BIGSHELL

Tony Chan
Collaborators: Victoria Sok, Jacky Kuang
Operating Systems I, CS374
December 1, 2024

Abstract

The BigShell program emulates key functionalities of a POSIX-compliant shell, offering beginners an engaging way to learn process management, interprocess communication, and shell behavior while building an impressive portfolio piece. This project balances the complexity of shell language with a practical codebase to enhance learning.

BigShell implements built-in and external shell utilities, parsing and executing user commands. Commands may involve variable assignments, I/O redirections, or external programs. Variable assignments allow users to define variables (e.g., name=value), while I/O redirections manage file input and output seamlessly before execution.

Process management is central to BigShell, enabling commands to execute in separate processes with job control for foreground and background tasks. Users can chain commands with pipes, where the output of one becomes the input for another. Signal handling demonstrates inter-process communication by managing events like interrupts or modifications.

The program also highlights the Unix process API, offering insight into system calls for creating, controlling, and terminating processes. This fosters understanding of how processes work and communicate effectively.

I. INTRODUCTION

Project overview

BigShell is to duplicate some functionalities of a POSIX like shell program. It will take user input and parse commands and its arguments. It will determine whether each of these commands are built-in or external commands and then execute the command. It will implement built-in shell commands like cd and pwd. Before the command is executed, it will detect I/O redirection in the command and execute a set of I/O redirections. It will be able for users to assign and evaluate environment variables. If the user enters a keyboard stop signal then the program will put the running process into a background process. Other interrupt signals will be disabled in the code. Multiple commands in a pipelines will also be recognized and be controlled by multiple job control functions.

Additional context and background information

Multics ("MULTiplexed Information and Computing Service") is an influential early time-sharing operating system [3][16]. Our modern Linux and POSIX-based operating systems [2]

have their roots in early developments like Multics, which significantly influenced the design of many operating systems, from microcomputers to mainframes.

This project is important

In POSIX-compliant systems, a shell serves as a vital interface for users to interact with the operating system. Bash (Bourne Again SHell) is among the most widely used shells. It lets users execute commands, manage files, and automate tasks through scripts. Learning Bash is valuable as it enables efficient system control, much like Windows PowerShell for Windows. Mastery of Bash is particularly useful for system administration, programming, and working with Unix-based servers or environments.

Similar existing command line interfaces

Other existing command line interfaces like Bash shell are Z shell for MacOS, Windows PowerShell for Windows, C shell (csh) and Korn shell (ksh).

II. Program Structure

BigShell program overall structure

To understand how commands in a pipeline work, I used separate control operators: is_pl, is_bg, and is_fg as major filters. See Figure 2 for the two major loops: run_command_list() and cnt2_for_loop of control operators. Each control operator shows how built-in or external commands, in child or parent processes, are executed. By separating the control operators, I could see how pipeline, foreground, or background commands are executed in their respective processes.

Figure 1 shows the BigShell Program Flow with 3 major functions: I. command_list_parse(), II. run_command_list(), and III. command_list_free(). The command_list_parse function in parser.c prompts for input, parses it into commands, arguments, and types, and stores this information in structures. The parsed commands are stored in a command_list structure, which contains an array of command structures with command words, assignments, I/O redirections, and control operators.

The run_command_list() function in runner.c is the main loop that executes commands. It separates stored commands into pipelines (is_pl), foreground (is_fg), and background (is_bg). It detects I/O redirection and variable assignments for each command, decides between built-in and external commands, and uses execvp() for external commands. It manages file descriptors and ensures proper execution of the command list.

After processing the input, the command_list_free() function frees the memory allocated for the command list, ensuring all resources are released. It iterates through each command and frees associated memory, preventing memory leaks.

A very important decision in the program is determining whether a command is built-in or external is crucial as it dictates execution. External commands and built-in commands for background/foreground run in child processes, while pipeline built-ins run in parent processes.

Command list cycle

The cnt2 for loop is the major loop to go through multiple commands in a command list. See Figure 2. To explain a complete command list cycle see the example in Figure 5. All the following will refer to Figure 5 example.

