Bhargav Desai (bhd20)

Kyle Shin (ks1141)

Joe Yu (jdy28)

SNFS: Simple Network File System

INFORMATIONAL:

To Run after extraction:

1. Run make in each folder (clientSNFS and serverSNFS)
2. For clientSNFS: ./clientSNFS -port [port number] -address [address of server] -mount [mount_path for client]
3. For serverSNFS: ./serverSNFS -port [port number] -mount [mount_path for server]

Note: The misc contains additional test cases that we used to test our code, in addition to the ones provided in this report.

IMPLEMENTATION:

The high level overview of our implementation is very simple. When clientSNFS is run, it creates a FUSE mount point at a user specified path. To use FUSE to vector basic IO file commands to a server, we needed to implement struct fuse_operations as follows:

```
static struct fuse_operations client_oper = {
  .create = client_create,
  .getattr = client_getattr,
  .open = client_open,
  .flush = client_flush,
  .release = client_release,
  .read = client_read,
  .truncate = client_truncate,
  .opendir = client_opendir,
  .mkdir = client_mkdir,
  .readdir = client_readdir,
  .releasedir = client_releasedir,
  .write = client_write,
  .unlink = client_unlink,
  .rmdir = client_rmdir
};
```

Whenever the corresponding system calls open, read, write, etc. are called within the FUSE mounted directory, the functions we implemented are used. In order to pass data from these functions to the server, we used sockets. Every time one of the above functions is called within the FUSE mount, a new socket is opened which connects to the running server, and passes the to the server at a user specified port. The server, which is multi-threaded, will constantly listen and accept new connections. Whenever a new connection is accepted, a new thread is spawned. This thread will detach itself, indicating that it will not need to be joined later. Then, the thread will parse the socket buffer for the corresponding IO function that is implemented on the server. After finishing the request, the thread will write back to the shared socket, returning success or error (errno), along with any other outputs from the called function (e.g: read returns a buffer a data from the file). The thread then terminates afterwards. The client reads from this socket, parses the data, and accordingly returns error or success, along with any data needed.

RPC IMPLEMENTATION:

As described above, our RPC was very low-level, as we passed raw data over sockets. Below are the important structs and functions we needed to use in order to create working sockets:

int server_fd = socket(AF_INET, SOCK_STREAM, 0) : creates an IPV4 socket with reliable two-way byte streams and returns the file descriptor for it

struct sockaddr_in address : this struct allows us to fill in information about our IP address, and is used later to bind the IP address to a socket
int ret = setsockopt(server_fd, SOL_SOCKET, SO_REUSEPORT, &optval, sizeof(optval)) : allows us to set attributes for a socket. In this case, we set the socket to allow for port reuse

int ret = bind(server_fd, (struct sockaddr *) &address, sizeof(address)) : binds an IP address to a socket using the struct sockaddr

int ret = listen(server_fd, 10) : will mark the socket as a passive one that will use accept for new connections and sets the queue size for waiting connections to 10

int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen) : accepts a connection on a socket

Used the below functions to prevent potential Endianness between the clients and server:

```
uint32_t htonl(uint32_t hostlong);

uint16_t htons(uint16_t hostshort);

uint32_t ntohl(uint32_t netlong);

uint16_t ntohs(uint16_t netshort);
```

In the client, these two functions were necessary:

int getaddrinfo(const char *node, const char *service, const struct addrinfo *hints, struct addrinfo

**res) : allows URL host names to be resolved as IPV4 addresses

int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen) : allows client to connect to an address at a specific socket, at which the server and client can pass data between each other by using the basic write functions read/write on the socket file descriptor.

Passing structs across sockets is quite difficult. As a result, we converted RPC calls into strings. Parameters in each string were separated by a unique token that only the server and client have. As a result, whenever the the server/client reads from their shared socket, they are able to parse the string for a token, which they both know. As a result, arguments can be separated and function calls can be identified. For every RPC call, the client puts the name of the function to be called on the server side first, and appends any additional arguments.

