Lab 1: Basic File System

Hand out: Sept 26th, 2021

Deadline: Oct 11th 23:59 (GMT+8) No Extension

Get Ready

In this lab, you will learn how to implement your own file system step by step. In **Part1**, you will implement an inode manager to support your file system. In **Part2**, you will start your file system implementation by getting some basic operations to work.

To do this lab, you'll need to use a computer that has the FUSE module, library, and headers installed.

In the past, we will provide you a VM image running on VMware so you can directly use this VM image without any environment configuration. However, in this year, we decided to use Container (Docker) for all of your CSE Labs. Also, we will provide a Container Image including all the environments your need for these labs. The default **username** is stu and **password** is 000. If you are not familiar with Container (Docker), read the [docker tutorial] first.

Getting Started

```
$ git clone https://ipads.se.sjtu.edu.cn:1312/lab/cse-2021-fall.git cse-lab -b lab1
$ chmod -R o+w cse-lab
$ docker pull shenjiahuan/cselab_env:1.0
$ sudo docker run -it --rm --privileged --cap-add=ALL -v `pwd`/cse-lab:/home/stu/cse-lab shenjiahuan/cselab_env:1.0 /bin/bash
  # now you will enter in a container environment, the codes you downloaded in cse-lab will apper in /home/stu/cse-lab in the container
$ cd cse-lab
$ make
```

Notes: Only files in the volume will be persistent, put all the files useful to the volume directory, in the above example: /home/stu/cse-lab directory.

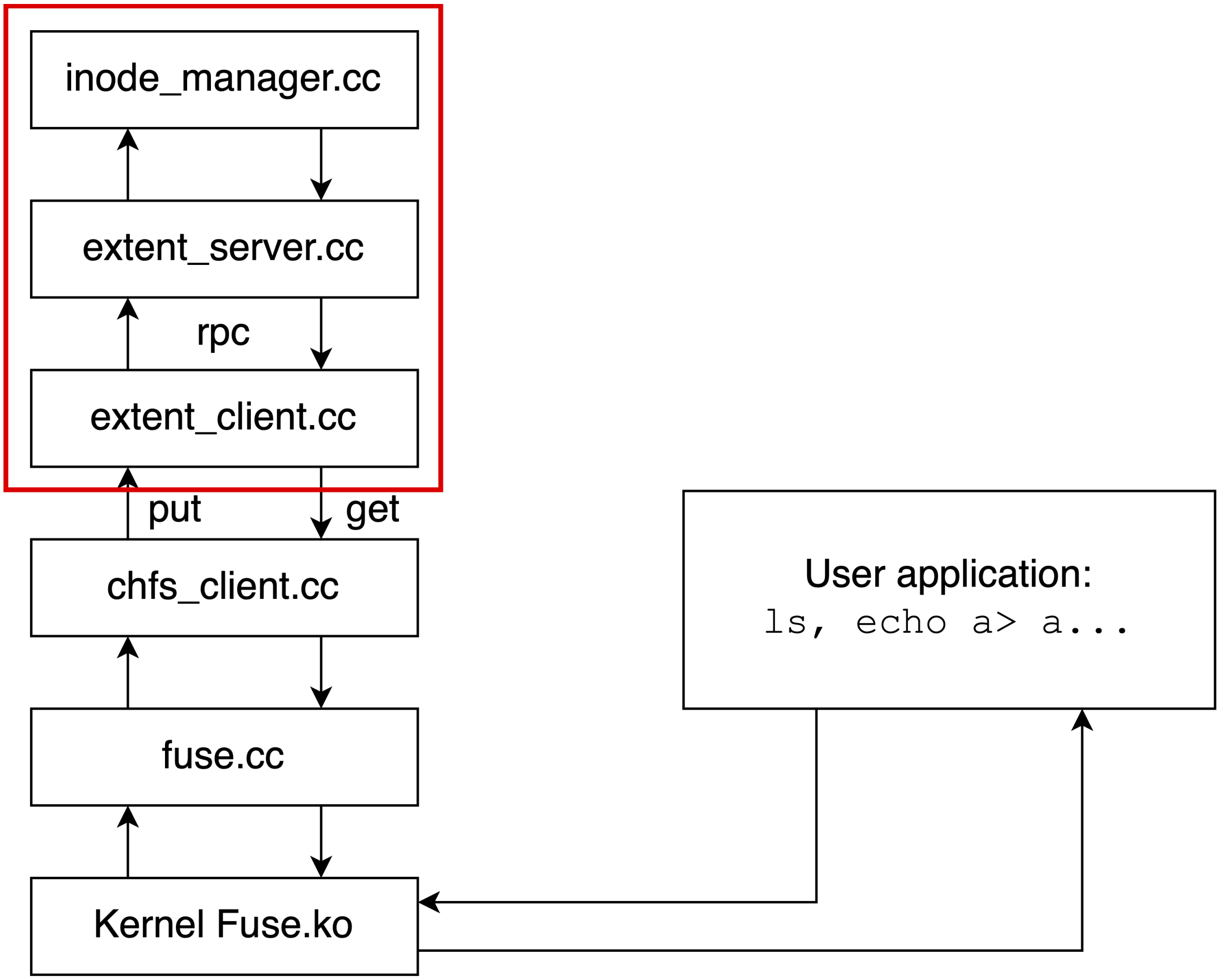
If you have questions about this lab, please ask TA: Shen Jiahuan.

