**PEX 2 A Simple Linux Shell - 75 Points**

Due: 2300 hrs on Lesson 14, 11/12 February 2014

# Help Policy

|  |
| --- |
| **AUTHORIZED RESOURCES:** Any, except another cadet’s assignment or published solutions to the assigned problem.  **NOTE:**   * Never copy another person’s work and submit it as your own. Here are a few blatant examples of copying:   + Making an electronic copy of another cadet’s solution and then modifying it slightly to make it appear as your own work.   + Reading a printout or other source of another cadet’s work as you implement your solution.   + Completing your entire solution by following explicit instructions from another cadet, while he/she refers to his/her own solution * Do not jointly implement a solution (outside designated pair programming partners). * Helping your classmates learn and understand the homework concepts is encouraged, but extensive assistance should generally be provided by DFCS instructors. Only provide assistance up to your depth of understanding, beyond which assistance by more qualified individuals is more appropriate and will result in greater learning. If you have to look at your solution while giving help, you are most likely beyond your depth of understanding. * Help your classmates maintain their integrity by never placing them in a compromising position. Do not give your solution to another cadet in any form (hard copy, soft copy, or verbal). * **DFCS will recommend a grade of F for any cadet who egregiously violates this Help Policy or contributes to a violation by others.** **Allowing another cadet to see your assignment to help them will result in a zero on this assignment.** |

# Documentation Policy

|  |
| --- |
| * You must document all help received from sources other than your instructor or instructor-provided course materials (including your textbook). * The documentation statement must explicitly describe WHAT assistance was provided, WHERE on the assignment the assistance was provided, and WHO provided the assistance. * If no help was received on this assignment, the documentation statement must state “NONE.” * If you checked answers with anyone, you must document with whom on which problems. You must document whether or not you made any changes, and if you did make changes you must document the problems you changed and the reasons why. * Vague documentation statements must be corrected before the assignment will be graded and will result in a grade deduction equal to 5% (ceiling) of the total possible points. |

**OBJECTIVES**

* Be able to effectively use the Ubuntu operating system.
* Be able to program in the C programming language.
* Be able to manage a programming project with the make utility.
* Be able to explicitly manage memory within a program.
* Be able to effectively use POSIX system calls and constructs.
* Be able to manage inter-process communication.

**OVERVIEW**

This PEX focuses on POSIX system calls, POSIX constructs, and inter-process communication. You will create a simple shell program that allows the user to execute commands. Your shell’s main loop will prompt for a line of input and execute the appropriate command or fork to execute the appropriate program(s). Your program must be robust. Check for errors and ensure your shell performs as expected. **You must write your shell in C on your Ubuntu virtual machine. It is also required that you use gcc and make to manage your project. Do NOT program in Windows or Eclipse.**

**PAIR PROGRAMMING REQUIREMENTS**

For this assignment you may choose to use the [Pair Programming](http://en.wikipedia.org/wiki/Pair_programming) approach or you may work independently. If you choose to work with a partner, both students must worth together on all parts of the solution and will receive the same grade. Because you will have an in-class work day, your partner should be from your section. You will report who your partner is or if you are working independently in class on lesson 8.

In the pair programming technique, two programmers work together on one computer. One is designated as the driver and writes the code; the other is designated as the navigator and reviews each line of code as it is typed. The two programmers switch roles frequently. For this programming exercise, the programmers should switch roles at least every twenty minutes.

Part of your submission for this programming exercise will be a log file detailing the time spent by each programmer as the driver and navigator. Remember, **you must be with your partner at all times while working on this programming exercise**, logging who is acting in which role.

**SHELL REQUIREMENTS**

The basic logic of the shell consists of a loop in which it:

Prints a prompt

Reads a command line

Stores that command line into the history, deleting duplicate entries if necessary

Parses the command line into tokens (words)