Parse command line input

The function command_list_parse() processes the user's input line, filtering commands and arguments with pipelines. It initializes variables, allocates memory for the command list structure, and parses the input character by character. Leading whitespaces are removed, assignment variables and I/O redirections are identified, and command words with arguments are extracted. Control operators such as “&,” “;,” and “|” are recognized to determine execution behavior. Each parsed command is stored in a command list structure. The entire pipeline isn’t executed at once. Commands are parsed and stored for later execution.

Built-in commands and external commands

User input commands are categorized as built-in or external and executed in a child or parent process or within the shell. See Figure 2 for command execution types. For example, a user enters 3 pipeline commands in BigShell. The program parses the input into individual commands. The first command “echo hello |” is recalled from the command list in run_command_list(). With a pipe, the control operator is_pl is set true, and the external command echo is executed in a child process. The second command is executed in the next cnt2 for loop iteration, also in a child process due to its pipeline control operator. Each control operator dictates whether commands run in a child or parent process. The third command “wc -l” has the control operator is_fg true and is executed in a child process.

Builtins.c implements built-in commands like cd, pwd, exit, export, unset, fg, bg, and jobs, which are executed within the shell without creating separate processes. The get_builtin function identifies and executes the built-in command directly.

I/O redirections

To perform a variety of I/O redirections on behalf of commands to be executed, the shell uses a function that handles the redirection operators. This function processes each redirection request, opening the necessary files for reading or writing, and maintains a virtual file descriptor table. This allows the shell to manage redirections without affecting the actual file descriptors, ensuring that built-in commands can be executed without disrupting the shell's environment. For example, if you run the command echo hello > filename.txt, the shell will redirect the output to filename.txt by opening the file and writing to it instead of displaying the output on the screen.

Shell variables to the environment

To assign, evaluate, and export shell variables to the environment, the shell parses variable assignments in the form of "name=value". These variables are stored in the shell's environment

and can be accessed or modified by subsequent commands. The shell provides mechanisms to export these variables, making them available to child processes and ensuring that they can be used in various commands and scripts. For example, if you set a variable with export TONY_VAR=tony, this variable will be available to any command or script executed in the same shell session, allowing you to use TONY_VAR in subsequent commands like echo $TONY_VAR.

Signal handling appropriate for a shell and executed commands

Signal handling in a shell involves defining functions to respond to signals, such as those from keyboard shortcuts like Ctrl-C and Ctrl-Z. For example, pressing Ctrl-Z stops the current process and moves it to the background. This is achieved using a signal handler function triggered when the signal is received. In the code, a signal handler checks if the signal is SIGTSTP (Ctrl-Z). If so, it prints a message and stops the process, enabling the shell to manage processes and respond to inputs. The signal handler in signal.c interrupts system calls and stops processes, while the wait() function manages their continuation and foreground/background states.

Manage processes and pipelines of processes using job control concepts

When there are multiple commands in a command list, each command can be executed separately in its own process and managed using job control concepts. This allows for the management of processes and pipelines of processes, enabling features such as foreground and background execution, as well as handling pipelines where the output of one command is the input to the next. For instance, you can add new jobs, check the job list, and remove jobs using functions like jobs_add(), jobs_get_joblist(), and jobs_remove_pgid.

III. Implementation Details

Tools, programming languages, and resources used

VSCode and VIM were used as the code editors. Compiling the C program was done using GCC GNU compiler on OS1 server. Running BigShell could be done on OS1 or any of the flip servers. Submission to Gradescope used Github as a source code tracking and Gradescope is connected to my Github account. A Makefile script was used to help compile and link programs. It has rules and dependencies on how to build executables. GDB (GNU Debugger) [9] was used as a debugging tool that can control the execution of programs. It supports functionalities such as setting breakpoints, inspecting variables, stepping through code, and diagnosing run time errors to streamline troubleshooting and development. Some documents that I used are POSIX 2008 [2], The class book The Linux Programming Book by Michael Kerrisk was very helpful [8] and IEEE Reference Guide [7]. YouTube videos by Benjamin Brewster [5], Tonnsman and Gombard [1].

