Raj Patel, Derek Sagun, Luiz Rivera

PA3 Report

In order to run our program:

First create a mount point inside of /tmp. For an example, you can run the command "mkdir /tmp/mounty" to create a folder called mounty inside of tmp which we will use as the mount point.

Then run "mkdir mount" on the server side. Mount is the where the server is going to use a filesystem.

Secondly, go into the ServerSNFS and run the make command. The server executable follows the same structure as specified in the assignment document, so you can run the server as such: "./serverSNFS -port 12457 -path ./mount" After the server runs, it will print out "Waiting for incoming connections" . The server will use the directory mount as its local filesystem.

Then, go into the clientSNFS and run the make command. Afterwards, execute the following command "./ssfs -f /tmp/mounty" , including the name of the mount path after -f. Afterwards, the client will wait for the port number and IP "-port 12457 -ip 127.0.0.1" Enter this command and the client will connect to the server. We always use the IP address 127.0.0.1

After this, the connection will be made between the client and the server, this will be noticeable as a print statements will appear stating the connection has been made.

In order to test our program:

We have included a folder called test cases, which has a bunch of programs that test a lot of the function calls. We will list each function and describe how to test each one. We have created a program to test the following functions: create, open, read, write, truncate. Within each of the functions, we also open and close the files accordingly thus, flush and release are also tested. In order to test the following: readDir, openDir, relaseDir and getAttr again, we can run "dir" on the directory to print out its contents. Lastly, we can test the mkdir function by explicitly running "mkdir /tmp/mounty/newDir" or something similar. Next we will describe the command line structure for parameters for the programs in testcases. For each program, you can enter the directory and run make.

Here are the test cases we would like to run:

Mkdir:
Run the following:

mkdir /tmp/mounty/dir1
mkdir /tmp/mounty/dir2

-mkdir should have created dir1 and dir2 in the ./mount
Readdir:

Run the following:

dir /tmp/mounty

-the output should be dir1 and dir2

testCreate: takes in path of the file to create appended to the mount path

Run the following commands (after running make in terminal of testCreate directory):

./testCreate /tmp/mounty/newFile
./testCreate /tmp/mounty/newFile2.txt
./testCreate /tmp/mounty/newFile3.pdf

This creates 3 new files in the server's mount path, all with the S_IRWXU permission
The testCreate program will print out the file descriptor of the file that was returned by the command. Closes the file afterwards

testOpen: takes in path of the file to open appended to the mount path

Run the following commands (after running make in terminal of testOpen directory):

./testOpen /tmp/mounty/newFile3.pdf
./testOpen /tmp/mounty/newFile2.txt

Opens these two files and returns file descriptors, closes the files after use

testWrite: takes in path of file to write to appended to mount path, and a string to write into the file (MAX 50 bytes)

Run the following commands (after running make in terminal of testWrite directory):

./testWrite /tmp/mounty/newFile "test1 test1 test1"
./testWrite /tmp/mounty/newFile2.txt "this is the second test of write"

This writes the strings in quotes to the following files. To test this, you can run
cat /tmp/mounty/newFile or cat /tmp/mounty/newFile2.txt
or you can also just open the files manually

testRead: takes in path of the file to read from appended to mount path (tries to read 50 bytes)

Run the following commands (after running read in terminal of testRead directory):

./testRead /tmp/mounty/newFile
./testRead /tmp/mounty/newFile2.txt

This will print out the first 50 bytes of the file at the path.

testTruncate: takes in path of the file to truncate and the size to truncate the file to

Run the following command

./testTruncate /tmp/mounty/newFile2.txt 5

This will truncate the file to 5 bytes. This can be examined by clicking the file and examining its size.

For all the functions above except for truncate, the file is CLOSED at the end of every test program; thus flush & release are called at the end of testCreate, testOpen, testRead, and testWrite.

