3: Performance of NFS and our network file system under sequential writes workload

the concurrent writes. we think all the writes before `close` are ”unofficial” and should not become visible to other users by uploading it to the server.

3. File creation. In this workload, we create 100 4KB files. The performance of two systems is shown in Figure 4. As shown by the figure, our system takes much more timer creating file than NFS does (49 seconds vs. almost 0 seconds). This meets our expectation because of overhead imposed by the remote file downloading. However, NFS will only create files in local cache and synchronize changes at the background. When file is closed, NFS will initiate the synchronization to the remote.

As we can see from workloads, our network file system has performance advantage over NFS when write to the file at the sacrifice of potential file loss. However, our network file system performs poorly when `open` operation is involved as remotely downloading file from server is inevitable.

4.1.3 vs. NFS

From the design and actual experiment, our network file system shares great similarities with NFS: both of them are stateless and are designed under the assumption of not many concurrent accesses in a one-server-multiple-clients setting. Both of systems implement last-writer-wins policy.

NFS has advantage in terms of file consistency and expensive remote file downloading is avoided with the help of attibute cache and file cache. Attributes cache periodically checks with remote and ensure the file cache is valid at the best effort (client can still with stale copy if the file is changed during the interval between attribute check). In addition, NFS consistently synchronize local writes to the remote to avoid serious consequence of local client power loss. On the other hand, since we

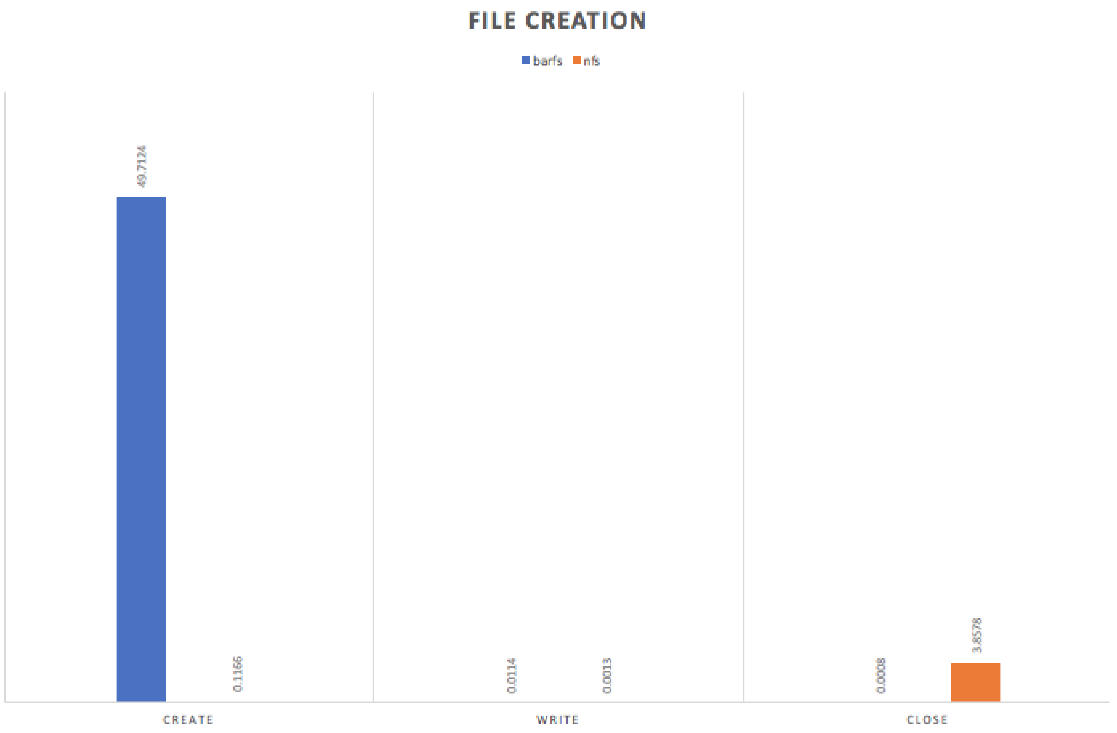


Figure 4: Performance of NFS and our network file system under file creation workload

follow the principle of frugality (use the least powerful solution to a given problem) [12], our network file system can reap some performance gain during read and write. Specifically, we only synchronize local copy with remote on close and work with local cache most of time, we avoid overhead during the basic I/O operations.

