BIGSHELL

Tavner Murphy
Operating Systems I
Dec. 1, 2024

BigShell is an implementation of a POSIX (Portable Operating Systems Interface) shell. A shell is a command language interpreter program. Its primary purpose is to parse the user's command-line input and execute the extracted commands either by forking separate processes or running built-in commands within the shell environment. This shell supports core functionalities such as I/O redirection, signal handling, variable management, and job control, providing the user a robust command-line interface adhering closely to POSIX standards.

The project's objectives included implementing built-in commands such as cd, exit, and unset, enabling command word and variable expansions, implementing foreground/background process management using fork, exec, and wait system calls, implementing I/O redirection and shell pipelines, and finally enabling signal handling to manage interrupts and termination of the shell and its child processes.

BigShell showcases a comprehensive understanding of Unix operating system concepts, including process creation, inter-process communication, signal management, and shell behavior. Additionally it demonstrates the ability to interpret POSIX specifications and implement them effectively in C, showing a clear proficiency with C language concepts, including the use of pointers, structures, and various library functions. BigShell serves not only as a robust, practical tool for command line interaction with the operating system, but as an advanced demonstration of C programming proficiency and UNIX system design concepts.

I. INTRODUCTION

The UNIX (Uniplexed Information and Computing System) operating system was developed in the early 1960's at Bell Labs by a team led by Ken Thompson and Dennis Ritchie. [1] One of the major factors contributing to UNIX's success was its open architecture, which allowed users to customize and extend the system to meet their specific needs. [2] This led to "… the development of a rich ecosystem of tools and applications built on top of UNIX" [2]. The SuS (Single Unix Specification) and POSIX (Portable Operating System Interface) standards date back to the 1980s when they were created to ensure a "clear, consistent, and unambiguous standard" across UNIX-like operating systems in response to the increased fragmentation of the UNIX operating system [3]. These two standards define the basic features of such an operating system, and their philosophy is said to have been widely influential throughout the history of operating systems and software development. The goal of SuS/POSIX is to promote compatibility between different operating systems and to make it easier for software developers to write applications that can run on different platforms [2]. Core principles of the guidelines include "interoperability, portability, and standardization" [2].

The POSIX shell is a powerful, widely used system interface in Unix-like operating systems and is outlined in the POSIX standards mentioned above. When it was first introduced it revolutionized user-operating system interaction by allowing users to directly input commands into the operating system using text. The user inputs human readable commands on the command line, and then the system interprets them and gives immediate feedback, facilitating efficient navigation, and execution of operations [5]. Over time there have been many widely used implementations of the POSIX compliant shells, including bash, fish, zsh, and ksh, which continue to be integral on the modern, UNIX-like Linux, Windows, and macOS operating systems [13]. Thus, a POSIX-compliant shell is a very interesting project to take on. It demonstrates not only the developer's deep understanding of how operating systems function, but much about the fundamental philosophy of POSIX, the very principles underpinning all of modern computing.

BigShell is "…based loosely on the shell command language specification in POSIX.1 2008, with several features removed or simplified" [1]. Its purpose is to process user input from the command line, parse commands, and execute them either by forking separate processes or running internal commands. The shell command language follows a certain syntax and structure; all shell commands must be one of the following: a simple command, a pipeline, a compound command, a function definition, or a list compound-list [7]. These commands can be used to accomplish a myriad of tasks limited essentially only by the imagination of its user, as the shell provides features to provide control flow over commands and the shell is capable of spawning new child processes to run commands in the background [4]. Additionally, commands can be linked together via a pipeline, or run simultaneously. The shell, therefore, is quite powerful. This project serves as an exercise in understanding and implementing core UNIX operating system concepts, including but not limited to process forking, shell language syntax, inter-process communication, job control, and signal handling. It provides the user with a robust command-line interface adhering closely to the POSIX standards.

II. PROGRAM STRUCTURE

Main Shell Loop (bigshell.c)

The main shell loop is the core of BigShell. When initialized, it runs continuously through a read, evaluation, and print loop (REPL) until exiting. First, it initializes the parser and signal modules, then prompts the user for input and calls the command parser module to read and interpret the user's input to a simple list of commands. It receives back and prints the parsed command list for the user and notifies them via print statements it is executing the parsed command list, noting the number of commands to be executed. Next, it calls the Command Execution module to run the list of parsed commands. Finally it cleans up after itself, and restarts the loop. During the REPL loop it handles any exceptions and both halts the loop and reports to the user if any of the other modules encounter errors, ensuring the shell operates smoothly.

