Assignment 2 Shell

Contents

1. Description

We developed and implemented a basic command-line interpreter, like a UNIX shell like BASH or ZSH, using the C programming language. This shell is designed to handle at least 100 commands in a command line and up to 1000 arguments per command. While not a commercial-grade equivalent, it includes essential features. For example, users can change the shell prompt using the built-in command 'prompt' (e.g., typing '% prompt john$' changes the prompt to 'john$'). The 'pwd' command displays the current directory, and the 'cd' command can change the directory, including setting it to the user's home directory if no path is specified.

Additionally, the shell supports wildcard characters for filename expansion (e.g., '% ls *.c' expands to list all '.c' files), which can be implemented using the C function 'glob'. The shell also includes redirection of standard input, output, and error using '<', '>', and '2>' symbols, respectively. For example, '% ls -lt > foo' redirects output to the file 'foo', while '% cat < foo' redirects input from 'foo'.

Pipelines are supported, allowing the output of one command to be used as input for another (e.g., '% ls -lt | more'). The shell can run commands in the background (e.g., '% xterm &') and supports sequential job execution (e.g., '% sleep 20 ; ps -l').

Command history is also available, enabling users to navigate and repeat previous commands with the Up/Down arrow keys or the 'history' command. Special features include repeating commands using '!' followed by the command number or a string.

The shell inherits its environment from the parent process and can be terminated using the built-in 'exit' command. However, it should not be terminated by CTRL-C, CTRL-\, or CTRL-Z.

It is crucial that the implementation does not rely on existing shell programs (e.g., through the 'system' function). Commands like 'ls', 'cat', 'grep', 'sleep', 'ps', and 'xterm' are examples and not limited to these; the shell should handle any command or executable. Built-in commands like 'prompt', 'pwd', 'cd', 'history', and 'exit' must be implemented within the shell, not as external commands, with behavior closely matching those in the Bash shell. A significant component of this shell involves a command line parser, for which specific implementation suggestions are provided.

2. List Of Files

Following is a list of files included in the submission archive.

```
Shell/
 |-- Makefile
 |-- src/
 | |-- main.c
 | |-- command.c
 | |-- parser.c
 | |-- shell_builtins.c
 | |-- utils.c
 |-- include/
 | |-- utils.h
 | |-- log.h
 | |-- command.h
 | |-- parser.h
 | |-- shell_builtins.h
 |-- build/
```

2.1. Purpose Of Files

MAKEFILE

This Makefile automates the build and management of a simple UNIX shell program. It sets the default build mode to ‘release’, which creates an optimized binary without debug statements, and organizes the project into directories for build files, source files, header files, and tests. The shell program, named ‘Shell’, is compiled using ‘gcc’ with specified compiler flags for warnings and optimization. The Makefile includes utilities for directory management, file operations, and converts lowercase letters and hyphens to uppercase and underscores. It also defines color codes for terminal messages. Verbosity of the build output can be controlled using a variable, allowing for either detailed or minimal output. Key targets include ‘all’ for building the shell, which links object files into the final executable, and ‘init’ for setting up the project structure, including creating necessary directories and initializing a ‘main.c’ file. The ‘valgrind’ target enables memory checking using Valgrind, while the clean target removes build files to clean up the project. A ‘test’ target is also included for running a test suite located in the ‘test’ directory. This Makefile ensures efficient, consistent building, testing, and management of the shell project, minimizing manual intervention.

MAIN.C

This code implements the main functionality of a simple UNIX shell program. It includes necessary headers and libraries for handling commands, parsing input, and managing shell built-ins. The shell can operate in both interactive mode and script mode, with the latter allowing for automated testing by executing commands from a script file. The ‘getInput’ function reads user input from the command line or a script, handling various scenarios like EOF or signal interruptions.

The program also sets up signal handlers for common signals like SIGINT (Ctrl-C), SIGTSTP (Ctrl-Z), and SIGQUIT (Ctrl-\), ensuring proper handling of these interruptions. It maintains a global shell state, including the shell's prompt and history of commands. The main function initializes the shell state and enters a loop where it continually reads input, processes it into tokens, and then parses and executes the resulting commands. The shell supports basic command execution, including command chaining and history management, allowing users to re-run previous commands. The program cleans up resources, such as allocated memory and command history, before exiting.

COMMAND.C

The code in ‘command.c’ defines the implementation for functions declared in ‘command.h’, primarily focusing on managing command and command chain structures for a shell program. It includes functions for initializing and setting up commands (‘initSimpleCommand’, ‘initCommand’, and ‘initCommandChain’), adding commands to a command chain, and pushing arguments to a command's argument list. The code provides mechanisms for executing command chains and individual commands, handling both foreground and background processes, as well as built-in and external commands. It also includes comprehensive cleanup functions to release memory and resources associated with commands and command chains. Additionally, utility functions like ‘printCommandChain’ and ‘printSimpleCommand’ are provided for debugging and logging the structure and details of the commands being executed. The use of macros, such as ‘CHAINED_WITH’, enhances code readability by simplifying checks for command chaining operators.

PARSER.C

The provided code defines a function ‘parseTokens’ that parses an array of tokens to generate a ‘CommandChain’, representing a sequence of commands to be executed in a shell environment. It includes headers such as "parser.h" and "shell_builtins.h", and standard libraries like ‘<fcntl.h>‘ and ‘<glob.h>‘. The function initializes a command chain and iterates over the tokens, creating ‘Command’ and ‘SimpleCommand’ structures as needed. It handles various scenarios, including command

execution, piping, input/output redirection, and background execution. The function uses utility macros like ‘COMPARE_TOKEN’ for token comparison and includes error handling and cleanup mechanisms to manage memory allocation and resource usage. The final command chain is built by linking commands and their respective simple commands, taking care of different chaining operators and special cases like history commands and wildcards.

SHELL_BUILTINS.C

The provided code defines the implementation of various built-in shell commands and utility functions for a shell program. It includes functions for managing the shell's command history, such as ‘add_to_history’ to add commands to the history list and ‘get_command’ to retrieve commands from the history. The ‘init_shell_state’ function initializes the shell's state, while ‘clear_shell_state’ cleans up resources. The code also defines functions for manipulating file descriptors, such as ‘setUpFD’ and ‘resetFD’, to handle input/output redirection. Built-in commands like ‘cd’, ‘pwd’, ‘exit’, ‘history’, and ‘prompt’ are implemented, each handling specific shell functionalities. Additionally, an ‘executeProcess’ function is provided to handle the execution of external processes. The ‘commandRegistry’ struct maps command names to their corresponding execution functions, allowing the shell to identify and execute commands accordingly.

