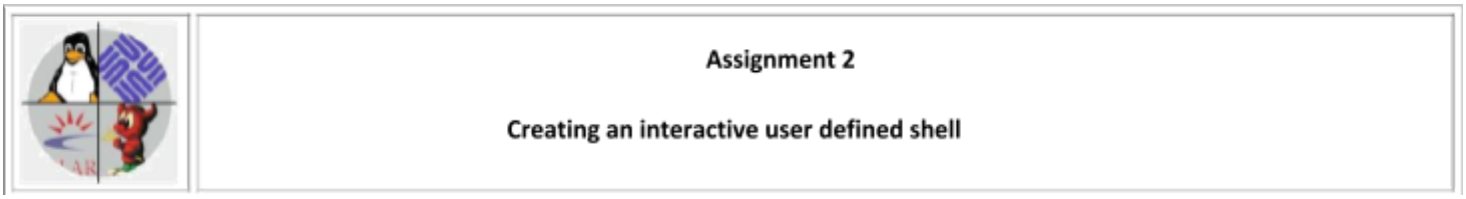

Assignment 2

Creating an interactive user defined shell

Deadline: **August 22, Thursday, 5:00 pm.**

This is an individual Assignment. Assignment 3 will be a direct continuation of this assignment.

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 assignment 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>
Example: <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" i.e change dir then the corresponding change must be reflected in the shell as well.

Example: ./a.out
<Name@UBUNTU:~>cd newdir
<Name@UBUNTU:~/newdir>

Specification 2: Builtin commands

Builtin commands are contained within the shell itself. Checkout 'type *commandname*' in the terminal (eg. 'type echo'). 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**. Don't use 'execvp' or similar commands for implementing

these commands.

Specification 3: ls command

Implement **ls [al]** – (it should be able to handle ls, ls -l, ls -a, ls -al/la and ls<Directoryname>. For ls , ls -a and ls<Directoryname> outputting the entries in a single column is fine.

Specification 4: 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.*

Example:
<Name@UBUNTU:~> ls &
This command when finished, should print 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

Specification 5: pinfo command (user defined)

-**pinfo** : prints the process related info of your shell program.
Example:
<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.
Example:
<Name@UBUNTU:~>pinfo 7777
pid -- 7777
Process Status -- {R/S/S+/Z}
memory -- 123456 {*Virtual Memory*}
Executable Path -- /usr/bin/gcc

Specification 6: Finished Background Processes

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

-**nightswatch [options] <command>**: Look up the man page entry for the 'watch' command - 'man watch'.
You will be implementing a modified, very specific version of watch. It executes the command until the '**q**' key is pressed.
***Options*:**
-n seconds: The time interval in which to execute the command (periodically)
***Command*:**
Either of these two:
1) interrupt – print the number of times the CPU(s) has(ve) been interrupted by the **keyboard controller (i8042 with IRQ 1)**. There will be a line output to stdout once in every time interval that was specified using -n. If your processor has 4 cores (quadcore machine), it probably has 8 threads and for each thread, output the number of times that particular CPU has been interrupted by the keyboard controller. Eg.
<Name@UBUNTU:~> nightswatch -n 5 interrupt
CPU0 CPU1 CPU2 CPU3 CPU4 CPU5 CPU6
CPU7

2 13 2 1 0 2 1 0
2 13 4 1 0 4 1 0
...
...
...
A line every 5 seconds until '**q**' is pressed.

2) dirty – print the size of the part of the memory which is **dirty**. Eg.

<Name@UBUNTU:~> nightswatch -n 1 dirty
968 kB
1033 kB
57 kB
...
A line every 1 second until '**q**' is pressed.

-**history:**

Implement a **'history'** command which is similar to the actual history command. The maximum number of commands it should output is 10. The maximum number of commands your shell should store is 20.

Example:

<Name@UBUNTU:~> ls

<Name@UBUNTU:~> cd

<Name@UBUNTU:~> cd

<Name@UBUNTU:~> history

ls

cd

history

- history <num>:

Display only latest <num> commands.

Example:

<Name@UBUNTU:~> ls

<Name@UBUNTU:~> cd

<Name@UBUNTU:~> history

<Name@UBUNTU:~> history 3

cd

history

history 3

General notes

Useful commands/structs/files : uname, hostname, signal, waitpid, getpid, kill, execvp, strtok, fork, getopt, readdir, opendir, readdir, closedir, sleep, watch, struct stat, struct dirent,/proc/interrupts, /proc/meminfo, etc. and so on. Type: man/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.

1. 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.
2. Use of system() call is prohibited, if you use it you'll get zero marks.
3. You can use both printf and scanf for this assignment.
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/contact TAs
9. **Do not take codes from seniors or in some case, your batchmates, by any chance. We will evaluate cheating scenarios along with previous few year submissions (MOSS). So please take care.**
10. **Viva will also be conducted during the evaluation related to your code and also the logic/concept involved. If you're unable to answer them, you'll get no marks for that feature/topic that you've implemented.**

Submission Guidelines

- Upload format **rollnum_assgn2.tar.gz**.Make sure you write a **makefile** for compiling all your code (with appropriate flags and linker options)
- Include a readme briefly describing your work and which file corresponds to what part.