Advanced Operating Systems
Assignment 2: POSIX Shell Implementation
[Monsoon 2024]

Deadline: 31st August 2024 11:59:00 PM

Task:

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

The goal of the assignment is to create a user-defined interactive shell program using c/cpp 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. Your shell can handle background and foreground processes and also 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:

Requirements:

1. **Display requirement**: When you execute your code, a shell prompt of the following form must appear along with it. *Do NOT hard-code the user name and system name here*. _username_@_system_name_:_current_directory_>

   Example:

   <Name@UBUNTU:~> vim &

   <Name@UBUNTU:~/newdir/test> cd ..

   <Name@UBUNTU:~> ls -a -l ; cd test

   <Name@UBUNTU:~/test>

   The directory from which the shell is invoked will be the home directory of the shell and should be indicated by "~".
   Your shell should account for random spaces and tabs.
   This is NOT hard to implement, just tokenize the input string appropriately.

2. **cd, echo, pwd**: . If the user executes "cd" i.e changes the directory, then the corresponding change must be reflected in the shell as well. If your current working

directory is the directory from which your shell is invoked, then on executing command "cd .." your shell should display the absolute path of the current directory from the root.

echo and pwd are also shell built-ins. Make sure you implement cd, pwd, and echo. **DON'T use 'execvp' or similar commands for implementing these commands.**

Note :

- For echo, handling multi-line strings and environmental variables is **not** a requirement
- You **do not need to handle escape flags and quotes**. You can print the string as it is. However, you must handle tabs and spaces Example: echo "abc 'ac' abc" "abc 'ac' abc"
- For cd apart from the basic functionality, implement the flags ".", "..", "-" and "~".
- It is an error for a cd command to have more than one command-line argument (print: *Invalid arguments for error handling*). **If no argument is present then you must cd into the home directory**.

3. **ls**: Implement the ls command with its two flags "-a" and "-l". You should be able to handle all the following cases also:

    - ls
    - ls -a
    - ls -l
    - ls .
    - ls ..
    - ls ~
    - ls -a -l
    - ls -la / ls -al
    - ls <Directory/File_name>
    - ls -<flags> <Directory/File_name>

Example: <Name@UBUNTU:~> ls -al test_dir

Note:

- For ls, ls -a and ls <Directory_name> outputting the entries in a single column is fine.
- You can assume that the directory name would not contain any spaces.

● For the “-l”, kindly, print the exact same content as displayed by your actual Shell. You can leave out the colors but the content should be the same.

● DON’T use ‘execvp’ or similar commands for implementing this.

● Multiple flags and directory names can be tested. Your shell should also account for these arguments in any order.

Example:
<Name@UBUNTU:~> ls -l <dir_1> -a <dir_2>
<Name@UBUNTU:~> ls -la <dir_1> <dir_2>

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.

4. **System commands (background/fg), with and without arguments:**

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 a background command. This implies that your shell will spawn that process and doesn't wait for the process to exit. It will keep taking other user commands. Whenever a new background process is started, print the PID of the newly created background process on your shell also.

Example: *<Name@UBUNTU:~>* gedit &
456

<Name@UBUNTU:~> ls -l -a

Note:

● You do NOT have to handle background processing for built-in commands (ls, echo, cd, pwd, pinfo). Commands not implemented by you should be runnable in the background.

● Your shell should be able to run multiple background processes, and not just one. Running pinfo on each of these should work as well.

5. **pinfo**

This prints the process-related info of your shell program. Use of “**popen()**” for implementing pinfo is NOT allowed.

Example: *<Name@UBUNTU:~>*pinfo
pid -- 231
Process Status -- {R/S/S+/Z}
memory -- 67854 {Virtual Memory}
Executable Path -- ~/a.out

pinfo <pid> : This prints the process info about the given PID.

Example: <*Name@UBUNTU:~*> pinfo 7777
pid -- 7777
Process Status -- {R/S/S+/Z}
memory -- 123456 {Virtual Memory}
Executable Path -- /usr/bin/gcc

Process status codes:

1. R/R+: Running
2. S/S+: Sleeping in an interruptible wait
3. Z: Zombie
4. T: Stopped (on a signal)

Note: “+” must be added to the status code if the process is in the foreground

6. **search**

Search for a given file or folder under the current directory recursively. Output should be **True** or **False** depending on whether the file or folder exists

Example: <*Name@UBUNTU:~*> search xyz.txt
True

7. **I/O redirection**

Using the symbols <, > and >>, the output of commands, usually written to stdout, can be redirected to another file, or the input taken from a file other than stdin. Both input and output redirection can be used simultaneously. Your shell should support this functionality. Your shell should handle these cases appropriately: **An error message should be displayed if the input file does not exist**. The output file should be created (**with permissions 0644**) if it does not already exist. In case the output file already exists, it should be overwritten in case of > and appended to in case of >> .

Example: # output redirection
<*Name@UBUNTU:~*> echo "hello" > output.txt

input redirection
<*Name@UBUNTU:~*>cat < example.txt

input/output redirection
<*Name@UBUNTU:~*> sort < file1.txt > lines_sorted.txt

8. **pipeline** A pipe, identified by **|**, redirects the output of the command on the left as input to the command on the right. One or more commands can be piped as the following examples show. Your program must be able to support any number of pipes.