FUSE/CLIENT/SERVER RELATION

As mentioned earlier, the FUSE userspace filesystem causes certain necessary function calls for filesystems to be vectored into our code. As a result, we can specify these implementations to be RPC's to the server for the same function at the server level. Below are explanations as to how each function was implemented, and its role:

static int client_getattr(const char *path, struct stat *st) : For the specified path from the mount directory (which is "/"), we must fill in the corresponding stat structure for it. As a result, a RPC is made to the server as explained above. The server reads the path, and create a stat structure and fills it. If stat fails, errno is set, and passed to the client. Otherwise, we can cast the struct stat to a void* and pass its size (144 bytes), back to the client. The client reads from the socket, checks for error, and casts the void* structure back to a stat*. Some fields have to be removed from stat, such as inode number, device number, rdevice number, and block size, as they are OS specific references to the actual file on the server. Getattr is called anytime the filesystem needs to check for the existence of a file, and proceeds every other call. Access levels are checked.

static int client_create(const char *path, mode_t mode, struct fuse_file_info *fi) : Our socket RPC implementation is used again, except now we must also pass the mode to the server as well. The server parses through this information, and uses the function:

int creat(const char *pathname, mode_t mode)

to create the file on the server side. The server passes back the open file descriptor that the creat function returned. Again, errno is passed back to the client on error. The client parses the information the server wrote to the socket, and fills in the struct fuse_file_info fi→fh field. This field can be filled in with the open file handle, in other words, the file descriptor. By doing this, the file descriptor for the file we have just opened (which is specific to the server), can be stored by FUSE and be used by the read, write, flush, and release functions.

static int client_open(const char *path, struct fuse_file_info *fi) : This function behaves the same way as create, except creating a file is not necessary, and the referred path is opened on the server. The file descriptor or errno is returned back to the client, who store the file descriptor in the fi→fh field.

static int client_read(const char *path, char *buffer, size_t size, off_t offset, struct fuse_file_info *fi)
static int client_write(const char *path, char *buffer, size_t size, off_t offset, struct fuse_file_info *fi) : Read and write are implemented nearly the same. Both need to pass on the size of the data request, and the offset in the file. We do not use the path field as we have the stored the file descriptor to an already open file on the server with the fi→fh field. For write, the buffer will contain data that needs to be written, as a result, this information is also passed to the server. For read, the server must read the specific amount of bytes (size), and return the information to the client, who will fill the buffer field. This is done on the server using pread and pwrite, which allow us to specify an offset and size.

static int client_flush(const char *path, struct fuse_file_info *fi): Flush the contents of the file descriptor held in fi→fh if the fi→flush flag is set to true. Otherwise, it is not necessary to flush the contents to disk. If we must flush, the file descriptor is passed to the server, who will call the fsync() function to save the contents of the file to disk. Called in conjunction with release whenever a file is closed

static int client_release(const char *path, struct fuse_file_info *fi) : The file descriptor open on the server is closed. The client passes the stored fi→fh flag to the server, who then finally closes the file descriptor and tells the client if it was successful or not.

static int client_opendir(const char *path, struct fuse_file_info *fi) : Similar to open but for directories. The path is passed to the server, who uses the opendir() function to open the directory. A DIR* stream is created, or NULL, representing an error. Once the directory is open, we can use the dirfd() function to ascertain the file descriptor for the directory. This is then returned to the client, who fills in the fi→fh flag.

static int client_readdir(const char *path, void *buffer, fuse_fill_dir_t filler, off_t offset, struct fuse_file_info *fi) : Readdir will pass the file descriptor of the directory, fi→fh, to the server. The server will use readdir(), on a DIR* stream created by using fdopendir(). Note that fdopendir does not open the directory, as it is already opened, it just returns the associated DIR* stream. Then, using the function readdir() on the server, we can traverse through the directory and create a buffer that contains the name of each of the files in the directory. We do this by using a struct dirent and accessing the d_name field. All of the directory names are passed to the client, who fills the fuse_fill_dir_t and buffer fields with this information.

