BigShell
CS 374 - Operating Systems I
Sawyer Baar (baars@oregonstate.edu)
December 1, 2024

BigShell Report

Table of Contents

Abstract

The BigShell project is a portfolio project for Oregon State University Course CS 374 - Operating Systems I. It is a cumulative project for the term that uses knowledge learned about UNIX/Linux operating systems to develop a shell implementation using the C language that implements a large portion of the POSIX shell's functionality.

The key objectives are to implement a shell that is able to parse and execute built-in and external commands, perform i/o redirection, modify the variables in the environment, and manage processes, pipelines, and signals.

The key outcomes are to be able to explain and implement programs for UNIX processes, explain and implement signals and signal handling, and explain and implement I/o redirection in programs.

This report provides an overview of the structure of the program, the tools and methods used to implement the program, and a description of the challenges encountered during the process.

Introduction

The BigShell project implements a POSIX-like shell program for the Linux kernel using the C programming language. It is a cumulative course project that uses all the concepts learned about operating system fundamentals in an entry level operating systems I course at a university level including: processes, file systems, terminal interface, utilities, interactive shell, and the shell command language, as well as data access and storage and interprocess communication [1]. The BigShell functions are based on the design of the Unix Time-Sharing System outlined by Dennis Ritchie [3].

The shell allows user access and ability to give commands without the user overhead of writing specific programs and then running them. Predefined commands have been developed with an extensive capabilities to perform various tasks and functions on the computer. The need for the capabilities that a shell provides was envisioned early on in computer history by Louis Pouzin. Pouzin wrote a 37-page technical memo titled, "The SHELL: A Global Tool for Calling and Chaining Procedures in the System" [7]. In this memo Pouzin explained the need for the shell as well as the requirements, design, organization, and process flowchart. All these topics are relevant to understanding the BigShell project, but most importantly the memo states the need for a shell, which is the motivation of this project. Pouzin stated in his memo, "The purpose of such a procedure is to create a medium of exchange into which one could activate any procedure, as if it were called from inside of another program. [7]"

Before the shell, a program had to be written and executed to perform a function. This was tedious. A shell provides access to a comprehensive library of functions that can abstract the access to operating system abilities. It's a powerful tool that simplifies day-to-day tasks on the computer.

There are multiple implementations of the shell including PowerShell for the windows kernel, and sh (Shell), bash (Bourne Again Shell), and zsh (Z shell) to name a few, for unix-like systems such as MacOS and Linux. Like BigShell, other shell implementations such as sh, bash, and zsh are POSIX compliant [2, 8]. POSIX compliance is important because it ensures compatibility among operating systems and allows for application portability. Therefore, it is important the BigShell is also POSIX compliant.

Program Structure

The structure of a shell is a loop that repeats until it is stopped by the user. The function **main** in the **bigshell.c** file is responsible for this loop. Before the loop starts, the Shell program performs an initialization routine including initializing the parser and signals. From there, a loop that checks for prompts and executes them continues until the exit command is received, or until there is an error that forces the program to exit. The loop can be simplified to three steps.

In the first step, the program waits on all background functions and then proceeds to read commands from the standard input file. The program checks on any jobs in the background, using the functions in the **wait.c** file. This function uses **jobs.c**. to access all the jobs and PGID's and utilizes a non-blocking wait on all child processes.

Second, the shell program parses and expand the command(s) and optional arguments that are provided. Commands are parsed and expanded using **parser.c** and **expand.c** respectively.

Finally, the program executes the parsed command and returns any output and errors to the standard output file and the standard error file respectively. During this phase, the functions contained in **runner.c** are called and all the builtin and non-builtin commands are executed. As needed the functions will handle IO flags, IO redirection, moving file descriptors, and more. This is also where pipes and forks are created. Next, variables are set or unset, or variables are exported to the environment. Functions in the file **vars.c** are used to ensure variable names are valid and then assign variables as specified. If the **exit** function in **exit.c** has been called, then a hangup signal (SIGHUP) is sent to all jobs, the jobs and variables are cleaned up, and the shell exits and returns a status from **params.c**.