   Example: # two commands
   <*Name@UBUNTU:~*> cat file.txt | wc

   # three commands
   <*Name@UBUNTU:~*> cat sample2.txt | head -7 | tail -5

9. **Redirection with pipeline** Input/output redirection can occur within command pipelines, as the examples below show. Your shell should be able to handle this.

   Example:
   <*Name@UBUNTU:~*> ls | grep *.txt > out.txt; cat < in.txt | wc -l > lines.txt

10. **Simple signals**
    1. **CTRL-Z** It should push any currently running foreground job into the background, and change its state from running to stopped. This should have no effect on the shell if there is no foreground process running.
    2. **CTRL-C** It should interrupt any currently running foreground job, by sending it the SIGINT signal. This should have no effect on the shell if there is no foreground process running.
    3. **CTRL-D** It should log you out of your shell, without having any effect on the actual terminal.

11. **Autocomplete for all the files/directories under the current directory.** Pressing the **TAB** key should either complete the command or output a list of matching files/dirs (space separated) if there are more than one.

    Example: # assume alpha.txt, alnum.txt are 2 files in your current path:

    # single match
    <*Name@UBUNTU:~*> cat alp<TAB>: completes ‘ha.txt’

    # multiple matches
    <*Name@UBUNTU:~*> cat
    al<TAB> alpha.txt alnum.txt

12. **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. **You must overwrite the oldest commands if more than 20 commands are entered**. You should track the commands across all sessions and not just one.

Example: <*Name@UBUNTU:~*> ls
<*Name@UBUNTU:~*> cd
<*Name@UBUNTU:~*> cd
<*Name@UBUNTU:~*> history
ls
cd
history
<*Name@UBUNTU:~*> exit

When you run the shell again

<*Name@UBUNTU:~*> history
ls
cd
history
exit
history

Extensions of the above command are:

a. **history <num>**: Display only latest <num> commands.

Example: <*Name@UBUNTU:~*> ls
<*Name@UBUNTU:~*> cd
<*Name@UBUNTU:~*> ls
<*Name@UBUNTU:~*> history
ls
cd
ls
history

<*Name@UBUNTU:~*> history 3
ls
history
history

Note: For this part can be more than 10 but will always be less than 20 which is the total number of commands your shell remembers.

b. **Up Arrow Key**: On clicking the UP arrow key, you must loop over the previous commands present in your shell's history and show them on the prompt. In case you reach the first command or have no history, then stay on the same command if UP is pressed.

Example: <*Name@UBUNTU:~*> ls

● After a command is shown on the prompt using the UP arrow key, it can be

<*Name@UBUNTU:~*> cd

<*Name@UBUNTU:~*> echo hello

Now if we press UP once, “echo hello” should be displayed as the command on the prompt. If we press UP again, your shell should show “cd”. If we press UP again, show “ls”. Now, as this is the last command in history, nothing will happen on pressing UP again and the shell will continue to display “ls”.

Note:

modified. For Example, after getting “cd” in the above example, we can add a directory name in front of it to change it to “cd” and then execute it by pressing the Enter key.

- UP arrow key will ONLY be pressed when the prompt is empty i.e., no other input is written in the prompt

Submission:

Upload format: <roll_Number>_Assignment2.zip

Make sure you write a makefile for compiling all your code (with appropriate flags and linker options).
Kindly adhere to the following naming guidelines and directory structure:

<roll_number>_Assignment2
├── README.md
├── makefile
└── Other files and Directories

2. Include a README.md file briefly describing your work and which file corresponds to what part. **Including a README file is NECESSARY**

General notes

1. Useful commands/structs/files/routines: uname, getenv, hostname, signal, waitpid, getpid, kill, execvp, strtok, fork, getopt, readdir, opendir, readdir, closedir, getcwd, sleep, watch, struct stat, struct dirent, /proc/interrupts, fopen, chdir, getopt, pwd.h (to obtain username), /proc/loadavg, etc.
Type: man/man 2 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. Some helpful routines and systemcalls: signal, dup, dup2, wait, waitpid, getpid, kill, execvp, malloc, strtok, fork, setpgid, setenv and getchar.

3. Use the exec family of commands to execute system commands. If the command fails to 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 waiting for and reaping them.

5. Use signalhandlers to handle signals when background processes exit.

6. The user can type the command anywhere on the command line, leaving spaces and tabs in between. Your shell should be able to handle this.

8. You need not implement background functionality for internal commands such as cd, ls, etc.

9. You have to implement piping and redirection for internal commands.

10. You need not implement redirection operators like 2>&1, &>, >& or 2>.

11. The symbols <, >, >>, &, |, ; , - would always correspond to their special meaning and would not appear otherwise, such as in inputs to echo etc.

Guidelines:

1. **The Assignment must ONLY be done in C or C++. NO other languages are allowed**.
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. The user can type in any command, including running another process instance of your shell program. **You MUST do error handling** for both user-defined and system commands.
4. Use of **popen**, **pclose**, **system() call is prohibited**.
5. curses/ncurses library is not allowed.
6. You can use exec command for "any other" commands that are not listed in the pdf.
7. You are **not** allowed to use environment variables **OLDPWD, PWD**.
8. You are not allowed to use filesystem library.
9. You are allowed to use pipe() system call.
10. You can use both “printf” and “scanf” for this assignment.
11. 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.
12. 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.
13. If the code doesn't compile, no marks will be rewarded.
14. Segmentation faults at the time of grading will be penalized.
15. **Do NOT take codes from seniors or your batchmates, by any chance. We will extensively evaluate cheating scenarios along with the previous few year’s submissions**