CS380L: Lab 2

Hochan Lee

March 2019

1 Objective

The main objective of the lab #2 is to compare in-memory based NFS (Network File System) and FUSE based user space filesystem (UFS); For convenience, I call user-space or user-level filesystem as **UFS**. In case of NFS, file operations generated by clients are sent to a server and the server which receives the requests practically processes them. On the contrary, the proposed FUSE filesystem processes file operations on a client side and sends a processed result file back to a server by using 'scp'. Therefore, this lab mainly aims to understand performance differences according to the processing location and operation characteristics.

2 Hardware/Software Specification

Two UTCS machines are used for this lab: walking-stick and virgo. The walking-stick machine consists of a single socket with Intel Xeon E3-1270 CPUs with 4 cores and 8 hyper-threads. It exploits 16GB of DRAM and 8192K last level cache (LLC) and, transparent huge page (THP) is disabled. In addition, 4-way set associative and 4 entries for 1GB pages, and 4-way set associative and 64 entries for 4KB pages are used for DTLB. Instruction TLB consists of 4KB pages, 8 way set associative, and 64 entries. L2 TLB has 1MB, 4-way set associative, and 64 byte line size. The virgo machine has higher spec than the walking stick. It is composed of a single socket with Intel(R) Core(TM) i7-6700 CPUs with four cores and 8 hyper-threads. Additionally, 16GB of DRAM without THP and 8192K LLC are used. 4 entries and 4-way of DTLB is used for 1GB pages and 64 entries with 4-way of DTLB is used for 4KB pages. Instruction TLB consists of 4KB pages, 8-way set associative, and 64 entries. Lastly, L2 TLB has 1MB, 4-way set associative and 64 byte line size.

The main work of the project is to compare with NFS and FUSE based user file system. To do this, I used 2.9.7 version of FUSE, version 3 NFS, and 7.6 version of SSH and SCP.

3 Prerequisites

In this section, I introduce prerequisites for this lab: tmpfs, ssh, and cache invalidating. Most of them are already set on the UTCS machines, but when I tried to do this lab on my local machines, below information was necessary; correct experiments cannot be done without it. Therefore, I decided to spend spaces for introducing those requirements.

3.1 tmpfs

File systems to be exploited in this lab are in-memory based file system. Therefore, all files must locate in DRAM; the description requires students to manage files on '/tmp'. However, '/tmp' can be reside in either DRAM or storage. **tmpfs** is a temporary file system and resides in DRAM. Therefore, by adding below line to '/etc/fstab' and mounting '/tmp' through tmpfs, '/tmp' can certainly reside in DRAM with 1GB size; It is known as a default option, but I figured out that /tmp of my home machine was on storage.

/etc/fstab

```
tmpfs /tmp tmpfs rw,nodev,nosuid,size=1G 0 0
```

3.2 SSH

To communicate with server, the client UFS keeps performing file transmission operations through scp which exploits a ssh connection. In this case, ssh asks passwords for permission check. However, this process should be removed during UFS operations since it damages transparency of UFS. Therefore, user should set up SSH keys. By using below commands, we can easily configure the public and private keys.

ssh key registration-client side

```
# ssh-keygen -t rsa
# ls  /.ssh/
authorized_keys id_rsa id_rsa.pub known_hosts
# chmod 700  /.ssh
# chmod 600  /.ssh/id_rsa
# chmod 644  /.ssh/id_rsa.pub
# chmod 644  /.ssh/authorized_keys
# chmod 644  /.ssh/known_hosts
# scp  /.ssh/id_rsa.pub [ID]@[SERVER_HOSTNAME]:id_rsa.pub
```

In client side, the machine generates private key (id_rsa) and public key (id_rsa.pub). After setting up authority of each file, id_rsa.pub should be sent to server side. Lastly, in server side, transmitted id_rsa.pub is attached to /.ssh/authorized_keys. From now on, the client machine does not need passwords to communicate with the server.

ssh key registration-server side

```
# cat  /id_rsa.pub >>  /.ssh/authorized_keys
```

3.3 Cache Invalidating

Cache has significant impacts on this lab since file handling is internally memory and storage operations. For fair experiments, data cache is validated for all experiments by using below code; This code is already used by the previous lab.

```
void cache_flush(void) {
  char *tmp = (char *) malloc (L1D_CACHE_SIZE*sizeof(char));
  for (int i = 0; i < L1D_CACHE_SIZE; i++)
    tmp[i] = simplerand();
  free(tmp);
}
```

4 User-level File System (UFS)

4.1 Implementation

4.1.1 System Architecture

The proposed UFS supports 33 file operations including 4 extended file attribute operations. Note that this file system is networked file system. So local client server can process file operations on remote server files. In this case, all files are managed on '/tmp' of both server and client machine in order to accomplish in-memory file system. To enable networking, UFS utilizes **'scp'**. When open() is requested, it downloads the corresponding file from '/tmp' of server. Similarly, when close() is performed, UFS returns back the target file to the '/tmp' of server.

