Lab 2: Simple NFS
Due Tuesday, May 28, 2019, 11:59 p.m. PDT

Overview

In this lab, you will be writing fragments of client and server implementations of the Simple NFS protocol: a simplified version of SUN's NFS protocol; *SiN*FS is also an acceptable stylization. You will be programming in C.

An SNFS server listens on port 2048 (one less than the default NFS port, or `NFS--`) for connections from the client. The client listens for file system operations on the host machine. When a user on the machine performs a file system operation, the client connects to the server via TCP and sends binary requests to the server corresponding to the user's action. The server interprets the request, forms a corresponding reply, and sends it to the client. The client interprets the reply and responds to the user appropriately.

The Simple NFS protocol borrows liberally from both the NFSv1 protocol and the NFSv3 protocol. It is as stateless as the NFS protocol. In its current incarnation, the server supports the following operations:

- **`NOOP() -> noop`**

  Does nothing but reply to the client.

- **`MOUNT() -> fhandle`**

  This is the first message sent by a connecting client. The server returns the root fhandle.

- **`GETATTR(fhandle) -> fattr`**

  Returns the file attributes for the file referred to by the handle.

- **`LOOKUP(fhandle dir, filename name) -> (fhandle, fattr)`**

  Looks up the file named `name` in the directory referred to by the handle `dir` and returns the fhandle and attributes for the file if it is found.

- **`READDIR(fhandle dir, int count) -> (uint, entry[])`**

  Returns a list of at most `count` directory entries in the directory referred to by the handle `dir`.

- **`READ(fhandle file, int offset, uint count) -> (uint, bool, data[])`**

  Reads at most `count` bytes beginning at offset `offset` for the file referred to by the handle `file`. Returns the actual number of bytes read, a boolean indicating whether the `EOF` was reached, and the data read.

- **`WRITE(fhandle file, int offset, uint count, data[]) -> (uint)`**

  Writes `count` bytes of `data` beginning at offset `offset` for the file referred to by the handle `file`. Return the number of bytes actually written.

- `SETATTR(uint which, mode, uid, gid, size, atime, mtime) -> (uint)`

  Sets the file attributes for the attributes flagged in the `which` bitfield. Returns a bitfield indicating which file attributes were actually set.

The server and client implementations make use of several third-party libraries: **1)** Berkeley DB, a persistent and in-memory key-value store, **2)** nanomsg, a socket library, and **3)** FUSE: a loadable kernel-module for Unix-like operating systems that enables creation of user-level file systems. Programming against these libraries can be complicated, so we've provided abstractions on top of these libraries where possible. For example, we've provided `send_request` and `send_reply` functions for you to use when sending requests in client code and sending replies in server code that internally use nanomsg.

Berkeley DB is used on the server to store persistent forward and reverse mappings between randomly generated file handles and file names. The use of file names is an unfortunate consequence of writing the server in user space where there are no inode operations like `nametoi`. You will implement the `char *get_file(fhandle handle)` and `fhandle name_find_or_insert(const char *filename)` functions to enable simpler interactions with Berkeley DB. FUSE is used on the client to implement the file system.

The lab is divided into 7 phases and an additional extra credit phase. Each phase guides you towards implementing a part of the SNFS server, client, or both. After completing the lab, you will have written a working NFS-like client and server. There are unit tests corresponding to each phase, allowing you to mark your progress as you work through the lab. You may find the list of references at the bottom of the page useful as you proceed.

Phase 0: Getting Started

In order to minimize compatibility issues with the various libraries, we have provided you with a virtual machine on which you can develop the lab. We **highly recommend** that you develop the lab on the provided virtual machine. You may also use your own machine if it runs Linux or Mac OS X, but your code must compile and pass tests on the virtual machine: your lab will be graded on the virtual machine. See the notes section for instructions on how to set up a local development environment. Note that we are unable to provide support for local development environments, and cannot guarantee their functionality.

