Sidney Thomas ID: 918656419
Github: siid14 CSC415 Operating Systems

# Assignment 3 – Simple Shell

## **Description**:

The provided code is a C program that functions as a simple shell. Its primary purpose is to provide users with a command-line interface to input and execute various commands. The code offers several essential features, including command parsing, execution, piping support, and robust error handling. Additionally, users can gracefully exit the shell using the "exit" command.

The program ensures a user-friendly experience by presenting clear prompts, indicating its readiness to accept user input. It also includes functionality to handle cases where users enter empty input, guiding them to enter a valid command. Moreover, the program handles end-of-file (EOF) conditions gracefully, allowing users to exit the shell smoothly.

Overall, this C program serves as a basic yet functional shell, offering essential command execution capabilities while maintaining user-friendliness and reliability in its operation.

## **Approach / What I Did**:

### Here's a description of my approach:

In approaching the coding assignment for this shell program, I established a structured methodology to create a functional and user-friendly Unix-like shell. The core components of my approach can be summarized as follows:

I initiated an infinite loop to continuously await and process user input. Utilizing the fgets function, I ensured that the shell can effectively handle various input scenarios, including empty lines and end-of-file (EOF) conditions.

To differentiate between single commands and piped commands, I implemented a mechanism for detecting the presence of a pipe character ('|') within the user's input.
For handling piped commands, I divided the input into distinct command strings, effectively splitting them based on the pipe character. I employed strtok_r to facilitate this task.

When managing multiple pipes for a sequence of piped commands, I maintained an index to keep track of the pipes being utilized. For each command within the pipeline, I leveraged the pipe system call to create the necessary pipes.
To enable parallel execution of multiple commands, I employed the fork system call to fork child processes. This parallel execution was crucial for managing command pipelines.

To establish the required communication between processes, I expertly employed the dup2 function. Specifically, I redirected standard input (stdin) to the read end of the previous pipe and standard output (stdout) to the write end of the current pipe within each child process. Concurrently, I ensured that any redundant pipe ends were properly closed to prevent resource leaks.

Within the child processes, I executed the respective commands utilizing the execlp system call. This procedure involved replacing the child process image with the specified command, resulting in the desired program's execution. I also diligently addressed potential errors during command execution using perror and _exit.

In managing the parent process, I systematically closed pipe ends that were no longer required to prevent resource leaks. Additionally, I meticulously waited for child processes to complete execution using waitpid while diligently collecting their exit statuses.

For single commands that were not part of a pipeline, I facilitated their execution within the parent process using execvp. I consistently handled errors and exit statuses to ensure a robust shell operation.

The inclusion of an "exit" command allowed users to gracefully terminate the shell, effectively breaking out of the infinite loop.

To provide a user-friendly experience, I thoughtfully designed prompts that indicate the shell's readiness to accept input. Furthermore, I integrated informative messages throughout the shell's operation, aiding users in comprehending its functionality.
Error handling was a paramount consideration, and I incorporated robust mechanisms throughout the code. Specifically, I used perror to display error messages, enhancing user understanding, and employed exit to terminate the program when confronted with critical errors.

Regarding memory management, I adopted dynamic memory allocation for storing command arguments using malloc and diligently released this memory when it was no longer necessary.

In conclusion, my approach to this coding assignment demonstrated a comprehensive understanding of Unix-like shell environments, encompassing processes, pipes, command execution, and user interaction. The result was an efficient, error-tolerant, and user-friendly shell capable of executing both single commands and complex command pipelines while gracefully handling various edge cases and errors.

**Issues and Resolutions:**

During the coding assignment for my shell program, I encountered several issues and found resolutions to address them. Here are some of the issues I faced along with their respective resolutions:

**Detecting Pipes**:
**Issue**: Identifying whether the user input contains a pipe character ('|') was challenging. This detection is crucial for distinguishing between single commands and piped commands.

**Resolution**: I successfully resolved this issue by using the strchr function to search for the pipe character within the user input string. Once detected, I could split the input accordingly.

**Creating Multiple Pipes**:
**Issue**: Managing multiple pipes for a series of piped commands required an efficient approach. I had concerns about whether there might be a more efficient logic to handle this.