Command Parser (parser.c, parser.h, expand.c, expand.h)

The command parser module is responsible for reading and interpreting the user's input. It breaks down the input into manageable components called tokens, which can be interpreted as commands, arguments, or operators [10]. It also handles word and variable expansion as outlined in the POSIX standards for special characters and symbols, such as ~ [7]. If the command line input given by the user has invalid syntax, it flags errors for the main shell loop to notify the user.

Command Execution (runner.c, runner.h)

This module intakes the parsed command list, and flags whether each individual command is a built-in command to be run within the shell execution environment, a pipe command (pipes are a unidirectional data channel that can be used for interprocess communication), or an external command and/or background command, in which case it is responsible for forking a new process [19]. The child process then uses an exec system call to execute the command. If the user has indicated they want to run a command in the background, the module calls functions from the jobs module to run the process in the background, otherwise the shell calls the functions in the job module to wait for the process to complete before returning control back over to the user.

Parameters and Variables (params.c, vars.c)

This module is responsible for storing, validating, updating, and processing the environment and user-defined variables, which are used throughout the shell's other subroutines [3]. System level environment variables like $PATH and $HOME must be handled differently than the user defined variables according to POSIX standards [6].

Signal Handling (signals.c, signals.h)

This module handles everything to do with UNIX's signals, which are special ways for the operating system and the shell to control processes that are currently running [2]. This module enables signaling and defines the behavior of numerous signals for the shell and its children, and reports back to the main loop if any unexpected behavior or errors are encountered [21].

Jobs Management (jobs.c, jobs.h, wait.c, wait.h)

This module is responsible for controlling and tracking any jobs that are set to be run in the foreground or background by the command execution module. Those jobs go onto a jobs list, and this module then handles status reporting for foreground/background jobs, as well as frees up any resources when jobs are terminated. This module is essential for the shell to track and manage its various tasks and enables multiple processes to be run simultaneously [21].

III. IMPLEMENTATION DETAILS

Bigshell was developed using the C programming language, which was itself originally developed in parallel with the UNIX operating system by the team led by Ken Thompson and Dennis Ritchie [1] [3]. All of BigShell's subroutines are built on top of C library functions that are imported within the project, and of course this is no coincidence, as the C language and many of its library functions were developed for the very purpose of implementing a POSIX shell, along with the rest of the UNIX operating system [3].

The project used Makefiles to manage the compiling of the shell and the creation of the final BigShell executable. Makefiles are shell scripts built using the 'make' utility that automate the process of compiling large project with many source code files, defining the relationships between the files [9]. One feature is that it only recompiles files that have changed when creating the executable, rather than recompiling everything from scratch. Although I was the only developer on this project, Git was used for version control, allowing for efficient tracking and backtracking during project development. Finally, using VSCode in combination with GitHub Codespaces was my preferred IDE, because of its compatibility with the aforementioned makefiles and its excellent UI, ease of use, and range of useful plugins.

Testing was made far easier thanks to the 'debug' executable created by the makefiles, which added many print statements to the program during the compiling phase. This made it much easier to informally troubleshoot and fix issues myself even when large portions of the shell were not implemented yet, and I did not need to develop a testing suite. During testing I ran built-in commands like cd, exit, and unset, tried using complex I/O direction within my test commands, with operators like >, <, and pipes, and iterated many times on each module trying to make the program more and more functional as each new module was bolted on.

The following are descriptions of key functions implemented. The builtin_cd() function changes the current working directory using chdir() from the C library [10]. If no operand is provided it defaults to the path stored in the $HOME environment variable. Builtin_unset() unsets shell variables by calling vars_unset(), typically for cleanup. The builtin_exit() function updates the shell's status before closing out of the shell. Using the C library function strtol() it extracts the argument from a command structure, and converts it to a long before calling bigshell_exit() to perform cleanup and termination of the shell process [12].

The wait_on_fg_pgid() function in the Jobs Management Module handles foreground process groups. It restarts processes by sending the SIGCONT signal via the C library function kill(), which despite its blunt name is used to send *any* signal to a process, not just a termination signal [15]. If a process is interactive, it gives it access to the terminal. It waits on all processes in

the process group using the C library function waitpid(), which uses a pgid to wait for a process to stop or terminate [14]. Once finished, it updates the shell's status and removes the job from the list.