During this loop multiple processes can be formed and the program must wait on background jobs using the functions in the **wait.c** file. It is also **signals.c** to control the flow of multiple processes. The file **params.c** holds a structure that maintains the most recent exit status and the last background pid. The process is continually checking for errors that will cause the shell program to exit.

Implementation Details

The BigShell project was developed using the C programming language on the os1 linux server hosted by Oregon State. I used Visual Studio Code (VScode) as for the Integrated Development Environment (IDE). The GNU Compiler Collection (GCC) was used to compile the program since it is the standard compiler that comes with Linux distributions and it is a free [9]. Finally, GitHub was used for version control since git is industry standard.

The program was frequently compiled, ran, locally tested, and finally submitted to GradeScope (an automatic grading service) to ensure continuous progress in meeting the specifications. There are many files that make up the source code, but only one was edited at a time. In order to simplify this process, the GNU Make utility was used to compile all the source code [10]. GNU Make helps the development process because it "determines automatically which pieces of a large program need to be recompiled, and issues the commands to recompile them" [10]. In order to use GNU Make, a makefile was used to check for any edits to the source code and it returned a GCC compilation command.

An iterative workflow was used to develop the BigShell project. A function, or a portion of a large function was implemented, and then the program was compiled and tested to ensure the functionality met specifications [1], before moving on. When errors were found, the function was debugged until it worked correctly. Once the program compiled and the new functionality worked as expected, the source code was pushed to GitHub. There are many different options for debuggers, but in a program with this many files, I found that print statement debugging was the most efficient. Specifically, it was very helpful to write descriptive error messages for implemented functions to ensure that any errors that occurred during testing were easy to track back to the offending function.

In total, seventeen functions in five files were implemented during the BigShell project. Each of the functions implemented is described in more detail below, organized by source code file.

builtins.c - In GNU shell implementations, builtin functions are invoked directly without invoking another program [11]. In this implementation, “Builtins use pseudo-redirection to avoid accidentally changing the shell's actual open files. This is implemented as a virtual layer (pseudo-fds) on top of the existing file descriptor system” [1].

The builtin functions that were implemented were: CD (change directory), Exit, and Unset (unset variables). Each of these functions checks the number of arguments provided with the command and uses if statements to handle each according to the specification.

signal.c - As named, this file contains functions that handle signals for processes. The functions signal_init, signal_enable_interupt, signal_ignore, and signal_restore were all implemented. These functions handle signals for the shell using the sigaction() function from the C library signal.h to change the response of a process when it receives specific signals.

vars.c - The functions implemented in this file were is_valid_varname and vars_is_valid_varname. The latter is essentially a wrapper for the former. They are responsible for checking if the variable names given in a command are valid or not before allowing the variables to be set. This uses type checking functions from the ctype.h standard library for C to ensure the first character is alphabetic and not an underscore, and to ensure the rest of the characters are alphanumeric or an underscore in order to meet POSIX standards [2].

runner.c - The functions implemented were expand_command_words, do_variable_assignments, get_io_flags, move_fd, do_io_redirects, run_command_list. These functions were the most involved and complex to implement.

The functions expand_command_words and do_variable_assignments iterate through the commands and arguments given and handle each one individually. The function get_io_flags is responsible for setting the IO flags for redirection before files are opened. The function move_fd uses dup2() to move file descriptors between the source and destination. The function do_io_redirects takes the IO flags from get_io_flags, and uses them to move file descriptors and open the specified file. Finally, run_command_list is responsible for handling the list of commands provided from the standard input. It is where pipelines are created, and processes are forked. It also calls the above functions as helper functions.

wait.c - The functions to be implemented in this file are wait_on_fg_pgid (foreground paid) and wait_on_bg_jobs (background jobs). As the name implies, they wait on the foreground pgid and the background jobs respectively. These functions monitor the child processes and return exit statuses once the jobs complete.

Challenges and Solutions

As can be seen from the above sections, the BigShell project is fairly complicated with several thousand lines of code distributed through twenty-one source code files. Understanding the flow of a larger program and then modifying it can be very daunting at the beginning. Documentation is incredibly important. There wasn't a lack of documentation, but on the contrary there was a large ocean of information to wade through and digest before any progress could be made on the project.