UTILS.C

This code defines several utility functions for string manipulation. The ‘tokenizeString’ function splits an input string into an array of tokens based on a specified delimiter, handling quoted substrings as single tokens. The ‘getTokenCount’ function counts the number of tokens in an array of strings, while ‘freeTokens’ deallocates memory used by this array. Additionally, the ‘removeQuotes’ function removes enclosed quotes from a string if present, returning a new string without the quotes. These functions facilitate string processing tasks often needed in command-line or shell environments.

UTILS.H

This header file, ‘utils.h’, defines several macros and utility functions to aid in shell operations. It includes logging macros for error, debug, and general messages, as well as string-related macros such as ‘MAX_STRING_LENGTH’ for buffer sizes and ‘COPY’ for safely copying strings. The file provides function declarations for ‘tokenizeString’, which splits a string into tokens based on a delimiter while handling quoted substrings, ‘freeTokens’ for deallocating memory used by token arrays, ‘removeQuotes’ for removing enclosing quotes from a string, and ‘getTokenCount’ to count the number of tokens in an array. These utilities simplify string manipulation and logging within the shell.

LOG.H

The ‘log.h’ header file provides a flexible logging framework designed to enhance message clarity beyond simple ‘printf’ statements. It defines macros for colored output, making log messages visually distinct based on their type—errors, debug information, or regular prints—using ANSI escape codes. The file allows for different logging modes: errors (‘LOG_ERR’), debug messages (‘LOG_DBG’), and standard messages (‘LOG_PRI’). The debug mode can be toggled with a compiler flag, and detailed annotations, including file name, function name, and line number, can be included in logs when enabled. The ‘LOG’ macro handles the formatting and output of log messages, ensuring that critical errors and user messages are always displayed, while debug messages are conditional on the debug mode setting.

COMMAND.H

The ‘command.h’ file defines the structures and functions for handling commands and command chains in a shell-like environment. It includes the ‘SimpleCommand’ struct, which represents individual commands with their associated arguments, file descriptors, and execution details. It also defines the ‘Command’ struct, which can be a pipeline of ‘SimpleCommand’ instances, and the ‘CommandChain’ struct, which represents a sequence of such commands connected by chaining operators like ‘;’ or ‘&’.

The file provides functions for creating and managing these structures, including initialization, argument handling, memory cleanup, and execution. Additionally, it offers utility functions for debugging by printing the command structures. This setup supports the execution of complex command sequences and their management within the shell environment.

PARSER.H

The ‘parser.h’ file defines macros and a function for parsing tokens into a command chain structure. It includes macros for identifying various types of tokens, such as chaining operators (‘&’, ‘;’), pipes (‘|’), redirection operators (‘>‘, ‘<‘, ‘2>‘), and ignorable tokens (whitespace). The key function, ‘parseTokens’, takes an array of tokens and constructs a ‘CommandChain’ based on the tokenized input. This command chain represents a sequence of commands, potentially involving background execution, piping, and redirection. The caller is responsible for managing memory, including freeing the allocated structures after use.

SHELL_BUILTINS.H

The ‘builtins.h’ file provides structures and functions to manage shell built-ins and maintain the shell's internal state. It defines a linked list for storing shell command history, with functions to add commands to this history, retrieve commands by index, and find commands starting with a given prefix. It also includes a ‘ShellState’ structure to track the original file descriptors for standard input, output, and error, as well as the current shell prompt and command history. Functions for initializing and cleaning up the shell state are provided. Additionally, it declares built-in command functions for common operations such as changing directories (‘cd’), exiting the shell (‘exit’), printing the working directory (‘pwd’), and displaying command history (‘history’). The ‘getExecutionFunction’ function retrieves the appropriate execution function for a given command, while ‘executeProcess’ handles the execution of external commands by forking a child process.

3. Self-Diagnosis And Evaluation

All the functions specified in the manual have been properly implemented and thoroughly tested. The following is a list of the functions that have been implemented.

1. The shell features a customizable prompt, initially set to ‘%’. Users can change this prompt by issuing a command with a single argument to set a new prompt.

2. The built-in ‘pwd’ command has been implemented within the shell, replacing the default binary found in ‘/usr/bin’.

3. Directory navigation is supported, allowing users to change directories using ‘cd’ with a specified directory or return to the home directory if no argument is provided.

4. The shell supports wildcard character expansion, converting wildcard patterns in commands into matching filenames.

5. Redirection of standard input, output, and error streams to different files is supported.

6. Unix pipelines are available, enabling the chaining of multiple commands in a pipeline of arbitrary length.

7. Background job execution is enabled using the '&' operator, which starts a job in the background and immediately returns the prompt. Background jobs can be linked with other background or foreground jobs.

8. Sequential execution of jobs is supported, allowing multiple pipelines to run one after another using the ';' operator.

9. A history feature like bash is included, where each command is saved. The 'history' command displays previous commands, and the '!' operator can run commands from the history by index or by matching a prefix.

10. The shell inherits its environment from the parent process.11. A bash-like 'exit' built-in command is available, which can also accept an argument to specify an exit status, defaulting to 0.

4. Discussion Of Solution

The primary emphasis was on developing a modular and extensible architecture by employing various abstractions. This discussion delves into the design decisions that were made and highlights some of the benefits associated with these choices.

4.1. Modular Design And Abstractions

We employed a modular design strategy to improve the maintainability and extensibility of the shell. By defining simple commands, pipelines, and chains as separate structures or objects with their own methods, we achieved a clear separation of concerns. This approach allows for the independent incorporation of new features at each level. For example, the integration of logical chaining became more straightforward thanks to this modular design. **4.2. Data Structure And Algorithmic Choices Array Usage for Improved Cache Locality**

Command Arguments and Pipelines: Arrays were utilized to manage command arguments and the individual commands within pipelines. This choice was made to enhance cache locality, given that these elements are accessed repeatedly during execution.

Chains: Although sequential chains are accessed less often, linked lists were selected for their efficiency in adding new elements. A tail pointer was incorporated into the design to achieve O(1) write times, reflecting the infrequent nature of read operations.

History Management

Current Implementation: Given that history is accessed less frequently but updated more often, it is currently implemented using a linked list.

Potential Improvement: To optimize the balance between read and write operations, there is consideration of transitioning to an array-based approach for storing history, which could offer performance benefits.

4.3. Ease Of Feature Addition

The choice to utilize separate structures for commands, pipelines, and chains enhances both maintainability and the ease of adding new features. This approach allows each component to be extended individually, thereby increasing the shell's flexibility in adapting to changing needs.