Part 1

In this part, you will firstly implement an inode manager to support your file system, where following APIs should work properly:

```
CREATE, GETATTR
PUT, GET
REMOVE
```

Before implementing your inode manger, let's have a glance at the ChFS architecture:



In part 1, you can completely ignore the fuse and chfs_client, but just concern about the parts framed by the red box: **extent_client**, **extent_server** and **inode_manager**.

Extent_client acts as a block provider just like a disk. It will communicate with extent_server using rpc (which you will implement in the future, just now it only uses direct and local function call).

The inode manager mimics the inode layer of alloc_inode, free_inode, read_file, write_file, remove_file, getattr, which support the five APIs (**CREATE/GETATTR/PUT/GET/REMOVE**) provided by extent_server.

If there's no error in make, an executable file part1_tester will be generated, and after you type:

```
$ ./part1_tester
```

you will get following output:

```
========== begin test create and getattr ==========
...
[TEST_ERROR]: error ...
--------------------------------------------------
Part1 score is : 0/100
```

If you see additional warnings/errors, it's most likely because you don't have some specific libraries installed. Usethe apt-file utility to look up the correct package that contains the file you need, if you are on debian-based system.

Part 1 will be divided into 3 parts. Before you write any code, we **suggest** that you should read **inode_manager.h** first and be familiar with all the classes. We have already provide you some useful functions such as get_inode and put_inode.

In **part 1A**, you should implement disk::read_block, disk::write_block, inode_manager::alloc_inode and inode_manager::getattr, to support **CREATE** and **GETATTR** APIs. Your code should pass the **test_create_and_getattr()** in part1_tester, which tests creating empty files, getting their attributes like type.

In **part 1B**, you should implement inode_manager::write_file, inode_manager::read_file, block_manager::alloc_block, block_manager::free_block, to support **PUT** and **GET** APIs. Your code should pass the **test_put_and_get()** in part1_tester, which, write and read files.

In **part 1C**, you should implement inode_manager::remove_file and inode_manager::free_inode, to support **REMOVE** API. Your code should pass the **test_remove()** in part1_tester.

In part 1, you should only need to make changes to inode_manager.cc. (Although you are allowed to change many other files, except those directly used to implement tests.) Although maybe we won't check all the corner case, you should try your best to make your code **robust**. It will be good for the coming labs.

