**Project 1 - Implementing a Shell**

**Due Date: September 5, 2020, 11:59pm**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Language Restrictions: C/C++ only

Additional Restrictions: The system() system call may not be used.

# Purpose

The purpose of this project is to familiarize you with the mechanics of process control through the implementation of a shell user interface. This includes the relationship between child and parent processes, the steps needed to create a new process, shell variables, and an introduction to user-input parsing and verification.

Your shell program must be able to run on a Linux environment such as Ubuntu on Windows Linux Subsystem.

# Problem Statement

Design and implement a basic shell interface that supports the execution of other programs and a series of built-in functions, as specified below. The shell should be robust (e.g., it should not crash under any circumstance beyond machine failure).

# Part 0: General Shell Structure and Documentation (30 points)

The shell (command line) is just a program that continually asks for user input, perhaps does something on the user’s behalf, resets itself, and again asks for user input. Here is an example:

while(1)

{

/\* Get user input \*/

/\* Exit? \*/

/\* Do something with input \*/

}

# Part 1: The Prompt (10 points)

At this point, the prompt should indicate that the shell is ready to accept input from the user. Often times, it also shows useful information, such as the name of the user running the shell and the current directory. For now, you just need to implement a simple prompt.

* The prompt should look like the following:
  + myshell$
* There should be a space after the dollar sign so that the user input does not visually run into the prompt.

# Part 2: Command Line Parsing (Code provided)

Before the shell can begin executing commands, it needs to extract the command name and the arguments into “tokens”. It might be nice to store these tokens into an array so that you can then parse each one in order. In our shell, the first token will always be the name of the program we wish to execute, and all remaining tokens (perhaps including the first token) will be arguments to that program.

Take note of the following assumptions:

* No leading whitespace
* One space separates the command line tokens.
* No trailing whitespace
* You can assume that each token is no longer than 80 characters.
* You can assume that a command will have at most 10 space-separated tokens

Make sure that you can successfully print out your array of tokens through different iterations of your shell loop before moving on. If you see garbage in any of your commands or arguments, try using the C library call memset() or bzero() to clear out your input string and token array before and/or after you are done using them.

The C library call fgets() can gather user input from the screen and save it into a string (C character array). See the man pages for fgets for more information.

# Part 3: Command Execution (30 points)

Once the shell understands what commands to execute it is time to implement the execution of simple commands. Since the execution of another program involves creating another process, you will have to use the fork() system call to create another process. Once you have created the new child process, that process must use the execvp() system call to execute the program. Finally, the parent (shell) process must wait for the child process to complete before releasing the child’s resources using the waitpid() system call.

However, the execvp() system call may return if there is an error. If it does, your shell should print an error, reset, and prompt for new input. Here is an example:

myshell$ lalala -a

Error: Command could not be executed

myshell$

# Part 4: Built-in Commands (30 points)

Not all commands are actually programs, and your shell must implement two “built-in” commands. In other words, if you encounter any of these two commands, do not execute them using fork(), exec(), and waitpid(). Instead, your shell should call a subroutine that implements the following functionality:

* **exit** – terminates your running shell process and prints 'exit'.

myshell$ exit exit

(shell exits)

* **showpid** – shows the last 5 ***child*** process IDs created by your shell.

myshell$ showpid

4987

4992

5001

5002

5004

# Project Submission

Both the C code file and the Makefile should be uploaded to Blackboard as a Zip file.