4.4. Debugging

We also want to highlight the debugging strategies that significantly contributed to the project's development. We incorporated a header-only logging library to manage logs at various levels, enabling us to create both release and debug builds. The debug build provides detailed log outputs, which are excluded from the release build to avoid unnecessary verbosity while ensuring the release build remains optimized. Additionally, Valgrind was extensively utilized (integrated into the makefile) to identify memory leaks and address segmentation faults.

5. Test Evidence

All code is compiled with the following flags for gcc (from the makefile)

```
CFLAGS=-Wall -Wextra -Werror
```

For each requirement, test evidence is provided via running the shown commands and collecting their output and showing it verbatim.

5.1. Compilation With Make

```
% make clean; make; ./build/Shell
-- Cleaned: build/*
  CC       src/command.c
  CC       src/main.c
  CC       src/parser.c
  CC       src/shell_builtins.c
  CC       src/utils.c
  LD       build/Shell -- Build
successful in release mode.
%
```

The shell program compiles successfully and displays the prompt to the user. (It's worth noting that the compilation was performed within our own shell, demonstrating its effectiveness.)

5.2. Simple Commands

The shell can execute simple commands according to the specified grammar. For every valid input, where the input commands correspond to actual external executables, the shell executes them correctly.

```
% ls
Makefile  build  include  src
% touch tempfile
% ls
Makefile  build  include  tempfile
% rm -f tempfile
% mkdir -p random_ahh_dir
% rm -rf random_ahh_dir
% ls
Makefile  build  include  src
%
```

The test results indicate that all simple commands and basic built-ins, including pwd and exit, are functioning correctly.

5.3. Shell builtins

Our shell program supports four built-in commands: ‘cd’, ‘pwd’, ‘prompt’, and ‘exit’. Each of these commands is handled by the shell, and when given valid input, they produce the correct output. The home directory is configured to be ‘/home/Shell’.

```
% prompt
testuser$ testuser$ pwd
/home/josiah/c_code/assignment_2
testuser$ cd include
testuser$ pwd
/home/josiah/c_code/assignment_2/includ
e testuser$ cd ../src testuser$ pwd
/home/josiah/c_code/assignment_2/src
testuser$ ls
command.c  main.c  parser.c  shell_builtins.c  utils.c
testuser$ cd testuser$ pwd /home/josiah
testuser$ exit
#
```

The test results indicate that all simple commands and basic built-ins, including pwd and exit, are functioning correctly.

5.4. Tokenisation

```
$ make clean ; make BUILD_DEFAULT=debug ; ./build/Shell
-- Cleaned: build/*
  CC       src/command.c
  CC       src/main.c
  CC       src/parser.c
  CC       src/shell_builtins.c
  CC       src/utils.c
  LD       build/Shell
-- Build successful in debug mode.
[DEBUG]:  (main,135) Starting shell
% ls
[DEBUG]:  (main,195) Token 0: [ls]
[DEBUG]:  (printCommandChain,342) Printing command chain
[DEBUG]:  (printCommandChain,350) [Link 1]
[DEBUG]:  (printSimpleCommand,365) -- name: ls
[DEBUG]:  (printSimpleCommand,366) -- args:
[DEBUG]:  (printSimpleCommand,369) -- -- ls
[DEBUG]:  (printSimpleCommand,372) -- Input FD: 0
[DEBUG]:  (printSimpleCommand,373) -- Output FD: 1
[DEBUG]:  (printSimpleCommand,374) --------------------
[DEBUG]:  (executeCommand,214) Executing command : ls
[DEBUG]:  (executeProcess,421) Waiting for child process, with command name ls
Makefile  build  include  src
[DEBUG]:  (executeProcess,438) Finished executing command ls
[DEBUG]:  (executeCommand,230) Command executing with pid: 11743
[DEBUG]:  (main,206) Command executed with status 0
[DEBUG]:  (cleanUpSimpleCommand,257) Cleaning up simple command: ls
% sleep 1 & ps -l ; echo hello ; ls | grep b
[DEBUG]:  (main,195) Token 0: [sleep]
[DEBUG]:  (main,195) Token 1: [1]
[DEBUG]:  (main,195) Token 2: [&]
[DEBUG]:  (main,195) Token 3: [ps]
[DEBUG]:  (main,195) Token 4: [-l]
[DEBUG]:  (main,195) Token 5: [;]
[DEBUG]:  (main,195) Token 6: [echo]
[DEBUG]:  (main,195) Token 7: [hello]
[DEBUG]:  (main,195) Token 8: [;]
[DEBUG]:  (main,195) Token 9: [ls]
[DEBUG]:  (main,195) Token 10: [|]
```

```
[DEBUG]:  (main,195) Token 11: [grep]
[DEBUG]:  (main,195) Token 12: [b]
[DEBUG]:  (printCommandChain,342) Printing command chain
[DEBUG]:  (printCommandChain,350) [Link 1]
[DEBUG]:  (printSimpleCommand,365) -- name: sleep
[DEBUG]:  (printSimpleCommand,366) -- args:
[DEBUG]:  (printSimpleCommand,369) -- -- sleep
[DEBUG]:  (printSimpleCommand,369) -- -- 1
[DEBUG]:  (printSimpleCommand,372) -- Input FD: 0
[DEBUG]:  (printSimpleCommand,373) -- Output FD: 1
[DEBUG]:  (printSimpleCommand,374) --------------------
[DEBUG]:  (printCommandChain,350) [Link 2]
[DEBUG]:  (printSimpleCommand,365) -- name: ps
[DEBUG]:  (printSimpleCommand,366) -- args:
[DEBUG]:  (printSimpleCommand,369) -- -- ps
[DEBUG]:  (printSimpleCommand,369) -- -- -l
[DEBUG]:  (printSimpleCommand,372) -- Input FD: 0
[DEBUG]:  (printSimpleCommand,373) -- Output FD: 1
[DEBUG]:  (printSimpleCommand,374) --------------------
[DEBUG]:  (printCommandChain,350) [Link 3]
[DEBUG]:  (printSimpleCommand,365) -- name: echo
[DEBUG]:  (printSimpleCommand,366) -- args:
[DEBUG]:  (printSimpleCommand,369) -- -- echo
[DEBUG]:  (printSimpleCommand,369) -- -- hello
[DEBUG]:  (printSimpleCommand,372) -- Input FD: 0
[DEBUG]:  (printSimpleCommand,373) -- Output FD: 1
[DEBUG]:  (printSimpleCommand,374) --------------------
[DEBUG]:  (printCommandChain,350) [Link 4]
[DEBUG]:  (printSimpleCommand,365) -- name: ls
[DEBUG]:  (printSimpleCommand,366) -- args:
[DEBUG]:  (printSimpleCommand,369) -- -- ls
[DEBUG]:  (printSimpleCommand,372) -- Input FD: 0
[DEBUG]:  (printSimpleCommand,373) -- Output FD: 4
[DEBUG]:  (printSimpleCommand,374) --------------------
[DEBUG]:  (printSimpleCommand,365) -- name: grep
[DEBUG]:  (printSimpleCommand,366) -- args:
[DEBUG]:  (printSimpleCommand,369) -- -- grep
[DEBUG]:  (printSimpleCommand,369) -- -- b
[DEBUG]:  (printSimpleCommand,372) -- Input FD: 3
[DEBUG]:  (printSimpleCommand,373) -- Output FD: 1
[DEBUG]:  (printSimpleCommand,374) --------------------
[DEBUG]:  (executeCommand,214) Executing command : sleep
[DEBUG]:  (executeProcess,438) Finished executing command sleep
[DEBUG]:  (executeCommand,230) Command executing with pid: 12115
[DEBUG]:  (executeCommand,214) Executing command : ps
[DEBUG]:  (executeProcess,421) Waiting for child process, with command name ps
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000     788     495  0  80   0 -  1590 do_wai pts/4    00:00:00 bash
0 S  1000   11643     788  0  80   0 -   702 do_wai pts/4    00:00:00 Shell
1 D  1000   12115   11643  0  80   0 -   702 -      pts/4    00:00:00 Shell
0 R  1000   12116   11643  0  80   0 -  1872 -      pts/4    00:00:00 ps
[DEBUG]:  (executeProcess,438) Finished executing command ps
[DEBUG]:  (executeCommand,230) Command executing with pid: 12116
[DEBUG]:  (executeCommand,214) Executing command : echo
[DEBUG]:  (executeProcess,421) Waiting for child process, with command name echo
hello
[DEBUG]:  (executeProcess,438) Finished executing command echo
[DEBUG]:  (executeCommand,230) Command executing with pid: 12117
[DEBUG]:  (executeCommand,214) Executing command : ls
[DEBUG]:  (executeProcess,421) Waiting for child process, with command name ls
[DEBUG]:  (executeProcess,438) Finished executing command ls
[DEBUG]:  (executeCommand,230) Command executing with pid: 12118
[DEBUG]:  (executeCommand,214) Executing command : grep
```

```
[DEBUG]:  (executeProcess,421) Waiting for child process, with command name grep
build
[DEBUG]:  (executeProcess,438) Finished executing command grep
[DEBUG]:  (executeCommand,230) Command executing with pid: 12119
[DEBUG]:  (main,206) Command executed with status 0
[DEBUG]:  (cleanUpSimpleCommand,257) Cleaning up simple command: sleep
[DEBUG]:  (cleanUpSimpleCommand,257) Cleaning up simple command: ps
[DEBUG]:  (cleanUpSimpleCommand,257) Cleaning up simple command: echo
[DEBUG]:  (cleanUpSimpleCommand,257) Cleaning up simple command: ls
[DEBUG]:  (cleanUpSimpleCommand,257) Cleaning up simple command: grep
```