To avoid inconsistency, my UFS client automatically creates 'server_tmp' directory on the mounted directory. Networking operations can be executed only on the **'server_tmp'** directory. This is because the UFS does not mount a remote server directory. FUSE can basically mount a local source directory to a local destination directory. If the UFS supports networking operations on all locations of the mounted directory, there is no reason to mount a local directory; Some users want to treat files of the source directory. That is, user can handle a server's files by downloading and uploading them on 'server_tmp' directory.

Below command is used to mount a directory. Note that user has to specify server host name with account in order to communicate with the server.

```
# ./tmp/dustfs [FUSE OPTIONS] [SOURCE DIRECTORY] [TARET DIRECTORY TO BE MOUNTED] [USER ACCOUNT@HOST NAME]
```

Note that I do not use 'stat' on the server to implement getattr. Instead of using it, I assume that there is an initial file on '/tmp' of server. This is because without mounting, it is hard for user to list server files. Also, it deteriorates meanings of local mounting. To avoid this problem, I made 'server_tmp' directory mentioned above. Initially, it does not provide files which have exist in server. However, after processing file operations including create() and delete(), those files reside in not only server-side, but also client-side. Therefore, from that point, files are synchronized between server and client.

4.1.2 FUSE Calls Implemented

The implemented file system (UFS) named dustfs supports below operations.

struct fuse_operations (1)

```
.init = dust_init,
.statfs = dust_statfs,
.fgetattr = dust_fgetattr,
.getattr = dust_getattr,
.access = dust_access,
.open = dust_open,
.read = dust_read,
.write = dust_write,
.create = dust_create,
.flush = dust_flush,
.utime = dust_utime,
.rename = dust_rename,
.chmod = dust_chmod,
.chown = dust_chown,
.rmdir = dust_rmdir,
.mkdir = dust_mkdir,
.mknod = dust_mknod,
.readdir = dust_readdir,
```

struct fuse_operations (2)

```
.destroy = dust_destroy,
.readlink = dust_readlink,
.unlink = dust_unlink,
.fsync = dust_fsync,
#ifdef HAVE_SYS_XATTR_H
.setxattr = dust_setxattr,
.getxattr = dust_getxattr,
.listxattr = dust_listxattr,
.removexattr = dust_removexattr,
#endif
.truncate = dust_truncate,
.ftruncate = dust_ftruncate,
.fsyncdir = dust_fsyncdir,
.release = dust_release,
.releasedir = dust_releasedir,
.symlink = dust_symlink,
.link = dust_link
```

The proposed UFS named dustfs accepts three arguments: source directory, destination directory to be mounted, and [USERID]@[SERVER_HOSTNAME]. Those arguments are stored and managed by **'struct dust_state* dust_dinfo'** during execution. Among those values, destination path is managed by absolute path, not relative path. Since every tests are done in the destination directory, file system should avoid path inconsistency; it is possible that some users execute the file system in other directory with relative path. To get a real path, dustfs calls realpath() of FUSE.

All user level functions that are mapped through FUSE have similar flow. They internally call system calls and check whether these calls fail or not. If there is an error during the call, these functions return **'errno'** with a minus sign. Before calling the system calls, some functions require full paths. To treat it, I used **get_fullpath()** that attaches target file name to mounted directory path mentioned above.

```
char* get_fullpath(const char *filename) {
    char *fullpath = malloc(sizeof(char)*PAHT_MAX);
    strcpy(fullpath, dust_dinfo->destination_dir);
    strncat(fullpath, path, PATH_MAX);
    return fullpath;
}
```

After mounting the source directory, **dust_init()** automatically creates **'/server_tmp'** directory for communication with server. After constructing full directory path of 'server_tmp' by using strcpy() and strncat(), dust_init() creates this directory with read, write and execution permissions through mkdir().

