Assignment 3

Writing your own shell program. Due date: 11th November 2012.

The objective of this assignment is to gain experience with some advanced programming techniques in C/C++, use library functions and I/O facilities. You will also get familiarity with process creation and control, file descriptors, signals, and possibly pipes. Finally, you should learn how to make your program robust and crash-free.
To do this, you will be writing your own command shell "mysh" like csh, bash or the DOS command shell.

The shell program you write should support the following features (and additionally any other features for extra credit):

1. Your shell, called 'mysh' will read input lines from standard input, parse them into a command name and arguments, and then start a new process running that command.
When requesting input, your shell should print a prompt with the format similar to that in linux, i.e, with user@hostname:path to current dir.

2. Whenever a command with arguments is given, the prompt should parse the command and separate the arguments from the input and do the needful. Any illegal command,
Un-recognized arguments and arguments more than say 10, should throw meaningful error to the user and the shell should not crash or be killed.

3. To start with listing files, directories and their attributes, executing programs and terminating them, listing processes running in the system, drawing process tree diagram, much like pstree, but with added feature of displaying pID and whether the process is a user process or a system process, basic file operations, can be thought of as basic commands.

4. You should allow the user to specify commands either by relative or absolute pathnames. To read in the command line, you may want to consider the readline function from the GNU readline library as it supports user editing of the command line.

5. Optionally, at the end of any command input, can be a '&' character, which means that the parent process does not wait for the child process to complete before prompting for the next command. If any arguments follow the '&', your shell should print syntax error.

6. For any incorrect usage of a command, the shell should suggest correct usage of the command. You can support this by implementing help function for your commands.

7. Your shell should allow users to pass arguments with spaces in them through the use of backslash-space.

8. Your shell should support the special exit command, which causes the shell to exit after printing an exit message.

9. You should be able to redirect STDIN and STDOUT for the new processes by using < and >. For example, foo < infile > outfile would create a new process to run foo and assign STDIN for the new process to infile and STDOUT for the new process to outfile. Commands can be given in a file and taken as input with redirection.

10. You should maintain a history of commands previously issued and should also support editing those commands. A user should be able to repeat a previously issued command by typing "!no." where "no." indicates which command to repeat.

11. A built-in command is one for which no new process is created but instead the functionality is built directly into the shell itself. You should support the following built-in commands: jobs, cd, history, exit, help, and kill. "history" is to list the past few commands used and "jobs" is to check if there are any jobs spawned by your shell in the background. "exit" should check for incomplete jobs and warn the user before closing your shell.

12. Lastly you should try to provide support for pipe between two processes. The input of one process is obtained from the output of the other. You may want to start by supporting pipes only between two processes before considering longer chains. Use any POSIX IPC mechanism for this or in the worst case you can use files for IPC.

Optional features for extra credit:

1. You could implement more advanced I/O redirection than the simple in and out mentioned in 9th point.
2. You could support optional parameters to some of the built-in commands.
3. You could implement the built-in shell functions to move processes between the background and the foreground.
4. Up and down errors to scroll through the history list. The GNU history library makes easy.
5. Tab completion and command prompt editing. The GNU readline library makes this easy.
6. Any other features you can think of implementing are also welcome.

Tips:
1. Start simple. Try to write a shell program that first takes parameters and executes Linux shell commands in the beginning.
2. Use modular approach that uses separate programs for independent features. Write makefile to compile all of these together.
3. Test your program at each stage for various inputs and ensure that the program can handle any erroneous input. Learn to use the library functions with necessary signal handlers and features. This would help you to simplify your programs.
4. The following functions are likely to be helpful (consult the man pages for details):
fork, exec, execvp, wait, waitpid, kill, dup, pipe, strncmp, strlen, malloc, free, getcwd, chdir , open, close, readline, gets, fgets, getchar, signal.
5. Take the help of GNU C library, POSIX functions and use GDB for debugging. A good book that might be of help is Linux System Programming by Robert Love, OReilly Publishers.
6. Take special care in fork and ensure that dangling processes are not left or there are no explosion of processes that harms your system. Keep the bash shell ready to kill
your program if necessary.

Evaluation: Your program might be evaluated offline by TAs against test cases. So ensure that your code is documented well and all instructions to execute, input, cases for which the program does not work is mentioned in README. Your design document should contain the libraries used and methodology used to build the shell program. Your code will be evaluated against a variety of test cases and you should ensure that the program exits gracefully in any event.