We tested the execution of two distinct commands: a simple command and a more complex command involving multiple shell operators, to examine the debug output. The results demonstrated that the shell accurately tokenizes, parses, and executes these commands as expected.

5.5. Wildcard Expansion

The following session demonstrates the shell's ability to handle wildcard expansion. The shell supports bash-like wildcard expansion, including the "*" and "?" operators. When an argument includes a wildcard, it is expanded to match the corresponding filenames. If a wildcard in a command does not match any files, it remains unchanged. This behavior is illustrated in the second command example.

```
% ls *
Makefil
e
build:
Shell  build_mode  command.o  main.o  parser.o  shell_builtins.o  utils.o
 include: command.h  log.h  parser.h
shell_builtins.h  utils.h
 src: command.c  main.c  parser.c  shell_builtins.c
utils.c
% ls *.c
ls: cannot access '*.c': No such file or directory
% echo ./src/*.c
./src/command.c ./src/main.c ./src/parser.c ./src/shell_builtins.c ./src/utils.c
% wc -l src/*.c include/*.h
  376 src/command.c
  222 src/main.c
  305 src/parser.c
  495 src/shell_builtins.c
  111 src/utils.c
  197 include/command.h
   77 include/log.h
   43 include/parser.h
  113 include/shell_builtins.h
   70 include/utils.h
 2009 total
% wc -l src/*.c include/*.h | grep total | awk '{print $1}'
2009
% wc -l src/*.c include/*.h | grep total | awk '{print $1}' | xargs echo "Total
LOC: "
Total LOC:  2009
% touch file.txt_ez.c log.pop_ab.d
% echo *.txt_*.?
file.txt_ez.c
%
```

As anticipated, valid wildcard patterns expand to include all files in the current directory that match the pattern. Hidden files are excluded from the wildcard expansion. If no files match the pattern, the argument containing the wildcard characters remains unchanged.

5.6. Handling Of Interrupts

The shell correctly handles interrupts generated by the CTRL-Z, CTRL-C, CTRL-\ keys, as well as CTRL-D. By using the debug build, we can observe how the shell manages these signals.

```
% ^C
[DEBUG]:  (sigint_handler,76) CTRL-C pressed. signo: 2

% ^Z
[DEBUG]:  (sigtstp_handler,81) CTRL-Z pressed. signo: 20

% ^\
[DEBUG]:  (sigquit_handler,86) CTRL-\ pressed. signo: 3

EOF detected. Exiting shell.
```

The program demonstrates behavior identical to BASH. Signals are properly recognized by the shell, and the program does not terminate unexpectedly. Instead, the shell handles these signals gracefully: it reports the event without terminating, continuing its operation. Notably, CTRL-D will cause the shell to exit.