To start the virtual machine, your machine must meet these requirements:

- Run a supported OS: Windows, OS X, Linux
- Run a 64-bit variant of the OS
- Have at least 1GB of memory

Most computers released in the last ~10 years will meet these requirements. If you do not have access to a machine that meets these requirements, please contact the course staff and we'll find a solution.

Installing the prerequisite software

To run the virtual machine, you will need to install two pieces of software:

- Virtualbox for running the VM
- Vagrant for simple VM configuration

If you're on Windows, you'll also need to install Git for Windows. We recommend that you select "Use Git from the Windows Command Prompt" and "Checkout as-is, commit Unix-style line endings". Not selecting the latter will likely result in compilation errors later.

Ensure you've installed these software packages before continuing.

Getting the skeleton code and setting up the VM

To begin, clone the lab 2 skeleton git repository to your development machine:

```
git clone https://web.stanford.edu/class/cs240/lab2-skeleton.git lab2
```

To initialize the virtual machine, run the following command inside the `lab2` directory:

```
vagrant up
```

This will take a few minutes to complete. It will download the virtual machine machine, use VirtualBox to initialize it, and run a provisioning script to set up the development environment. When the virtual machine is ready, you may SSH into it be running the following command:

```
vagrant ssh
```

To put the VM to sleep, you can run `vangrant suspend`. `vagrant up` brings the machine back up. When you are done with the lab and want to reclaim your resources, run `vagrant destroy`. You must be in the `lab2` directory when you run these commands.

When the virtual machine has finished provisioning, SSH into it by running `vagrant ssh` while in the `lab2` directory. You will notice that there is a `lab2` directory in the VM's home directory as well. This directory mirrors the `lab2` directory on your local machine. Any changes you make to a file in either the VM or your local machine inside this directory is immediately reflected on the other machine. This means that you can use your local editor to make changes to files and then run tests on the VM. For the remainder of the lab, you should run commands in the VM.

Now, compile the third-party libraries. While still SSHd into the virtual machine, run `make` inside of the VM's `lab2` directory. This should only take a minute or two, depending on your machine's specs. You only have to compile the third-party libraries once. Every other `make` will use your already compiled libraries, and `make clean` won't remove them. You can read on while `make` does its thing.

Exploring the skeleton code

The code is split up into three directories in `src`. The first two, `server/` and `client/` contain the server and client code, respectively. The third, `common/`, contains code that is shared between the server and the client.

The `server/` directory contains the following files:

- `main.c` processes command line arguments and starts the server
- `handlers.c` contains the functions that are called when a request is received
- `fhandledb.c` contains the code to manage, put, and get from the database

The `client/` directory contains the following files:

- `main.c` processes command line arguments and starts FUSE
- `fuseops.c` contains the implementations of all of the FUSE callbacks
- `request.c` provides helper functions for sending requests to the server

The `common/` directory contains the following files:

- `comm.c` provides wrappers around the nanomsg API
- `strings.c` contains functions that return string representations of enums

There is also a `src/main.c` file. It provides a common starting point for both the server and client's main functions.

The `include/` directory contains the header files for these `src/` files where applicable and are named like their `src/` counterpart. `include/utils.h` defines the following helpful macros (among others):

- `debug(...)` like `printf`, but only prints if there is a `#define DEBUG` at the top of the file
- `assert(cond)` checks that `cond` evaluates to true. If it does not, prints an error message and terminates the program
- `err_exit(...)` like `printf` but terminates the program after printing the error message

Finally, the `test/` directory contains the unit testing code. The tests themselves are in `test/src/phase{n}_tests.c`. If a test you expect to pass fails, you can look at its test file to see why. Ensure that the **client and server are not running** before running `make test`.