static int client_releasedir(const char *path, struct fuse_file_info *fi) : Same as release(), except for the directory. We again use fdopendir() to get the corresponding DIR* stream, and close it using closedir().

static int client_mkdir(const char *path, mode_t mode) : Similar to create, except the mkdir function is called instead.

static int client_truncate(const char *path, off_t length) : Pass the path and offset to the server, which will call the truncate() function with the same arguments.

static int client_unlink(const char* path) : Passes the path to the server, and calls the same function unlink() with the same parameters. Used to remove a file

static int client_rmdir(const char* path) : Removes the directory associated with the path by passing the path to the server, and calling the rmdir() function

TEST CASE

Here are some common Linux command line functions that and the respective client methods that are called. Getattr() is used for everything and will not be listed:

ls : opendir(), readdir(), releasedir()

cat > hello_world.txt   : create() /open(), write(), flush(), release()

echo “hello” > hello_world.txt: truncate(), open(), write(), flush(), release()

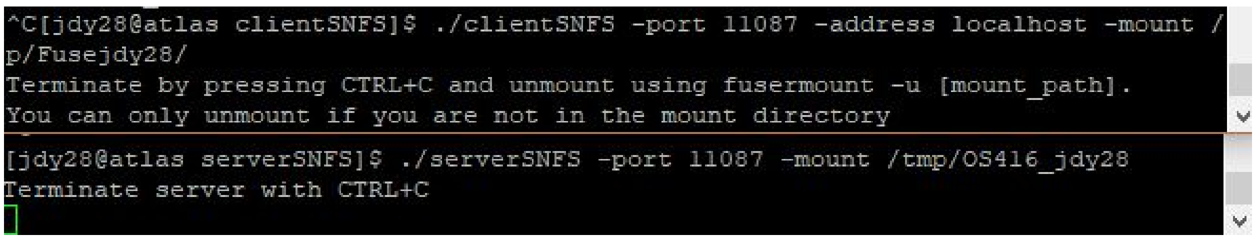
cat hello_world.txt : open(), read(), release()

mkdir newDir : mkdir(), opendir(), readdir(), releasedir()

rm -r newDir : rmdir()

rm hello_world.txt : unlink()

Below are test cases:

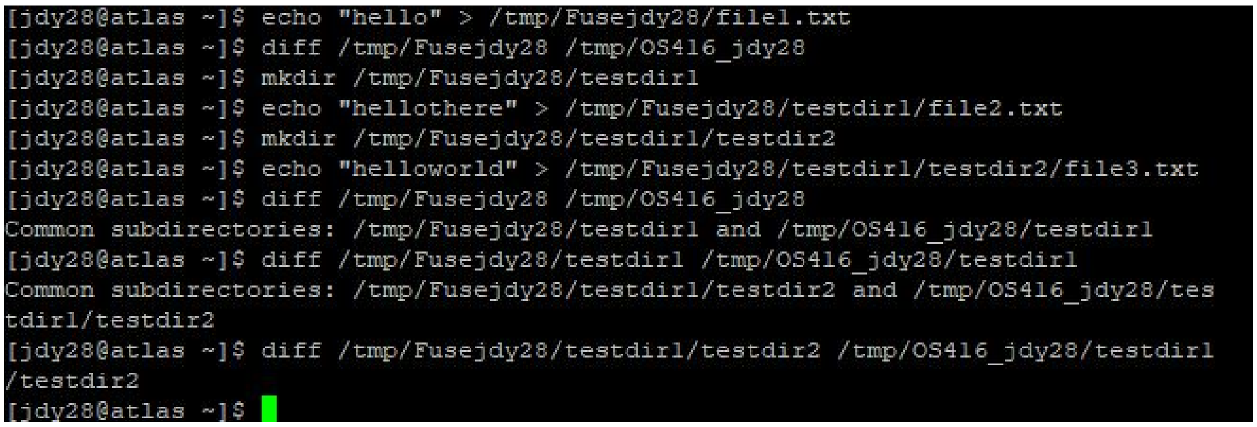


```
^C[jdy28@atlas clientSNFS]$ ./clientSNFS -port 11087 -address localhost -mount /
p/Fusejdy28/
Terminate by pressing CTRL+C and unmount using fusermount -u [mount_path].
You can only unmount if you are not in the mount directory
```

```
[jdy28@atlas serverSNFS]$ ./serverSNFS -port 11087 -mount /tmp/OS416_jdy28
Terminate server with CTRL+C
```