5.7. Claim Of Zombies

The shell is designed to properly clean up any background child processes once they have completed their execution. To verify this functionality, one can start a background process and later check the process list. If the shell has not reclaimed the process, it may appear as a defunct or zombie process.

```
% sleep 1 & ps -l
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000   20013     495  0  80   0 -  1562 do_wai pts/4    00:00:00 bash
0 S  1000   22264   20013  0  80   0 -   697 do_wai pts/4    00:00:00 Shell
0 S  1000   22815   22264  0  80   0 -   804 hrtime pts/4    00:00:00 sleep
0 R  1000   22816   22264  0  80   0 -  1872 -      pts/4    00:00:00
ps % ps -l
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000   20013     495  0  80   0 -  1562 do_wai pts/4    00:00:00 bash
0 S  1000   22264   20013  0  80   0 -   697 do_wai pts/4    00:00:00 Shell
0 R  1000   22834   22264  0  80   0 -  1872 -      pts/4    00:00:00 ps
% ps -aux | grep sleep
%
```

In the initial command, ‘sleep’ appears in the ‘ps -l’ output because it is executed in the background, and then ‘ps’ is executed immediately. After a short delay, once the ‘sleep’ process has finished, it no longer appears in the process list, as confirmed by the absence of results from the ‘grep’ command.

5.8. Auto Grading

A testing framework was implemented to streamline the testing of various commands. This was accomplished by introducing a script mode in the shell, allowing it to read inputs from a file instead of directly from the user. The framework processes a series of test files, executes the commands in these scripts, and compares the outputs with those from a reference shell, like bash. The following are the results from this testing process.

```
$ make test

SHELL TEST SUITE

Traces Directory:  Tests
Tests:  simple, advanced
Shell:  ../build/Shell
Reference Shell:  bash
Default Timeout:  1
Log To File:  False
...

Running tests : simple
     Running file: simple.test                      PASSED
Running file: builtins.test                PASSED
        Running file: exec_only.hidden              PASSED
        Running file: pipeline.test                 PASSED
        Running file: ioredir.test                  PASSED
        Running file: ioredir_one.hidden            PASSED
        Running file: pipeline_one.hidden           PASSED
        Running file: pipeline_two.hidden           PASSED
        Running file: pipeline_ioredir.hidden       PASSED
Running tests : advanced
        Running file: chaining.test                 PASSED
        Running file: wildcards.test                PASSED
        Running file: quotes.test                   PASSED
        Running file: wild_chaining.test            PASSED
        Running file: wild_chaining.hidden          PASSED
       Running file: wildcards_one.hidden          PASSED
...

SUMMARY

Total : 15, Passed : 15, Failed : 0
Finished in 3.11s
```

Our implementation successfully passed all the tests. The testing framework, along with all the relevant files, is included with the submission. These files provide comprehensive test coverage and thoroughly examine all the features of the shell. Additional features have been tested manually and are documented in the sections above.

Please note the inclusion of .test and .hidden files. During development, only .test files were used, while the final shell was tested with all files, including the hidden ones, to ensure no hardcoding issues were present. All script files are highly descriptive, featuring a variety of command combinations as well as simple commands.

6. Advanced Functionality

All the advanced features outlined have been successfully implemented, as demonstrated in the following session.

6.1. Sequential Commands

Sequential commands are handled using the ';' operator. When commands are listed and separated by this operator, the shell executes them sequentially, one after the other. Importantly, the shell will continue processing the remaining commands even if one of them fails. Additionally, both external and built-in commands can be included in these command chains.

```
% ls
Makefile  build  include  src
```

```
% echo hello
hello
% echo hello ; echo world ; echo 1 ; ps -l ; pwd ; ls ; cd src ;
ls hello world
1
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000   20013     495  0  80   0 -  1562 do_wai pts/4    00:00:00 bash
0 S  1000   22264   20013  0  80   0 -   697 do_wai pts/4    00:00:00 Shell
0 R  1000   34997   22264  0  80   0 -  1872 -      pts/4    00:00:00 ps
/home/josiah/c_code/assignment_2
Makefile  build  include  src
command.c  main.c  parser.c  shell_builtins.c
utils.c % cd ..
% ls
Makefile   build   include   src %
echo before ; false ; echo after
before after
%
```

Observations: The shell processes each command in a chain as expected. If a command fails (returns false), the shell continues to execute the subsequent commands without interruption.

6.2. Background And Concurrent Execution

Any valid command or pipeline that ends with `&` is executed in the background.

```
% sleep 2 &
% ls
Makefile  build  include  src
% sleep 5 &
% ps -l
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000   20013     495  0  80   0 -  1562 do_wai pts/4    00:00:00 bash
0 S  1000   22264   20013  0  80   0 -   697 do_wai pts/4    00:00:00 Shell
0 S  1000   36433   22264  0  80   0 -   804 hrtime pts/4    00:00:00 sleep
0 R  1000   36449   22264  0  80   0 -  1872 -      pts/4    00:00:00 ps
% sleep 1 & sleep 2 & ps -l ; echo done
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000   20013     495  0  80   0 -  1562 do_wai pts/4    00:00:00 bash
0 S  1000   22264   20013  0  80   0 -   697 do_wai pts/4    00:00:00 Shell
0 S  1000   36646   22264  0  80   0 -   804 hrtime pts/4    00:00:00 sleep
0 S  1000   36647   22264  0  80   0 -   804 hrtime pts/4    00:00:00 sleep
0 R  1000   36648   22264  0  80   0 -  1872 -      pts/4    00:00:00 ps
done
% sleep 5 & ps -l | grep sleep
0 S  1000   36936   22264  0  80   0 -   804 hrtime pts/4    00:00:00 sleep
%
```

We observe that the shell correctly manages background processes, reclaiming them once they have completed, as they no longer appear after execution. Background execution functions properly with both pipelines and sequential operators.

6.3. IO Redirection

Any valid command or pipeline ending with appropriate I/O redirection operators should have its input, output, or error streams redirected to the specified file.

```
% echo hello > hello.txt ; cat hello.txt
hello
% echo world > world.txt ; cat world.txt
world
% cat hello.txt world.txt | grep w
world
```

```
% cat < hello.txt
hello
% cat hello.txt world.txt > concat.txt ; wc < concat.txt
 2  2 12
% ls jajajaja 2> error.txt
% cat error.txt
ls: cannot access 'jajajaja': No such file or directory
% ls jaja > error.txt
ls: cannot access 'jaja': No such file or directory
% ls *.txt
concat.txt  error.txt  hello.txt  world.txt
% echo joooooo | wc > wc.txt ; cat wc.txt ; rm wc.txt
      1       1       8
%
```

We observe that the shell correctly handles I/O redirection. This is confirmed by checking the contents of the output file, which accurately captures the redirected output.