Whenever a file is opened, the file system calls **dust_open()**. First, this function checks whether the current directory is 'server_tmp' or not. If it is, dust_open() downloads the target file from server's '/tmp' directory by using scp. To execute scp command inside the code, I used **system()**. After constructing a command of scp, dust_open() passes the command to system(). Note that downloaded file must be located under /tmp directory in order to not only maintain in-memory file system characteristics, but also avoids to execute file operations on NFS directory; /tmp of UTCS machines resides in local DRAM.

On the contrary, **dust_flush()** uploads the target file to server's /tmp directory. Whenever an opened file is closed by using close(), dust_flush() is called. As dust_open() does, it constructs scp command which transfers the data and executes it by using system().

```
static int dust_open(path, fi) {
    ...
    if (strstr(path, "server_tmp") != NULL) {
        char *cmd = malloc(sizeof(char)*4096);
        char *fname = strrchr(path, '/')+1;
        strcpy(cmd, "scp ");
        strncat(cmd, dust\_dinfo->server, 4096);
        strncat(cmd, fname, 4096);
        strncat(cmd, " .", sizeof(" ."));
        system(cmd);
    }
    ...
}
```

dust_open(), **dust_opendir()**, and **dust_create()** update a file handle of file data information structure, **struct fuse_file_info**. Therefore, other operations can refer to this information easily. **dust_mknod()** is used to create a special file or a normal file. According to specified file mode, it calls open(), mkfifo(), or mknod() and creates the file. Other function implementation follows following flow: get_fullpath() -> systemcall(). Another important factor of implementation is a return value. If a function returns incorrect value, file system does not work correctly; for example, getattr() function must return '0' or '-errno'. Otherwise, it invokes permission error.

To test the implemented dustfs UFS, I executed below code.

```
int
main() {
   int fd0 = open("foo", O_RDWR);
   int fd1 = open("foo", O_RDONLY);
   char buf[100];
   buf[0] = 9;    buf[1] = 81;
   buf[2] = 'A'; buf[3] = 'q';
   buf[4] = '0';
   int nb0 = write(fd0, buf, 100);
```

```
    int nb1;
    close(fd0);
    nb1 = read(fd1, buf, 100);
    assert(buf[0] == 9); assert(buf[1] == 81);
    assert(buf[2] == 'A'); assert(buf[3] == 'q');
    assert(buf[4] == '0');
    close(fd1);
    printf("Wrote %d, then %d bytes\n", nb0, nb1);
    return 0;
}
```

This test was successfully executed and printed out **"Wrote 100, then 100 bytes"**.

4.1.3 Miscellaneous

I spent 30 hours for the lab. Lab of this class is always interesting and helpful.

4.2 Evaluation

4.2.1 Experiment Design

The main difference between NFS and UFS is file operation processing location. NFS server gets requests through clients' vnode or VFS, and processes them. On the other hand, UFS client handles file operation itself instead of the server. To do this, UFS client always downloads a target file from a server on '/tmp' directory and sends back it to the server's '/tmp' directory. Those different processing flows of NFS and UFS imply that main factors of performance would be two: 1) the number of network communications and 2) cache efficiency; for example, if two client keep trying to access or touch one file on NFS, it would lose cache hit opportunities and performance also would decrease.

To reflect impacts of above factors, following three experiments are designed: 1) changing a target file size: write and read file by increasing exponentially from 1KB to 8MB for write() test and from 1KB to 1GB for read() test; each range is decided by whether it sufficiently shows trends or not, and 2) adding NFS client which executes 'touch', while another client performs file operations, and lastly, 3) executing combination of write and read operations: at first, write a file as 1) case does, and read hundreds character repeatedly.

All the experiments are performed **third times** and before executing the experiments, **cache invalidation** code is carried out. To accomplish in-memory UFS, dustfs manages files on '/tmp' directory of both server and clients. NFS tests are done in UTCS HOME directory. In FUSE experiment, server is **UTCS virgo machine** and client is **UTCS walking-stick machine**. For NFS experiment, two machines are client of the same NFS server.

