Assignment 3

Extending the OS Shell

Computer Systems Engineering -1

Deadline : 23rd September, (Monday), 23:55

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 cost.

The goal of this assignment is to enhance your user defined interactive shell program so that it can handle **background and foreground processes** and **handle signals** sent to them. It should also be able to handle **input/output redirections and pipes**.

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

Specification 1: Foreground and background processes

Apart from the user defined commands, 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. If the background process exits then the shell must display the appropriate message to the user.

E.g:

Example 1

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

Example 2

<Name@Ubuntu:~> echo hello
<Name@Ubuntu:~> emacs &
<Name@Ubuntu:~> ls -l (display the output)
<Name@Ubuntu:~> emacs with pid 456 exited normally
<Name@Ubuntu:~>

Specification 2: Implement input-output redirection functionality

Output of running one (or more) commands must be redirected to a file. Similarly, a command might be prompted to read input data from a file and asked to write output to another file. **Appropriate error handling must be done** (like if the input file does not exist – display error message, if output file does not exist - create one with **permissions of 644**, etc.)

Output Redirection - Ex: <NAME@UBUNTU:~> diff
file1.txt file2.txt > output.txt

Input Redirection - Ex:
<NAME@UBUNTU:~> sort < lines.txt

Input-Output Redirection - Ex: <NAME@UBUNTU:~>
sort < lines.txt > sortedlines.txt

Note: There is another clause for output direction '>>', and that must be

implemented appropriately.

Specification 3: Implement command redirection using pipes

A pipe is identified by "**|**". One or more commands can be piped as the following examples show. Your program must be able to support any number of pipes.

Two Commands - Ex:
<NAME@UBUNTU:~> more file.txt | wc

Three Commands - Ex: <NAME@UBUNTU:~> grep "new"
temp.txt | cat somefile.txt | wc

Specification 4: Implement i/o redirection + pipes redirection

Eg:

<NAME@UBUNTU:~> ls | grep *.txt > out.txt
<NAME@UBUNTU:~> cat < in.txt | wc -l > lines.txt
Hint*: treat input output redirection as an argument to the command and handle it appropriately*

Specification 5: User-defined commands

The following commands must be supported by your shell:

- **setenv *var* [*value*]** : If environment variable *var* does not exist, then your shell must create it. Your shell must set the value of *var* to *value*, or to the empty string if *value* is omitted. Initially, your shell inherits environment variables from its parent. Your shell must be able to modify the value of an existing environment variable or create a new environment variable via the setenv command. Your shell must be able to set the value of any environment variable; **It is an error for a setenv command to have zero or more than two command-line arguments.**

- **unsetenv *var*** : Your shell must destroy the environment variable *var*. It is an error for an unsetenv command to have zero command-line arguments.

- **jobs** : prints a list of all currently running jobs along with their pid, in particular, background jobs, in order of their creation along with their state – Running or Stopped.

<NAME@UBUNTU:~> jobs

[1] Running emacs assign1.txt [221]

[2] Running firefox [430]

[3] Running vim [3211]

[4] Stopped gedit [3213]

Here [4] i.e. gedit is most recent background job, and the oldest one is [1] emacs.

- **kjob <jobNumber> <signalNumber>** : takes the job id of a running job and sends a signal value to that process

<NAME@UBUNTU:~> kjob 2 9 It sends SIGKILL to the process firefox, and as a result it is terminated. Here 9 represents the signal number, which is SIGKILL. For further info, lookup "***man 7 signal***"

- **fg <jobNumber>** : brings a running or a stopped background job with given job number to foreground.

<NAME@UBUNTU:~> fg 3

Either brings the 3rd job which is *vim* to foreground or returns error if no such background number exists.

- **bg <jobNumber>** : changes a stopped background job to a running background job. <NAME@UBUNTU:~> bg 4

Changes *gedit* from Stopped in the background to Running in the background or return error if no such stopped background job exists. If that job is already running,

do nothing.

- **overkill :** kills all background process at once.

- **quit** : exits the shell. Your shell should exit only if the user types this "quit" command. It should ignore any other signal from user like : CTRL-D, CTRL-C, SIGINT, SIGCHLD etc.

- **CTRL-Z** : It should change the status of currently running job to stop, and push it in background process.

- **CTRL-C** : It should cause a SIGINT signal to be sent to the current foreground job **of your shell**. If there is no foreground job, then the signal should not have any effect.

Important Note for CTRL-C – When you run your shell from the standard Unix shell, your shell is running in the foreground process group. If your shell forks any child processes, they will also be a part of the foreground process group. Typing in a CTRL-C sends a SIGINT to every process in the foreground group, including your shell, which is not the desired behaviour here. What is desired is for the SIGINT to be sent only to your shell's foreground child processes (if any). Find a workaround for this and implement it – your shell should only exit when **quit** is typed.

Bonus Questions:-

- **Command Recall using 'UP' arrow key:**

  - When you press the 'UP' key and then press the 'ENTER' key, a new prompt is displayed with the previous command and then that command is executed. (similar to normal ubuntu environment).
  - If the 'UP' key is pressed 'K' times consecutively, the K^{th} previous command should be executed.

    Given : K <= 10

Example:

```
<Name@UBUNTU:~> ls
shell.c
<Name@UBUNTU:~> echo “Hello”
Hello
<Name@UBUNTU:~> ^[[A
<Name@UBUNTU:~> echo “Hello”
Hello
<Name@UBUNTU:~> ^[[A^[[A^[[A
<Name@UBUNTU:~> ls
shell.c
```

Explanation:

1) ‘ls’ is executed.
2) ‘echo’ is executed.
3) ‘Up’ key is pressed and then ‘Enter’. Prompt displays with ‘echo’ and it is executed.
4) ‘Up’ key is pressed 3 times and then ‘Enter’. ‘ls’ was the 3rd previous command and is therefore printed and then executed.

- Cronjob:

Implement a **‘cronjob’** command which executes a particular command in fixed time interval for a certain period.

Example:

<Name@UBUNTU:~> cronjob -c ls -t 3 -p 6

This command should execute **‘ls’** command after every 3 seconds until 6 seconds are elapsed. In this example, **‘ls’** command should be executed 2 times, once after 3 seconds and then after 6 seconds.

General Notes:

1. Use of ***popen, pclose, system()*** calls is not promoted, Your marks will be deducted if you use any of these.

2. Useful Commands: getenv, signal, dup2, wait, waitpid, getpid, kill, execvp, malloc, strtok, fork, setpgid, setenv, getchar and so on.

3. Use exec family of commands to execute system commands. 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.

4. Use fork() for creating child processes where needed and wait() for handling child processes.

5. Use **signal handlers** to process signals from exiting background processes.

6. 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.

7. 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.

8. If code doesn't compile, it is zero marks.

9. Segmentation faults at the time of grading will be penalized.

10. You are encouraged to discuss your design first before beginning to code. Discuss your issues on Moodle and contact TAs, if you find any problem.

11. 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.

12. 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_assgn3.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.