6.4. Pipelines And Complex Command Lines

The shell accurately executes any valid pipeline of arbitrary length, even when combined with other operators.

```
% cat date.txt
Wed Nov 15 18:17:42 PKT 2023
% cat < date.txt | grep "Nov" | awk '{print $2}' > month.txt ; cat month.txt
Nov
% cat < month.txt | tr [:upper:] [:lower:] >> month.txt ; cat
month.txt Nov nov
% cat < month.txt | grep -c "t"
0
% rm -f date.txt month.txt
% ls
Makefile  build  include  src
% echo Josaih ; sleep 5 & ps | grep sleep
Josaih
  44166 pts/4    00:00:00 sleep
%
```

We observe that pipelines of arbitrary length work correctly, even when combined with other operators.

6.5. History

The purpose of this test is to verify if the history functionality in the shell program operates correctly. Several commands are executed to check if the history list updates as expected and if we can access this history list accurately using the '!' operator.

```
% ls
Makefile  build  include  src
% pwd
/home/josiah/c_code/assignment_2
% history
1 sleep 1 & ps -l
2 ps -l
3 ps -aux | grep sleep
4 clear
5 ls
6 echo hello
7 echo hello ; echo world ; echo 1 ; ps -l ; pwd ; ls ; cd src ; ls 8 cd ..
9  ls
10 echo before ; false ; echo after
11 clear
12 sleep 2 &
```

```
13 ls
14 sleep 5 &
15 ps -l
16 sleep 1 & sleep 2 & ps -l ; echo done
17 sleep 5 & ps -l | grep sleep
18 clear
19 echo hello > hello.txt ; cat hello.txt
20 echo world > world.txt ; cat world.txt
21 cat hello.txt world.txt | grep w
22 cat < hello.txt
23 cat hello.txt world.txt > concat.txt ; wc < concat.txt
24 ls jajajaja 2> error.txt
25 cat error.txt
26 ls jaja > error.txt
27 ls *.txt
28 echo joooooo | wc > wc.txt ; cat wc.txt ; rm wc.txt
29 clear
30 rm concat.txt
31 rm error.txt
32 rm hello.txt
33 rm world.txt
34 clear
35 cat date.txt
36 echo Wed Nov 15 18:17:42 PKT 2023 > date.txt
37 clear
38 cat date.txt
39 cat < date.txt | grep "Nov" | awk '{print $2}' > month.txt ; cat month.txt
40 cat < month.txt | tr [:upper:] [:lower:] >> month.txt ; cat month.txt
41 cat < month.txt | grep -c "t"
42 rm -f date.txt month.txt
43 ls
44 echo Josaih ; sleep 5 & ps | grep sleep
45 clear
46 ls
47 pwd
48 history % !38
cat: date.txt: No such file or directory
% !47
/home/josiah/c_code/assignment_2
% !pw
/home/josiah/c_code/assignment_2
% echo
hello
hello % !ec
hello %
echo a a %
echo b b %
echo c
c % !echo c
%
```

As expected, the shell displays the history list accurately. The history list is not persistent, as it was not a requirement; however, persistence can be easily implemented by saving the history to a file.

6.6. Handling of slow syscalls

The test shows that even during the execution of a slow system call, the shell remains responsive. Inputs are stored and processed the next time the shell becomes available.

```
% sleep 5
asdwsad
% asdwsad: No such file or directory
% sleep 2
pwd
```

```
% /home/josiah/c_code/assignment_2
%
```

Inputs provided while the shell is executing sleep commands are successfully stored and executed once the previous process completes.

7. Source Code Listings

```
# Set the Default build. Set to `release` to get the release build (optimized binary without
debug statements)
BUILD_DEFAULT = release

# Directories
BUILD_DIR=build
SRC_DIR=src
INCLUDE_DIR=include
TEST_DIR=test

# Target executable
TARGET_NAME=Shell
TARGET=$(BUILD_DIR)/$(TARGET_NAME)

# Shell Commands
CC=gcc
MKDIR=mkdir -p
```

```
RM=rm -rf
CP=cp

# useful utility to convert lowercase to uppercase.
UPPERCASE_CMD = tr '[:lower:][\-/]' '[:upper:][__]'

# Flags for compiler and other programs
CFLAGS=-Wall -Wextra -Werror
VALG_FLAGS = --leak-check=full --track-origins=yes
DEBUG_FLAGS = -g -DDEBUG
RELEASE_FLAGS = -O3 -march=native
LINKER_FLAGS =

# Color codes for print statements
GREEN = \033[1;32m
CYAN = \033[1;36m
RED = \033[1;31m
RESET = \033[0m

# Verbosity control. Inspired from the Contiki-NG build system. A few hacks here and there,
will probably improve later. ifeq ($(V),1)   TRACE_CC =
  TRACE_LD =
  TRACE_MKDIR =
  TRACE_CP =
  Q ?=

  BUILD_SUCCESS=
  BUILD_FAILURE=:
LINK_FAILURE=:
  INIT_SUCCESS=
  INIT_MAIN=
  RUN=
  VALGRIND_RUN=

  CLEAN=
  MK_INIT_ERROR= else

  TRACE_CC       = @echo "$(CYAN)  CC      $(RESET)" $<
  TRACE_LD       = @echo "$(CYAN)  LD      $(RESET)" $@
  TRACE_MKDIR    = @echo "$(CYAN)  MKDIR   $(RESET)" $@
  TRACE_CP       = @echo "$(CYAN)  CP      $(RESET)" $< "-->" $@
  Q ?= @

  BUILD_SUCCESS  =@echo "-- $(GREEN)Build successful in $(BUILD_DEFAULT) mode.$(RESET)"
  BUILD_FAILURE  =echo  "-- $(RED)Build failed.$(RESET)"; exit 1
  LINK_FAILURE   =echo  "-- $(RED)Linking failed.$(RESET)"; exit 1
  INIT_MAIN      =@echo "-- $(CYAN)Creating main.c$(RESET)"
  INIT_SUCCESS   =@echo "-- $(GREEN)Initialized the project structure$(RESET)"
  RUN            =@echo "-- $(CYAN)Executing$(RESET): $(TARGET_NAME)"
  VALGRIND_RUN   =@echo "-- $(CYAN)Running Valgrind on$(RESET): $(TARGET_NAME)"
  CLEAN          =@echo "-- $(GREEN)Cleaned$(RESET): $(BUILD_DIR)/*"
  MK_INIT_ERROR  =@echo "$(RED)Error: $(SRC_DIR) directory doesn't exist. Please run make
init to initialize the project.$(RESET)" endif

# phony targets
.PHONY: all run valgrind clean test

# Sets flags based on the build mode.
ifeq ($(BUILD_DEFAULT), release)
CFLAGS += $(RELEASE_FLAGS) else
  CFLAGS += $(DEBUG_FLAGS)
endif

# Find all the source files and corresponding objects
SRCS := $(wildcard $(SRC_DIR)/*.c)
OBJS := $(patsubst $(SRC_DIR)/%.c, $(BUILD_DIR)/%.o, $(SRCS))

# The all target, builds shell all:
$(TARGET)
```

```
                                        $(Q) echo "$(BUILD_DEFAULT)" >
$(BUILD_DIR)/build_mode
```

```
# The TARGET target depends on the generated object files.
$(TARGET): $(OBJS)
	$(TRACE_LD)
	  $(Q) $(CC) $(CFLAGS) -I$(INCLUDE_DIR) $^ -o $@ $(LINKER_FLAGS) || ($(LINK_FAILURE))
	$(BUILD_SUCCESS)

# The object files' targets, depend on their corresponding source files.
$(BUILD_DIR)/%.o: $(SRC_DIR)/%.c
	$(TRACE_CC)
	 $(Q) $(CC) $(CFLAGS) -I$(INCLUDE_DIR) -c $< -o $@ || ($(BUILD_FAILURE))
# Create the build, src and include directories if they don't exist. $(BUILD_DIR)
$(SRC_DIR) $(INCLUDE_DIR):
	$(TRACE_MKDIR)
	$(Q) $(MKDIR) $@

# Initializes the project directories, and creates a main.c file in the src directory. init:
$(BUILD_DIR) $(SRC_DIR) $(INCLUDE_DIR)
	$(INIT_SUCCESS)

# Runs the program in valgrind, for debugging purposes (if needed) valgrind:
$(TARGET)
	$(VALGRIND_RUN)
	$(Q) valgrind $(VALG_FLAGS) $(TARGET)

# Cleans the build directory. clean:
	$(Q) $(RM) $(BUILD_DIR)/*
	$(CLEAN)

ARGS:=
# Runs the test suite test:
$(TARGET)
	$(Q) cd $(TEST_DIR) && python3 test.py $(ARGS)
```