Make targets

You will use `make` to build, test, and create a submission file for this lab. The following targets are defined in the `Makefile`:

- `all`: compiles the server and client binaries
- `client`: compiles just the client binary
- `server`: compiles just the server binary
- `test`: runs the unit tests against your library
- `submission`: creates the `lab2.tar.gz` submission file
- `clean`: deletes server and client files that can be remade
- `superclean`: deletes all files that can be remade, including libraries

Calling `make` in your shell will implicitly run the `all` target. The other targets can be invoked using `make {target}`. Ensure that you are in the root of your lab directory before invoking `make`. Try calling `make test` inside the VM now. If this doesn't work, your machine may not meet the requirements.

Troubleshooting

FUSE, Berkeley DB, and nanomsg are complex pieces of software. As a result, you may run into issues with these libraries. We've provided a few troubleshooting steps in the notes section at the bottom of this lab text. You may wish to consult it if you run into unexpected issues. If your issue is not resolved there, please inform us.

Client and server binaries

When `make` has completed successfully, you can call the server and client binaries by running `./bin/client` and `sudo ./bin/server`. Running these without any arguments will cause the binaries to print a usage message. Note that `sudo` is required to execute the server binary: the server calls the privileged chroot function.

Try running `sudo ./bin/server -v /snfs/serve`. You should see:

```
$ sudo ./bin/server -v /snfs/serve/
SNFS serving '/snfs/serve' on tcp://*:2048...
```

This message indicates that the server has started and is waiting for client requests over TCP on port 2048. With the server still running, try running `./bin/client -dv tcp://localhost:2048 /snfs/mount/`. You should see:

```
$ ./bin/client -dv tcp://localhost:2048 /snfs/mount/
SNFS mounting at '/snfs/mount/'.
... FUSE output ...
ERROR: Server mount failed.
```

When the client binary begins, it attempts to retrieve the root file handle from the SNFS server at the URL provided in the command line arguments. If the mount succeeds, the binary invokes the FUSE main function with the mount directory. FUSE will then attempt to take over control of all file system operations referencing the provided directory and forward the operations to the calling binary, in this case, the client binary. In this way, FUSE allows a user-level program to implement a file system.

Once you implement the client/server `MOUNT` operation, the mount operation should succeed. Note that the `-v` flags to both the client and server are for "verbose", which cause the binaries to print extra information. The `-d` flag for `./bin/client` instructs FUSE not to daemonize your program and to run in debug mode. In this mode, FUSE will keep your binary in the foreground so that you can see

the client's printout. Pressing `^C` will unmount your file system. Without passing `-d`, you can unmount your file system by calling `fusermount -u /snfs/mount` or `fusermount -uz /snfs/mount` on Linux and `umount /snfs/mount` on OS X. We recommend that you use the `-d` flag during development.

Getting familiar

Before you continue, read through the `src/` files and some of the unit test files (`test/src/phase{n}_tests.c`) to see how the functions you will write (marked `FIXME`) are intended to be used and what we will test for. Also pay attention to the code style: how variables are named, how code is aligned, and how spacing is used. You should write your code with an equivalent style.

Then, carefully read the request, reply, and SNFS types in the `include/common/{requests, replys, ftypes}.h` headers. ("replys" is an intended misspelling!) It is important that you understand the structure of SNFS requests and replies. Once you have a good grasp of the SNFS client and server functions and operations, you're ready to start writing code.

Phase 1a: File Handle Database

In phase 1a, you will implement the `name_find_or_insert` and `get_file` functions in the server's `fhandledb.c`.

When implemented in the kernel, NFS can use inode numbers to map file handles to files. Unfortunately, inode numbers are not exposed directly to userspace. As a result, SNFS uses file names as a proxy for inode numbers and maps file handles to file names and vice versa. Because file handles must be valid even after a server crashes, this mapping must be made persistent.

To do this, we use Berkeley DB, an in-memory and on-disk key-value store. We've implemented functions that wrap the Berkeley DB interface: `init_db_if_needed`, `destroy_db`, `find`, and `insert`. We've also implemented functions that find and associate new file handles: `generate_new_fhandle`, and `new_fhandle`.