Fuse directory: /tmp/Fusejdy28/

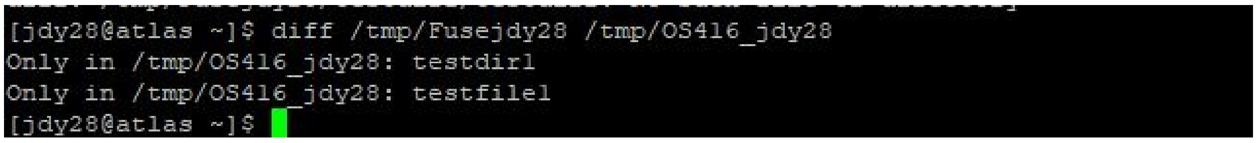
Server directory: /tmp/OS416_jdy28/



```
[jdy28@atlas ~]$ echo "hello" > /tmp/Fusejdy28/file1.txt
[jdy28@atlas ~]$ diff /tmp/Fusejdy28 /tmp/OS416_jdy28
[jdy28@atlas ~]$ mkdir /tmp/Fusejdy28/testdir1
[jdy28@atlas ~]$ echo "hellothere" > /tmp/Fusejdy28/testdir1/file2.txt
[jdy28@atlas ~]$ mkdir /tmp/Fusejdy28/testdir1/testdir2
[jdy28@atlas ~]$ echo "helloworld" > /tmp/Fusejdy28/testdir1/testdir2/file3.txt
[jdy28@atlas ~]$ diff /tmp/Fusejdy28 /tmp/OS416_jdy28
Common subdirectories: /tmp/Fusejdy28/testdir1 and /tmp/OS416_jdy28/testdir1
[jdy28@atlas ~]$ diff /tmp/Fusejdy28/testdir1 /tmp/OS416_jdy28/testdir1
Common subdirectories: /tmp/Fusejdy28/testdir1/testdir2 and /tmp/OS416_jdy28/tes
tdir1/testdir2
[jdy28@atlas ~]$ diff /tmp/Fusejdy28/testdir1/testdir2 /tmp/OS416_jdy28/testdir1
/testdir2
[jdy28@atlas ~]$
```

By the third file creation, there is a file "testfile1" in fuse directory, directory "testdir1" with file "testfile2" and directory "testdir2", and directory "testdir2" has file "testfile3". When we look at differences between each step, we see that all files are the same and all directories are the same.

On server and client termination, we see:



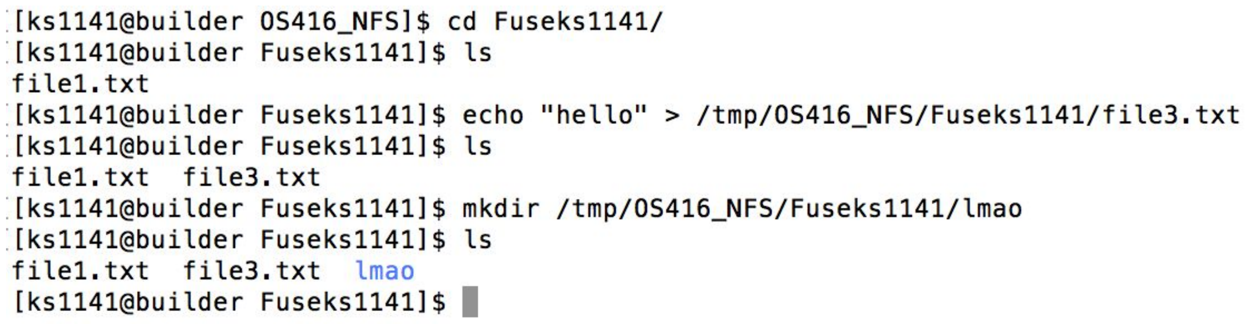
```
[jdy28@atlas ~]$ diff /tmp/Fusejdy28 /tmp/OS416_jdy28
Only in /tmp/OS416_jdy28: testdir1
Only in /tmp/OS416_jdy28: testfile1
[jdy28@atlas ~]$
```

This indicates that when the programs are terminated, the fuse directory does not store the files and directories, but the server program maintains its changes in disk.