Listing 1: Shell/Makefile

```
/**
*      @file command.h
*      @brief Header file defining structures and functions for handling commands
and command chains.  * @version 0.1
 */

#ifndef COMMAND_H
#define COMMAND_H

// Includes
#include "utils.h"
#include <stdbool.h>
#include <unistd.h>

/**
*      @brief Represents a simple command, which includes the command name,
arguments, file descriptors, and a function pointer to execute the command.
 *
*      A simple command consists of a command name and its associated arguments.
It may also include input, output, and error file descriptors. Commands are
executed in a sequence or pipeline, and I/O redirection is handled by the shell,
not by the command itself.
 */
typedef struct SimpleCommand {
    char* commandName; //< Command name, e.g., "ls"

    char** args;      //< Array of arguments for the command, including the command
name
    int argc;          //< Number of arguments, including the command name

    int inputFD;       //< Input file descriptor (default is 0 for stdin)
int outputFD;      //< Output file descriptor (default is 1 for stdout)
```

```
int stderrFD;       //< Error file descriptor (default is 2 for stderr)
int pid;            //< Process ID of the child process, default is -1
```

```
    char* inputFile;   //< Optional input file for redirection
char* outputFile;  //< Optional output file for redirection
    int noWait;        //< Flag indicating whether to wait for the command to
finish (0: wait, 1: no wait)

    int (*execute)(struct SimpleCommand*); //< Function pointer for executing the
command
} SimpleCommand;

/**
*      @brief Represents a command, which can be a pipeline of multiple simple
commands.
 *
*      A command is essentially a sequence of simple commands connected by pipes.
It also includes information about background execution and the operator used to
chain the command with the next one.
 */
typedef struct Command {
    struct SimpleCommand** simpleCommands;  //< Array of pointers to simple
commands in this command
    int nSimpleCommands;                    //< Number of simple commands in the
array
    bool background;                        //< Flag indicating background
execution (true if in background)

    char* chainingOperator;                 //< Operator used to chain this
command with the next one (e.g., ';', '&')
    struct Command* next;                   //< Pointer to the next command in
the chain } Command;

/**
*      @brief Represents a chain of commands connected by chaining operators.
 *
*      A command chain is a sequence of commands linked together using operators
such as ';' or '&'. It forms a linked list of commands.
 */
typedef struct CommandChain {
    struct Command* head;   //< Pointer to the first command in the chain
struct Command* tail;   //< Pointer to the last command in the chain }
CommandChain;

// Function declarations

// ------------------------ Initializers --------------------------------

/**
*      @brief Creates an empty SimpleCommand structure.
 *
*      Allocates memory for a SimpleCommand and initializes its members to
default values. The caller is responsible for freeing the allocated memory.
 *
*      @return SimpleCommand* Pointer to the newly created SimpleCommand
structure, or NULL on failure
 */
SimpleCommand* initSimpleCommand();

/**
*      @brief Creates an empty Command structure.
 *
*      Allocates memory for a Command and initializes its members to default
values. The caller is responsible for freeing the allocated memory.
 *
*      @return Command* Pointer to the newly created Command structure, or NULL
on failure
 */
```

```
Command* initCommand();
```

```
/**
*      @brief Creates an empty CommandChain structure.
 *
*      Allocates memory for a CommandChain and initializes its members to default
values. The caller is responsible for freeing the allocated memory.
 *
*      @return CommandChain* Pointer to the newly created CommandChain structure,
or
NULL on failure
 */
CommandChain* initCommandChain();

// ------------------------ Pushers --------------------------------

/**
*      @brief Adds an argument to the arguments array of a SimpleCommand.
 *
*      If the command name is not set, it will be set to the provided argument.
The function increases the size of the arguments array and updates the argument
count.
 *
*      @param arg Argument to be added
*      @param simpleCommand Pointer to the SimpleCommand structure to which the
argument will be added
*      @return int Status code (0 for success, -1 for failure)
 */
int pushArgs(char* arg, SimpleCommand* simpleCommand);
/**
*      @brief Adds a Command to a CommandChain.
 *
*      Appends the provided command to the end of the chain and updates the tail
pointer.
 *
*      @param chain Pointer to the CommandChain to which the command will be
added
*      @param command Pointer to the Command to be added
*      @return int Status code (0 for success, -1 for failure)
 */
int addCommandToChain(CommandChain* chain, Command* command);

/**
*      @brief Adds a SimpleCommand to a Command.
 *
*      Increases the size of the array holding simple commands in the command and
appends the provided SimpleCommand. Uses realloc to adjust the array size.
 *
*      @param command Pointer to the Command to which the SimpleCommand will be
added
*      @param simpleCommand Pointer to the SimpleCommand to be added
*      @return int Status code (0 for success, -1 for failure)
 */
int addSimpleCommand(Command* command, SimpleCommand* simpleCommand);
// ------------------------ Cleaners --------------------------------

/**
*      @brief Frees the memory allocated for a SimpleCommand.
 *
*      Deallocates memory for the command name, arguments array, and any
associated files. Sets the pointer to NULL.
 *
*      @param simpleCommand Pointer to the SimpleCommand structure to be cleaned
up
```

```
 */
void cleanUpSimpleCommand(SimpleCommand* simpleCommand);

/**
*      @brief Frees the memory allocated for a Command.
 *
```

```
*      Deallocates memory for the array of SimpleCommand pointers and the
chaining operator string. Sets the pointer to NULL.
 *
*      @param command Pointer to the Command structure to be cleaned up
 */
void cleanUpCommand(Command* command);

/**
*      @brief Frees the memory allocated for a CommandChain.
 *
*      Deallocates memory for all commands in the chain and updates the head and
tail pointers to NULL.
 *
*      @param chain Pointer to the CommandChain structure to be cleaned up
 */
void cleanUpCommandChain(CommandChain* chain);

// ------------------------- Execute --------------------------------

/**
*      @brief Executes a chain of commands.
 *
*      Traverses the command chain and executes each command in sequence. Handles
chaining operators to determine execution order. Returns the exit status of the
last command.
 *
*      @param chain Pointer to the CommandChain to be executed
*      @return int Status code (exit status of the last command)
 */
int executeCommandChain(CommandChain* chain);

/**
*      @brief Executes a Command.
 *
*      Traverses and executes each SimpleCommand in the Command. Manages the
execution sequence and returns the exit status of the last executed command.
 *
*      @param command Pointer to the Command to be executed
*      @return int Status code (exit status of the last command)
 */
int executeCommand(Command* command);

// ------------------------- Debug --------------------------------

/**
*      @brief Prints the details of a CommandChain in a human-readable format.
 *
*      Useful for debugging to visualize the structure of the command chain.
 *
*      @param chain Pointer to the CommandChain to be printed
 */
void printCommandChain(CommandChain* chain);

/**
*      @brief Prints the details of a SimpleCommand in a human-readable format.
 *
*      Useful for debugging to visualize the details of the simple command.
 *
*      @param simpleCommand Pointer to the SimpleCommand to be printed
 */
void printSimpleCommand(SimpleCommand* simpleCommand);
#endif // COMMAND_H
```