Experiment 1: Changing target file size and the number of I/O

As a first and a basic experiment, I execute read and write operations respectively by increasing target file size from 1KB to 8MB for write() and from 1KB to 1GB for read(). Each operation unit is 1B. Therefore, as file size grows up, the number of requests also grows up significantly. A motivation of this experiment is to check trade-offs between communication amount and performance: execution time. Since FUSE always downloads and uploads a target file with open() and close(), I expect that UFS will get lower performance than NFS with small file size. However, as file size grows up, overheads of NFS will proportionally increase in order to send requests to the server; FUSE will get performance benefits since it locally processes file operations before turning it back. Below is a pseudo-code used for this experiment.

```
void write_test(long long target_size) {
    int ofp = open("write.output", O_RDWR);
    char c[1] = {'0'};
    if (ofp == NULL) abort();
    for (long long i = 0; i < target_size; i++) write(ofp, c, 1);
    close(ofp);
}
void read_test(long long target_size) {
    char c[1];
    int ofp = open("write.output", O_RDONLY);
    if (ofp == -1) abort();
    for (long long i = 0; i < target_size; i++) read(ofp, c, 1);
    close(ofp);
}
```

Experiment 2: Cache disturbance of NFS clients

One thing to know is that NFS uses write-back cache leases for clients. Therefore, if a client gets a write lease of the cache, server will accumulate read requests from other clients. It implies that one client can affect performance of another client; for example, 'touch' operation will update file access time (write), and this update also invalidates another clients' local cache memory and deteriorate read operations. To check each client's impacts, I executed above experiment 1, read operation part while executing 'touch' command in another client.

Experiment 3: Combination of write and read operations

The final experiment is to understand a case that shows benefits of FUSE based UFS. This experiment is also based on the above experiment 1. In addition to the source code, read intensive code is attached. To be specific, after write a file with specified size, it reads 512 characters one by one. Then, the code reads the characters 4 times in order to increase locality. At the same time, another client of NFS keep touching the files in the current directory. I expect that FUSE based UFS shows better performance than NFS since the number of requests would be extreme for NFS and FUSE also can improve cache performance. Below source code is used.

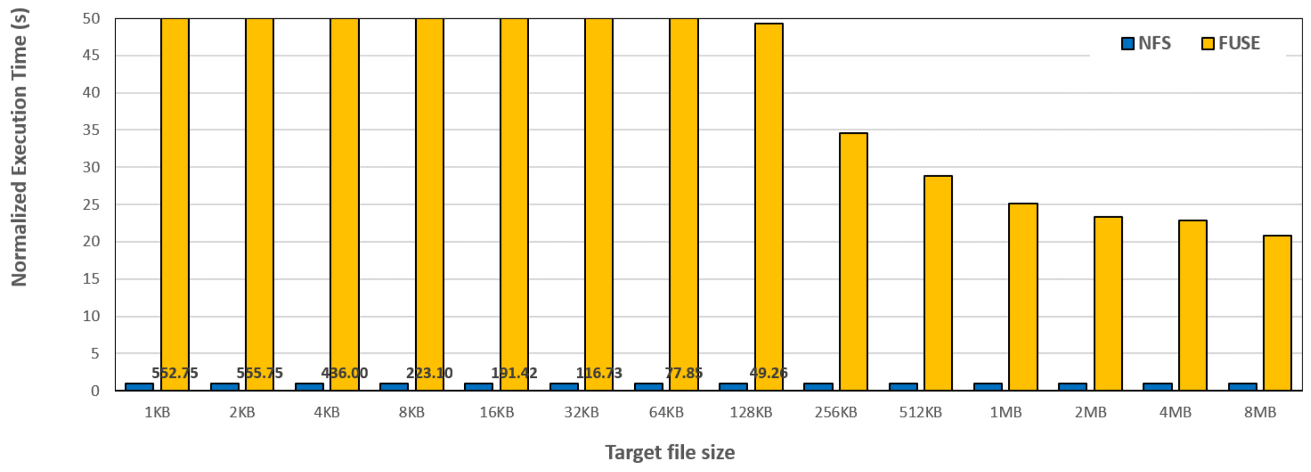
```
void write_read_test(long long target_size, int iter) {
    char buf[512];
    char c[1] = {'*'};
    int ofp = open("write.output", O_RDWR);
    if (ofp == -1) abort();
    for (int i = 0; i < iter; i++)
    for (long long i = 0; i < target_size; i++) write(ofp, c, 1);
    close(ofp);

    // intensive read operations with considering locality
    ofp = open("write.output", O_RDONLY);
    if (ofp == -1) abort();
    for (long long i = 0; i < target_size-512; i++) {
        for (int j = 0; j < 512; j++) read(ofp, buf, 1);
        for (int iter = 0; iter < 4; iter++) {
            volatile char *a;
            char c;
            for (int k = 0; k < 512; k++) a = &buf[k];
        }
    }
    close(ofp);
}
```

