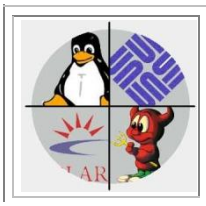

Assignment 1 (Phase 2)

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

Deadline: 18 th September (Friday)

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

Specification 1: Foreground and background processes

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

<NAME@UBUNTU:~>emacs &
<NAME@UBUNTU:~> ls -l -a

.
.
. *Execute other commands*
.
.

<NAME@UBUNTU:~> echo hello

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:~>

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 etc.)
E.g. **Output redirection**
<NAME@UBUNTU:~>diff file1.txt file2.txt > output.txt
E.g. **Input redirection**
<NAME@UBUNTU:~>sort < lines.txt
E.g. **Input-Output redirection**
<NAME@UBUNTU:~>sort < lines.txt > sorted-lines.txt

Note: There is another clause for output direction '>>', and must be implemented too 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.
E.g. **Two Commands**
<NAME@UBUNTU:~>more file.txt | wc
E.g. **Three commands**
<NAME@UBUNTU:~>grep "new" temp.txt | cat - somefile.txt | wc

Specification 4: Implement i/o redirection + pipes redirection

E.g.
<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 the shell

-**jobs** : prints a list of all currently running jobs along with their pid, particularly background jobs, in order of their creation.
<NAME@UBUNTU:~>jobs
[1] emacs Assign.txt [231]
[2] firefox [234]
[3] vim [5678]

Here [3] i.e vim is most recent background job, and the oldest one is 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 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.

-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

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* 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 portal and contact us, if you find any problem
11. 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.

OStas,ICS231