5 System Tools Exercise

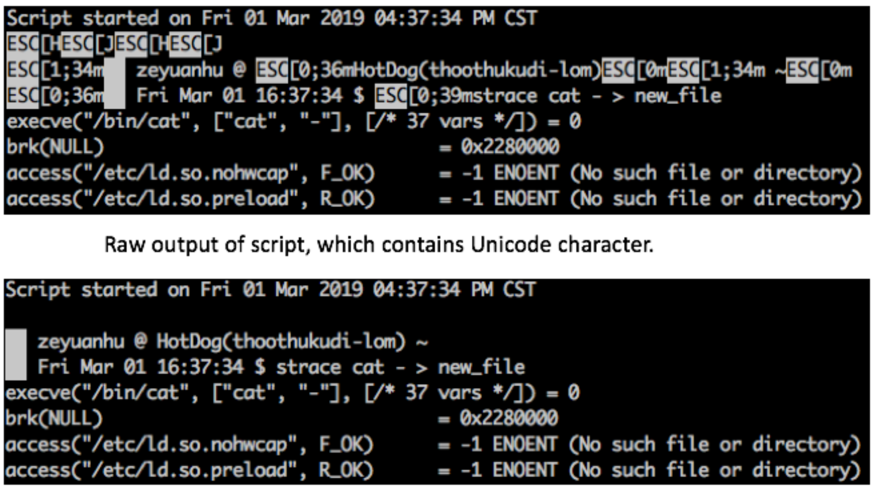
5.1 strace

`script` command allows user to record terminal printout into a file [?]. Per the lab instruction, we use `strace` to trace the syscalls and signals of a target process [?]. In our case, we trace the process involving `cat`. One thing I notice is that `script` contains some unicode as shown in Figure 5. 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 5 [4].

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



```
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 5: `script` output before and after cleanup

5.2 lsof

`lsof` lists all open files [?]. `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
- `/dev/ptmx`: a character file to create a pseudoterminal master

6 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 [?], 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] "Executable and Linkable Format." https://en.wikipedia.org/wiki/Executable_and_Linkable_Format.

[2] "Eexcutable and Linkable Format (ELF)." http://www.skyfree.org/linux/references/ELF_Format.pdf.

[3] "Program Header." http://www.sco.com/developers/gabi/latest/ch5.pheader.html.

[4] "Linux Standard Base Core Specification 3.1." http://refspecs.linuxbase.org/LSB_3.1.1/LSB-Core-generic/LSB-Core-generic.html#PROGHEADER, 2018.

[5] "Acronyms relevant to Executable and Linkable Format (ELF)." https://www.cs.stevens.edu/~jschauma/631/elf.html.

[6] "Linker and libraries guide." https://docs.oracle.com/cd/E19957-01/806-0641/6j9vuqujs/index.html#chapter6-71736.

[7] "How programs get run: Elf binaries." https://lwn.net/Articles/631631/.

[8] "personality(2) - linux man page." http://man7.org/linux/man-pages/man2/personality.2.html.

[9] ".bss." https://en.wikipedia.org/wiki/.bss.

[10] "System v application binary interface." https://software.intel.com/sites/default/files/article/402129/mpx-linux64-abi.pdf.

[11] 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.

[12] H. Massalin and C. Pu, "Threads and input/output in the synthesis kernal," in *ACM SIGOPS Operating Systems Review*, vol. 23, pp. 191–201, ACM, 1989.