Compares the command to the shell’s built-in functions

Executes the built-in function

OR

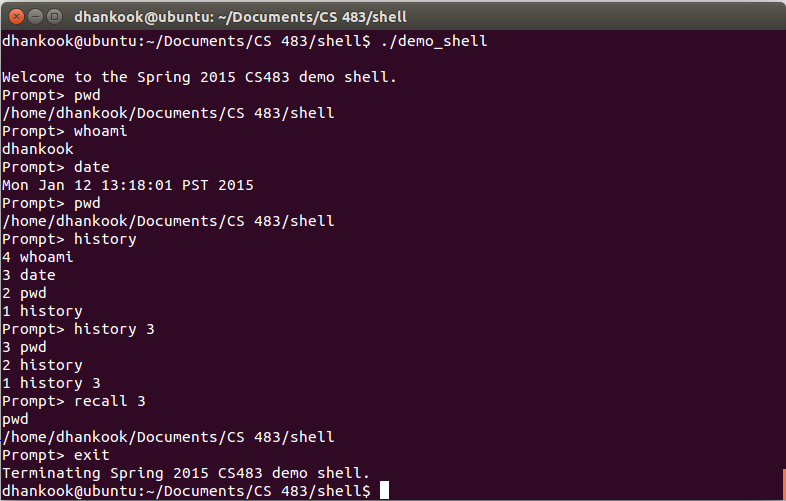
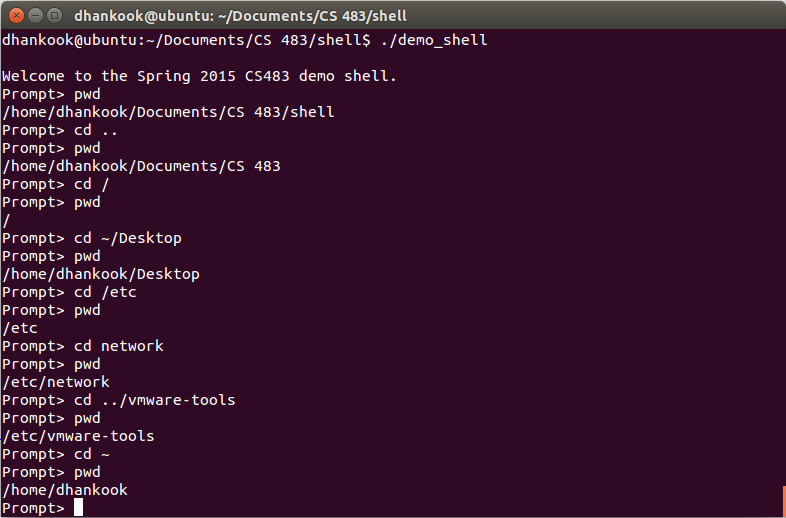
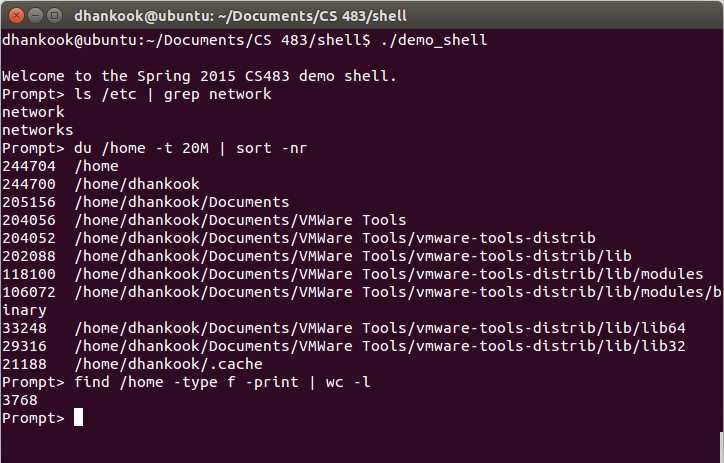
Forks a child/children

Which loads and executes the program(s)

Waits for the child(ren) to terminate

Each command line consists of a series of tokens, separated by "white space" (specifically, space(s) or tab(s)). Your program should read an entire command line as a single string, and then parse it into tokens. You may assume that no command will be longer than 127 characters by declaring a single array of 128 characters for capturing input. This is the only statically sized structure you are allowed. Hint: The standard C library has a function named strtok that is really good at this parsing a string into tokens.

In addition to executing standalone programs, your shell should be able to recognize and handle the following commands internally:

* exit – Terminates your shell and return control to the caller.
* history – Prints a numbered list of the most recent calls. Only remember calls within the current run of your shell. By default, print up to the 10 most recent calls (most recent call is #1), but the user can specify to print the x most recent calls by typing history x. Only store/display unique calls (most recent call takes precedence—i.e. remove any previous entries of the same call from the history and place it at the most recent position). There is no need to validate calls before storing them.
* recall x – Prints and executes the xth most recent call, as stored for the history call. An example run depicting program execution, exit, history, and recall is below.  
  