Your task is to use these functions to implement `name_find_or_insert` and `get_file`. The former finds an existing file handle, or creates and inserts a new file handle, for a given filename. The latter retrieves the filename for a given file handle. After you have implemented these functions, you should pass the `db` phase 1 unit tests.

Warning: `find` returns a `malloc()`d value! Ensure it's `free()`d before returning.

Hint: These can be implemented using only `find` and `new_fhandle` in `name_find_or_insert`, and `find` in `get_file`.

Phase 1b: Client Mount Request

In phase 1b, you will implement the `MOUNT` request function, `server_mount`, in the client's `request.c`.

When the client initializes, it requires the root file handle to make any further requests. The root file handle is the means by which the client can discover the rest of the files through future `READDIR`, `LOOKUP`, and other operations. The client makes this request using the `server_mount` function in `client/request.c`, which you will implement. The `server_mount` function is called from `snfs_init`. `snfs_init` is called by FUSE when it initializes. The code to set this up can be found in the client's `main.c` file.

To construct a request message on the stack, you may use the `make_request` macro. To send a request, you should typically use the `send_request` function. The specification for `send_request` is above its definition in `client/fuseops.c`. `send_request` attempts to send a request; it times out if the server takes too long to response. For `MOUNT`, we'd like to wait indefinitely for the server to respond, and so you should use `snfs_req_rep` to implement `server_mount`. The specification for `snfs_req_rep` is above its definition in the client's `request.c`.

If the reply is of fixed size (no flexible array members), you may use the `snfs_req_size` macro to determine its size. Simply doing `sizeof(snfs_req)` will return a value that is usually larger than the true size of your message, wasting precious bandwidth. The `make_request` macro takes two parameters: a message type and a request structure. Request structures are defined in `include/common/requests.h`. The specification for `send_request` is above its definition in `client/fuseops.c`. Note that you must free the returned reply using `nn_freemsg`. The `snfs_rep_size` macro takes one parameter: the message type in lower-case.

The following C code constructs a request of type `READ`:

```
snfs_req request = make_request(READ, .read_args = {
  .file = file_handle,
  .offset = offset,
  .count = count,
});

snfs_rep *reply = send_request(&request, snfs_req_size(read));
// or, to wait forever for a reply:
//   snfs_rep *reply = snfs_req_rep(sock, url, &request, snfs_req_size(read));
if (!reply) {
  print_err("The server did not reply with a valid response!\n");
  return -EIO;
}

/* ... use reply... */

nn_freemsg(reply);
```

You may also find it useful to construct a request that begins with *invalid* properties and get a handle to its arguments for later filling in. The following C code does this for a `LOOKUP` request:

```
snfs_req request = make_request(LOOKUP, /* fill in later */);
snfs_lookup_args *args = &request.content.lookup_args;

/* ... do work ... */

args->dir = some_handle;

/* ... do more work ... */
```

You can access reply properties through the reply union. To retrieve the `handle` for a `LOOKUP` reply, for instance, you can use:

```
handle = reply->content.lookup_rep.handle;
```

With this, you are ready to implement `server_mount` in `client/request.c`. You can find the function's specification above its definition. Once `server_mount` has been implemented successfuly, you should pass the `client_mount` phase 1 unit test.

Phase 1c: Server Mount Handler

In phase 1c, you will implement the `MOUNT` request handler in the server's `handlers.c`.

When the server receives a request, it is processed by the `serve_loop` function in `server/main.c` which identifies the request type and calls the appropriate function in `servers/handlers.c`. The handler receives the argument structure for the request sent by the client and eventually calls `send_reply` to send a response to the client. When a request of type `SOMETYPE` is successfully fulfilled, the protocol states that the server's reply should be of the same type, `SOMETYPE`. Otherwise, the reply should be of type `ERROR` with a property indicating what kind of error occurred.