4.2.2 Results

This section shows results of experiments explained above. I got similar results that I expected above in all of experiments. First, NFS shows faster execution time with small number of operations regardless of write or read. Second, FUSE based UFS shows better performance than NFS as the number of operations is increased, as well as data reuse ratio is increased. Third, one NFS client's 'touch' to change access time of files has bad impact on another NFS client's performance.

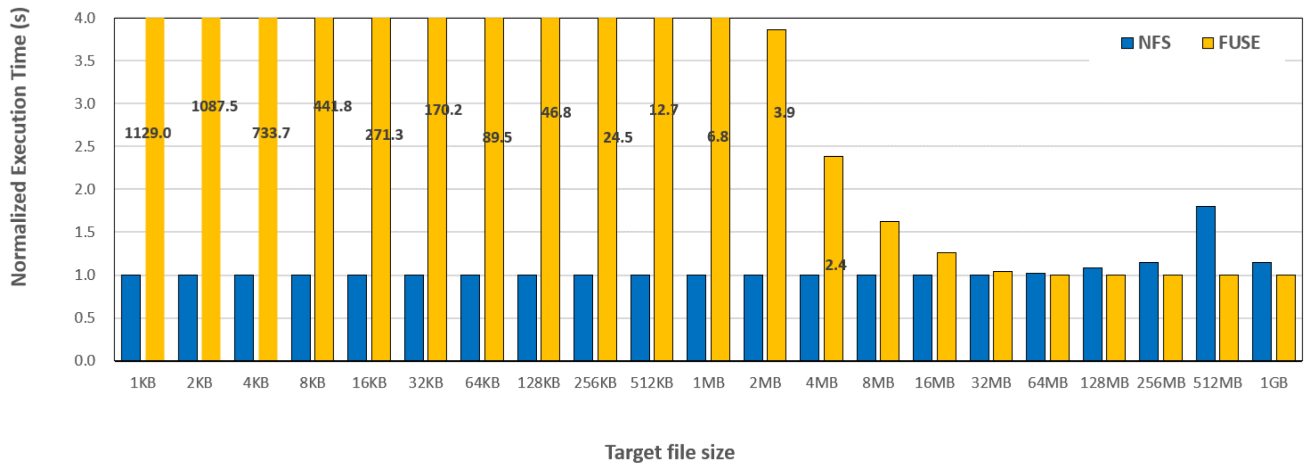
Figure 1 shows results of **Experiment 1**. By increasing a target file size from 1KB to 2GB, writing one byte operation is performed until satisfies the thresholds. Y-axis of the graph is normalized execution time that shows

Figure 1: **Sequential Write Operation Test**



slow down factors compared to the fastest cases. X-axis is target file size. For relatively small size files from 1KB to 128KB, FUSE based UFS got extremely slow downs about from x50 to x553 compared to NFS. This is because UFS executes scp file transfer tasks whenever open() and close() are performed even for small size files. Transferring files is the majority of overheads of UFS. On the contrary, NFS clients only send file operation requests instead of sending entire files. However, as the target file size increased, the performance gap between NFS and FUSE is decreased.