In order to get a handle on the BigShell project and begin to make progress, I had to trace the flow of the program line by line, starting from the **main** function in **bigshell.c**. Initially, I looked at the main flow of the program, and then I familiarized myself with the more detailed paths that it could potentially take. Other students provide visual diagrams on Ed Discussions that helped visualize the flow as well, but it wasn't as helpful as slowly going through each function step by step.

When implementing the incomplete functions, it was helpful to reference the GNU documentation on the bash shell as well as the linux man pages [5, 11]. It was also incredibly helpful to review examples of the individual functions that I was trying to implement, as well as review examples on stack overflow as well because other users would explain the function in layman's terms [12].The Ed Discussion board was also used, but not as often as Stack Overflow due to the search feature in Ed not working as well [13].

This process is very relevant to future work. It's likely that I will be given access to a source code for a project from a previous developer and I will need to update or add functionality. I will first need to understand the flow of the code and the objective, and then use outside resources to complete the task. It's also important that I think of the developers that will come after me. It's important to provide concise and organized information. It's also important to write clean code with helpful comments throughout. Finally, descriptive error messages should be provided because they can help the end-user as much as they can help the developers.

Conclusion

The BigShell project implements basic functionality of a POSIX shell, but it could be expanded to provide more functionality comparable to bash or zsh. For example, job control functionality could be added in the future. This is the function to select processes and manually pause and resume their execution. Each job is assigned a PID and it can be accessed using this number.

Another potential improvement to this program is to modify the builtins so that they don’t use pseudo-direction, so the shell program could be ran stand-alone. Currently, the shell program must be run in another shell for testing purposes. More research would be required to implement this correctly.

Overall, this implementation is a start to a shell program that has basic functionality. It implements the basics of POSIX compliant shell including parsing and expanding commands, executing builtin and external commands, performing IO redirection, exporting to the environment, implementing signal handling, and managing processes and pipe lines.

References

Information that was referenced to learn more about building a shell implementation and complete the assignment are referenced below.

[1] R. Gambord, "BigShell Specification," Operating Systems I [Online], August 8 2024. Available: https://rgambord.github.io/cs374/assignments/bigshell/

[2] *IEEE Standard for Information Technology - Portable Operating System Interface (POSIX(R))*, IEEE Std 1003.1-2008. doi: 10.1109/IEEESTD.2008.4694976.

[3] D. M. Ritchie, and K. Thompson, "The UNIX time-sharing system," Commun. ACM vol. 17, no. 7, pp. 365-375, Jul. 1974. doi: https://doi.org/10.1145/361011.361061

[4] *Bash*. (2024). GNU Project.

[5] Michael Kerrisk, "man-pages," Linux Man Pages Online, https://man7.org/linux/man-pages/index.html (accessed Nov. 1, 2024).

[6] "System Interfaces," The Open Group Base Specifications Issue 7, 2018 edition IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008), https://pubs.opengroup.org/onlinepubs/9699919799/functions/contents.html (accessed Nov. 1, 2024).

[7] L. Pouzin, "The SHELL: A Global Tool for Calling and Chaining Procedures in the System", mit.edu, https://people.csail.mit.edu/saltzer/Multics/Multics-Documents/MDN/MDN-4.pdf (accessed Nov. 3, 2024).

[8] "Shell Command Language," Shell Command language, https://pubs.opengroup.org/onlinepubs/9699919799/utilities/V3_chap02.html (accessed Nov. 5, 2024).

[9] "GNU Project," GCC, the GNU Compiler Collection, https://gcc.gnu.org/ (accessed Nov. 5, 2024).

[10] "GNU Make," GNU, https://www.gnu.org/software/make/manual/make.html (accessed Nov. 10, 2024).

[11] "GNU bash manual," GNU Project - Free Software Foundation, https://www.gnu.org/software/bash/manual/ (accessed Nov. 29, 2024).

[12] Stack Overflow, https://stackoverflow.com/ (accessed Nov. 29, 2024).

[13] "CS 374 400 – Ed Discussion," Ed Stem, https://edstem.org/us/courses/67724/discussion (accessed Dec. 1, 2024).