Part 1A: CREATE/GETATTR

Your job in Part 1A is to implement the read_block and write_block of disk and the alloc_inode and getattr of inode_manager, to support the **CREATE** and **GETATTR** APIs of extent_server. You may modify or add any files you like, except that you should not modify the part1_tester.cc. (Although our sample solution, for lab1-part-1, contains changes to inode_manager.cc only.)

The tips can be found on the codes of inode_manager.[h|cc]. Be aware that you should firstly scan through the code in inode_manager.h, where defines most of the variables, structures and macros you can use, as well as the functions get_inode and put_inode of inode_manager I leave to you to refer to.

Meanwhile, pay attention to one of the comments in inode_manager.cc:

```
// The layout of disk should be like this:
// |<-sb->|<-free block bitmap->|<-inode table->|<-data->|
```

It may be helpful for you to understand most of the process of the data access. After you finish these 4 functions implementation, run:

```
$ make
$ ./part1_tester
```

You should get following output:

```
========== begin test create and getattr ==========
...
...
========== pass test create and getattr ==========
========== begin test put and get ==========
...
...
[TEST_ERROR] : error ...
--------------------------------------------------
Part1 score is : 40/100
```

Part 1B: PUT/GET

Your job in Part 1B is to implement the write_file and read_file of inode_manager, and alloc_block and free_block of block_manager, to support the **PUT** and **GET** APIs of extent_server.

You should pay attention to the indirect block test. In our inode manager, each file has only one additional level of indirect block, which means one file has 100 direct block and 1 indirect block which point to a block filled with other blocks id.

After you finish these 4 functions implementation, run:

```
$ make
$ ./part1_tester
```

You should get following output:

```
========== begin test create and getattr ==========
...
...
========== pass test create and getattr ==========
========== begin test put and get ==========
...
...
========== pass test put and get ==========
========== begin test remove ==========
...
...
[TEST_ERROR] : error ...
--------------------------------------------------
Part1 score is : 80/100
```

Part 1C: REMOVE

Our job in Part 1C is to implement the remove_file and free_inode of inode_manager, to support the **REMOVE**API of extent_server.

After you finish these 2 functions implementation, run:

```
$ make
$ ./part1_tester
```

You should get following output:

```
========== begin test create and getattr ==========
...
...
========== pass test create and getattr ==========
========== begin test put and get ==========
...
...
========== pass test put and get ==========
========== begin test remove ==========
...
...
========== pass test remove ==========
--------------------------------------------------
Part1 score is : 100/100
```

