Operating Systems. Home Work #2.

Due November 29, 2018, 23:59.

In this assignment, you will use tools you've learned to create a simple shell program. We provide (attached to the assignment) a skeleton shell program that reads lines from the user, parses them into commands, and calls a **process_arglist()** function to handle the command. You just have to implement **process_arglist()** and any initialization/finalization code required to implement the required shell functionality (more details below).

To complete this assignment, learn and use the following: ***execvp*** *(*man **3** execvp, *not to be confused with execv!),* ***pipe, dup, dup2*** (man 2), ***SIGCHLD, wait, setpgid***.

Cheating Policy

Part of the course requirements is to program assignments by yourself. You can discuss concepts and ideas with others, but looking at other people's code, looking at code from previous semesters, sharing your code with others, coding together, etc. are all not allowed. Students caught violating the requirement to program individually will receive a "250" grade in the course and will have to repeat it. *Even if you're under pressure, ask for help or even don't submit, rather than cheat and risk having to repeat the course.*

If you put your work in your TAU home directory, set file and directory permissions to be accessible only by you.

Late Submission Policy

You have 5 grace days throughout the semester, you can use all of them, some of them or none of them for this exercise. A 10 point deduction will be applied for every late day past the grace days.

Submission Guidelines

The provided code file contains the *main* function of the shell. You should create another file (***myshell.c***) implementing only the *process_arglist, prepare* and *finalize* functions, and link it together with the provided file for testing.

Submit just this single **myshell.c** file. **Document it** properly – with a main comment at the beginning of the file, and an explanation for ***every non-trivial*** part of your code. Help the grader understand your solution and the flow of your code.

Do not modify the code file we provide, and **do not** submit it. Only use it with your own code for testing (in ***myshell.c***).

We check

1. Programs implement the specified behavior.
2. C program compiles without warning with "`gcc -O3 -Wall -std=gnu99`"

Code structure

The skeleton we provide reads & parses input lines into an array "*arglist*" with "*count*" elements. It then invokes *process_arglist(count, arglist)*. The *arglist* array is just a split of the input line into words (a word is a non-empty sequence of non-whitespace characters, where whitespace is space, tab, or newline).

Empty lines are detected and ignored (already handled by the skeleton). If Ctrl-D is pressed, the skeleton exits.

Shell specification

1. **Behavior of process_arglist()**
   - Commands (a program and its arguments) specified in the *arglist* should be executed by the *process_arglist()* function as a child process using ***fork*** and ***execvp*** (man **3** execvp, not *execv*). Note that the *arglist* array is not necessarily the final argument for ***execvp*** (see below).
   - In the original (shell/parent) process, *process_arglist()* should always return 1. This makes sure the shell continues processing user commands.
   - Support background processes
     - If the last argument of the *arglist* array is "&" (ampersand only), run the child process in the background: the parent should not wait for the child process to finish, but instead continue executing commands.
     - Do not pass this argument (&) to ***execvp***!
     - Assume background processes don't read input (*stdin*), that "&" can only appear as the last argument, and that it's always separated by a whitespace from the previous argument.
   - Support nameless pipes
     - If *arglist* contains "|" (pipe only), run two child processes, correctly piping their *stdin* and *stdout* together.
     - To pipe input and output of processes, use the ***pipe*** *and* ***dup2*** system calls.
     - Use the same array for all ***execvp*** calls by referencing items in *arglist*, no need to duplicate, malloc, etc.
   - The *process_arglist()* should not return until the last foreground child process it created exits.
   - You can assume the following:
     - No more than a *single* pipe (|) is provided, and it is correctly placed (at least 1 arg before and after, and separated by a white-space).
     - Pipes and background processes will not be combined, i.e., nameless pipes will never be run in the background (however, other background processes might still be running from previous commands).
     - You do not need to support built-in shell commands such as *cd* and *exit*, only execution of program binaries as described above.
     - No need to support arguments with quotation marks or apostrophes. Assume these characters are never provided.
     - The results of the provided parser are correct.
2. **Error handling**
   - If an error occurs in the parent process, print a proper error message and have *process_arglist()* return 0. There's no need to notify anything to any running child processes. If an error occurs in a child

process (before it execs), output a proper error message and terminate **only** the child process with *exit(1)*. Nothing should change for the parent or other child processes.

- The user command might be invalid! (e.g., a non-existing command/program). This should be treated as an error in the child process (i.e., it must *not terminate the shell*).
- Print error messages to *stderr*.
- Error message do not have to be worded exactly as the existing terminal, anything returned by *strerror* is ok.
- Unlike in HW#1, you do not have to exit "cleanly" on error. You may terminate the child or parent processes (depends on where the error originated) without freeing memory.

3. **General requirements**
   - You should prevent zombies and remove them as fast as possible.
   - Below is the specification for how the processes you create should handle receiving SIGINT:
     - After prepare() finishes, the parent (shell) **should not terminate** upon a SIGINT.
     - Foreground child processes (regular commands or parts of a pipe) **should terminate** upon SIGINT.
     - Background child processes **should not terminate** upon SIGINT.
     - Be sure to ignore SIGINT correctly – a process must **always** follow the above rules, in any point of its execution. (Recall that ***fork*** duplicates a process, including its signal handlers.)
   - The skeleton calls a *prepare()* function when it starts. This function returns **0** on success; any other return value indicates an error. You need to provide this function. You can use it for any initialization and setup that you think are necessary.
   - The skeleton calls a *finalize()* function before it exits. This function returns **0** on success; any other return values indicates an error. You need to provide this function. You should use it to free any resources—but not the arglist, that is taken care of by the skeleton code.

Guidelines

If you are unsure how something should work, base it on the existing terminal and bash shell in your VM. *If your shell behaves like the standard bash shell – it is correct!* **But there is one exception to this rule:** standard shells use a special mechanism that prevents SIGINT from being sent to background processes when the user presses Ctrl-C (read about *setpgid()* and this link if you're curious). Your shell won't use this mechanism, so the behavior of background processes receiving a SIGINT won't be the same as in bash.

If you're still unsure – ask in the forums. **As always**, closely follow the forums and all questions & answers provided there. Explanations, guidelines and relaxations may be given there, and they are **mandatory**.