The function run_command_list() within the Command Execution Module executes a list of commands according to their flagged control type (foreground, background, or pipeline). Looping through the command list it starts by handling word expansions for the command according to POSIX shell syntax [7]. Next it prepares to read from the pipeline of the previous command, if it exists, or creates a new pipeline, if needed, using the C library function, pipe() [19]. Next, it forks a child process if the command is not a built-in and/or is flagged as a background command using the C library function fork(), which creates a child process and returns its process ID [16]. If the command is part of a pipeline, a new process group ID is assigned if it is the first command in that pipe, or alternatively it is updated to match the other commands in the pipe, using C library functions setpgid() and getpgid(), which are used to set and get the process group ID's from a process, respectively [20]. The function then handles the remaining redirect operators from the command, and performs variable assignment using helper functions. It executes a built-in command directly in the shell environment.

For external commands that have forked, run_command_list() sets up pipeline redirection using the standard C library function dup2(), which duplicates a file descriptor over top of a different file descriptor [18]. Then it executes the command using the C library function execvp(), which replaces the parent's process image with a new process image constructed from an executable file called the 'new process image file' [17]. The parent waits on the child if its a foreground command and after execution cleans up after itself, calling the C library function exit() to terminate the child process, and closing unnecessary pipe ends [21].

The functions signal_init(), signal_enable_interrupt(), signal_ignore(), and signal_restore() within the Signals Module are built on top of the C library function sigaction(), which defines the behavior of the shell and its children when receiving certain signals [21]. Together these functions enable the shell to control and interact with its processes.

Finally functions is_valid_varname() and vars_is_valid_varname() of the Variables and Parameters module ensure variable names provided as arguments strictly follow the POSIX guidelines for validity [22].

IV. CHALLENGES AND SOLUTIONS

The scope of this project needless to say necessitated a high degree of knowledge regarding POSIX standards and the shell language. The central challenge I faced was the pure volume of information required to intake in order to get up to speed and to know which eddies in which to sink my oar. This meant lots and lots of reading and digging through the specifications, not only of the POSIX standards themselves, but of instructional and educational resources throughout the internet. There are many aspects to the POSIX shell specifications that "…are left vague or that require extensive and esoteric background knowledge of the shell and its concepts" [1]. From high-level concepts such as how child process creation works, to the specific inputs and outputs of lower level system call functions, the breadth and depth of important topics seemed to have no limit. I read through dozens and dozens of man pages for individual

commands such as dup2(), execve(), getpgid(), strtol(), fork(), and more, in order to glean just how all of these library functions could be used in accordance with the shell's tasks.

Of particular note was the Command Execution Module in runner.c. This was the largest module requiring by far the most work on my part in order to reach my secondary, but no less important starting goal— a compiling, debuggable executable. Following the suggested order of implementation I began by implementing the built-in commands, then the exec and wait functionality, and when the BigShell program finally compiled at this point I was able to take on my third goal of debugging the work I had done thus far and make sure there was a foundation from which to begin implementing the remaining modules [1]. After that, my fourth goal was to implement I/O handling, pipelines, and the Signal Modules. I iterated step by step, continuing to review man pages for C library functions and shell commands as I went.

Coding such a major project in the C language also required extensive knowledge of its syntax, libraries, and various quirks, all of which I had only cursory knowledge of at the outset. I had certainly not implemented something of this scale in the C language before. So this project proved to be an enormous crash course in the C language itself; I learned how to use pointers effectively, how to handle signals and structs, how to handle environment variables, as well as the proper usage of many functions within the C libraries.

Thankfully, this project had already been 'designed' long before I was born, in fact quite brilliantly so, and the carefully laid out standards and specifications referenced above made it much much easier to follow along. More than other projects I had completed to this date, this was much more akin to assuming the role of a chef following a recipe, than the usual design-from-scratch sort of approach, and required far less problem-solving and thinking creatively about system architecture than it might have were there no Ritchie and Thompson. I would go as far to say that more than anything I have learned a great deal of respect and admiration for the founding ideology of UNIX and for its creators.

V. Conclusion

This implementation of a UNIX shell program proved to be a difficult but immensely rewarding project. It was a deep dive into Unix operating system principles, the POSIX standards, and shell programming concepts. By implementing process management, I/O redirection, and signal handling, I have a shell that offers a functional and robust command-line interface adhering to the POSIX standards. It is a living breathing ode to the modularity and extensibility of the Unix philosophy, as well as the flexibility and usefulness of the C programming language.