To construct a reply message on the stack, you may use the `make_reply` macro. To send a reply, you should use the `send_reply` function. If the reply is of fixed size (no flexible array members), you may use the `snfs_rep_size` macro to determine its size. Simply doing `sizeof(snfs_rep)` will return a value that is usually larger than the true size of your message, wasting precious bandwidth.

The `make_reply` macro takes two parameters: a message type and a reply structure. The specification for `send_reply` is above its definition in `server/handlers.c`. The `snfs_rep_size` macro takes one parameter: the message type in lower-case.

The following C code constructs a reply of type `ERROR` with an error type of `error_type` on the stack and sends it to the client:

```
snfs_rep reply = make_reply(ERROR, .error_rep = {
  .error = error_type
});

if (send_reply(sock, &reply, snfs_rep_size(error)) < 0) {
  print_err("Failed to send error message to client.\n");
}
```

Reply structures are defined in `include/common/replys.h`. Ensure you understand how `serve_loop` dispatches requests and how handlers send replies to the client before proceeding. You may consult existing handlers, like `handle_noop` and `handle_error` for reference.

Implement `handle_mount` in `server/handlers.c` by changing the one line marked `FIXME`. You can find the function's specification above its definition. After you have implemented this function, you should pass all of the phase 1 unit tests.

Hint: The `name_find_or_insert` function that you implemented in phase 1a should look attractive here.

Phase 2: Lookup

In this phase, you will implement the crucial `lookup` function in the client's `fuseops.c`.

The SNFS (and NFS) protocol does not provide a way to determine the file handle associated with a file given a pathname. Given that every useful operation operates on file handles, it is important to be able determine the file handle for a given pathname. Instead, its `LOOKUP` operation requires you to look up a pathname component by component, passing in a file handle for one component and the filename for the next. The `lookup` function in `fuseops.c` does exactly this: given a pathname, it determines the file handle for the file by iteratively sending `LOOKUP` requests to the server. The `lookup` function will be called by many FUSE operation callbacks, but it itself is not a callback.

The root file handle is stored in the client's `STATE->root_fhandle`. It was set as a result of the `server_mount` operation in `client/main.c`. We have partially implemented `lookup` for you: it currently returns the file handle for the root path. Your task is to implement the remainder of `lookup`. Note that the `filename` property of the `LOOKUP` request should contain at most `SNFS_MAX_FILENAME_LENGTH` characters plus a `'\0'` character (= `SNFS_MAX_FILENAME_BUF`). You can find the function's specification above its definition. After you have implemented this function, you should pass the phase 2 unit tests.

Hint: You may find the libc `strtok` function useful.

Phase 3: Getattr

In this phase, you will implement `snfs_getattr` in `client/fuseops.c` and `handle_getattr` in `server/handlers.c`.

When a user performs a file system operation such as `open` or `stat` on a FUSE mounted directory, FUSE forwards the call to the process responsible for the directory. The process must have registered a `callback` operation for the corresponding system call; if it has not, FUSE returns a "not implemented" error to the user. Not all system calls map with callbacks 1-to-1: FUSE calls whichever callbacks it needs to manage its internal state and fulfill the user's requests.

To register a callback, FUSE requires a `fuse_operations` structure to be filled in with function pointers that refer to the callback for a given operation. The client code we have provided already fills this structure and passes it to FUSE in `client_main` in `client/main.c`. You will notice that the `getattr` callback function is `snfs_getattr` in `client/fuseops.c`.

FUSE calls the `getattr` callback for a large number of reasons: to manage its cache, to partially determine directory structure, and to fulfill a `stat` request, among others. It is essential that this callback exists for a FUSE file system to operate. The `getattr` callback's responsibility is to fill in the `struct stat *stbuf` parameter with as much information as possible. The `stat` structure is a POSIX file system structure that contains information about a file. The stat(2) man page has more information about the structure.

Your `snfs_getattr` FUSE callback should send a `GETATTR` request to the server for the file handle associated with the file at `path` and populate the `stbuf` with the `fattr` structure in the server's reply. Your `GETATTR` handler should respond to this request by finding the pathname for the `fh` file handle using the `get_file` function declared in `fhandledb.c`, calling `stat` on the file, filling in the `fattr` structure in the reply, and then sending the reply to the client.

