BIGSHELL

M. Saltz

Operating Systems I

August 6, 2024

Abstract-- POSIX (Portable Operating System Interface) is a set of standards used for the design of application programs and utilities in Unix using the programming language C for implementation [1]. The goal of Bigshell is to adhere to these standards while implementing a small shell program. The metrics by which the program is evaluated are its ability to execute some key Unix shell features. Bigshell accesses the standards via the Unix process API, creating a shell program with limited capabilities. The outcome of this project is a program that shows my ability to utilize signals, write programs using this API, and employ I/O redirection [6]. The program was able to fulfill each of the assigned requirements and implement the required shell functionality, as well as an additional implementation of job handling. The shell is capable of executing built-in utilities, redirection, pipelines, signal handling, background and foreground commands, and external commands. This shell does not handle every single edge case but does pass the assigned tests. This project is far from a true shell, and instead serves as a learning tool for interacting with low-level operating systems programming tools. To fully implement a shell, however, requires more work. The future of this work involves implementing a command history and line clearing using the upwards arrow key and curses library.

I. INTRODUCTION

Bigshell was introduced as an assignment for the class CS 374 – Operating Systems I at Oregon State University. It is an implementation of a lightweight Unix shell with limited capabilities based on the POSIX standards. Historically the Unix shell has seen many revisions, some of the most influential of which have been the PWB or Mashey shell [3] and the Bourne Shell [6]. Piping and shell redirection has existed ever since it was introduced in the Thompson shell, the first Unix shell [4]. The Mashey shell implemented utilization of several pipes as well as better handling of variables by implementing shell variables [3]. The Bourne shell is still in use today and implemented scripting and integrated signal handling [6]. This history of Unix shell creation culminated in the POSIX standards of today, which Bigshell is based on. The POSIX standards are built off the Bourne shell and define "a standard operating system interface and environment, including a command interpreter (or "shell"), and common utility programs to support applications portability at the source code level [2]." The goal of this project is to implement a shell with a few of the typical Unix shell features. It is important to do this so I can expand my knowledge of C programming, familiarize myself with POSIX, and most importantly learn to read another's code. Part of the shell has been implemented already, but it needs to be completed to meet the following requirements:

Parse command-line input into commands to be executed

Execute a variety of external commands (programs) as separate processes

Implement a variety of shell built-in commands within the shell itself

Perform a variety of i/o redirection on behalf of commands to be executed

Assign, evaluate, and export to the environment, shell variables

Implement signal handling appropriate for a shell and executed commands

Manage processes and pipelines of processes using job control concepts [5]

Bigshell is not intended to be an all-encompassing shell program, but rather a learning tool. Its scope is the above requirements, and it is not meant to handle every error or edge case. This project improved my understanding of a shell program flow and enabled me to work with a larger codebase with another contributor. Similar software projects to Bigshell include the Bourne shell, the Mashey shell, and the current Unix shell that is available with each Linux distribution. Such projects have contributed to the POSIX standards of today, which is what this shell is based on.

II. PROGRAM STRUCTURE

The architecture and flow of this shell program involves the user inputting a command into the terminal, the command words being parsed and undergoing expansion, then the execution of those commands which are either built-in or external commands. Depending on the command, such execution may also include I/O redirection and variable assignment. At any point the program may receive a signal, which it handles accordingly. The main modules of Bigshell

are bigshell, builtins, exit, runner, wait, expand, jobs, parser, and vars. These modules interact with each other to form the Bigshell program.

After initialization, the program Bigshell operates as an infinite loop. It takes in the user input, then calls the parser to turn it into a list of commands. It checks for signals and errors and then runs the commands by calling on runner. Afterwards it cleans up and prepares to begin again.

Bigshell checks for signals in the main module bigshell.c by calling on signals.c, which handles the signals of SIGSTP, SIGINT, SIGTTOU. This module can initialize signals to new actions, restore their old actions, set them to interrupt any ongoing processes, or ignore them entirely. Bigshell uses signals.c to first allow SIGINT to interrupt any background jobs and then, after parsing the command list, sets SIGINT to be ignored. After runner.c finishes piping any commands, it restores the signals to their original values, and the cycle repeats again.

