## Homework 1 W4118 Fall 2012 UPDATED: Friday 9/7/2012 at 12:29am EST DUE: Monday 9/17/2012 at 11:59pm EST

All homework submissions are to be made via [Git](http://git-scm.com/). You must submit a detailed list of references as part your homework submission indicating clearly what sources you referenced for each homework problem. You do not need to cite the course textbooks and instructional staff. All other sources must be cited.

The Git repository you will use can be cloned using git clone gitosis@os1.cs.columbia.edu:UNI/hmwk1.git (Replace UNI with your own UNI). A public/private SSH key-pair will be emailed to you which you will use to access the Git server. Be aware that commits pushed after the deadline will not be considered. Refer to the homework policy section on the [class web site](http://www.cs.columbia.edu/%7Enieh/teaching/w4118) for further details.

### Non-Programming Problems:

Exercise numbers refer to the course textbook, *Operating System Concepts Essentials*. Each problem is worth 5 points.

1. Exercise 1.13
2. Exercise 1.24
3. Exercise 1.30
4. Exercise 1.31
5. Exercise 2.10
6. Exercise 2.16
7. Exercise 2.23
8. Exercise 2.24

### Programming Problems:

For all programming problems you will be required to submit source code, a README file documenting your files and code, and a test run of your programs. The README should explain any way in which your solution differs from what was assigned, and any assumptions you made. Refer to the homework submission page on the class web site for additional submission instructions.

Simple Shell (60 Pts.) Even the shell itself is just another user program. The files /bin/sh, /bin/bash, and /bin/tcsh are all executable files for shells. The only thing special about your login shell is that it is listed in your login record so that /bin/login (the program that prompts you for your password) knows what program to start when you log in. If you run ``ypmatch id passwd'', replacing id with your user ID, you'll see your login record, with your login shell program listed as the last field.

1. Write a simple shell in C. Your shell should read a line from standard input, parse the line to get the command and its arguments, fork, and exec the command. Your shell should wait for commands to finish, and should exit when it receives the command exit.
   * Using the system function is not allowed, as it just invokes the system's /bin/sh shell to do all the work.
   * You may assume command line arguments are separated by whitespace. Don't do anything special for backslashes, quotes, ampersands or other characters that are ``special'' in other shells. Note that this means commands in your shell will not be able to operate on filenames with spaces in them!
   * You may set a reasonable maximum on the number of command line arguments, but your shell should handle input lines of any length.
   * The executable file for your shell should be named w4118\_sh. When testing, make sure you execute your w4118\_sh and not /bin/sh.
2. A few shell commands are not separate programs at all, but are handled directly by the shell program. You have already implemented exit, which is an example of this. Now implement cd, the change directory command. You will need to invoke the chdir system call to do this. Note that if the call to chdir fails, you probably don't want to exit the shell but instead should handle the error appropriately.
3. The path variable holds a list of possible paths in which to search for executables. The list of paths is empty by default, but may grow to any arbitrary size. You should implement a built-in command to control this variable: path [+|- /some/dir]
   * path (without arguments) displays all the entries in the list separated by colons, e.g. "/bin:/usr/bin".
   * path + /some/dir appends the given pathname to the path list.
   * path - /some/dir removes the given pathname from the path list.
4. Most shell contain a history of commands that have been launched in the past. Usually typing the word history at the shell prompt will display this list of commands in chronological order, along with a number indicating the command. In addition, typing !n, where n is a number, will launch the nth command in the history list. The history command works as follows:
5. [me@myhost /home/me]$ history
6. [1] vi foo
7. [2] firefox
8. [3] ls
9. ...
10. [79] quake
11. [80] ssh clic.cs.columbia.edu
12. [me@myhost /home/me]$ !3
13. myfile1 myfile2 myfile3

Augment your shell to utilize a linked list in order to contain the history of the last 100 commands launched. Once 100 commands have been launched, the list should remove the first command in the list whenever it needs to accomodate more commands. Your shell should support the history command, and print out the list of past commands whenever history is typed. In the case where !n is typed, the nth command should be launched. In addition, be sure to free any memory allocated to support the linked list and to support file executions.

### *Additional Requirements*

1. *All code (including test programs) must be written in C.*
2. *Make at least ten commits with Git. The point is to make incremental changes and use an iterative development cycle.*
3. *Follow the following coding style rules:* 
   * *Tab size: 8 spaces.*
   * *Do not have more that 3 levels of indentations (unless the function is extremely simple).*
   * *Do not have lines that goes after the 80th column (with rare exceptions).*
   * *Do not comment your code to say obvious things. Use /\* ... \*/ and not // ...*
   * *Follow the* [*Linux kernel coding style*](http://www.kernel.org/doc/Documentation/CodingStyle)*. Use* [*checkpatch*](https://github.com/mirrors/linux-2.6/blob/master/scripts/checkpatch.pl)*, a script which checks coding style rules.*
4. *Use a makefile to control the compilation of your code. The makefile should have at least a default target that builds all assigned programs.*
5. *When using gcc to compile your code, use the -Wall switch to ensure that all warnings are displayed.* ***Do not*** *be satisfied with code that merely compiles; it should compile with no warnings. You will lose points if your code produces warnings when compiled.*
6. *Check the return values of all functions utilizing system resources. Do not blithely assume all requests for memory will succeed and all writes to a file will occur correctly. Your code should handle errors properly. Many failed function calls should not be fatal to a program. Typically, a system call will return -1 in the case of an error (malloc will return NULL). If a function call sets the errno variable (see the function's man page to find out if it does), you should use the result of strerror(errno) as part of your error message. As far as system calls are concerned, you will want to use one of the mechanisms described in* [*Reporting Errors*](http://www.cs.cf.ac.uk/Dave/C/node18.html#SECTION001810000000000000000) *or* [*Error Reporting*](http://www.gnu.org/software/libc/manual/html_mono/libc.html#Error%20Reporting)*.*
7. *Testing will be completed in part by an automated script. Therefore it's important that your shell's output is consistent. Your shell prompt text should always be "$ ". All error messages must be of the form "error: %s", strerror(errno). If an error occurs for which* errno *is not set, ensure the message begins with "error:". The remaining text is up to you. To verify the sanity of your shell's output (and functionality), run* [*this*](http://www.cs.columbia.edu/%7Enieh/teaching/w4118/homeworks/hw1_test.py) *simple version of the grading script. Once downloaded, you can run this script on the CLIC machines using the following command python hw1\_test.py "path\_to\_shell". To run on your own machine you will need* [*pyexpect*](http://www.noah.org/wiki/pexpect)*, which is available in most linux distro repositories.*

### *Tips*

1. *For this assignment, your primary reference will be* [*Programming in C*](http://www.cs.cf.ac.uk/Dave/C/CE.html)*. You might also find the* [*Glibc Manual*](http://www.gnu.org/software/libc/manual/) *useful.*
2. *Many questions about functions and system behaviour can be found in the system manual pages; type in man function to get more information about function. If function is a system call, man 2 function can ensure that you receive the correct man page, rather than one for a system utility of the same name.*
3. *If you are having trouble with basic Unix/Linux behavior, you might want to check out the resources section of the class webpage.*
4. *A lot of your problems have happened to other people. If you have a strange error message, you might want to try searching for the message on* [*Google*](http://www.google.com/)*.*