Project No. 1: Quite a Shell (quash)

Anjali Pare and Victoria Maldonado

**Objective:**

Implement Quite a Shell (quash) using UNIX system calls. This shell has to be similar to bash including the following functionality:

* Run executables with and without arguments.
* Set environment variables
* Exit properly
* Change directories
* Child process environment inheritance
* Allow foreground and background execution
* Report current background processes running
* File redirection using > or <
* Allow the use of one pipe

**Implementation:**

In the quash shell code, we created a loop that gets user input and continues executing the specified command until “quit” or “exit” is typed. For each of the command line typed, we first split the same and then called specific functions depending on the keywords in the line read in.

1. Run executables without arguments.

On reading the user input, “launch” function is called with the command parameters. First, an existing process forks itself into two separate ones. Then, the child uses execvp() to replace itself and run a new program. If not a background process, parent waits for the child to finish execution using the system call wait(). It is important to send NULL at the end of the arguments list because execvp requires it to be a null terminated list. The first parameter of execvp is the executable name and the next is the argument list for that executable (In this case it is just NULL, because we are running executables with no arguments). We tested this using a dummy program that required no arguments and also using simple commands such as ls, pwd and quit.

1. Run executables with arguments

This has the same execution as the one above, only difference being in the argument list sent to execvp. In this case, it will contain the arguments required for the executable and complete execution.

1. set for HOME and PATH work properly

For this feature, when the user input contains the keyword “set”, the setPaths function is called. This function is responsible for first splitting the environment variable name from its value (these are passed to the function as a single string). Then setenv function is used to appropriately change the HOME and PATH environment variables. Depending on whether the setenv function was successful in changing the path or not, we print appropriate messages on the terminal. Once quash is done implementing this command, printenv VARIABLE\_NAME can be used to see if the environment variable was updated or not (VARIABLE\_NAME can be either HOME or PATH).

1. exit and quit work properly

Every command entered is first checked if it is equal to “quit” or “exit”. If yes, the quash shell exits successfully.

1. cd (with and without arguments) works properly

For this feature when the user input contains the keyword “cd”, the changeDir function is called. Depending on whether cd is executed with or without arguments, changeDir is given the HOME environment variable of the user entered directory name as input respectively. This function then call chdir with the passed in parameter and updates the PWD environment variable to correctly display the current directory the user is currently in. Since we use PWD to print the working directory of the user each time a new command is entered, this was easy to test.

1. PATH works properly. Give error messages when the executable is not found

For this feature to work, we made use of the execvp function call again in the launch function. If execvp returned a value less than 0, we printed the error message saying the executable was not found. We tested the execution by using a dummy executable that did not exist and the error message was successfully printed.

1. Child processes inherit the environment.

The property of a child process inheriting the environment from the parent is something that every process has automatically. We only had to make sure that we were able to set the PATH and HOME environment variables properly (explained in point 3: set Home and PATH variables works properly) to make sure that the changes will also be reflected in the child process once an executable was called. We tested this functionality by creating a program called “ls” in our current directory. After that, we called “ls” normally and we would see the normal and expected output of ls since the path was the default one. However, once we set the PATH variable to have a “.” at the beginning of the path, we were able to change the default way to look up things so if “ls” was called again, the ls dummy program would be executed instead of the default “ls” since it was at the beginning of the path.

1. Allow background/foreground execution

To implement this functionality, we first had to check if the input line had a “&” symbol at the end of it to make sure that the user wanted to run the process in the background. Once we knew what the user wanted to do, we would do the same steps as in point 1 (Run executables without arguments), with the only exception that the parent of the background child process would not wait for it. Instead, we implemented a way to send a SIGCHLD signal from every process that was ending. The job of the parent of that process was to handle the signal with a signal handler called childSignalHandler(). Once every parent got the signal of their child terminating, we only allowed the parents of zombie processes to proceed and print that the background process was terminated.

Separately from all background and foreground functionality, we created a vector to keep track of every background process that was being started and assign them JOBIDs to identify those processes. Also, we created another vector that was the one in charge of keeping track of the command used to start that process since we had to print that command once the process was terminated.

1. Printing/reporting background processes (jobs)

This feature of printing the background processes in progress was implemented using the printJobs() function that is called as soon as we check that the first command entered by user is “jobs”. In the last paragraph (8: Allow background and foreground processes), we mentioned that two vectors were created to store information about the background processes. One vector is used to store JOBIDs and PIDs and the other one to store JOBIDs and COMMANDs to easily print the jobis, pids, and commands of all the currently running background processes. Once the jobs finish, we just change the pid value to -100 and stop printing any job that has that value or was a foreground process to begin with.

1. Allow file redirection (> and <)

To allow file redirection in quash, we first had to see whether the input had a < or a > in the second to last command since we accounted for file redirection being made from an executable that required extra parameters. Once we figured out that < or > was used, we would call the function redirect(). This function oversaw whether the user called < or >. Once this was known, we would use the freopen() C function to redirect the stdin or stdout (depending on < or > respectively) from or to the one given in file. Therefore, everything in between freopen() and close() would be redirected to output to file (in the case of >) or to input from file (in the case of <).

1. Allow one pipe

To allow a pipe in our shell, we first check whether the character “|” was typed between commands. If is was, our program would call the pipeCommand() function with the two separated commands on each of | sides. Inside the function, we created a pipe and then forked the process to allow the child to redirect the output of the process into the pipe. After that, we created another fork so the corresponding child could redirect the input of the process from the pipe. All the pipe work was, of course, done by closing the end (read or write) that was not in use, running the command, and then closing the pipes completely. At the end, the main (or parent) process also would close the pipe and wait for the children to finish.

**Discussion:**

Things we learned maybe

**References:**

Sources