



Assignment 2

Creating an Interactive User Defined Shell

Course : Operating Systems (Monsoon 2018)

Deadline : 2nd September, Sunday, 23:59

This is an Individual Assignment, you need to be pretty strong with the basics and please do start early. The deadline will not be extended at any costs.

Implement a shell which supports semi-colon separated list of commands. Use 'strtok' to tokenise the command. Also your shell needs to have support of the '&' operator which lets a program to run in the background after printing the process id of the newly created process. Write this code in a **modular fashion**. In the next assignment, you would be adding 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 similar to the following must appear:
<username@system_name:curr_dir>

Example: <Name@UBUNTU:~/Documents/>

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: Built-in 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 these, using these doesn't mean it's built-in to your shell.

Specification 3: ls command

Implement **ls [al]** – (it should be able to handle **ls**, **ls -l**, **ls -a** and **ls -al/la**. For **ls** and **ls -a**, 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.

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

Specification 5: pinfo command (user defined)

pinfo : prints the process related info of your shell program.

Ex: <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.

Ex: <Name@UBUNTU:~>pinfo 7777

pid -- 7777

Process Status -- {R/S/S+/Z} memory

- 123123 { *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:

1) Personal Reminder:

Implement a reminder command 'remindme' which reminds you with a custom message after the given number of seconds elapse. You should allow the shell to take in commands during this time.

Example:

```
<Name@UBUNTU:~> remindme 20 "Go to OS Class."
```

```
<Name@UBUNTU:~>
```

```
....
```

```
Reminder: Go to OS Class.
```

```
<Name@UBUNTU:~>
```

2) Clock Command :

Implement a '**clock**' command which displays dynamic date and time in your shell. Updates should be done based on fixed time interval given as an argument. The usage of 'date' or similar commands is not allowed.

Example:

```
<Name@UBUNTU:~> clock -t 3
```

```
11 Aug 2018, 15:18:21
```

```
11 Aug 2018, 15:18:24
```

```
11 Aug 2018, 15:18:27
```

```
.....
```

Hint: Look for RTC driver files/logs.

General Notes:

1. **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.
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 think about 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.