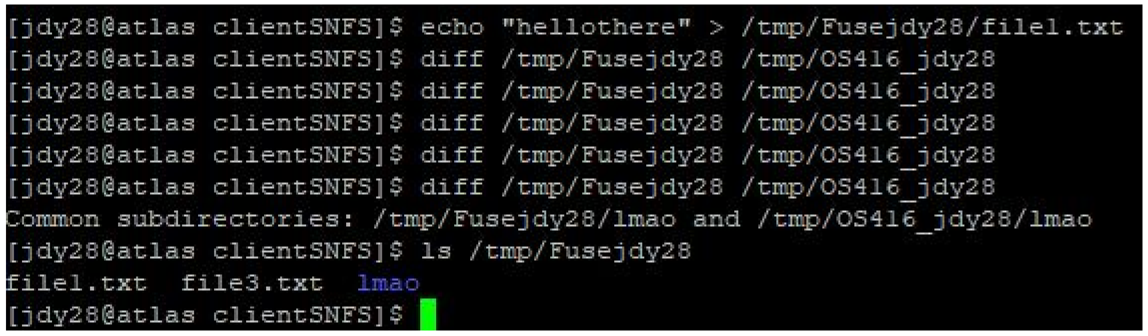
Another case we tested is if we had two instances of the client running at the same time and making changes to their respective directories.

Client 1 fuse directory: /tmp/Fusejdy28/

Client 2 fuse directory: /tmp/OS416_NFS/Fuseks1141/



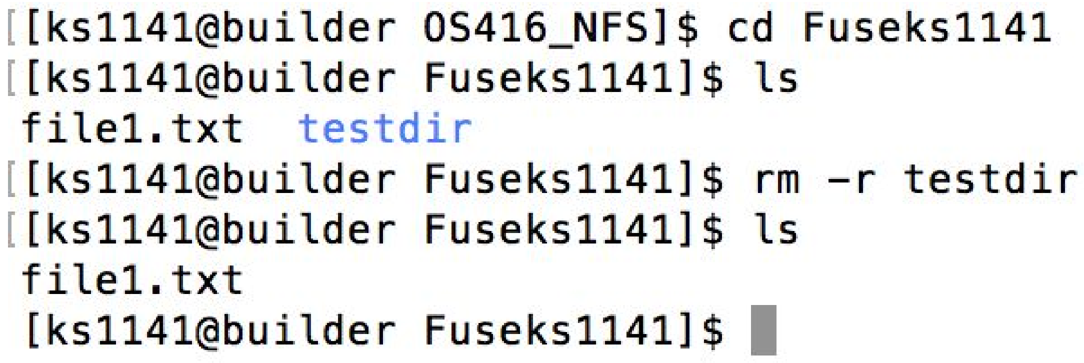
```
[ks1141@builder OS416_NFS]$ cd Fuseks1141/
[ks1141@builder Fuseks1141]$ ls
file1.txt
[ks1141@builder Fuseks1141]$ echo "hello" > /tmp/OS416_NFS/Fuseks1141/file3.txt
[ks1141@builder Fuseks1141]$ ls
file1.txt  file3.txt
[ks1141@builder Fuseks1141]$ mkdir /tmp/OS416_NFS/Fuseks1141/lmao
[ks1141@builder Fuseks1141]$ ls
file1.txt  file3.txt  lmao
[ks1141@builder Fuseks1141]$
```



