Operating Systems (CS 444/544)

Programming Assignment: Implementing a Shell

OVERVIEW

A shell is an interface between the user and the internals of an operating system. It is the shell's job is to parse the user's commands and ask the OS to launch processes in response.

In this assignment, you will write your own command shell to gain experience with some advanced programming techniques like process creation and control, file descriptors, signals and possibly pipes. We have provided a simple parser for you to use and you need to implement certain features in your shell.

You will submit your code with a README file to describe your assignment's status (noting any any references you used, any required features that do not work, and any extra features you may have included).

REQUIRED FEATURES

Here are the features you should support in your shell although you will probably not implement them in this order :

1. **Print prompt with current working directory, username, and hostname.**
   For example: `username@hostname:/bin/foo/bar$`
   Hint: use getcwd(), getlogin(), gethostname() syscalls

2. **Support shell built-ins including** `jobs, cd, pwd, history, exit, echo, kill, help, clear`**.**
   jobs: provide a numbered list of processes currently executing in the background. Try waitpid with WNOHANG option to check without blocking.

   cd: changes the working directory

   pwd: print working directory

   history: print the list of previously executed commands. The list of commands should be numbered such that the numbers can be used with ! to indicate a command to repeat. Store at least 10 previous commands. A user should be able to repeat a previously issued command by typing !number where number indicates which command to repeat.

   clear: clears the terminal

help: lists the available built-in commands and their syntax.

exit: should terminate your shell process (template handles this!).

echo: prints out the rest of the arguments (template handles this!)

kill %num: terminates the process numbered in the list of background processes returned by jobs (by sending a SIGKILL signal). Hint: use the kill(pid, SIGKILL) syscall. Note: Usually kill num refers to the process with ProcessId num; while kill %num refers to the process in the jobs list with number num

3. **Execute commands (exec_cmd function) returned by the parser including pipes, I/O redirection, backgrounding. Allow the user to specify commands either by relative, or absolute pathnames. Hint: use execvp() to handle searching paths automatically.**

    a. **You should be able to place commands in the background with an & at the end of the command line. (You do not need to support moving processes between the foreground and the background (ex. bg and fg). You also do not need to support putting built-in commands in the background.)**

    b. **You should be able to redirect STDIN and STDOUT for the new processes by using < and >. For example, foo < infile > outfile would create a new process to run foo and assign STDIN for the new process to infile and STDOUT for the new process to outfile. In many real shells it gets much more complicated than this (e.g. >> to append, > to overwrite, >& redirect STDERR and STDOUT, etc.)! You do not have to support I/O redirection for built-in commands (it shouldn't be too hard but you don't have to do it.)**

4. **If the user chooses to exit while there are background processes, notify the user that these background processes exist, do not exit and return to the command prompt. The user must kill the background processes before exiting.**

OPTIONAL EXTRA CREDIT FEATURES

If you enjoy this assignment and would like to add more advanced features to your shell, here are some sugguestions:

1. **You could support optional parameters to some of the built-in commands. For example, history -s num could set the size of the history buffer and history num**

could return the last num commands. You could also support additional built-in commands like which, pushd/popd or alias. If you make modifcations of this type, I would recommend help command to return more detailed information on a single command.

2. **You could support | , a pipe, between two processes. For example, foo | bar would send the STDOUT of foo to the STDIN of bar using a pipe. You may want to start by supporting pipes only between two processes before considering longer chains. Longer chains will probably require something like handle process n and then recursively handle the other n-1.**
3. **You could implement more advanced I/O redirection (>&, >!, etc.).**
4. **You could implement the built-in shell functions, fg and bg, to move processes between the background and the foreground.**
5. **You could support the editing of shell variables with built-in shell functions like printenv and setenv.**
6. **I wouldn't recommend it :-), but you could even write support for shell programming (e.g. if/then, while, for constructs).**
7. **Terminal support: You may notice that some programs like more or emacs are aware of the screen real estate they need - that requires terminal emulation support.**
8. **You could port the code to run on Windows as well using the CreateProcess/WaitForSingleObject. (count as two features)**