Facebook-like Asssembly Server Part 2

Introduction This project is part of an assignment I had to do at Trinity College Dublin in the module Computer Architecture Two. With the second part of this project came creating the client program that encodes requests to the server in order to run the program with a specific client. Client to server request encoding:

```
./client <u1> create               -> create <u1>
./client <u1> add <u2>             -> add <u1> <u2>
./client <u1> post <u2> <message> -> post <u1> <u2> <message>
./client <u1> display              -> display <u1>
```

The above shows that the client needs to encode its inputs like this: `argv[2] argv[1] argv[3] argv[4]`. Pipes were a big portion of this project in which I was tasked to use them to facilitate communication between the client and server. These FIFO pipes were manuauly created before testing. Making sure to run the opening, reading, writing and closing sequences correctly to avoid both blocking and causing a deadlock.

The project code can be found here: https://github.com/ThomasCreagh/learn-x86/tree/main/assignments/01

RISCV64 vs x86 The client program was to be written in riscv64. The syntax had similarities to x86 especially with the pseudoinstructions. One of the recommended practices was to use a C driver to assist in helping pass inputs to the assembly program. I originally used a C driver but due to my implementation I was able to rename the start of my assembly program to `main` and use argv and argc as you would in C. Original C driver:

```
#include <stdio.h>

extern int start(int argc, char **argv);

int main(int argc, char **argv) {
    start(argc, argv);
    return 0;
}
```

Design Decisions Some of the design decisions I made were as follows:

- Using blocking pipes. This is allowed for more robust communication between the client and the server.
- Client reading pipe in a loop till close of pipe from server. This was because of how my `cat` function worked. It writes to the pipe in multiple writes to allow for large files. But this caused my client to have to read multiple lines until EOF or else some lines would get stuck in the pipe.
- Had my utility subroutines in different files to create better separation.

- My server didn't have much changes compared to just using stdout. I just created a global client and server file descriptor variables to allow the entire program to read and write from them respectively.
- Each loop of the server it tries to create a connection with the client which allows the server to stay running even if multiple different clients want to access it in one run.
- Implemented the server input parsing like bash that would trim any white space, handle quotes correctly and remove the quote character from inputs. One of the ways I could improve on my input parsing, would be to hadle more input as I only have a 512 byte buffer to read stdin.

Struggles My client originally was originally adding a next line (`\n`) and null (`\0`) character to the end of the request to the server to allow it to read till the next line or null. This wasn't permitted in the tests so I had to read the file in first and place the null byte after it was read based on the read bytes in order to tokenise the request by the server.

The biggest struggle I had was the results differing in my enviorment and the testing enviorment. I had 2 tests that returned `Client Crashed`. With no indication of whether that was a segmentation fault or any other error.

```
Running test: Add Friend when they don't exist (worth 5 points)
Test failed: client crashed
Running test: Display Wall (worth 5 points)
Test failed: client crashed
```

This was especially confusing to me because testing both of these on my local machine resulted in success.

```
$ qemu-riscv64 ./client tom and bob
nok: user bob does not exist

$ qemu-riscv64 ./client tom display
start_of_file
anthony: hi
end_of_file
```

Between the ambiguous test results and the fact that it was working on my machine it was very hard to debug and I was practically just guessing what was the problems.

These two tests never appeared to be working unfortuanetly. I tried to make sure all the files were closed when they needed to be and that the server looks for a new client connection after every request whether successful or not. I also tried making sure that args weren't being read from the clients termainal if they were not given but none of these things stopped the "client crash" in the testing enviorment.

My hypothesis is that it may be due to my buffered reading and writing which meant that I had to implement my client with a reading loop which differed to my peers implementation. I don't know exactly what is going wrong with it or if it has to do with the way the pipes are blocking in an edge case when a "display wall" request is sent. I do wish further errors were provided in order to assist me in finding the root cause of these crashes.

I didn't use AI tools very frequently in the creation of this project. I mainly used it to see if there was a better impementation of a subroutine after I wrote it or used it to gather information about how pipes worked.

Debugging The main method of debugging I used was to do debugging writes to stdout in order to see the strings that were building at different stages of the program. I did attempt to use gdb with 2 terminals to go step by step in my project but I never got it to work properly. If I was able to get gdb working it would have made debugging much more efficient.

Conclusion Overall I really enjoyed looking at RISCV64 and its benifits. I learned a lot about FIFO pipes and how they interact with blocking. My goal, with the knowlege I've gained making this project, is to understand my operating system better and how you can interact with it in unique ways. I would love to continue to refine this project and allow for multiprocessing so many clients could communicate to the server at once. And also to see the differences between this project and one thats in a production enviorment written in assembly.

This report can be seen here: https://thomascreagh.github.io/blogs/html/Facebook-like_Assembly_Server_Part_2.html