Operating Systems. Home Work #2.

Due to December 2, 2017, 23:55.

In this assignment, you will use tools you've learned to create a simple shell program. For your shell, learn and use the following: ***execvp*** *(*man **3** execvp, *not to be confused with execv!),* ***pipe***, ***dup***, ***dup2*** (man 2), ***SIGCHLD***.

Code structure

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).

The skeleton 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). It is not necessarily the final argument for ***execvp*** (see below).

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()**
   - Create a child process that executes the command specified in the *arglist* using ***fork*** and ***execvp*** (man **3** execvp, not *execv*).
   - In the original (shell/parent) process, *process_arglist()* should always return 1 from *process_arglist*. This makes sure the shell continues processing user commands.
   - Unless specified otherwise (see next bullet), the function should not return until the child process exits.
   - 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*), and that "&" can only appear as the last argument.
   - Support nameless pipes
     - If *arglist* contains "|" (pipe only), run two child processes, correctly piping their *stdin* and *stdout* together.
     - You may create two child processes of the original shell, a hierarchy where one process is the parent of the previous one, etc. You're free to choose whatever option so long as it works correctly.
     - To pipe input and output of processes, use ***pipe*** *and* ***dup2*** system calls.
     - Use the same array for all ***execvp*** calls by referencing items in *arglist*, no need to duplicate, malloc, etc.

- 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).
  - 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 special shell commands (e.g., *cd, exit*); only those specified, and executing commands.
  - 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, terminate it with a proper error message and *exit(1)*; 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.
- Handling of interrupts (receiving SIGINT):
  - 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 (recall that ***fork*** duplicates a process, including its signal handlers).
  - When the child process finishes, restore the parent signal handler to the previous (**not necessarily default!**) handler.
- 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.

Submission Instructions

The provided code file contains your *main* function. You should create another file (***myshell.c***) implementing the *process_arglist, prepare* and *finalize* functions.

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 given code file, and **do not** submit it. Only use it with your own code (in ***myshell.c***).

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!*

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**.

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