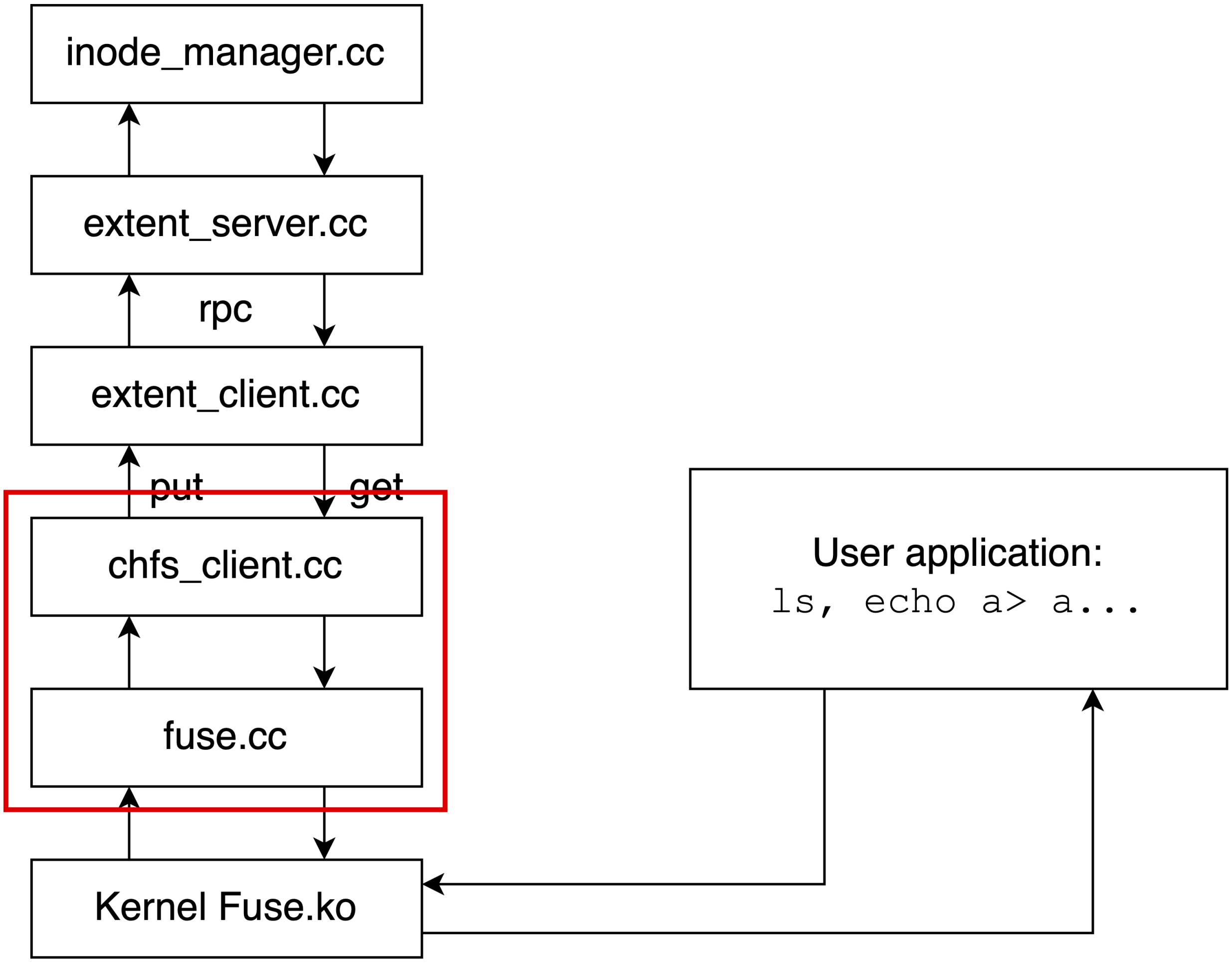
Part 2

In Part 2, you will start your file system implementation by getting the following FUSE operations to work:

```
CREATE/MKNOD, LOOKUP, and READDIR
SETATTR, WRITE and READ
MKDIR and UNLINK
SIMBOLIC LINK
```

(For your own convenience, you may want to implement rmdir to facilitate your debugging/testing.)

At first, let's review the ChFS architecture:



In part 2, what you should concern about are the parts framed by the red box above: **FUSE** and **ChFS** client.

The **FUSE** interface, in **fuse.cc**. It translates FUSE operations from the FUSE kernel modules into ChFS client calls. We provide you with much of the code needed to register with FUSE and receive FUSE operations. We have implemented all of those methods for you except for Symbol Link. So don't modify **fuse.cc** unless you want to implement **Symbol Link**.

The **ChFS client**, in **chfs_client.{cc,h}**. The ChFS client implements the file system logic. For example, when creating a new file, your chfs_client will add directory entries to a directory block.

We provide you with the script **start.sh** to automatically start **chfs_client**, and stop.sh to kill previously started processes. Actually, **start.sh** starts one **chfs_client** with **./chfs1** mountpoint. Thus you can type:

```
$ make
$ sudo ./start.sh
$ sudo ./test-lab1-part2-a.pl ./chfs1
$ sudo ./test-lab1-part2-b.pl ./chfs1
$ sudo ./stop.sh
```

Note 1: Since you need to mount fuse file system, so you should add **sudo** to above commands;

Note 2: If **stop.sh** reports "Device or resource busy", please **keep executing stop.sh** until it says "not found in /etc/mtab", such as:

```
fusermount: entry for /home/stu/cse-lab/chfs1 not found in /etc/mtab
...
```

Part 2 will be divided into **4 parts**:

At the beginning, it will be helpful to scan the **interfaces** and **structs** in **chfs_client.h** and some other files. The functions you have implemented in part 1 are the fundament of this part.