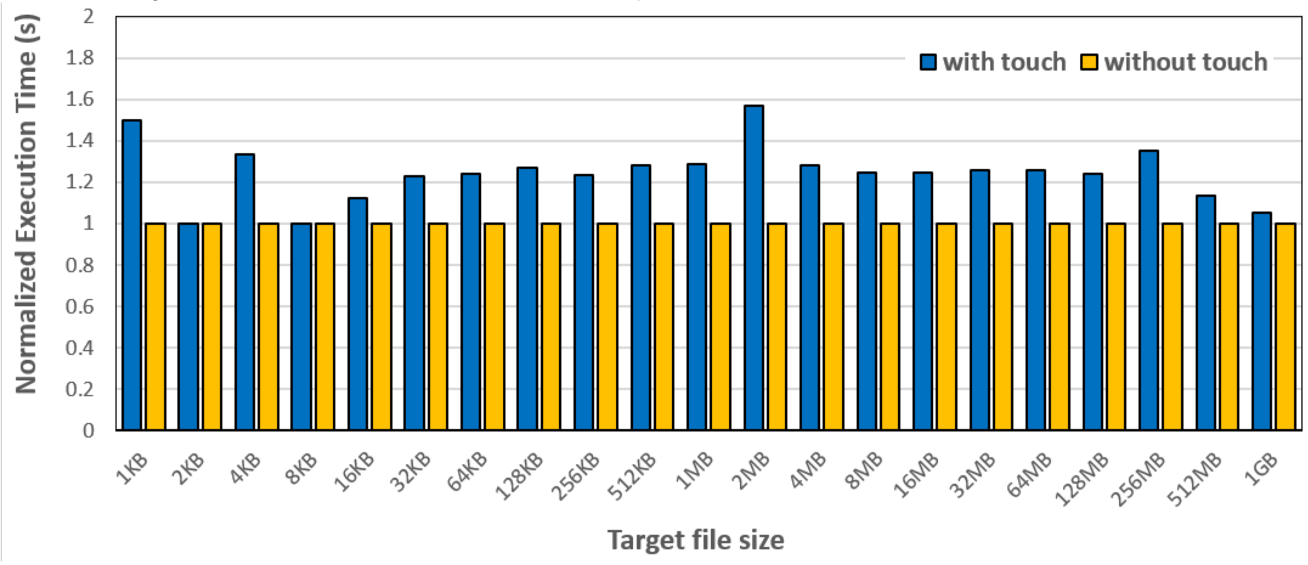
Figure 2: **Sequential Read Operation Test**



Sequential read operation test of the Experiment 1 also shows similar trends that FUSE based UFS shows better performance as the target file size is increased: Figure 2. For small file size from 1KB to 16MB, FUSE is extremely slower than NFS about x1129 to x1.3. However, when the target file size exceeds 64MB, FUSE based UFS shows comparable or better performance than NFS; For example, in 512MB read() test, NFS is slower than UFS about x1.7.

Figure 3 shows experiment results of **Experiment 2** that compare sequential read() performance between with cache invalidation from another client and without it on NFS. The yellow bars show execution time results with a client that keep touching the target file and the blue bars are execution time results without disturbances of another client. The X-axis is normalized execution time. The case with 'touching' disturbance shows about 1.3 slow downs on geomean compared to the case without 'touching'. So to speak, the touching disturbance from a NFS client causes local cache invalidation for another client; the better way to evaluate cache invalidation is to use

Figure 3: **Sequential Read Performance with/without another client cache disturbance in NFS**



perf. However, UTCS machine does not allow students to use perf.

