Assignment 1

Creating an interactive user defined shell

Deadline: August 24th, Monday, 9pm.

Implement a shell which supports semi-colon separated list of commands. Use '*strtok*' to tokenize the command. Also support '&' operator which lets a program run in background after printing the process id of the newly created process. Write this code in a modular fashion. In the next assignment, you will add more features to your shell

The goal of the project is to create a user defined interactive shell program that can create and manage new processes. The shell should be able to create a process out of a system program like emacs, vi or any user-defined executable

The following are the specifications for the project. For each of the requirement an appropriate example is given along with it.

Specification 1: Display requirement

When you execute your code a shell prompt of the following form must appear: <username@system_name:curr_dir>
E.g., <Name@UBUNTU:~>

The directory from which the shell is invoked will be the home directory of the shell and should be indicated by "~" If the user executes "cd" change dir then the corresponding change must be reflected in the shell as well.

E.g., ./a.out
<Name@UBUNTU:~>cd newdir
<Name@UBUNTU:~/newdir>

Specification 2: Builtin commands

Builtin commands are contained within the shell itself. When the name of a builtin command is used as the first word of a simple command the shell executes the command directly, without invoking another program. Builtin commands are necessary to implement functionality impossible or inconvenient to obtain with separate utilities.

Make sure you implement cd , pwd and echo.

Specification 3: System commands with and without arguments

All other commands are treated as system commands like : emacs, vi and so on. The shell must be able to execute them either in the background or in the foreground.

Foreground processes: *For example, executing a "vi" command in the foreground implies that your shell will wait for this process to complete and regain control when this process exits.*
Background processes: *Any command invoked with "&" is treated as background command. This implies that your shell will spawn that process and doesn't wait for the process to exit. It will keep taking user commands.*

E.g
<Name@UBUNTU:~> ls &
This command when finished, should print its result to stdout.

<Name@UBUNTU:~>emacs &
<Name@UBUNTU:~> ls -l -a (Make sure all the given flags are executed properly for any command and not just ls.)

.
.
. *Execute other commands*
.
.

<Name@UBUNTU:~> echo hello

Bonus:

If the background process exits then the shell must display the appropriate message to the user.

For example :-
After emacs exits, your shell program should check the exit status of emacs and print it on stderr <Name@UBUNTU:~>
emacs with pid 456 exited normally <Name@UBUNTU:~>

Bonus: User defined commands

-**pinfo** : prints the process related info of your shell program. <Name@UBUNTU:~>pinfo
pid -- 231
Process Status -- {R/S/S+/Z} memory
-- 67854 {*Virtual Memory*}
Executable Path -- ~/a.out

-**pinfo <pid>** : prints the process info about given pid. <Name@UBUNTU:~>pinfo 1

General notes

1. **Useful commands** : uname, hostname, signal, waitpid, getpid, kill, execvp, strtok, fork,getopt etc. and so on. Type: man 2 <command_name> to learn of all possible invocations/variants of these general commands. Pay specific attention to the various data types used within these commands and explore the various header files needed to use these commands.
2. If the command cannot be run or returns an error it should be handled appropriately. Look at perror.h for appropriate routines to handle errors.
3. Use of system() call is prohibited, if you use it you'll get zero marks.
4. The user can type the command anywhere in the command line i.e., by giving spaces, tabs etc. Your shell should be able to handle such scenarios appropriately.
5. The user can type in any command, including, ./a.out, which starts a new process out of your shell program. In all cases, your shell program must be able to execute the command or show the error message if the command cannot be executed.
6. If code doesn't compile it is zero marks.
7. Segmentation faults at the time of grading will be penalized.
8. You are encouraged to discuss your design first before beginning to code. Discuss your issues on portal, as well as you can contact TAs
9. You can work together but implement your code independently to maximize the learning aspect. Please adhere to the anti-plagiarism policies of the course and the institute.
10. **Do not take codes from seniors or in some case, your batchmates, by any mistake. We will evaluate cheating scenarios along with previous few year submissions. So please take care.**