Part 2A: CREATE/MKNOD, LOOKUP, and READDIR

Your job

In Part 2A your job is to implement the **CREATE/MKNOD**, **LOOKUP** and **READDIR** of **chfs_client.cc** in ChFS. You may modify or add any files you like, except that you should not modify the test scripts. Your code should pass the **test-lab1-part2-a.pl** script, which tests creating empty files, looking up names in a directory, and listing directory contents.

On some systems, FUSE uses the MKNOD operation to create files, and on others, it uses CREATE. The two interfaces have slight differences, but in order to spare you the details, we have given you wrappers for both that call the common routine **createhelper()**. You can see it in **fuse.cc**.

As before, if your ChFS passes our tester, you are done. If you have questions about whether you have to implement specific pieces of file system functionality, then you should be guided by the tester: if you can pass the tests without implementing something, then you do not have to implement it. For example, you don't need to implement the exclusive create semantics of the CREATE/MKNOD operation.

Detailed Guidance

1. When creating a new file (fuseserver_createhelper) or directory (fuseserver_mkdir), you must assign a unique inum (which you've done in part1).

   Note: Though you are free to choose any inum identifier you like for newly created files, FUSE assumes that the inum for the root directory is 0x00000001. Thus, you'll need to ensure that when chfs_client starts, it is ready to export an empty directory stored under that inum.
2. Directory format: Next, you must choose the **format for directories**. A directory's content contains a set of name to inode number mappings. You should store a directory's entire content in a directory (recall what you learned). A simple design will make your code simple. You may refer to the **FAT32** specification (http://staff.washington.edu/dittrich/misc/fatgen103.pdf) or the EXT inode design (http://en.wikipedia.org/wiki/Inode_pointer_structure) for an example to follow. **Note**: As is mentioned in Wikipedia (http://en.wikipedia.org/wiki/Ext3), the EXT3 filesystem which we go after supports any characters but '\0' and '/' in the filename. Make sure your code passes when there's '$', '_', ' ', etc, in the filename.
3. FUSE:When a program (such as ls or a test script) manipulates a file or directory (such as chfs1) served by your **chfs_client**, the FUSE code in the kernel sends corresponding operations to chfs_client via FUSE. The code we provide you in **fuse.cc** responds to each such operation by calling one of a number of procedures, for create, read, write, etc. operations. You should modify the relevant routines in fuse.cc to call methods in **chfs_client.cc**. **fuse.cc** should just contain glue code, and the core of your file system logic should be in **chfs_client.cc**. For example, to handle file creation, fuseserver_createhelper to call chfs->create(...), and you should complete the create(...) method to **chfs_client.cc**. Look at getattr() in **fuse.cc** for an example of

how a fuse operation handler works, how it calls methods in **chfs_client**, and how it sends results and errors back to the kernel. ChFS uses FUSE's "lowlevel" API.