In order to test getAttr, readDir, openDir, & releaseDir, we can run the terminal command: Since get_attr is called first to see if the directory is real. Then openDir to get the Dir struct followed by readDir then by releaseDir, who closes that opened dir struct. You can look at the terminal for the clinetSNFS when get_attr is called. It will print out the specifics details for a file or directory.

dir /tmp/mounty

This will print out the contents of the directory and list all of the following files.

Cat and Dir are terminal programs that work for our implementation of FUSE

Responsibilities:

Raj : Open, Truncate, Create , Wrote testCreate, testOpen

Derek: Read, Write, Flush, Release, Wrote testRead, testWrite, testTruncate
Luiz: MkDir, Readdir, ReleaseDir, GetAttr, OpenDir

Some Things about our program:

For the RPC: So for our api for the rpc server, we have created a multithreaded tpc server and the messages it receives are encoded in a certain way in order to tell the server which function to call to pass the rest of the message. Since there is 12 functions, the first two chars of the message are devoted to numbers that single out certain functions. For example if the client sends over a readdir command to the server with the path ./. The message sent to the server is "00./"". The first two chars of the message of "00" tell the server that the message is a readdir command and from there, the path is concatenated with the path parameter and the path is sent to the corresponding function. Depending on the functions, the function may ead more data from the client side fuse program.

For the client side: Once our fuse functions are called from fuse. We just send the target path with the associated function number to the server. Depending on the function. The client may need to send more information to the server

For get_attr: The client sends the path for get_attr to the server which test if the file/dir exist or not using stat(). If it doesn't it notifies the client that it doesn't, if it does, then it sends the corresponding information one by one to the client.

For open_dir: After the server gets the path for open_dir. It opens up the directory in order to get the DIR struct. The server then stores the DIR struct in a global linked list.

For readdir: After the server gets the path for readdir. It searches the link list for the path in order to get the DIR struct. From there it reads through the contents of the directory and sends the paths to the client, one by one, who adds each path to the buffer provided by fuse.

For release_dir. After the server gets the path for release_dir, it searches the link list for the path and calls the closedir method on the DIR struct stored inside of that node.

For mkdir: After the server gets the path for the client, it just creates the directory in the that corresponding target directory
Note: IF any function fails on the server side, we send out the associated errno to the client and return that multiplied by one to the fuse function. IF the function works, we just return 0.

We created a FUSE File system that implements all of the functions asked for in the assignment document (open, create, read, write, flush, release, truncate, opendir, readdir, mkdir, relasedir, getattr). Our RPC involves a system of codes that correspond to each function, along with the path, since essentially every single FUSE function needs the path of the file or directory. For instance, the code for Open is 22. So if an open function was called on the mount path, FUSE would send "22/mountpath/file" to the server. The server would examine the first two characters and call the handle open function, which just awaits the specific parameters for the open function (the flags) and sends back confirmation. After receiving the parameters from the client, the server executes the system call and sends back the result, possibly with an errno if the system call fails. Most of the file call implementations work in this manner. Some difficulties we came across included our initial idea for RPC which was to send structs over a socket. However, this did not work, and we decided to go the simple manner of just sending a code and awaiting parameters for each individual function. Our server is also multithreaded and can handle multiple clients, as long as they make requests to operate on different files. Another challenge we faced was determining which calls were dependent on which. For an example, I had tried to implement the create function first, but I was unsuccessful, and it took me some time to realize that create actually called get_attr as well as open before actually routing to the create call. These dependencies proved to be a challenge since they could only be tested after most of the calls were complete.

For the functions read, write, flush and release, they basically start of the same as open and create, by sending the code to access the function and path for the file. They all began by calling open() in order to retrieve the file descriptor. After this they each called their own respective file calls. By obtaining the file descriptor we were able to access and manipulate the files data. Read called read(), write called write(), flush called fsync(), and release called close(). A problem that we had was trying to access the fuse call do_read and do_writes parameters size_t size and off_t offset. The way we countered this issue was by making the offset = 0 and size = 50. Because we did this our program can read and write up to the first 50 bytes max, but still allows for reading and writing with values smaller than 50.