Facebook-like Asssembly Server

Introduction This project is part of an assignment I had to do at Trinity College Dublin in the module Computer Architecture Two. In this project, I was tasked with implementing a simplified Facebook-like social media server using IA-32 NASM assembly on Linux. The goal was to reproduce the core functionalities of a server that manages users, friends, and message exchanges, but to do so entirely through low-level system calls rather than high-level language abstractions. Throughout this project, I created assembly programs capable of creating users, adding friends, posting messages, and displaying message walls.

The project code can be found here: https://github.com/ThomasCreagh/learn-x86/tree/main/assignments/00

NASM vs C The main difference I had to deal with implementing this project in NASM vs C is that I had to rewrite a lot of the C library. Others doing the same project didn't do this but I though I would like to learn more about the C library and how these functions might have been implemented.

This is my file tree for the project:

```
.
|-- Makefile
|-- README.md
 -- src
    |-- add_friend_main.asm
    |-- create_user_main.asm
    |-- display_wall_main.asm
    |-- post_message_main.asm
    |-- server_main.asm
    |-- core
    |   |-- add_friend.asm
    |   |-- create_user.asm
    |   |-- display_wall.asm
    |   |-- post_message.asm
    |    -- server.asm
     -- lib
        |-- append_line.asm
        |-- cat.asm
        |-- exit.asm
        |-- get_txt.asm
        |-- mkdir.asm
        |-- parse_input.asm
        |-- touch.asm
        |-- user_exists.asm
        |-- user_in_file.asm
        |-- io
        |   |-- close.asm
```

```
|   |-- open.asm
|   |-- print.asm
|   |-- printf.asm
|   |-- read.asm
|   |-- read_line.asm
|    -- write.asm
 -- string
    |-- strcat.asm
    |-- strcmp.asm
    |-- strcpy.asm
    |-- strlen.asm
     -- tokenize_str.asm

6 directories, 33 files
```

I had to flatten this due to the way submittion works but this was the intended file structure. As you can see I wrote a lot of wrapper functions that would handle the specific system call numbers for me or to process strings and memory. Creating all these functions allowed me to program the main application in a higher level fashion.

I tried to embrase the Unix philosophy:

> Write programs that do one thing and do it well. - Doug McIlroy.

One of the great advantages of seperating the subroutines into different files was that my lables like `.loop:` didnt need any unique name because it wasn't conflicting with other loop labels of other subroutines. To help myself, I added a subroutine comment header to each file describing what the subroutine does, what inputs it takes (like `[ebp+8]`) and what it returns in eax.

Design Decisions Some of the design decisions I made were to:

- Use strcat for accessing the files instead of changing directorys using system calls for better efficiency.
- Using a file/chunk buffer for reading files. This ment I was not using a system call for every byte which would be extermely slow.
- Implemented a `cat.asm` subroutine that would print in 1024 bytes to improve performance.
- 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.
- I created sepreate main files for each of my executables which allowed me to directly call the subroutines in the `server.asm` program instead of having to use the `exec` system call to increase efficiency.
- I did also implement symetric friending which I had to remove due to the submittion portal test cases not liking it, even though it allowed it in the specifications.

Struggles Encapsulating the subroutines as such, worked well but it did pose some challanges like in `read_line.asm` and in how I was going to implement `printf.asm`. `read_line.asm` was more difficult to implement as a seprate routine as I had to pass a lot of input varible to keep track of where I was in the buffers and file: `file_descriptor, line_buffer, file_buffer, file_buffer_offset, bytes_read` where all needed for each line.

My main fucntions were required to print out responses to executing the programs. One of the challanges of this was to dynamically print out the input names given e.g. `./add_friend tom bob` should output `nok: user tom does not exist` if tom hasn't been created yet. I had created a `print.asm` subroutine and I wanted to create `printf.asm` subroutine to allow me to insert strings into a print statement dynamically. I tought I would seprate the formating to another subroutine called `format.asm` that would take N inputs and dynamaically write to a new buffer. The problem I faced was that since `printf.asm` had N inputs I wasn't able to just forward the inputs to `format.asm` without making it much more complicated. So in this case I just implemented the formatting inside `printf.asm`.

Having a `printf` function that replaced `$` with what ever input you pushed allowed me to make my `section .data` sections much cleaner instead of having to split up all the strings in order to string concatenate them together.

In the creation of this project I didn't use AI tools very frequently. I mainly used them to see if there was a better impementation of a subroutine after I wrote it. I did find some forums interesting to read on the implementation of C fucntions such as the `strcpy` function. In this stackoverflow discussion https://stackoverflow.com/questions/3561427/strcpy-return-value, they made a great point about why functions in the `string.h` library return the pointer to the start of the destination string that you give it.

This means you may have to recompute the size of the string if you did `strcpy` or another simalar fucntion that iterates over the destination string. So in my strcpy implementation I return the pointer to the null terminating character of the destination string. This allows me to use concatonate strings much more efficiently as I don't have to compute the length of the destination string after a strcpy.

Debugging In order to debug my project I used the program `gdb`. It was extemely useful to be able to step through my program line by line and print out memory address's or register values. e.g.

```
(gdb) x/s *(char**)($ebp+8)
0xffff9bd0: "tom"
```

This allowed me to see the string that was past as the first argument into the subroutine.

Conclusion Overall I learned a lot in creating this project and I am very happy with the outcome. I hope to bring some of the knowlege I gain in this to my future projects.

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