Ben Davidson & Jay Offerdahl

Quash - Now That’s Quite a Shell!

Running Executables (with & without arguments):

- Example: [EXEC] [ARGS...]
- Used the execv() version of the execute commands. If the input command is not in absolute path format, we check if we have access to execute the input command in the current subdirectory. If the input command is in absolute path format, we search for it in the path before executing. In either case, if we found an executable that matched, we executed the command with its list of arguments.
- To test, we spent some time ensuring our PATH searching function was working correctly by entering a command, and printing out the returned PATH/command string, if we found a valid executable at that location. Once this was working, we tested specific command such as find, ls, and others. We also tested non absolute path format executables. For this part of the testing, we wrote a program that terminated after 5 seconds.

Setting PATH and HOME Environment Variables:

- Example: set [var]=[location]
- Used the built in function setenv(..), which took the name of the variable, the value we wanted to set it to, and a flag to determine if we wanted to overwrite the current variable. Quash outputs the variable and its new value.
- Testing this was obviously straightforward, since quash outputs the new variable value we were able to confirm that the value of PATH or HOME had indeed been changed to the new value.

Changing Directories (cd) and Printing the Working Directory (pwd):

- Example: cd OR cd [LOCATION]
- Example: pwd
- If there were no arguments passed along with cd, the user was routed to their HOME directory using the chdir(path_to_home_directory) function call. If arguments were passed with the command, the chdir() function targeted the given directory. If the directory didn’t exist in either case, an error was given.
- To test “cd”, we typed “cd”. This took us to the location specified in the HOME variable. We then changed the home variable using set and typed cd again. Assuming the new HOME was valid, we were taken to that location. If an argument was sent in with this command, quash tried to change directory to that location. If it was a valid location, the function succeeded. If not, an error message was printed out.
- We were able to test “pwd” with cd because after changing directories, cd prints out the current working directory as well. (pwd uses the built in getcwd() command.

Exit & Quit to exit Quash:

- Example: [quit] OR [exit]
- “quit” was already set up for us in the base version of the program, all we had to do to add “exit” was another string comparison check on the command. If either condition was met, the program announced it was exiting, then terminated the instance. Testing was minimal as there were only two cases to test, which both worked.

Foreground & Background Processes (Allowing for background processes):

- Example: [EXEC] [ARGS...] &
- To accomplish this, we needed a method which forked a child and didn’t wait for the process to complete. To test, we created a process with the command ./h (a dummy program to wait 5 seconds before exiting), it prints out the newly created process jobid and pid, and upon completion, prints out a message saying it finished. We also tested this with a nested instance of quash. We entered a parent process of quash, started quash in the background (it would still request input), upon exiting the nested child process of quash, the parent process would print out a “finished” statement with the child’s pid.

Printing Current Running Jobs (background processes):

- Example: jobs
- When creating background processes, we stored the newly created process id, jobid, and command of a background process in a custom struct. We then store all the jobs in an array of job structs. In order to print out the list of jobs, or background processes, we loop through our array of jobs and print the necessary information.
- To test, we created several instances of the program “h” by typing in “./h &” several times, after words, I printed out the job list and could see each individual job by jobid and process id.
- We can confirm that only active jobs are listed because after a job shows that it has terminated, it no longer appears as output from the “jobs” command.

Child Process Inheritance (environment variables):

- Example: See testing
- Child processes in any of our commands that forked retained the environment variables from the parent because of the properties of fork(). Forking a child process creates an exact copy of the currently running process, so any environment variables stored in the parent process would be copied into the child process.
- To test this, we ran a nested quash instance. We opened quash inside of quash, then echoed the $PATH and $HOME variables, and they were the same as the parent’s variables. To test this further, we exited the child quash instance, changed the environment variables, and tested again. Lo and behold, the environment variables were the same in the child process as the newly changed variables in the parent process.

I/O Redirection:

- Example: [EXEC] [ARGS...] > [FILE]
- Example: [EXEC] [ARGS...] < [FILE]
- Quash implemented I/O redirection so that users of the shell had the ability to both write command output to files as well as read already existing file content and process this content with their desired command. For example, if a user was to type the following command: ls > abc.txt, the shell would store all of the files/directories located in the current working directory into the abc.txt file.
- If the file didn’t exist before running the command, the shell would create the file for the user and store the data in the file. If the file already existed before execution, whatever content that was originally within the file would be replaced by whatever value was returned to stdout. This worked simply by opening the file in quash.c and designating the file to writable and assigning stdout to the file itself. On the other hand, the user could change the direction of i/o by typing in a command such as: sort -n < numbers.txt. This simply assigned stdin to the file desired to be read from and allows the function on the left hand side of the ‘<’ to run on the data from the file on the right hand side.

- We tested this by running commands, such as ls, that would generate data to write to files and, on the other hand, running commands such as sort that handle incoming data. It’s worth noting that running commands that use files as input require that the target file actually exists before running the command.

Implementing the Pipe (|) command:

- Example: [EXEC] [ARGS...] | [EXEC] [ARGS...]
- We implemented pipes similarly to lab 3. We store the command on the left side of the | as well as the right hand side of the | in char arrays. We redirected stdout and stdin accordingly which effectively created the pipe between the two processes.
- To test, we ran multiple tests with different commands piped together. One of the tests we used to make sure the pipe function was working properly was: find quash.h quash.c | sort, which would return quash.c quash.h as opposed to just quash.h quash.c which would be returned by running find alone.
- Please note: We overwrote this functionality in favor of the extra credit version.

Support of Multiple Pipes in one Command:

- Example: [EXEC] [ARGS...] | [EXEC] [ARGS...] | ... (as many pipes as desired)
- This method still tokenizes the string by breaking it into two halves of the command (left up to first |, right is the rest). Next, the left side parsed and executing with execv(...) which actually ran the command. STDOUT was redirected to the right half of the command (after the first pipe). This command was then sent to our normal command handler.
- To test the multiple pipe functionality we ran the command “find quash.h quash.c | sort | sort --reverse”, which sent the list “quash.h quash.c” to sort, which would be sorted to”quash.c quash.h”, and then this would be reversed. Another test was “cat numbers.txt | ./A | ./A | ./A | sort”. The program “A”, which we wrote, took in three numbers, and multiplied them all by two. So, if we ran this three times in a row, the numbers would effectively be multiplied by 8, and then sorted.
- We were surprised at how little we had to change from our original function to get it working. In fact, this implementation even cleaned up our original pipe function because we didn’t have to store so many variables for the second execution since we were just passing the whole right side of the command.

Delivering ‘Kill’ Command to Background Processes:

- Example: kill [SIGNUM] [JOBID]
- We implemented a “kill” method which took in the argument list. Since we already had the capability to run processes in the background and stored these background processes in a global variable called jobList, we could easily validate the existence of the process. We searched the jobList variable to see if the specified process actually existed in the list, and if the job was found in the list, we used the kill() function provided in C. This effectively killed the specified background process and accomplished the task at hand.
- In order to test our kill function, we ran a few background processes, typed “jobs”, and then specified a background process to be killed. We confirmed that the kill function was working because the confirmation that the background process had finished never arrived, and the job was no longer present as output from the “jobs” command.
- Obviously, when the process had time to run and compete, the shell would alert the user that the process had completed, however we knew that we successfully killed the background process because that alert never arrived after we ran the kill command.