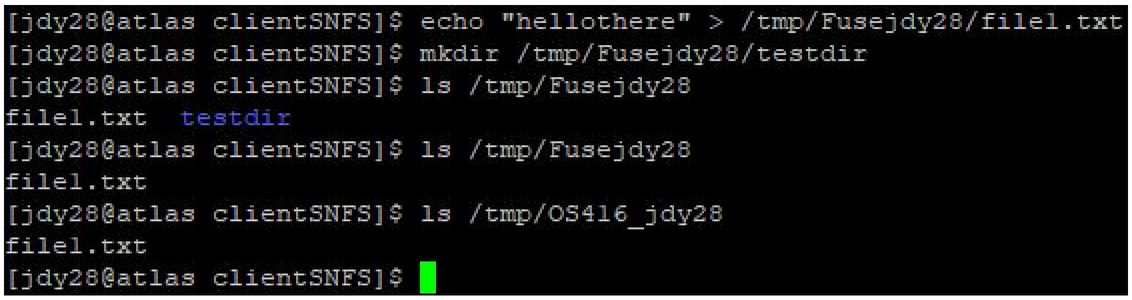
```
[jdy28@atlas clientSNFS]$ echo "hellothere" > /tmp/Fusejdy28/file1.txt
[jdy28@atlas clientSNFS]$ diff /tmp/Fusejdy28 /tmp/OS416_jdy28
[jdy28@atlas clientSNFS]$ diff /tmp/Fusejdy28 /tmp/OS416_jdy28
[jdy28@atlas clientSNFS]$ diff /tmp/Fusejdy28 /tmp/OS416_jdy28
[jdy28@atlas clientSNFS]$ diff /tmp/Fusejdy28 /tmp/OS416_jdy28
[jdy28@atlas clientSNFS]$ diff /tmp/Fusejdy28 /tmp/OS416_jdy28
Common subdirectories: /tmp/Fusejdy28/lmao and /tmp/OS416_jdy28/lmao
[jdy28@atlas clientSNFS]$ ls /tmp/Fusejdy28
file1.txt  file3.txt  lmao
[jdy28@atlas clientSNFS]$
```

We both put in text files into our respective directories as well as a new folder in one of the directories. The results synched up within our directories.

We also implemented directory removal.



```
[[ks1141@builder OS416_NFS]$ cd Fuseks1141
[[ks1141@builder Fuseks1141]$ ls
 file1.txt  testdir
[[ks1141@builder Fuseks1141]$ rm -r testdir
[[ks1141@builder Fuseks1141]$ ls
 file1.txt
 [ks1141@builder Fuseks1141]$ 
```



```
[jdy28@atlas clientSNFS]$ echo "hellothere" > /tmp/Fusejdy28/file1.txt
[jdy28@atlas clientSNFS]$ mkdir /tmp/Fusejdy28/testdir
[jdy28@atlas clientSNFS]$ ls /tmp/Fusejdy28
file1.txt  testdir
[jdy28@atlas clientSNFS]$ ls /tmp/Fusejdy28
file1.txt
[jdy28@atlas clientSNFS]$ ls /tmp/OS416_jdy28
file1.txt
[jdy28@atlas clientSNFS]$ 
```

In this case, removing the directory from one client reflected in being removed from another client as well.

CHALLENGES/DIFFICULTIES

The goal of this project was pretty simple compared to the previous projects. We knew that or the function implementations for the client, the same corresponding would have to be called on the server. However, the main challenge with this project was understanding how the FUSE filesystem actually worked. We were not sure how to implement some functions, such as getattr(), since the documentation on the FUSE website was not descriptive. For example, initially, we did not know what the purpose of the struct fuse_file_info was, and ignored it. However, later we realized that it was necessary to have that struct, as we had to fill it with the file descriptor whenever opening a file or directory. Additionally, even Google at times did not have a solution to the errors. Educated guessing and long hours of research was the resolution.

Additionally, network sockets are always tricky to work with. We had trouble dealing with passing structs across sockets, as a result, we converted everything to char* strings instead. This task was tedious and long, but worked well. All in all, this project was very long, but interesting and awesome to see when it works.