4. ChFS code:The bulk of your file system logic should be in **chfs_client**, for the most part in routines that correspond to fuse operations (create, read, write, mkdir, etc.). Your **fuse.cc** code should pass inums, file names, etc. to your **chfs_client** methods. Your **chfs_client** code should retrieve file and directory contents from the extent client with get(), using the inum as the extent ID. In the case of directories, your **chfs_client**code should parse the directory content into a sequence of name/inum pairs (i.e. chfs_client::dirents), for lookups, and be able to add new name/inum pairs.
5. A reasonable way to get going on **fuss.cc** is to run **test-lab1-part2-a.pl**, find the function in fuse.cc whose missing implementation is causing the tester to fail, and start fixing that function. Look at the end of **chfs_client1.log** and/or add your own print statements to **fuse.cc**. If a file already exists, the CREATE operator (fuseserver_create and fuseserver_mknod) should return EEXIST to FUSE.
6. **start.sh** redirects the STDOUT and STDERR of the servers to files in the current directory. For example, any output you make from **fuse.cc** will be written to chfs_client1.log. Thus, you should look at these files for any debug information you print out in your code.
7. If you wish to test your code with only some of the FUSE hooks implemented, be advised that FUSE may implicitly try to call other hooks. For example, FUSE calls LOOKUP when mounting the file system, so you may want to implement that first. FUSE prints out to the **chfs_client1.log** file the requests and results of operations it passes to your file system. You can study this file to see exactly what hooks are called at every step.

About Test

The Lab tester for Part 2A is **test-lab1-part2-a.pl**. Run it with your ChFS mountpoint as the argument. Here's what a successful run of **test-lab1-part2-a.pl** looks like:

```
$ make
$ sudo ./start.sh
starting ./chfs_client /home/stu/cse-lab/chfs1  > chfs_client1.log 2>&1 &
$ sudo ./test-lab1-part2-a.pl ./chfs1
create file-yyuvjztagkprvmxjnzrbczmvmfhtyxhwloulhggy-18674-0
create file-hcmaxnljdgbpirprwtuxobeforippbndpjtcxywf-18674-1
...
Passed all tests!
```

The tester creates lots of files with names like file-XXX-YYY-Z and checks that they appear in directory listings.

If **test-lab1-part2-a.pl** exits without printing "Passed all tests!", then it thinks something is wrong with your file server. For example, if you run **test-lab1-part2-a.pl** on the skeleton code we give you, you'll probably see some error message like this:

```
test-lab1-part2-a: cannot create /tmp/b/file-
ddscdywqxzozdoabhztxexkvpaazvtmrmmvcoayp-21501-0 :
    No such file or directory
```

This error message appears because you have not yet provided code to handle the CREATE/MKNOD operation with FUSE. That code belongs in fuseserver_createhelper in **fuse.cc**.

Note: testing Part 2A on the command line using commands like touch will not work until you implement the SETATTR operation in Part 2B. For now, you should do your testing via the creat/open, lookup, and readdir system calls in a language like Perl, or simply use the provided test script.

Note: if you are sure that there is not any mistake in your implementation for part1 and still cannot pass this test, maybe there are some **bugs in your part1**, especially read_file and write_file. Remeber that passing the test do not guarantee completely correct.

Part 2B: SETATTR, READ, WRITE

Your job

In Part 2B your job is to implement SETATTR, WRITE, and READ FUSE operations in **fuse.cc** and **chfs_client.cc**. Once your server passes **test-lab1-part2-b.pl**, you are done. Please don't modify the test program or the RPC library. We will use our own versions of these files during grading.

Detailed Guidance

1. Implementing SETATTR. The operating system can tell your file system to set one or more attributes via the FUSE SETATTR operation. See fuse_lowlevel.h for the relevant definitions. The **to_set** argument to your SETATTR handler is a mask that indicates which attributes should be set. There is really only one attribute (the file size attribute) you need to pay attention to (but feel free to implement the others if you like), indicated by bit **FUSE_SET_ATTR_SIZE**. Just AND (i.e., &) the to_set mask value with an attribute's bitmask to see if the attribute is to be set. The new value for the attribute to be set is in the attar parameter passed to your SETATTR handler. The operating system may implement overwriting an existing file with a call to SETATTR (to truncate the file) rather than CREATE. Setting the size attribute of a file can correspond to truncating it completely to zero bytes, truncating it to a subset of its current length, or even padding bytes on to the file to make it bigger. Your system should handle all these cases correctly.
2. Implementing READ/WRITE:A read (fuseserver_read) wants up to size bytes from a file, starting from a certain offset. When less than size bytes are available, you should return to fuse only the available number of bytes. See the manpage for read(2) for details. For writes (fuseserver_write), a non-obvious situation may arise if the client tries to write at a file offset that's past the current end of the file. Linux expects the file system to return '\0's for any reads of unwritten bytes in these "holes" (see the manpage for lseek(2) for details). Your write should handle this case correctly.