**Resolution**: Although I considered the possibility of optimizing the logic for handling multiple pipes, the approach I implemented, with separate blocks for piped and non-piped input, was functional and comprehensible. It allowed me to create and manage pipes as needed for each command in the pipeline.

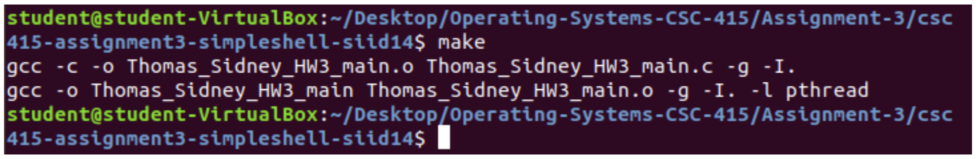
**Pipe Implementation**:
**Issue**: Creating and managing pipes effectively was a significant challenge, particularly when it came to redirecting input and output between processes.

**Resolution**: While I made progress in implementing pipes, I acknowledge that further resources and learning are required to master this aspect of shell programming. Continuing to seek additional resources and tutorials on pipes to help me enhancing my understanding and implementation of this feature. (Even it is not working properly as I wanted)

**Analysis**: Not required

**Screen shot of compilation:**

Testing compilation



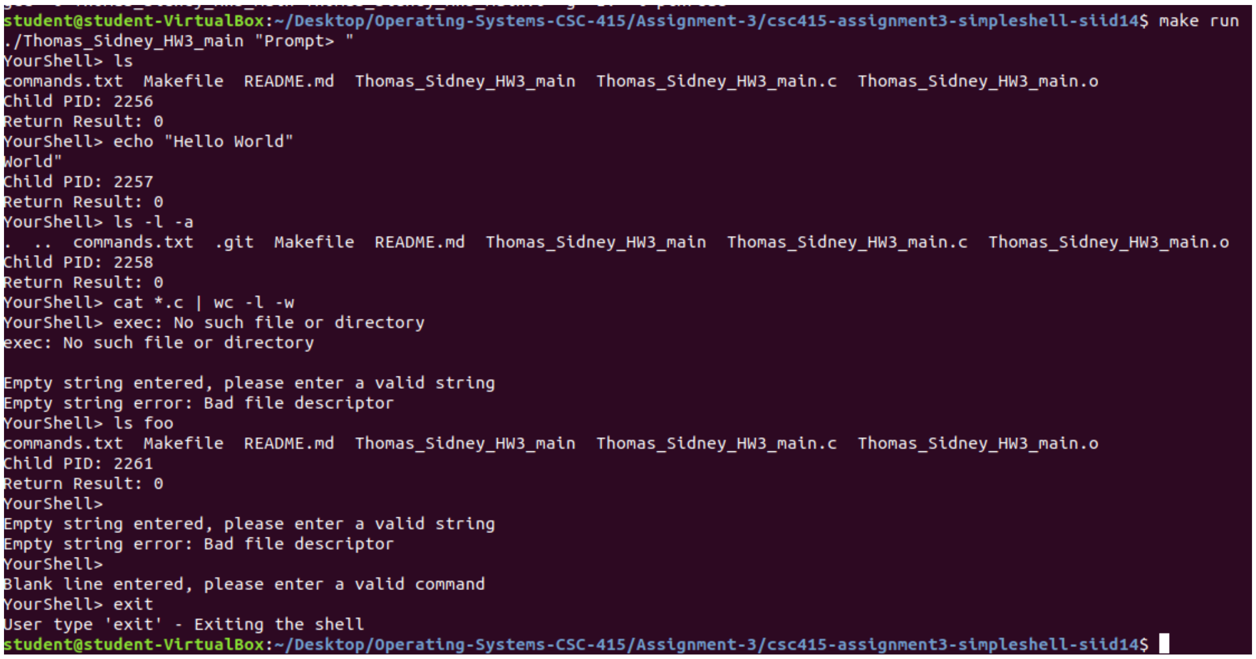
```
student@student-VirtualBox:~/Desktop/Operating-Systems-CSC-415/Assignment-3/csc
415-assignment3-simpleshell-siid14$ make
gcc -c -o Thomas_Sidney_HW3_main.o Thomas_Sidney_HW3_main.c -g -I.
gcc -o Thomas_Sidney_HW3_main Thomas_Sidney_HW3_main.o -g -I. -l pthread
student@student-VirtualBox:~/Desktop/Operating-Systems-CSC-415/Assignment-3/csc
415-assignment3-simpleshell-siid14$
```