Workflow, the build system, and modified functions

I followed the suggested order of implementation in the BigShell specifications: Built-in Commands, Non-built-in commands, Foreground/Background process waiting, Redirection, Pipelines, Variable Assignment, Signal Handling, and Job Control Functionality.

The C language build system I used is a Makefile script that calls the GNU gcc compiler to compile and link the source code. The Makefile can create separate debug and release directories with specific purposes, depending on the build configuration. It can have custom flags for debugging purposes, typically using the CFLAGS variable.

The functions that were modified the most due to TODO comments or major modifications are run_command_list(), command_list_parse(), get_builtin(), wait_on_fg_pgid(), wait_on_bg_jobs(), interrupting_signal_handler(), and others. If there was a TODO in a function, I modified it to fulfill its purpose.

Testing to the specifications

In the BigShell specification, example input commands are provided as well as their corresponding should be output. Also, each line of code was stepped through using GDB to see intermediate variables. I used many gprintf() statements to see the intermediate output. The ultimate test to see BigShell meets specifications is passing the Gradescope tests.

IV. Challenges And Solutions

Challenges encountered and solutions

One challenge was debugging a C program on a remote server without VSCode's integrated debugger. The code had to be compiled on OS1 due to a special configuration. The solution was to use GDB debugger instead of the integrated debugger in VSCode, with both VSCode and vim as editors on the OSU flip servers.

Another problem was determining what executed in a child process versus a parent process. The original pseudocode managed this by placing the fork function strategically. After separating is_pl, is_bg, and is_fg into different branches, See Figure 2. I stepped through each line of code to identify if it was in a child or parent process.

Understanding the group process ID, crucial for waiting commands and job control, was another challenge. The group process ID is the process ID of the first command in the command list, which is always the first element in the cmd structure. By using GDB to step through the code and print variables, I identified the first command and determined the group process ID.

Lessons learned

If I do another C language project, then I will find a really good visual debugger that can work on remote servers. Visual GDB only works on Visual Studio but I need one for VSCode that does work on the OS1 Engineering servers.

When debugging I would like to constantly monitor the current line of code is in a child process. Another thing to look at it is what the scope of a block.

To really understand the program flow and how it works I highly recommend stepping through the program code line by line. You will learn everything about the program.

V. Conclusion

Key points

BigShell is a project that emulates core features of a POSIX-compliant shell, offering a hands-on learning experience in process management, inter-process communication, and shell behavior. BigShell allows users to input commands, handle variable assignments, manage input/output redirections, and execute commands either directly or through external programs. Its functionality includes job control for handling pipelines, background, and foreground processes. Built-in commands like cd and pwd are executed internally, while external commands run as separate processes. Signal handling ensures smooth process management, responding effectively to user inputs like interruptions. Additionally, BigShell supports environmental variable assignments and exports. The implementation emphasizes modularity and clear separation of command processing stages, facilitating learning and debugging. This project demonstrates practical application of Unix concepts. BigShell has met the specifications.

Improvements and future directions

For future work, more built-in functions could be implemented. I finally realized child and parent processes are parallel processes. Some more exercises or side projects on doing parallel processing would be useful knowledge.

VI. References

[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] "- YouTube." Accessed: Nov. 28, 2024. [Online]. Available: https://www.youtube.com/watch?v=1R9h-H2UnLs&t=1994s&ab_channel=BenjaminBrewster

[6] "GDB: The GNU Project Debugger." Accessed: Nov. 30, 2024. [Online]. Available: https://www.sourceware.org/gdb/

[7] "Publish with IEEE Journals," IEEE Author Center Journals. Accessed: Dec. 01, 2024. [Online]. Available: https://journals.ieeeauthorcenter.ieee.org/

[8] M. Kerrisk, The Linux programming interface: a Linux and UNIX system programming handbook. San Francisco: No Starch Press, 2010.