Failure is certain!

Be wary of failure conditions. At the client, if the file at `path` does not exist, your function should return `-ENOENT`. If communication with the server fails, your function should return `-EIO`. At the server, if a supplied file handle is invalid, you should call `handle_error` with an `SNFS_ENOENT` error to send an `ERROR` response to the client. For any internal issues, you should send an `SNFS_EINTERNAL` error.

You should use the supplied `fattr_to_stat` function in your client implementation and the `stat_to_fattr` function in your server implementation. You can find the specification for all of the referenced functions above their definition. After you have implemented these functions, you should pass the phase 3 unit tests.

Phase 4: Readdir

In this phase, you will implement `handle_readdir` in the server's `handlers.c`.

A client sends a `READDIR` request to determine the contents of a directory. We have already implemented the FUSE `readdir` callback for you in the client's `snfs_readdir` function. Read it now and ensure you understand what the client expects from the server.

Your `handle_readdir` function should determine the path associated with the file handle or return an `SNFS_ENOENT` error if the handle is invalid. If the file referred to by the handle is not a directory, your handler should send an `SNFS_ENOTDIR` error to the client. For internal issues, send `SNFS_ENOTINTERNAL`. Finally, your handler should read `count` `snfsentry`s into the flexible array `entries[]` of the reply. Note that you will not be able to use the `make_reply` macro here: you must `malloc` a reply structure of the expected size and fill it in manually. Remember to set the reply type to `READDIR`. You can determine the size of the structure as follows:

```
size_t reply_size = snfs_rep_size(readdir) + bytes_for_entries;
```

The client can (and does) request more entries than actually exist: you shouldn't send more bytes than you need to. You can find the specification for `handle_readdir` above its definition. After you have implemented this function, you should pass the phase 4 unit tests.

Hint: Use functions like `opendir` and `readdir` to iterate through directory entries.

Warning: `readdir` returns NULL under normal *and* error conditions! Read the man page carefully.

Hint: While usually a bad idea, `goto` works wonders for garbage collection.

Phase 5: Open

In this phase, you will implement `snfs_open` in the client's `fuseops.c`.

The open callback is fairly straightforward: it associates a `path` with a FUSE handle that is later passed to calls like `read`. This FUSE handle is the `fh` property of the `fuse_file_info` structure. You can think of the handle as a FUSE file descriptor that you control.

Your task is to implement `snfs_open` so that it sets `fi->fh` to be the SNFS file handle for the file at `path` or return `-ENOENT` if the server doesn't recognize the file or if there's an issue communicating with the server. After you have implemented this function, you should pass the phase 5 unit tests.

Phase 6: Read

In this phase, you will implement `handle_read` in the server's `handlers.c`.

A client sends a `READ` request to read from a file. We have already implemented the FUSE `read` callback for you in the client's `snfs_read` function. Read it now and ensure you understand what the client expects from the server.

Your `handle_read` function should determine the path associated with the file handle or return an `SNFS_ENOENT` error if the handle is invalid or the file doesn't exist. Then, it should seek to the user provided `offset` and read at most `count` bytes into the `data[]` flexible array member in the reply. If the end of the file was reached during the read, the `eof` reply property should be set to 1. Note that you will not be able to use the `make_reply` macro here: you must `malloc` a reply structure of the expected size and fill it in manually. Remember to set the reply type to `READ`. You can determine the size of the structure as follows:

```
size_t reply_size = snfs_rep_size(read) + bytes_to_read;
```

The client can request to read more bytes than exist in the file: you shouldn't send more bytes than you need to. You can find the specification for `handle_read` above its definition. After you have implemented this function, you should pass the phase 6 unit tests.

Hint: Use functions like `lseek` and `read` in your implementation.