The runner.c module interprets and executes commands. Is called on by bigshell.c in the main loop. Depending on the command, runner.c will either implement redirection, execute built-in commands, execute external commands, or handle the setting of variables. All of this depends on what the parsed command words are evaluated to be. The main function for this purpose is run_command_list, which runs the commands by taking in the parsed commands, looping over them, expanding them with expand.c, and handling processes, piping, and jobs. It does this by setting flags for background, foreground, or pipeline tasks if the command words are evaluated as such and handling each accordingly. Then runner.c goes on to implement job control and joins or creates a new process group. If it is determined there is a built-in command, runner.c executes it by handling I/O redirection, variable assignment, and running the builtin.c module specific to that command. Otherwise, it handles external commands by pointing the pipes to the shell for reading and writing, handles io redirection, variable assignment and executes the command with execvp. Finally, runner.c calls wait.c to implement waiting on any processes.

builtins.c handles any built-in commands and does this by using a function that calls the other functions based on the parsed commands it receives. It controls all the internal commands and is called whenever an internal command is identified in runner. builtin_null handles commands that contain only redirections and assignments, in which case it does nothing. builtin_cd changes the directory. It does this by first checking the number of commands and setting the environment variable PWD to the passed in command. If nothing is there, PWD is home. Then it switches to that directory. builtin_exit controls the shell's exit status and exits the shell if that command is received. If it receives an exit status number, it assigns it as the status. If it does not, it uses the previous status. Then it exits the shell. builtins.c can handle multiple variables and searches for the assignment operator. Once it encounters that, it sets the variable and exports it. builtins.c can also unassign all variables it receives. If builtins encounters a “bg”, “jobs”, or “fg” command, it handles each accordingly by either placing a specified job in the background, listing any background jobs, or placing a specified background job in the foreground. builtins.c uses vars.c to handle any variable assignments, unsetting, exporting, or checking the validity of variable names.

wait.c is called by builtins.c, bigshell.c, and runner.c to wait on any processes that need to finish, and either suspends the process until the child changes state or is nonblocking if it is a background process. jobs.c handles any jobs, which are separate process groups that get their own specific IDs (job IDs). exit.c exits the shell completely when called.

An example series of commands that follow the outlined process would be when the user inputs “X=123”, “echo $X”, and then “unset X”. To execute the first command, bigshell.c first parses it, then calls runner.c on the list of command words. runner.c expands the command words, spawns a new foreground process which is not forked because it is modifying the shell’s execution environment. It executes the command by calling vars.c to perform the variable assignment. No redirection operators are used; therefore no redirection is needed. vars.c first checks if the variable name is valid, then it checks if it already exists. Because it does not, it creates “X” as a new variable. bigshell.c then cleans up the command list and returns to first checking on background jobs, of which there are none, then parsing and executing the next command. Because “echo” is an external command, this follows much of the same process, calling “execvp” to execute the command. Because this is a foreground command, runner.c calls wait.c to perform a nonblocking wait. The next command is a built-in command, so instead of calling “execvp”, builtins.c is called to unset the variable. This calls on vars to handle the unsetting, which first checks if the variable is of a valid name, then removes it. If the variable does not exist, it does not exit with an error but simply does nothing. Finally, bigshell cleans up and begins again until the “exit” command is run, in which it exits with either a designated status or the status of the most recently terminated foreground command.

