CPSC 4420 PROJECT 1: SHELL SPRING 2016

Due: Wednesday, February 18th, at the beginning of class, through Canvas (source code). *Also either bring a paper printout of your source code or upload a pdf to Canvas (in addition to your source code) to class.*

*** If your program does not compile, you will receive no credit!! Double check that your program *** will compile before submitting it.

Overview:
Write a robust program that operates as a simple shell. This assignment will help you to become familiar with process creation, robust error handling, and the function of a shell. Your program will read commands from the user (stdin), parse each line into a command name and arguments, and start a new process that runs that command. Your program must be robust, handle all values returned from functions, and output meaningful error messages.

Specifics:
1. Your program will begin by displaying a prompt to the screen. This prompt must be different from the normal bash shell prompt.
2. Your program will accept commands of standard UNIX form (command name followed by a series of arguments: *cmd arg1 arg2 arg3* ... etc), with one command per input line. (I.e. you do not have to handle the ';' character.) Your program will continue to accept commands until the user types "exit" (without the quotes).
   a. The maximum number of characters that this shell allows in a single command is 256. (You must enforce this limit, and issue an error message if the limit is exceeded. Use a #define statement to encode the value of 256.)
   b. Arguments are separated by one or more whitespace characters. Tabs and spaces are considered whitespace. Any number of them can appear in any place, before or after the command, or between or after the argument(s).
   c. A carriage return ('\n') signals the end of the line of input.
3. Your program will execute each command entered, providing it with the arguments entered by the user. The shell must support the following:
   a. An internal shell command "exit" that quits the shell.
   b. A command with no arguments. (Ex: *ls*)
   c. A command with arguments. (Ex: *ls -l* or *cat filename* or *cat file1 file2* or *gcc project1.c*)
   d. Your shell does not need to support >, <, &, or |.
4. Your program should not crash. Ever. Errors should be handled gracefully and meaningful error messages produced.
   a. Blank lines (consisting of only whitespace) should not produce an error message, just a new prompt.
   b. If the user enters a command that *execvp()* cannot find (ex: *garbagename*) then your program should report an error similar to the error that a bash shell would produce, and give a new prompt.
   c. Save all return values from functions (*fork(), execvp(), waitpid(), malloc()*), and check to see if the function returned an error. If so, print an error message, and gracefully continue to run, if possible.
   d. You do *not* need to have separate error messages for each of the different errors that can be returned by a function. A general error message that the call failed is sufficient.

Style:
As always, follow good programming style. Comment your code as discussed in class (all variables commented, purpose of all functions described, sufficient comments in the code to make it clear the purpose of each code segment).

Hints:
1. You will probably want your prompt to be dramatically different from your normal prompt, lest you forget that you are inside your own shell, rather than bash.
2. Sample starter code is available on Canvas that handles the initial allocation of memory for the array of strings needed by *execvp()*.
3. As in many programs, the user interface is the most time consuming part to program.
   a. You can read input from the user in any fashion that works. However, you may want to consider using *getc()* in this instance, since it allows the most control.
   b. Think carefully as to how to handle the array of strings that contains the arguments. *execvp()* expects arg[0] to be the command name, and arg[i>0] to be the arguments...and expects the last argument to be followed by a NULL pointer. In other words, if there are no arguments, argv[1] must be a NULL pointer. This can be a major hassle, and requires careful thought to avoid segmentation faults on subsequent input lines, as well as to avoid memory leaks.
4. All UNIX commands of the standard form should behave nearly the same as in the bash shell.
   a. Note that some commands are in fact shell built-ins, not UNIX programs! You do not need to provide any built-ins, except for the command "exit".
   b. A successful shell should be able to use vi to edit its own source code, recompile it, and execute the resulting binary file (resulting in a shell running within a shell).
5. Your program should not crash. Ever.