Phase 7: Write

In this phase, you will implement `snfs_write` in the client's `fuseops.c`.

The `write` FUSE callback functions exactly like the `write` system call except that FUSE considers any return value that is not `count` an error. Your task is to implement the `snfs_write` FUSE callback so that it sends a `WRITE` request to the server for the file handle at `fi->fh` which was set by your `open` callback. The write should begin at `offset` and write `count` bytes of data from `buf`. The `WRITE` handler is already implemented in `server/handlers.c`. Read it now and ensure you understand what the server expects from the client.

Note that you will not be able to use the `make_request` macro here: you must `malloc` a request structure of the expected size and fill it in manually. Remember to set the request type to `WRITE`. You can determine the size of the structure as follows:

```
size_t request_size = snfs_req_size(write) + count;
```

If there is a problem communicating with the server, your callback should return `-EIO`. You can find the specification for `snfs_write` above its definition. After you have implemented this function, you should pass the phase 7 unit tests.

Submission

Once you've completed the tasks above, you're done and ready to submit! Any changes made to files outside of the `src/`, `include/`, and `test/` directories will not be submitted with your lab. Ensure that your submission is not dependent on those changes, if any.

Warning: Ensure that everything runs as you expect on the virtual machine where we will grade your lab.

Finally, run `make submission` and then proceed to Gradescope to upload your submission.

Phase 8: Extra Credit

This phase is entirely optional and lack of completion will have no adverse effects on your score. Successful completion of this phase will result in a **15%**, **25%**, or **40%** boost to your score. You must also successfully complete all previous phases to be awarded the extra credit.

There are two options for this extra credit phase:

1. **Implement a lookup cache.** (**15%**)

   The client's `lookup` function is exceptionally slow: it sends a `LOOKUP` message to the server for every component in the path. To speed up lookups, a cache that maps paths to file handles can be kept at the client. The cache can be become invalid or inconsistent; care must be taken to prevent this.

   To implement this extra credit option, you should implement `cached_lookup` in the client's `fuseops.c`, which should use a cache to handle lookups. The cache should be consistent even in the face of file deletions, creations, or renames. The cache should be kept in memory. Consider using Berkeley DB as an in-memory key-value store by `open`ing the database with the `file` parameter set to `NULL`. The Berkeley DB API documentation has more information.

   You should add some unit tests that assert that `cached_lookup` is much faster than `lookup` for paths with many components and that it doesn't return stale file handles even when other clients are adding, removing, and modifying files in the background. **When you are done implementing this option, add a file named `extra_credit.cache.txt` inside the `src/` directory explaining your implementation and unit tests. Then, resubmit.**

2. **Add support for file creation, deletion, and renaming.** (**25%**)

   The present implementation of SNFS does not support file creation, deletion, or renaming. This makes SNFS less useful than one might hope. To implement this extra credit option, you will need to mount the present SNFS file system and attempt to create files (`touch`), delete files (`rm`), and rename files (`mv`) to see which calls FUSE makes. Then, you'll need to design request and reply message structures for fitting operations, implement the FUSE callbacks at the client, and implement handlers at the server. Your request and reply structures should not deviate much from those specified in the NFSv3 protocol. Finally, you'll need to add unit tests to the test suite that confirm that your implementation is working.

   At a minimum, the following shell code should execute as expected when run inside an SNFS mounted directory:

```
$ touch hello
$ stat hello
16777220 28447165 -rw-r--r-- ... and so on ...
$ rm hello
$ stat hello
stat: hello: stat: No such file or directory
$ echo 'Hello, world!' > hello
$ cat hello
Hello, world!
$ cat hello > hello_there
$ rm hello
$ mv hello_there bye_now
$ echo " Bye." >> bye_now
$ stat hello
stat: hello: stat: No such file or directory
$ cat bye_now
Hello, world!
 Bye.
```

Of course, clients should be able to read, modify, rename, and remove files created at or modified by other clients. **When you are done implementing this option, add a file named `extra_credit.ops.txt` inside the `src/` directory explaining your design, implementation, and unit tests. Then, resubmit.**

You may implement one or both options. Successfully implementing only the first option will result in a **15%** boost to your score. Successfully implementing only the second option will result in a **25%** boost to your score. Successsfully implementing *both* options will result in a **40%** boost to your score. To receive credit, ensure that you create the `extra_credit.{option}.txt` files with explanations of your implementations and that your submission contains these files.