III. IMPLEMENTATION DETAILS

To build this program, I used the C programming language, the Linux manpages, and POSIX documentation. The skeleton code provided was heavily commented and described exactly what needed to be implemented and gave tips on which resources to use. I also relied on old code from a previous assignment to aid in my completion of this project. I relied on a Github codespace to compile, run, test, and debug the project. Much of my testing also took place in Gradescope. Gradescope contained a list of tests and subtests that my implementation had to pass. It compared my code’s output to its own, and if the two outputs matched, the test was considered a success. While the Gradescope tests do not cover every edge case, they performed basic testing to ensure each implemented section met the requirements. My workflow involved first familiarizing myself with the program and attempting to understand the skeleton code and what needed to be done, and then first attempting to implement the built-in commands. To get the built-in commands to pass the Gradescope tests, I discovered that wait.c and runner.c also had to be minimally implemented. I began with the goal of implementing the exit command but did not actually succeed in passing this test until much later. As I progressed through implementation, I grew more familiar with the program’s structure. I created a list of functionalities I had to incorporate and set an initial goal of one per day, which quickly fell apart as I picked up pace. Instead of only implementing one functionality per day, I ended up implementing every single assigned functionality of the span of a few days. I discovered that much of what I had to implement were single lines of code that relied heavily on either already

implemented code or functions from the Linux manpages. I implemented minimal logic at most. Each of the functions and my responsibilities involving each are below:

A. *builtins.c*

In this module I implemented builtin_cd, where I set the environment variable, changed the directory, and set the directory to default to home if the command is empty. I also implemented builtin_exit and was responsible for setting the exit value to the correct status. The last function in this module that I implemented was builtin_unset, where I unset the variables using a prewritten function.

B. *runner.c*

In runner.c I implemented expand_command_words, where I looped through the list of commands and expanded each using a prewritten function, do_variable_assignment, where I set the variable and exported it if necessary, get_io_flags, in which I set each I/O flag value, and move_fd, in which I duplicated and moved a file descriptor. I also implemented do_io_redirects to handle much of the opening, closing, and moving of the file descriptor and run_command_list to handle forking, piping, and execution of external commands using execvp.

C. *signals.c*

This module relied on me implementing signal_init to initialize signals and store their old actions, signal_enable_interrupt to set the signal disposition for interruption and not store the old action, signal_ignore to ignore a received signal without storing the old disposition, and signal_restore to restore the signals to what they were on bigshell initialization.

D. *vars.c*

In this module I implemented is_valid_varname, giving it the functionality to check if the variable name is valid as well as providing a wrapper function, vars_is_valid_varname.

E. *wait.c*

I implemented wait_on_fg_pgid, providing the functionalities of continuing the process group, making the process group the foreground group by setting the foreground ID, setting params.status to the correct value, removing the job from the job list, handling any stopped child processes, and then making bigshell the foreground process again. I also implemented wait_on_bg_jobs by incorporating a nonblocking wait for the process group.

IV. CHALLENGES AND SOLUTIONS

I knew this project would be a massive undertaking, so I began early. I set specific goals for myself to reach each day. However, I quickly began to struggle with getting even the seemingly simplest builtins to function properly and pass the Gradescope tests. To combat this, I first relied on my old code, then used print statements for debugging, then came to the realization that external commands and proper exit statuses for processes had to be implemented for one of the builtins to run properly. I posted my questions to the discussion board and did not hesitate to reach out for assistance. The initial undertaking of understanding the code and how each module

was supposed to work together was very overwhelming, especially when I was impatient to begin coding. But I found that by reading the comments closely and reminding myself of how few lines needed to be implemented, the project became more manageable. I tried to begin by completing each module and functionality separately, but this was not a good strategy for a program that depended so much on interconnectivity. Reading and understanding the skeleton code was the most difficult and time-consuming portion of the assignment despite how well commented it was, and how well documented the assignment requirements were. However, I believe that was the most important part of the assignment and the most useful skill to acquire from it. This assignment also showed me how important comments are, especially for joint codebases.

V. CONCLUSION

Bigshell facilitated the creation of a small shell program through the implementation of POSIX standards. Completing this project enabled me to gain experience in using the Unix process API to fulfill the project requirements such as signal handling, implementing built-in commands, and managing processes. The project ultimately met all the graded requirements with the addition of job control. Encountering the challenges I faced while working on this project led to a more in-depth understanding of the code itself and improved my debugging abilities. Bigshell is not a fully fleshed shell but has many of the utilities of one. After speaking with the instructor, I determined that to improve on Bigshell and take this project further I could implement a command history feature like the existing command history in current Unix shells using the curses library. Another idea for taking this project further would be to have the shell spawn its own GUI, which is typically handled by a separate program and would rely on a GUI library. Working on this project has inspired me to take it further and continue improving my C programming abilities.

