Shell Implementation

Agathe Benichou and Shira Wein

Operating Systems Spring 2018
Due: February 18, 2018

Project Description

The assignment was to implement a mini shell for Linux. The purpose of the project was to improve our understanding of the C programming language and our system programming skills, as well as to better our knowledge of the Unix programming environment and shells in particular. Tasks associated with the assignment include creating pipelines and implementing stdin, stdout, stderr, PATH, and environment variables.

Shell and Piping

A Unix shell is a command line interpreter. The Unix shell provides a command line user interface where users direct the operation of the computer by entering commands as text for the command line interpreter to execute.

A pipeline is a sequence of processes chained together by their standard streams so that the output of each process feeds directly as input to the next one. Pipelines are created under program control. Piping allows you to chain together commands to pass output from one program to input of another to get a desired end result.

Project Contributors

Agathe Benichou and Shira Wein worked on the project. As we worked in a pair on this assignment, we practiced pair programming. This indicates that one person sat at the keyboard while the other contributes to the design and watches for errors. We switched who occupied which role of the pair programming so that both partners got equal coding and watching time. We primarily worked next to each other on the project, but occasionally implemented requirements individually for the other to check later.

Obstacles Overcome

The primary obstacle faced during the completion of this project was implementing piping after developing an understanding of pipelines. Pipelines proved difficult because it was necessary to provide support for multiple pipes. Our initial design of piping developed one set of stdout and stdin for each pipe, but if there are multiple pipes in the full command, each command would need to receive input from or to output to another command, rather than a simplistic, linear output to input approach. Therefore, we needed to design a more full implementation of piping. We overcame this obstacle by researching pipes, seeking help from the professor, and expanding on our initial design. This expanded design evaluates whether to pipe to stdin and stdout based on how many pipe symbols come before and after the current pipe.

Additional challenges faced during the project include environment variables and signal handling. Specifically, we had to overcome many segmentation faults by manually allocating memory for necessary pointers. This strengthened our understanding of memory allocation for low-level languages, C in particular. Finally, creating, storing and pushing jobs to either foreground or background served as a challenge. Jobs proved difficult to implement because we did not understand how to fully test them. Therefore, we implemented them by creating a job struct which contained the job's process ID and an array of job structs, able to list them upon user command. We incorporated functionality to create and add new jobs as they come, organized by process ID.

Learning Outcomes

Major things that we learned during the implementation of this program include the concept and implementation of piping, the importance of removing environment variables and closing pipes in system programming, and how stdin, stdout, and stderr work. In particular, piping is fully implemented for any number of pipes which contributed extensively to our understanding of pipelining. Additional learning outcomes include how PATH and environment variables work. We also improved our C programming skills and familiarity with the Unix programming environment. Practicing pair programming was also valuable.

Enhancements

A valuable enhancement that would be good for the project would be to expand on the implementation of the Linux signals list in Figure 8.26 in the Computer Systems textbook, specifically, those relating to Jobs. Additionally, our shell has some visual formatting issues when the result of a user command overlaps with the lsh prompt, but this is a visual interface issue and not a functional one.

Conclusion

The target learning outcomes were attained and the implemented Linux mini shell meets all of the requirements outlined in the Moodle project description. Fulfilling these aforementioned requirements, the mini shell: is written in C, is called lsh, prints a prompt **lsh>**, inherits environment variables from the process that starts it, allows any process created to execute a command to inherit all of the shell's current environment variables, supports pipes, supports the requirements of Homework Problem 8.26 in the Computer Systems textbook, follows the Unix convention for the arguments to main, and enables process statistics on each external command executed.