Troubleshooting

Have you tried turning it off and on again?

Try restarting your VM first.

File system lock up

During development, you may find that an incorrect implementation of a FUSE callback causes your local file system to lock up. When this happen, you have three options:

1. Wait until FUSE gives up.
2. Try to kill the offending process: `sudo pkill test`, `sudo pkill client`, etc.
3. Try to manually unmount FUSE (see above).

Build errors when switching platforms

When switching between working in the VM and a local machine, you may mind that your project no longer builds. This is likely because the third-party libraries need different compilations, and neither the `clean` nor `all` targets will recompile third-party libraries for performance reasons. To recompile the libraries, call `make superclean` and then try `make` again.

Notes

This section provides notes on the lab and the SNFS protocol and its implementation that may be useful as you work through it.

Setting up a local development environment

If you would like to work on the lab on your own machine and your machine runs a modern distribution of Linux or OS X, you will need to install several libraries and set up testing paths.

On Linux:

- Install `pkg-config, libbsd-dev, libfuse-dev, g++-4.9, libdb6.0-dev`
- Ensure `which g++` refers to `g++-4.9`
- To unmount: `fusermount -u mount-dir`

On Mac:

- Install FUSE for OS X
- Install Homebrew
- Install Berkeley DB: `brew install berkeley-db`
- To unmount: `umount mount-dir`

On both:

- `sudo mkdir -p /snfs/mount`
- `sudo mkdir -p /snfs/serve`
- `sudo chown -R [your-user-id] /snfs`

Restrictions

The current implementation of the protocol assumes that the client sends well-formed messages. Further, it imposes the following restrictions:

1. The protocol should only be used over a trusted local network and the server should be protected by a firewall from untrusted connections. Since the server will accept any MOUNT call, any client can retrieve the root handle and therefore access all remote files. Further, data is sent unencrypted, so a malicious bystander node can retrieve any data sent over the network if the traffic is public. Finally, since the server has no way to identify which client (via IP or otherwise) a message is coming from, it is trivially susceptible to DOS attacks.

2. The SNFS server must be the only process modifying the directory being served. The internal mapping from fhandles to local files depends on paths being consistent across server calls. If the server is unaware of unlinks, links, renames, etc., then it cannot maintain a reliable mapping.

References

NFSv1 Protocol RFC
SUN's 1989 RFC detailing the full NFSv1 protocol specification.

NFSv3 Protocol RFC
SUN's 1995 RFC detailing the full NFSv3 protocol specification.

CS135 FUSE Documentation
Concise documentation for FUSE operation structure functions from Harvey Mudd's CS135 course.

FUSE Tutorial
A pithy FUSE tutorial showing the basics on how to build a FUSE file system - complete with code!

FUSE Mount Options
The missing documentation to FUSE's command line arguments.

FUSE Operations Struct

Official FUSE documentation specifying the fields of the FUSE operation structure.

VirtualBox Download
The VirtualBox download page. A VMM, similar to VMWare, but free and open source.

Vagrant Download
The Vagrant download page. Allows you to textually declare a virtual machine image, among other things.

C File System Interface
The GNU libc manual page that details high-level file system interfaces in C.

C Low-Level I/O
The GNU libc manual page that details low-level I/O operations such as open/close, read/write, and seek.

Berkeley DB Getting Started
The official Getting Started guide for Berkeley DB. Includes prose and examples for using the database.

Berkeley DB C API
The C API for Berkeley DB. Note that the most common function are listed near the top of the table.