REFERENCES

[1] Anon. “A Backgrounder on IEEE Std 1003.1” Open Group. Accessed: August 5, 2024. [Online]. Available: https://www.opengroup.org/austin/papers/backgrounder.html

[2] Anon. “The Open Group Base Specifications Issue 8” Open Group. Accessed: August 5, 2024. [Online]. Available: https://pubs.opengroup.org/onlinepubs/9799919799/

[3] J. R. Mashey. “Using a command language as a high-level programming language.” in *Proceedings of the 2nd international conference on Software engineering (ICSE '76)*., 1976, pp.169–176. [Online]. Available: https://dl.acm.org/doi/pdf/10.5555/800253.807670

[4] M. Jones, “Evolution of Shells in Linux” in *IBM Developer*. Dec. 2011. [Online]. Available: https://developer.ibm.com/tutorials/l-linux-shells/#a-history-of-shells0

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

[6] S. R. Bourne, "Unix time-sharing system: the unix shell," in The Bell System Technical Journal, vol. 57, no. 6, pp. 1971-1990, July-Aug, 1978. Accessed: August 5, 2024, doi: 10.1002/j.1538-7305.1978.tb02139.x. [Online]. Available: https://ieeexplore.ieee.org/document/6770407

FIGURES

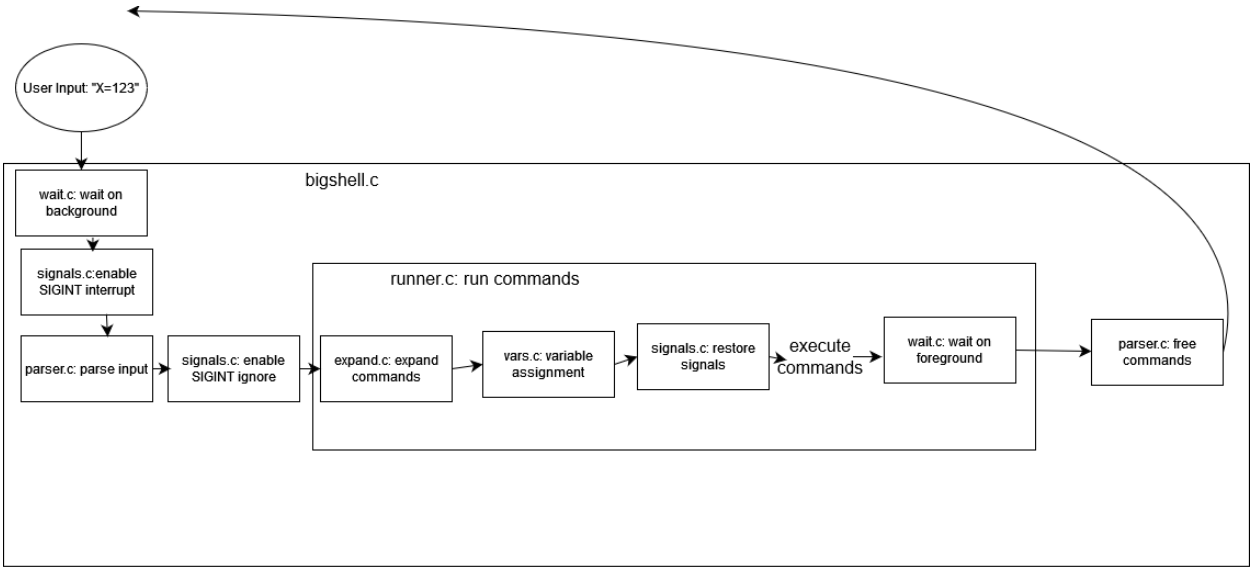

Figure 1: Rough outline of Bigshell flow when receiving variable assignment command

*Note: Does not include jobs.c implementation