Morgan Martin & Jennai Jackson

The objective of this assignment was to understand the workings of a command-line interface and to obtain a working knowledge of process forking and signaling. Our first step was to execute built-in commands such as *"pwd"*, *"cd"*, *"echo*", *"exit"*, *"env*", and *"setenv"*. We have created a file 'shell.c' where we developed a simple shell to execute these commands from the command line.

Our shell prompts the user with the current working directory followed by ">". The core functionality of our shell is implemented within the 'main' function, which prompts the user for input and processes the commands accordingly. Within is a do-while loop to ensure the user is asked for input until a command that's not empty is given. We were able to parse through the input using *'strtok'* which tokenizes the input command into arguments to distinguish between parameters and flags and stores the command into the variable, "arguments". If argument[0] == 'cd', we change the current working directory to the desired directory using the *'chdir()'* function to take in the next argument entered on the command line. If argument[0] == *'pwd*', we printed the current working directory by printing *prompt* which is a variable that holds the working directory.. If argument[0] == *'echo'*, we implemented a loop to iterate over the command arguments to print the message following the echo command. If argument[1] began with the *'$'* symbol, we retrieved the environment variable using the *'getenv()'* command to return the variable value. This allows echo to still obtain the variable value and place it in the argument array. In this way, variables can be used with all commands. In order to be able to exit our shell and return to the terminal shell, the condition if argument[0] == *'exit'*, exits the shell using the '*exit()'* function. If argument[0] == *'env'*, we accessed the global environment

variables and returned them to the screen. However, if *'env'* is followed by another argument, we iterate through the global environment variables until the argument is found using the *'strncmp()'* function. If *'strncmp()'* returns a 0, the variable is found and we return the value of the variable. If argument[0] == 'setenv', we check to see that it is followed by two arguments. If so, we make argument[1] a variable with argument[2] as the value using the *'setenv()'* function. If the *'setenv()'* function returns a 0, the variable was set successfully; however, if it doesn't return a 0, an error message is shown because the variable already exists.

If the input command is not a built-in command, it forks a child process, loads the program from the file system, passes the arguments, and executes it. The *'execvp()'* function is used to execute the external command and the parent process waits for the child process to finish. If the command ends with *'&'*, the shell doesn't wait and continues immediately. To implement this, we initialized the boolean *'background'* as false. From here, we check if the argument ends with the *'&'* symbol. If it does we set *'background'* to true and remove the & from the arguments by setting it to *'NULL'*. The background boolean is able to determine whether to wait for the child process to complete. If it's true, the shell doesn't wait, but if it's false, the shell waits for the child process to finish.

Signal handling is implemented to prevent the shell from quitting when pressing '*CTRL C*'. To catch this signal, we used the *'SIGINT'* signal so that when *'CTRL C'* is pressed, the shell returns to the prompt instead of quitting the shell. A timer is implemented to terminate foreground processes after 10 seconds have elapsed using the '*kill()*' function. If the process finishes before the 10 seconds are over, the shell returns the prompt for the user to run their next command. We did this using the kill_process function, which we designed to send a '*SIGTERM'* signal to a given process ID *'pid'* and print a message indicating that the process has been killed.

Our simple shell displays basic command-line interface concepts and our implementation of built-in commands, execution of external commands, background processing, and handling of signals demonstrate the core functionalities expected from a basic shell.