About Test

test-lab1-part2-b.pl tests reading, writing, and appending to files. To run the tester, first start one chfs_client using the **start.sh** script.

```
$ make
$ sudo ./start.sh
```

Now run test-lab1-part2-b.pl by passing the chfs1 mountpoint.

```
$ sudo ./test-lab1-part2-b.pl ./chfs1
Write and read one file: OK
Write and read a second file: OK
Overwrite an existing file: OK
Append to an existing file: OK
Write into the middle of an existing file: OK
Check that one cannot open non-existant file: OK
Check directory listing: OK
Passed all tests

$ sudo ./stop.sh
```

If **test-lab1-part2-b.pl** exits without printing "Passed all tests!" or hangs indefinitely, then something is wrong with your file server. After you are done with Part 2, you should go back and test with **test-lab1-part2-a.pl**again to make sure you did not break anything.

Part 2C: MKDIR and UNLINK

Your job

In Part 2C your job is to handle the MKDIR and UNLINK FUSE operations. For MKDIR, you do not have to create "." or ".." entries in the new directory since the Linux kernel handles them transparently to ChFS. UNLINK should always free the file's extent; you do not need to implement UNIX-style link counts.

About Test If your implementation passes the **test-lab1-part2-c.pl** script, you are done with part 3. The test script creates a directory, creates and deletes lots of files in the directory, and checks file and directory mtimes and ctimes. **Note** that this is the first test that explicitly checks the correctness of these time attributes. A create or delete should change both the parent directory's mtime and ctime (here you should decide which level you can **modify the 3 time attributes**, and think about why?). Here is a successful run of the tester:

```
$ make
$ sudo ./start.sh
$ sudo ./test-lab1-part2-c.pl ./chfs1
mkdir ./chfs1/d3319
create x-0
delete x-0
create x-1
checkmtime x-1
...
delete x-33
dircheck
Passed all tests!
$ sudo ./stop.sh
```

Note: Now run the command sudo ./grade and you should pass A, B, C and E.

Part 2D: SYMLINK, READLINK

Please implement symbolic link. To implement this feature, you should refer to the FUSE documentation available online and figure out the methods you need to implement. It's all on yourself. Also, look out for comments and hints in the hand-out code. **Note**: remember to add related method to fuse.cc. You may want to refer to http://stackoverflow.com/questions/6096193/how-to-make-symbolic-links-in-fuse and https://fossies.org/dox/fuse-3.10.5/structfuse__operations.html .

GRADING

Finally, after you've implemented all these features, run the grading script:

```
$ sudo ./grade.sh
Passed A
Passed B
Passed C
Passed D
Passed E
Part2 score: 100/100
```

Note that if you encounter a **"chfs_client DIED"**, your filesystem is not working. In such cases the requests are served by the system's file system (usually EXT3 or btrfs or tmpfs). **You would not be awarded credits if your chfs_client crashes, but could get partial credit if it produces incorrect result for some test cases**. So do look out for such mistakes. **We've seen dozens of students every year thinking that they've passed lots of tests before realizing this**.

Handin Procedure

After all above done:

```
$ make handin
```

That should produce a file called lab1.tgz. Change the file name to your student id:

```
$ mv lab.tgz lab1_[your student id].tgz
```

Then upload **lab1_[your student id].tgz** file to Canvas before the deadline. Make sure your implementation has passed all the tests before final submit. (If you must re-submit a new version, add explicit version number such as "V2" to indicate).

You will receive full credit if your software passes the same tests we gave you when we run your software on our machines.