Quash Report

EECS 678

Cole Jurden, Evan Nichols

Overview

Quash (Quite A Shell) uses UNIX system calls to simulate a terminal, its behavior similar to popular shells such as csh and bash. The program allows all functionality included in a regular shell (background execution, I/O redirection, piping, etc.), along with a selection of built-in functions: set, echo, cd, pwd and jobs. An explanation of Quash's main execution function is below.

Main Execution Function

The main execution follows a series of if/else statements. In main, we first check to see if there were other arguments passed in the running of quash. If a file is passed in with commands to read from, quash reads through that file, splits its contents by `\n`, and executes each line accordingly. If there are no other arguments, quash waits until a command is received from the command line and the string containing the command is passed to `parse_command()`. This method first searches the strings for `|`, `&`, `>` and `<`. The results of these `strchr()` calls are stored in variables to be used in the conditionals.

The first conditional relates to the presence of a pipe. If there is a pipe, `parse_command()` splits the command string into two parts: a string of commands before the pipe and after the pipe. Then the program follows standard IPC format, declaring the pipe, `fork`ing the first process and redirecting `STDOUT`, `execvp`ing the first argument. In the second process, `STDIN` is redirected and the second argument is executed.

If there is no pipe detected, the program moves into an `else` conditional where it checks for other commands. First, it divides the command string using `strtok` with a whitespace delimiter and storing each individual part of the command in a `char*` array. This is mainly because we check the contents of `cmds[0]` for individual commands in our conditionals.

The next conditional check returns true if a command is directed to run in the background. In this case, we remove the `&` from the command, fork a child process, and call `parse_command()` using the truncated command string. We add non-completed jobs to our `jobs` array, which is displayed with the `jobs` command.

I/O commands relate to the next two statements. In both cases, we isolate the commands before the `<` or `>` and run them in a child process using the files given after the I/O indicators as redirects for `STDIN` and `STDOUT` respectively. When redirecting, we utilize the `dup2` command. This is also true for pipes.

Custom Commands

Once all these checks have been completed, it is time to look for our redefined commands, `set`, `cd`, `pwd`, `echo` and `jobs`. If the command is `set`, the command arguments is parsed using `strtok` to isolate only the desired path, which is then passed into the `setenv` UNIX system call . If the command is `cd`, the second command argument is passed into the `chdir` UNIX system call, updating the current working directory. If the command is `pwd`, the `getcwd` UNIX system call is invoked and printed to the screen.

Child processes inherit their parents' environmental variables. In order to correctly execute the `echo` function, it was necessary to use two global variables, `HOME_ENV` and `SET_ENV`. Each time the `set` function is called on HOME or PATH, the `HOME_ENV` or `PATH_ENV` variable are updated to reflect the change. This way, when the user executes `echo $HOME`, the `HOME_ENV` variable is used instead of the `getenv` UNIX system call, which would return the parents' environmental variable .

If the command is `jobs`, we call the function `print_jobs()`. This function utilizes the globally declared job count integer `jc` and the globally declared array of `struct job_t`'s to print current jobs in the same format as `bash` would.

Issues and Bugs

Setenv in a child process

Setting environment variables in a child process will set them for that process, but when an echo is called, it will echo the parents' environment variable. While the setenv function was being executed correctly, we thought it was wrong as the `echo` command (which initially used the `getenv` UNIX system call), would always return the parents' environmental variable.

Parsing and Cleaning Command Line Input

we initially struggled to parse and store the command line argument in such a way that it could be passed into system calls and `exec` calls without resulting in error. Using the `strtok` C method was an excellent solution, which allows for a string to be broken into tokens (sequences of contiguous characters) separated by delimiters. In our case, the delimiters were " , <, >, =, | ", depending on the command to be executed.

Removal of Jobs

We were not able to effectively remove a job from our `jobs` array. Code for the function `check_jobs()` demonstrates the technique we attempted to implement in principle. Essentially, for all jobs where `waitpid()` returned a value greater than 0, we would decrement job count and iterate through the job array replacing the pid in question. The `check_jobs()` function would be called at the beginning of `parse_command()` and act in the same way that `bash` reports finished background processes.

File Input

Reading input from a file, for certain commands, is functional, although it runs the function three times. We attempted to take the I/O redirection out of a child process entirely, out of one child

process, and in two child processes (within the initial conditional check and then with a recursive call to `parse_command()`) and the only solution that kept quash running was the latter. In attempts to debug we made a very simple program and attempted to incorporate similar functionality. This process yielded the same results we found in the more complicated quash.