Figure 4: **Sequential Write and Read operation Test**

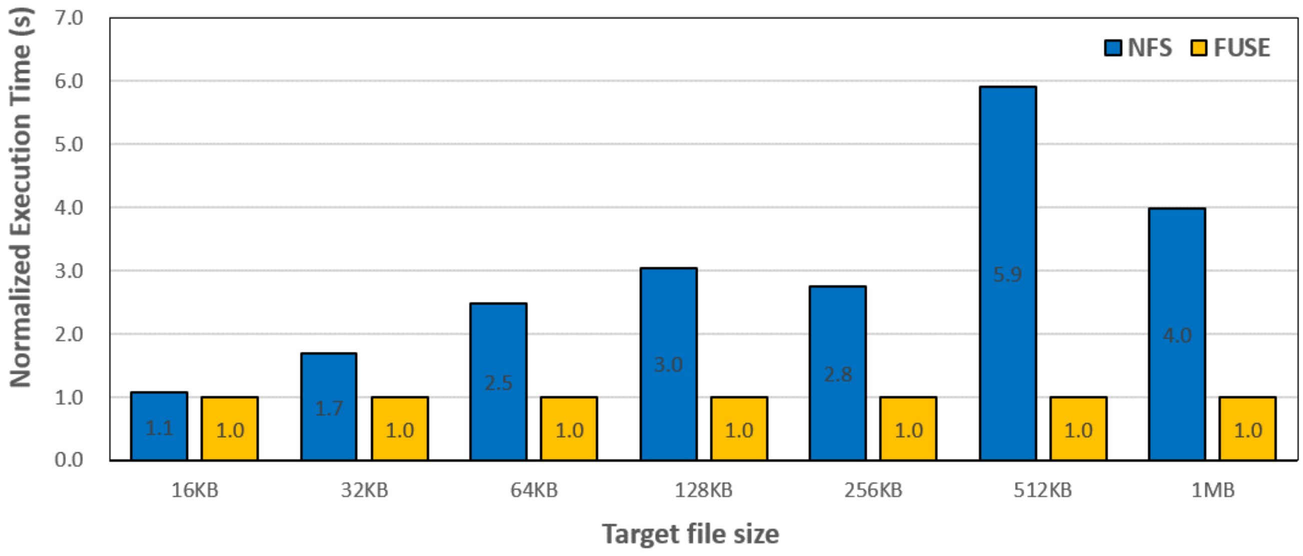


Figure 4 is normalized results of the scenario **Experiment 3** which is intended to design a case that FUSE based UFS shows better performance than NFS. Based on **Experiment 1**'s code, it intentionally performs read operations of the generated file during writing tasks. In this case, in order to improve locality, accessing the same data several time is carried out; In this case, one NFS client starts to touch the target file in order to hinder another client's cache efficiency. In this scenario, FUSE based UFS gets faster execution time about **x3** on geomean. Since even for from 16KB to 1MB target file sizes, UFS shows enoughly better performance, I stopped the experiments. The number of operations and locality intensive operations are beneficial to UFS.

5 System Tools Exercises

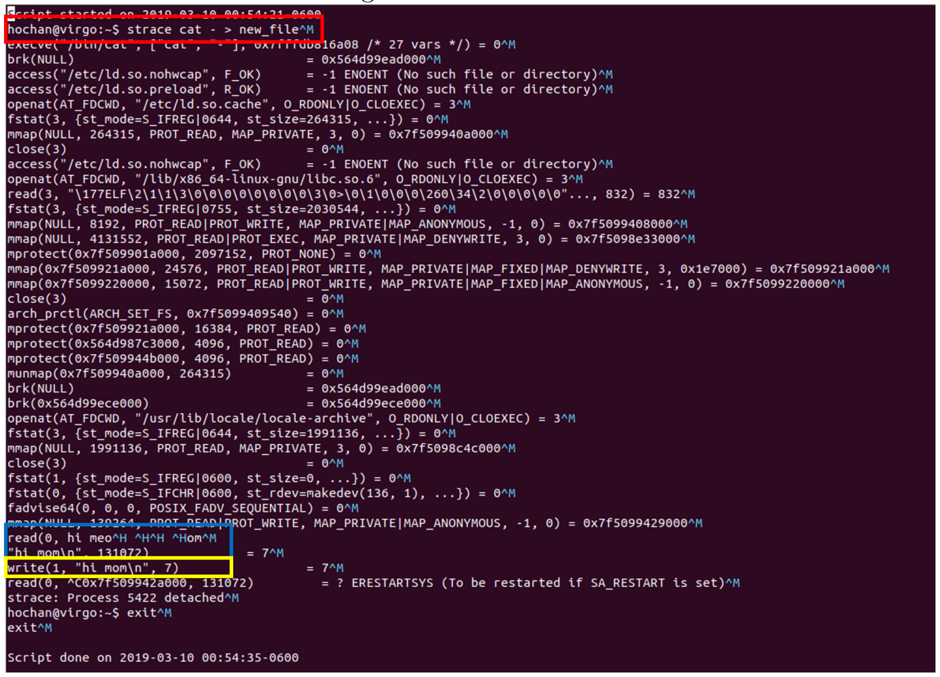
A strace

On the virgo machine, below command is executed.

```
# script session_record
strace cat - > new_file
hi mom
∧Dexit
```

This is first time for me to use this wonderful '**script**' command. 'script' command records all logs of the current session. So, above commands act following this; At first, '$ script session_record' command starts to trace all operations of the current terminal session and record those logs to 'session_record.' To test the command, I typed 'strace cat - > new_file' command which input texts of stdin are passed to 'new_file'; '-' with cat means reading stdin, and, at the same time, records system calls used for this storing operation. Now, user can type any inputs to be written to 'new_file' through stdin. Lastly, 'ctrl+D' or 'exit' stop the scripts session and the save the file. Below Figure 5 shows the result file of 'script' command. I mainly focused on three parts. The red box shows the command that I typed. Then, system calls used to load libraries or file operations are listed. The blue box shows a 'read' system call, 'read(0, ', which waits for input to stdin (fd is 0). Interestingly, after typing 'hi mom' text, read() system call finished. Lastly, this input was stored through write() to stdout (fd is 1) which passes the inputs to 'new_file' file (This file is attached.).

