Muzaffar Sharapov ID: **921548703**
Github: mansursh CSC415 Operating Systems

Description:

The file defines three functions: countArguments, parseCommand, and freeCommand. The main function serves as the entry point of the program, and it sets a maximum buffer size and prompts the user for input. It creates a child process using the fork function, counts the number of arguments in the user input using the countArguments function, and parses the user input into an array of C-String pointers using the parseCommand function. If the execvp function fails, it reports an error and exits the child process with a status code of -1. In the parent process, it waits for the child process to terminate using the wait function and frees the memory allocated for the userInput and prompt strings.

Approuch/What I did:

First, I read the instructions multiple times and checked the provided template. I made notes on what was needed and then followed the instructions. Whenever I didn't understand something, I read the official libraries to learn more. Most of the time, I was reading and taking notes. The coding process steps were:

• Wrote a function called "countArguments" that took a string (cmdline) as input and returned the number of arguments (tokens) in the command. (This function is needed to count the number of arguments in the user input so that we can allocate memory for the argv array in the "parseCommand" function.)

• Created a copy of the input string using the "strdup" function. (It's necessary because the "strtok_r" function modifies the input string, and we don't want to modify the original user input.)

• Initialized a counter variable (nTokens) to zero. (This variable will be used to keep track of the number of tokens found.)

• Used the "strtok_r" function to extract each token from the input string. (This function breaks up the input string into individual tokens based on a delimiter.)

• Incremented the counter variable (nTokens) for each token extracted. (This allows us to keep track of how many tokens were found.)

• Freed the copy of the input string using the "free" function. (It's necessary to prevent memory leaks.)

• Returned the number of arguments (tokens) plus one (for the NULL argument used in the execvp function). (It's necessary because the execvp function requires a NULL pointer as the last argument.)

• Wrote a function called "parseCommand" that takes a string (cmdline) and an integer (numArguments) as inputs, and returns an array of C-String pointers (char** argv). (This function is needed to parse the user input into an array of C-String pointers so that we can pass it to the execvp function.)

• Allocated memory for the argv array using the "malloc" function. (It's necessary to create an array that can hold the arguments.)

• Created a copy of the input string using the "strdup" function. (It's necessary because the "strtok_r" function modifies the input string, and we don't want to modify the original user input.)

• Initialized a counter variable (nTokens) to zero. (This variable will be used to keep track of the number of tokens found.)

• Used the "strtok_r" function to extract each token from the input string. (This function breaks up the input string into individual tokens based on a delimiter.)

• Allocated memory for each argument (token) using the "malloc" function. (It's necessary to create a C-String that can hold each token.)

• Copied each argument (token) into the argv array using the "strcpy" function. (It's necessary to copy each token into the argv array.)

• Incremented the counter variable (nTokens) for each token extracted. (This allows us to keep track of how many tokens were found.)

• Set the last element of the argv array to NULL. (It's necessary because the execvp function requires a NULL pointer as the last argument.)

• Freed the copy of the input string using the "free" function. (It's necessary to prevent memory leaks.)

• Returned the argv array. (It's necessary so that we can pass the array to the execvp function.)

• Wrote a function called "freeCommand" that takes an array of C-String pointers (char** argv) and an integer (numArguments) as inputs, and freed the memory allocated for the argv array and its contents. (This function is necessary to deallocate the memory that was allocated for the argv array and its contents in the "parseCommand" function.)

• Used a loop to call the "free" function on each argument in the argv array (except the last one). (It's necessary to deallocate the memory that was allocated for each argument.)

• Freed the argv array using the "free" function. (It's necessary to deallocate the memory that was allocated for the argv array.)

• Set a maximum buffer size (bufferSize = 102 + 1(NULL)) to prevent buffer overflow when reading user input. (It's necessary to prevent the user from entering a command that is too long and causing a buffer overflow.)

• Allocated memory for the prompt string (prompt) using the "malloc" function. (It's necessary to create a string that can hold the prompt.)

• Checked the command line arguments to see if a custom prompt was specified. If not, use the default prompt ("> "). (It's necessary to allow the user to specify a custom prompt if they want to.)

• Allocated memory for the user input string (userInput) using the "malloc" function. (It's necessary to create a string that can hold the user input.)

• Entered a loop that reads user input until end-of-file or an exit command is entered. (It's necessary to allow the user to enter multiple commands.)

• Displayed the prompt using the printf function. (It's necessary to show the user the prompt and prompt them to enter a command.)

• Read a line of input from the user using the fgets function. (It's necessary to get the user's command.)

• Checked if the user input is empty or only contains the new line character. If so, report an error and prompt the user for input again. (It's necessary to prevent the user from entering an empty command.)

• Removed the new line character from the user input string using the null character. (It's necessary to clean up the user input string.)

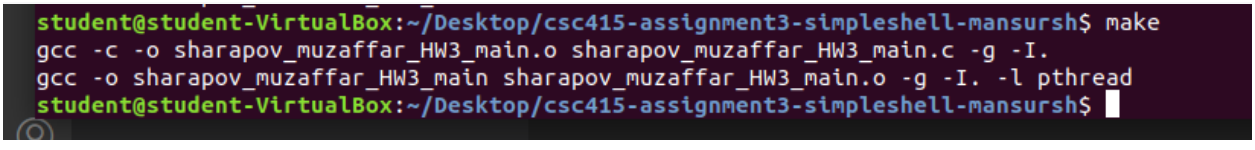
• Created a child process using the fork function. (It's necessary to run the user's command in a separate process.)

• In the child process:
Counted the number of arguments in the user input using the "countArguments" function. (It's necessary to determine how much memory to allocate for the argv array.)
Parsed the user input into an array of C-String pointers (cmd) using the "parseCommand" function. (It's necessary to pass the user's command to the execvp function.)
If the "execvp" function fails, report an error using the "fprintf" function and exit the child process with a status code of -1. (It's necessary to handle errors when running the user's command.)
Exit the child process. (It's necessary to end the child process and return control to the parent process.)

• In the parent process:
Waited for the child process to terminate using the "wait" function. (It's necessary to wait for the child process to finish running the user's command.)
If the child process terminated normally, print its process ID and exit status using the "printf" function. (It's necessary to show the user that their command was executed and whether it was successful.)
• Return 0 to indicate successful program execution.

Analysis

Screen shot of compilation of the program:



Screen shot of compilation and the execution of the program: