CMPUT 379
hdesmara
Hannah Desmarais

CMPUT 379 Assignment 1 Report: msh379

Objectives

This assignment is aimed to teach a student about how to manipulate processes with the use of signals, forking, run commands, and waiting for changes of state. It also is intended to show the student how they can track processes and the time it takes for them to run as well as how to allocate resource limits. Lastly, it helps them also practice extracting and using environment variables as well as parsing and executing user commands.

Design Overview

Important features:

- Commands are case sensitive.
- There is a maximum of 32 tasks that may be accepted. Terminated tasks do not get removed from the list.
- A task is only accepted if the execlp() command is successful, not if there is a successful fork so that faulty tasks do not fill the 32 slots.
- In order to catch the status of the child and determine if execlp() was successful, the parent does sleep for a second to allow the execlp() to run or exit.
- A list of available commands is shown if there is an improper command entered.
- Tasks are stored in a vector for more flexibility than an array.
- All command lines are in their own method for readability and organization but the main loop will handle the decision of who gets called.
- I chose not to remove the $ in front of environment variables when the user makes a “cdir …” command. I did this so that when I go to parse the command, I can know that what follows is an environment variable and I can grab all characters up until the first “/”. This allows me to generalize my getenv() call instead of having to check for the actual variable names.
- For the check command, I chose to store any PID that qualified as a match with the PID given by the user in a vector so that I don’t need to know how many descendants there are right off the bat. I can do this as I go because the ps command will sort the processes by their PID so I know that any child of the head process, or a subsequent one, will be shown further down in the list. This way, all I must do is check the PPID for each line and see if my vector contains a matching PID. If it does, I know it is a descendant and print its information and store it's PID so I can check later if it has any children as well. Using this implementation, I am able to get all descendants of a PID, not just the direct children.

CMPUT 379
hdesmara
Hannah Desmarais

Project Status

As of now, my project is finished. Everything works as it should and has been tested by myself. Sometimes when I switch to the lab machine instead of using Ubuntu on my laptop, the code has some odd behaviour with tasks being accepted, but I believe I have managed to fix it. The trickiest part of the project for myself was getting the execlp() to work and run files. I had some issues getting the myclock code to work that was posted on eclass but managed it in the end. Another difficulty was getting the task to only be accepted after execlp() ran. I didn't realize at first that it would exit the child and wouldn't execute any code below it in the child.

Testing and Results

The following was tested:

- cdir pathname: I tested this by feeding the code pathnames that didn't exist and checking that an error message would be given and the user would be prompted for their next command. This was also tested by giving it proper pathnames to change directories to. I first would make sure no error message would come up, and the use pdir to check if I was in the same directory as what I had given for a pathname.
- pdir: This was tested by calling it immediately after running the shell and making sure it matched where the code was stored on my laptop. Then I would use cdir to change the directory and print the directory again, checking if it now matched the directory I gave as an argument. If there was ever an error with the actual getcwd() call, I made sure an error message would appear. I tested this by manually passing it a bad path and checking that the error message would appear if pdir was entered on the command line.
- lstasks: To check that this was working correctly, I first would enter it immediately on starting the shell to make sure that it didn't display anything as I shouldn't have any tasks yet. I made sure that no tests that didn't successfully run wouldn't be added to the task list by trying to run a program that doesn't exit, waiting for the error message, and then typing lstasks and making sure it was still empty. Then I would run a bunch of programs and call lstasks. If all were there, I then terminated one and called lstasks again. If that process had disappeared from the list, I would terminate all of them and call lstasks again and make sure no processes remained in the list. While doing all this, I would have another window open and repeatedly call ps to make sure all of the PID's matched and that lstasks really wasn't showing anything that was terminated.
- run pgm arg1 · · · arg4: To test run, I provided names of programs that didn't exist and would make sure that the error message would be presented. If I gave it a proper program (such as myclock) I would check that it had begun running, as well as any children it had too, by using the ps command in another window. If I gave arguments for the program, I would make sure they also applied to the program as they should. Such as making sure if I specified an outfile to myclock that it was generated.
- stop taskNo, continue taskNo, terminate taskNo: First, I would try to run each stop, continue, and terminate with an index of 0 right when the shell starts to verify that it gives an error when there aren't any tasks. Then I ran a program like myclock. To test stop I would run a

program and open a window and use the ps command to make sure it was running. I would then enter stop and the index of the process. To make sure the program was paused I would use the ps command again and verify the status. If it had successfully been paused, I would then use the command continue with the same index. Checking with ps again, I would make sure the process was then continue. After this I would then use enter terminate, again with the same index, and verify it worked as well with the ps command. I also entered all three with an index of -1 and an index larger than how big the vector was to verify that error messages would be given.

- check target pid: To test check, I first used a false pid to make sure that an error message would be presented. Then I would run a program and use the check command with the PID of the child (found by running ps in another window) to make sure it would print that it was running as well as the information of the process as well as any of it's children as well. I would then check the PID of msh379 and see if it printed all of it's descendants as expected. I then would terminate the child of msh379 and use the check command on the child pid again. If it printed that the process had been terminated and only the information for it's pid, no children, then I knew it worked.
- Exit and quit: To check these I would run programs and then call either one. If I chose exit, I used the ps command in a window to check that the process was gone from the list. If I chose quit, I would make sure that the program was still running even after the shell closed.
- Times(): To test the times function I would simply open the shell and time how long it was open with my phone and enter quit and stop my timer at the same time. If they matched it worked.
- Split(): I tested this by simply putting in words on the command line and then printing out the token array and making sure it was split by the blank spaces.
- Wrong args: I made sure that any incorrect command line entries would result in the printing of commands and waiting for a proper one by entering random words and making sure it didn't call anything.

Acknowledgements

- http://webdocs.cs.ualberta.ca/~cmput379/W22/379only/lab-tokenizing.html - I consulted this to learn how to build a split function.
- https://man7.org/linux/man-pages/man2/kill.2.html - This website was helpful in how to use the kill() function.
- https://linux.die.net/man/3/execlp - This website was used for information on the execlp() command
- https://www.educative.io/edpresso/splitting-a-string-using-strtok-in-c - This was used for more information on strtok()
- https://pubs.opengroup.org/onlinepubs/009696699/functions/pclose.html - This was used for more information on what exactly pclose() returns
- https://pubs.opengroup.org/onlinepubs/009696699/functions/pclose.html - This was used for information of waitpid(), WNOHANG, and WIFEXITED
- Lastly, the AUPE(3/E) textbook was used in the sections mentioned in the assignment.