Figure 5: **session_record**



```
Script started on 2019-03-10 00:54:21-0600
hochan@virgo:~$ strace cat - > new_file^M
execve("/bin/cat", ["cat", "-"], 0x7fffdb816a08 /* 27 vars */) = 0^M
brk(NULL)                               = 0x564d99ead000^M
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)^M
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)^M
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3^M
fstat(3, {st_mode=S_IFREG|0644, st_size=264315, ...}) = 0^M
mmap(NULL, 264315, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f509940a000^M
close(3)                                = 0^M
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)^M
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3^M
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\260\34\2\0\0\0\0\0"..., 832) = 832^M
fstat(3, {st_mode=S_IFREG|0755, st_size=2030544, ...}) = 0^M
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f5099408000^M
mmap(NULL, 4131552, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f5098e33000^M
mprotect(0x7f509901a000, 2097152, PROT_NONE) = 0^M
mmap(0x7f509921a000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1e7000) = 0x7f509921a000^M
mmap(0x7f5099220000, 15072, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7f5099220000^M
close(3)                                = 0^M
arch_prctl(ARCH_SET_FS, 0x7f5099409540) = 0^M
mprotect(0x7f509921a000, 16384, PROT_READ) = 0^M
mprotect(0x564d987c3000, 4096, PROT_READ) = 0^M
mprotect(0x7f509944b000, 4096, PROT_READ) = 0^M
munmap(0x7f509940a000, 264315)          = 0^M
brk(NULL)                               = 0x564d99ead000^M
brk(0x564d99ece000)                     = 0x564d99ece000^M
openat(AT_FDCWD, "/usr/lib/locale/locale-archive", O_RDONLY|O_CLOEXEC) = 3^M
fstat(3, {st_mode=S_IFREG|0644, st_size=1991136, ...}) = 0^M
mmap(NULL, 1991136, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f5098c4c000^M
close(3)                                = 0^M
fstat(1, {st_mode=S_IFREG|0600, st_size=0, ...}) = 0^M
fstat(0, {st_mode=S_IFCHR|0600, st_rdev=makedev(136, 1), ...}) = 0^M
fadvise64(0, 0, 0, POSIX_FADV_SEQUENTIAL) = 0^M
mmap(NULL, 139264, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f5099429000^M
read(0, hi meo^H ^H^H ^Hom^M
"hi mom\n", 131072)               = 7^M
write(1, "hi mom\n", 7)                 = 7^M
read(0, ^C0x7f509942a000, 131072)         = ? ERESTARTSYS (To be restarted if SA_RESTART is set)^M
strace: Process 5422 detached^M
hochan@virgo:~$ exit^M
exit^M

Script done on 2019-03-10 00:54:35-0600
```

B lsof

'**lsof**' command shows not only open file lists, but also detail information including process, device and types of files. Therefore, with 'grep /dev' command, it shows only cached /dev information. When I executed 'lsof | grep /dev/null', /dev/null and /dev/pts/0 for bash, lsof and grep are shown.

C Network Tools

Include the name of the interface your machine is using to communicate externally (<eth0>). Questions:

1. Are DHCP messages sent over UDP or TCP?
   Answer: UDP

2. What is the link-layer (e.g., Ethernet) address of your host? (Feel free to obscure the last couple bytes for privacy's sake)
   Answer: 00:0c:29:ce:13:. (** means private bytes)**

3. What is the IP address of your DHCP server?
   Answer: 192.168.181.141. I checked this address in '/var/lib/dhcp/dhclient.leases' directory.

4. What is the purpose of the DHCP release message?
   Answer: The purpose of the DHCP release message is to deallocate the lease of an IP address that is given by the DHCP server.

5. Does the DHCP server issue an acknowledgment of receipt of the client's DHCP request?
   Answer: Yes. When the client sends the DHCPREQUEST message to the DHCP server, it sends receipt, a DHCPACK, to the client and enters its final phase.

6. What would happen if the client's DHCP release message is lost?
   Answer: If the client's DHCP release message is lost, then the client starts to release the IP address, but the server will not assign new IP address to the client until the lease runs out.