Beyond its purely technical execution, this project offered more than one lesson in software engineering. The process of dissecting complex specifications, understanding UNIX design principles, and implementing functions one by one within a larger codebase step by step cultivated in me a newfound appreciation for these foundational operating system concepts. BigShell's completion not only demonstrates my technical proficiency in system level programming but also provides a foundation for future exploration into advanced system-level design topics. It highlights the elegance of the original UNIX philosophy and the relevance of the POSIX standards in modern computing. This project was a pivotal experience during my growth

as a software developer. It merges theoretical knowledge with practical application and paves the way for a future in both system-level programming and software development as a whole.

References

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

[2] R. Gambord, “A Brief History of Unix,” Operating Systems I [Online], 2024. Available: https://rgambord.github.io/cs374/modules/01/introduction/history-of-unix/.

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

[4] R. Gambord, “Shell,” Operating Systems I, 2024. [Online]. Available: https://rgambord.github.io/cs374/modules/01/shell/.

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

[6] The Open Group, “Base Specifications Issue 7, 2018 Edition, IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)," IEEE and The Open Group, 2018. Available: https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap08.html.

[7] The Open Group, “Utilities, The Single UNIX Specification,” Version 4, 2018. Available: https://pubs.opengroup.org/onlinepubs/9699919799/utilities/V3_chap02.html.

[8] D. M. Ritchie, “The Evolution of the Unix Time-sharing System,” Bell Labs [Online]. Available: https://www.bell-labs.com/usr/dmr/www/chist.html.

[9] GNU Project, “GNU make: A Program for Directing Recompilation,” Free Software Foundation, Inc. Available: https://www.gnu.org/software/make/manual/make.html.

[10]. The Open Group, “cd - Change the working directory,” Linux Manual Pages, 2024. [Online]. Available: https://man7.org/linux/man-pages/man1/cd.1p.html.

[11] The Open Group, “chdir - Change the working directory of the calling process,” The Single UNIX Specification, Version 2, 2004. [Online]. Available: https://pubs.opengroup.org/onlinepubs/009695299/functions/chdir.html.

[12] The Open Group, “strtol - convert a string to a long integer,” Linux Manual Pages, 2021. [Online]. Available: https://man7.org/linux/man-pages/man3/strtol.3.html.

[13] Hyperpolyglot, “Unix Shells Comparison,” 2024. [Online]. Available: https://hyperpolyglot.org/unix-shells.

[14] The Open Group, “waitpid(3p) - process status,” Linux Manual Pages, The Open Group Base Specifications Issue 7, IEEE Std 1003.1-2017, 2018. [Online]. Available: https://man7.org/linux/man-pages/man3/waitpid.3p.html.

[15] The Open Group, “kill(2) - send a signal to a process,” Linux Manual Pages, The Open Group Base Specifications Issue 7, IEEE Std 1003.1-2017, 2018. [Online]. Available: https://man7.org/linux/man-pages/man2/kill.2.html.

[16] The Open Group, “fork(2) - create a child process,” Linux Manual Pages, The Open Group Base Specifications Issue 7, IEEE Std 1003.1-2017, 2018. [Online]. Available: https://man7.org/linux/man-pages/man2/fork.2.html.

[17] The Open Group, “exec(3p) - execute a program,” Linux Manual Pages, The Open Group Base Specifications Issue 7, IEEE Std 1003.1-2017, 2018. [Online]. Available: https://www.man7.org/linux/man-pages/man3/exec.3p.html.

[18] The Open Group, “dup(2) - duplicate a file descriptor,” Linux Manual Pages, The Open Group Base Specifications Issue 7, IEEE Std 1003.1-2017, 2018. [Online]. Available: https://man7.org/linux/man-pages/man2/dup.2.html.

[19] The Open Group, "pipe(2) - create a pipe,” Linux Manual Pages, The Open Group Base Specifications Issue 7, IEEE Std 1003.1-2017, 2018. [Online]. Available: https://man7.org/linux/man-pages/man2/pipe.2.html.

[20] The Open Group, “setpgid(2) - set process group ID,” Linux Manual Pages, The Open Group Base Specifications Issue 7, IEEE Std 1003.1-2017, 2018. [Online]. Available: https://www.man7.org/linux/man-pages/man2/setpgid.2.html.

[21] The Open Group, "sigaction(2) - examine and change a signal action,” Linux Manual Pages, The Open Group Base Specifications Issue 7, IEEE Std 1003.1-2017, 2018. [Online]. Available: https://man7.org/linux/man-pages/man2/sigaction.2.html.”

[22] The Open Group, “Base Specifications Issue 7, IEEE Std 1003.1-2017,” The Open Group, 2018. [Online]. Available: https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap03.html.