* cd – cd (change directory) is a shell command, not a Linux program. In order to make your shell more useful, you will implement the cd command. You should also interpret the ~ shorthand for the HOME environment variable. The getenv() and chdir() functions from unistd.h will make the implementation of cd much easier. An example run is provided below.  
  
* | (pipe) – It is common to direct the output of one program into the input of another. Most shells allow a user to do this using the | (pipe) token. Not surprisingly, this involves setting up a pipe IPC structure between the two child processes. You should add functionality for a single pipe to your shell. You do NOT need to allow pipes to be called with another shell command (e.g. history 100 | grep gcc), but recall should be able to retrieve a command from the history that includes a pipe and execute it. Note that this will require your shell to fork() and exec() two child processes. You may find [this webpage](http://jineshkj.wordpress.com/2006/12/22/how-to-capture-stdin-stdout-and-stderr-of-child-program/http:/jineshkj.wordpress.com/2006/12/22/how-to-capture-stdin-stdout-and-stderr-of-child-program/) useful in understanding how to set up the pipe between the children. An example run is provided below.  
  

**PLAN OF ATTACK/HINTS**

1. Start early. Compile and test often. Backup your code. Schedule and come in for EI early.
2. Think about your design first. In particular, think about what structures you will need and how your program will be organized. Also, how will you test as you go along? What functionality depends on what else, both for proper execution and testing?
3. Start by parsing the command-line into tokens. Be sure your solution handles extraneous spaces or tabs (before, middle, after).
4. Check for and handle the exit, cd, commands.
5. Build a data structure to store your call history.
6. Check for and handle the history and recall commands.
7. Implement the code to fork a child and have it exec the program. Note that exec refers to a family of functions; most likely, you will want to use execvp in the child. The parent will need to call wait or waitpid.
8. Implement the | (pipe) functionality by forking two children and manipulating file descriptors. Functions involved in the pipe setup/teardown include pipe, dup2, and close.
9. Test the functionality with various inputs from the command line. Do the correct things happen? Are the correct number of processes running at any given time?
10. Test input validation, appropriate output messages, the ability to quit, etc. Test your program with invalid inputs. Ensure that no logic, syntax, or runtime errors occur.

**SUBMISSION INSTRUCTIONS**

Submit your PEX via the “Submit Assignments” link to Moodle on the Course Piazza site. Submit all your c source (.c) and header (.h) files and your makefile. Do NOT structure your project into subfolders. Submit each of your files separately. Do NOT zip or otherwise archive your files. Your instructor will download your submission into one folder and should be able to type make and then run your shell.

Provide your documentation as a separate documentation.txt file. You may include in-line documentation, but please consolidate your documentation into one file.

If you are pair programming, part of your submission for this programming exercise will be a log file detailing the time spent by each programmer as the driver and navigator

NOTE:

* Moodle documentation has been disabled for this assignment. All your documentation must be in your documentation file. As you will not have Moodle to enforce the quality of documentation, be sure you are thorough in your documentation of WHAT assistance was provided, WHERE on the assignment the assistance was provided, and WHO provided the assistance.
* There is a 2MB file size limit on Moodle. If you exceed this limit for this assignment there is probably something drastically wrong with your code, but you may submit your files via email if that becomes necessary.
* You are also limited to uploading a maximum of 20 files.

**EXPECTATIONS**

|  |  |
| --- | --- |
| **Requirements**   1. Design/Implementation/Coding Standards    1. Proper decomposition, commenting, naming conventions, indentation, etc.    2. makefile    3. Memory management       1. Dynamically allocates memory of appropriate size       2. Free of memory leaks | **+10/10** |
| 1. Functionality    1. Correctly parses input       1. Error handling    2. cd functions properly       1. Handles ~ shorthand       2. Error handling    3. Executes external programs       1. Child process creation       2. External program call       3. Command line argument passing       4. Error handling    4. History and recall function properly       1. Correct insertion into list       2. Correct display of list       3. Handles size parameter       4. Recall of correct command       5. Command executes as if entered by user       6. Error handling    5. exit functions properly       1. Cleanly terminates shell    6. | (pipe) functions properly       1. Token parsing       2. Child process creation       3. File descriptor manipulation       4. Closes file descriptors appropriately       5. Error handling | **+10/10**  **+10/10**  **+15/15**  **+10/10**  **+5/5**  **+15/15** |
| **Penalties**   1. Vague/Missing Documentation (5%) 2. Late Submission (25% cap/day) | **-0/5** |
| **Total** | **75** |