Problem: In our slides we talked about a simple shell. For this assignment you will implement your own shell that runs on top of the regular command-line interpreter for Linux.

Your shell should read lines of user input, then parse and execute the commands by forking/creating new processes. For each command, your shell should call fork() followed by execvp(). Following each command, your shell should wait for its child process to complete, and then print the child PID and the **return result from the child process**. The user should be able to specify the command to execute by giving a path to the executable file (e.g. /bin/ls) or by using path expansion to locate the executable file (i.e. searching each directory in the PATH environment variable). (Note that the execvp() function perform this processing automatically; you do not need to program it yourself.) This assignment does not use pipes.

If your shell encounters an error while reading a line of input it should report the error and exit. If your shell encounters EOF while reading a line of input, it should exit gracefully without reporting an error. Ensure that you do not overflow a 102 byte buffer when fetching the line of input (functions that do not accept the size of your buffer are not able to prevent overflows whereas functions that do accept a size generally do; be sure to check the manpage of any function you use carefully). You do not need to report an error if the user's input line is larger than the 102 byte buffer; just use the truncated input as the command.

You can test the EOF by running make run < commands.txt or from the prompt enter CTRL-D

Before your shell forks a new process to call execvp(), it should parse the input string and separate it into a collection of substrings representing the executable file and any command-line arguments. If the user entered an empty line, report an error and fetch a new line of input. Your code must handle at least four command-line arguments (in addition to the name of the executable file itself).

You should store pointers to the substrings in an array (similar to the “argv” array passed to main()) and pass this array of arguments to execvp(). Note that the number of command-line arguments is variable; this is indicated in the array by including a NULL pointer in the array after the last substring. (This means that if the user specifies N substrings, your array must hold N + 1 pointers where the last pointer is NULL.) If the user enters the **exit**command, your shell should terminate (returning to the regular shell).

Note: your shell does not need to support cd (change directory).

Your program must also accept a command line argument which is the prefix prompt. If no value is specified use “> ” as the prompt.

Here is a sample execution:

student@student-VirtualBox:~/CSC415/assignment3$ ./assn3 prompt$

prompt$ ls -l

total 20

-rwxr-xr-x 1 student student 13216 Feb 23 13:44 assn3

-rw-r--r-- 1 student student 1583 Feb 23 13:44 assn3.c

Child 2124, exited with 0

prompt$ ls foo

ls: cannot access 'foo': No such file or directory

Child 2125, exited with 2

prompt$ exit

student@student-VirtualBox:~/CSC415/assignment3$

You should submit your source code file(s), and Makefile along with a writeup in PDF format (using the template) that includes a description of what you did issues you had and resolutions and the compilation and execution output (screen shots) from your program to GitHub and just the PDF also to Canvas. Your execution output should include commands with command-line arguments. Then use the exit command to exit your program and show the output of the same commands in the regular command-line interpreter for that machine to ensure they match.

All filenames should be <lastname>\_<firstname>HW<#>\_<optional>.<proper extension>

* Imported necessary C standard libraries (stdio.h, string.h, stdlib.h, unistd.h, sys/wait.h) using the #include directive.
* Wrote a function called "countArguments" that took a string (cmdline) as input and returned the number of arguments (tokens) in the command.
* In the "countArguments" function:
  + Created a copy of the input string using the "strdup" function.
  + Initialized a counter variable (nTokens) to zero.
  + Used the "strtok\_r" function to extract each token from the input string.
  + Incremented the counter variable (nTokens) for each token extracted.
  + Repeated steps 3-4 until no more tokens are found.
  + Freed the copy of the input string using the "free" function.
  + Returned the number of arguments (tokens) plus one (for the NULL argument used in the execvp function).
* 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).
* In the "parseCommand" function:
  + Allocated memory for the argv array using the "malloc" function.
  + Created a copy of the input string using the "strdup" function.
  + Initialized a counter variable (nTokens) to zero.
  + Used the "strtok\_r" function to extract each token from the input string.
  + Allocated memory for each argument (token) using the "malloc" function.
  + Copied each argument (token) into the argv array using the "strcpy" function.
  + Incremented the counter variable (nTokens) for each token extracted.
  + Repeated steps 4-7 until no more tokens are found.
  + Set the last element of the argv array to NULL.
  + Freed the copy of the input string using the "free" function.
  + Returned the argv array.
* 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.
* In the "freeCommand" function:
  + Used a loop to call the "free" function on each argument in the argv array (except the last one).
  + Freed the argv array using the "free" function.
* Wrote the main function as the entry point of the program.
* Set a maximum buffer size (bufferSize) to prevent buffer overflow when reading user input.
* Allocated memory for the prompt string (prompt) using the "malloc" function.
* Checked the command line arguments to see if a custom prompt was specified. If not, use the default prompt ("> ").
* Allocated memory for the user input string (userInput) using the "malloc" function.
* Entered a loop that reads user input until end-of-file or an exit command is entered.
  + Displayed the prompt using the printf function.
  + Read a line of input from the user using the fgets function.
  + 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.
  + Removed the new line character from the user input string using the null character.
  + Created a child process using the fork function.
  + In the child process:
    - Counted the number of arguments in the user input using the "countArguments" function.
    - Parsed the user input into an array of C-String pointers (cmd) using the "parseCommand" function.
    - If the "execvp" function fails, report an error using