[9] "GDB Command Reference - Index page." Accessed: Dec. 01, 2024. [Online]. Available: https://visualgdb.com/gdbreference/commands/

[10] "Git - Working with Remotes." Accessed: Dec. 01, 2024. [Online]. Available: https://git-scm.com/book/en/v2/Git-Basics-Working-with-Remotes

[11] "cppreference.com." Accessed: Dec. 01, 2024. [Online]. Available: https://en.cppreference.com/w/

[12] "Linux man pages online." Accessed: Dec. 01, 2024. [Online]. Available: https://man7.org/linux/man-pages/index.html

[13] "CONTENTS." Accessed: Dec. 01, 2024. [Online]. Available: https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/contents.html

[14] "Understanding Man Pages in Linux [Beginner's Guide]," It's FOSS. Accessed: Dec. 01, 2024. [Online]. Available: https://itsfoss.com/linux-man-page-guide/

[15] "GNU make." Accessed: Dec. 01, 2024. [Online]. Available: https://www.gnu.org/software/make/manual/make.html

[16] "Multics," Wikipedia. Nov. 17, 2024. Accessed: Dec. 01, 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Multics&oldid=1257893378

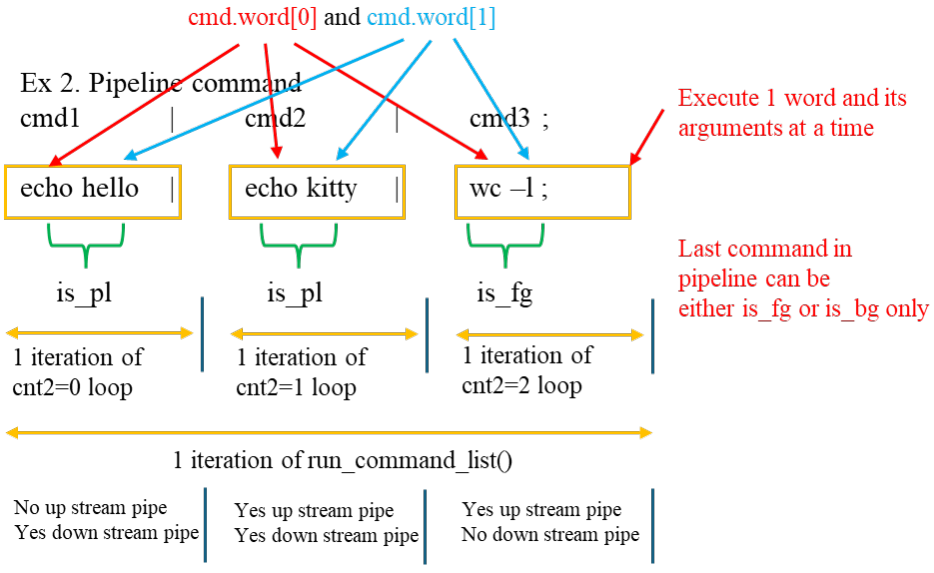
[17] S. Prata, C++ primer plus, 6th ed. Upper Saddle River, NJ: Addison-Wesley, 2012.

VII. FIGURES

Figure 1 BigShell Program Flow

BigShell Program Flow		
File Name	Major Functions in filename	Minor Functions within major functions
bigshell.c	main()	
parser.c	I. command_list_parse()	
		a) match_command() b) add_command() c) getline() -> user input
runner.c	II. run_command_list()	
expand.c		a) expand_command_words()
runner.c		cnt2 for loop is_pl, is_bg, is_fg
builtins.c		b) get_builtin()
runner.c		c) do_builtin_io_redirects() / io_redirects() d) do_variable_assignment() e) builtin = get_builtin()
signal.c		f) signal_restore()
parser.c	III. command_list_free()	