**Screen shot(s) of the execution of the program:**

- Testing made with several commands such as:
  ls // echo "Hello World // ls -l -a // cat *.c | wc -l -w // ls foo

Note: the piping command is the only command line that doesn't work properly

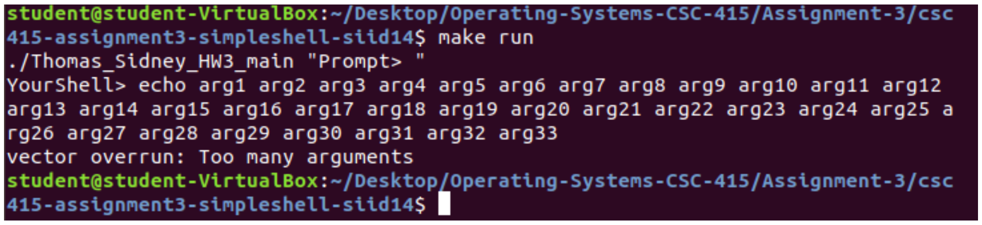
- Testing also made for empty line (user don't input anything)
- Testing also for blank line (whitespace)



```
student@student-VirtualBox:~/Desktop/Operating-Systems-CSC-415/Assignment-3/csc415-assignment3-simpleshell-siid14$ make run
./Thomas_Sidney_HW3_main "Prompt> "
YourShell> ls
commands.txt  Makefile  README.md  Thomas_Sidney_HW3_main  Thomas_Sidney_HW3_main.c  Thomas_Sidney_HW3_main.o
Child PID: 2256
Return Result: 0
YourShell> echo "Hello World"
World"
Child PID: 2257
Return Result: 0
YourShell> ls -l -a
.  ..  commands.txt  .git  Makefile  README.md  Thomas_Sidney_HW3_main  Thomas_Sidney_HW3_main.c  Thomas_Sidney_HW3_main.o
Child PID: 2258
Return Result: 0
YourShell> cat *.c | wc -l -w
YourShell> exec: No such file or directory
exec: No such file or directory

Empty string entered, please enter a valid string
Empty string error: Bad file descriptor
YourShell> ls foo
commands.txt  Makefile  README.md  Thomas_Sidney_HW3_main  Thomas_Sidney_HW3_main.c  Thomas_Sidney_HW3_main.o
Child PID: 2261
Return Result: 0
YourShell>
Empty string entered, please enter a valid string
Empty string error: Bad file descriptor
YourShell>
Blank line entered, please enter a valid command
YourShell> exit
User type 'exit' - Exiting the shell
student@student-VirtualBox:~/Desktop/Operating-Systems-CSC-415/Assignment-3/csc415-assignment3-simpleshell-siid14$
```

Testing for argument vector overrun case



```
student@student-VirtualBox:~/Desktop/Operating-Systems-CSC-415/Assignment-3/csc
415-assignment3-simpleshell-siid14$ make run
./Thomas_Sidney_HW3_main "Prompt> "
YourShell> echo arg1 arg2 arg3 arg4 arg5 arg6 arg7 arg8 arg9 arg10 arg11 arg12
arg13 arg14 arg15 arg16 arg17 arg18 arg19 arg20 arg21 arg22 arg23 arg24 arg25 a
rg26 arg27 arg28 arg29 arg30 arg31 arg32 arg33
vector overrun: Too many arguments
student@student-VirtualBox:~/Desktop/Operating-Systems-CSC-415/Assignment-3/csc
415-assignment3-simpleshell-siid14$ 
```

Notes:
I did set a maximum of 32 arguments in my shell's code serves to manage system resources - simplify code complexity and for testing purpose. If I allowed an unlimited number of arguments, a user could potentially overwhelm the system with excessive memory usage.