Simon Owens
Operating System CS470
Shell Project

High-level Design

The python script creates a shell for the user. This shell also allows users to see and enter their previous commands.

This program contains 3 main functions: child(), parent(), main(). This design was chosen to clearly show when a subprocess or main process is executing code. In this program, the subprocesses execute the user's commands for them. The main() function takes input from the user. They can either enter a command to be executed, see their history, execute their history, or quit. If they choose to executer commands, they make a call to the parent() function.

The parent process forks() - this creates a child process. Before the user input is executed, the parent function checks to see if the user wanted this task executed in the background. If they wanted this executed in the background, then the "double fork" method is used to prevent from subprocesses never being destroyed. If the user does not want their command executed in the background, they simply call the child() function to be run.

The child parses the user's command. The child then executes the users command. Once the command has finished, the child destroys itself. Execution then returns to the main program again. The user can choose to execute more commands, see their history, or quit.

To run this program, enter "python3 shell-project.py" in the same directory as the script. Python 3.5 must be on the system.

Test Plan

The test plan covers 4 main areas
- Parsing User input
- Executing User input
- Creating and Destroying sub-processes
- Allowing the user to execute previous commands

Follow the test procedure for ensuring the shell has full functionality
1) Run "python3 shell-project.py"
2) Type "ps" - notice there should only be pyhon3 and ps running
3) Type "ls" - you should see the entire contents of your current directory
4) Type "ps -aux" - this tests for arguments being passed to the command
5) Type "emacs test &" - this tests running background processes, you should still be able to type in commands
6) Type "dfasdfasdf" - this should not run any commands, but should not crash anything either. You should still have full functionality after garbage input

7) Type “history” - you should see all of the commands you’ve typed
8) Type “ps” again, then “!!”, this should execute the ps again
9) Type “history” again, then “!3” to bring up the emacs prompt in the background again
10) Type “./test-hello.py” to test for running other simple programs
11) Type “quit” to exit the shell
12) Type “ps” again to ensure all subprocesses were killed

Additional Questions

1) It was really weird keeping track of the process ID. How inside the program, the child is 0, but outside the program it is a regular process number. The parent process has the same process ID inside and outside of the file.
2) Fork() was super easy to understand. If you want another process, just fork() and boom, you got a kid that you don’t have to feed.
3) It doesn’t make any sense to have a child() function and parent() function. It makes much more sense to have a function that executes a command, a function that creates and deletes a process, and a function that parses the users input. I left the current format because it makes more sense for understanding how parent ↔ child processes work/relate.
4) I’m not sure that I found anything interesting or surprising about process manipulation. The hardest and most surprising part was working with execvp().