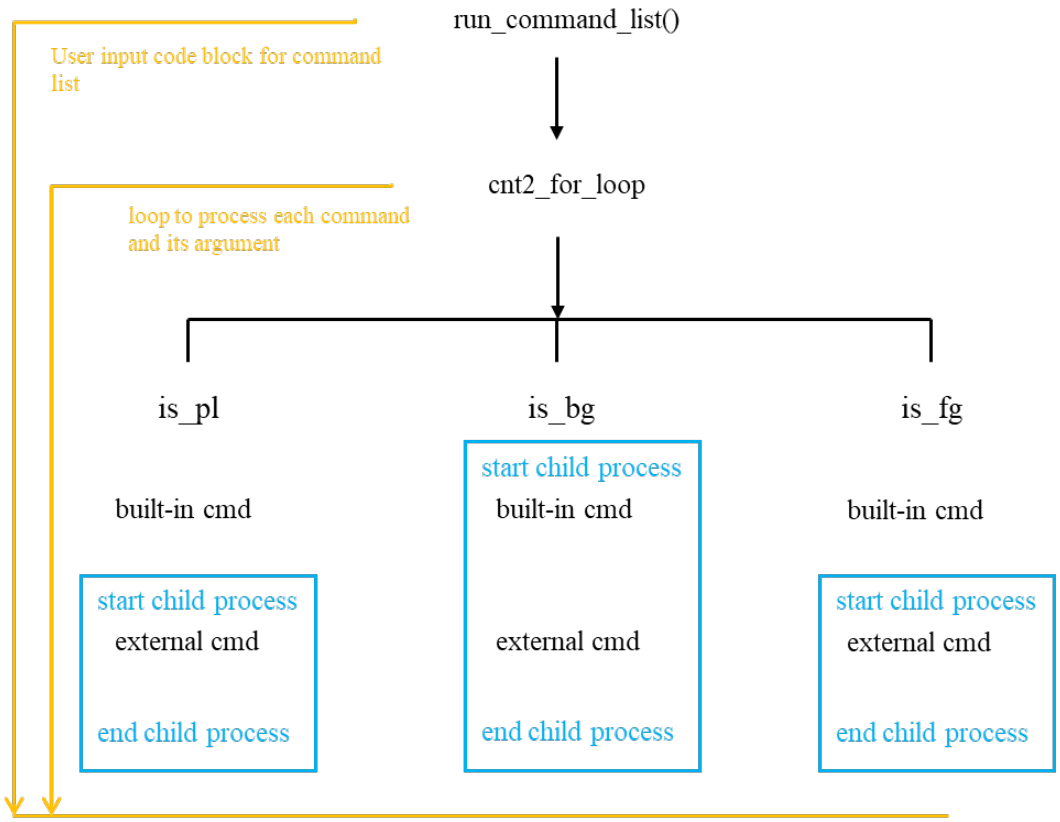
Figure 2 Three Commands in Pipeline Command List



Note: 1 The first command will determine the group process ID for the entire command list

Note: 1 The last command in a pipeline list is either a is_fg or is_bg, never is_pl.

Figure 3 Two major loops: run_command_list() and cnt2_for_loop of control operators



Note: 2 All pipeline, background and foreground external commands will be executed in a child process. In addition, background and foreground built-in commands will be executed in a child process. Pipeline built-in commands will be executed in parent processes or shell processes.

Figure 4 A Single Foreground Command

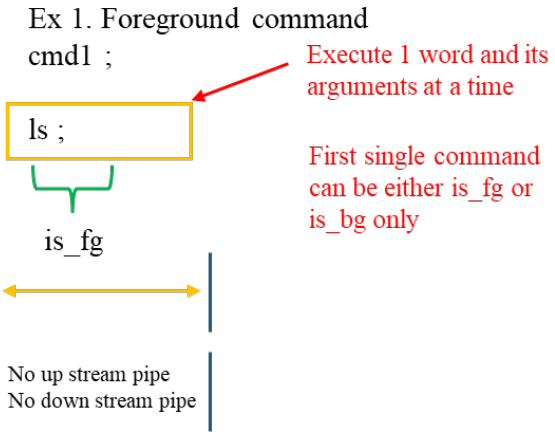


Figure 5 Single Background Command