Listing 2: Shell/include/command.h

```
/**
* @file log.h
```

```
*      @brief Provides macros and functions for logging messages with different
severity levels and optional annotations.
*      @version 0.1
 */

#ifndef LOG_H
#define LOG_H

#include <stdio.h>

// ANSI escape codes for colored log output
#define LOG_RESET   "\033[0m"   /**< Reset color to default */
#define LOG_RED     "\033[1;31m" /**< Red color for error messages */
#define LOG_GREEN   "\033[1;32m" /**< Green color for success messages */
#define LOG_YELLOW  "\033[1;33m" /**< Yellow color for warnings */
#define LOG_BLUE    "\033[1;34m" /**< Blue color for informational messages */
#define LOG_CYAN    "\033[1;36m" /**< Cyan color for debug messages */
#define LOG_WHITE   "\033[1;37m" /**< White color for general messages */
// Log levels and their respective display modes
#define LOG_ERR     0  /**< Log level for critical errors, always printed */
#define LOG_DBG     1  /**< Log level for debug messages, printed only in debug
mode */
#define LOG_PRI     2  /**< Log level for regular messages, always printed
without special annotations */

// Default colors for different log types
#define LOG_COLOR_ERR   LOG_RED    /**< Default color for error messages */
#define LOG_COLOR_DBG   LOG_CYAN  /**< Default color for debug messages */
#define LOG_COLOR_PRI   LOG_WHITE /**< Default color for regular messages */
// Debug mode configuration
#ifdef DEBUG
#define ANNOTATIONS 1      /**< Set to 1 to enable annotations in debug mode
(file, function, line info) */
#else
#define DEBUG 0
#define ANNOTATIONS 0
#endif

// Configuration for annotations
#define ANNOTATIONS_INFO 1  /**< Set to 1 to include file, function, and line
number annotations */

// Enable/disable specific annotation types
#define ANNOTATIONS_FILE 0 /**< Set to 1 to include file name in annotations */
#define ANNOTATIONS_FUNC 1 /**< Set to 1 to include function name in annotations
*/
#define ANNOTATIONS_LINE 1 /**< Set to 1 to include line number in annotations */
// Default output function for logging
#define LOG_OUT(...) printf(__VA_ARGS__)

// Constructs the annotation string with file, function, and line information
#define ANNOTATION_INFO_STRING do {\
    if (ANNOTATIONS_INFO) {\
LOG_OUT(" ("); \
        if (ANNOTATIONS_FILE) printf("%s", __FILE__); \
if (ANNOTATIONS_FILE && ANNOTATIONS_FUNC) printf(","); \
if (ANNOTATIONS_FUNC) printf("%s", __func__); \         if
(ANNOTATIONS_FUNC && ANNOTATIONS_LINE) printf(","); \
        if (ANNOTATIONS_LINE) printf("%d", __LINE__); \
        LOG_OUT(") ");\
    }\
} while (0)
```

```
// Macro for logging messages with optional annotations and color
#define LOG(type, prefix, color, ...) \
```

```
    do { \
        if (DEBUG) {\
            LOG_OUT("%s%s%s: ", color, prefix, LOG_RESET); \
            ANNOTATION_INFO_STRING; \
if (type == LOG_DBG) { \
LOG_OUT(__VA_ARGS__); \
break; \
            }
\         }\
        if (type == LOG_ERR || type == LOG_PRI) { LOG_OUT(__VA_ARGS__); } \     }
while (0)
#endif // LOG_H
```

Listing 3: Shell/include/log.h

```
/**
*      @file parser.h
*      @brief Contains macros and the definition for the main parser function
that processes command tokens into a command chain.
*      @version 0.1
 *
*      @copyright Copyright (c) 2023
 *
 */

#ifndef PARSER_H
#define PARSER_H

#include "command.h"

// Useful macros for readability

/**
*      @brief Checks if the given token is a chaining operator.
 *
*      Chaining operators are used to separate commands in a command chain, such
as `&` and `;`. This macro returns 1 if the token matches one of these
operators, otherwise 0.
 *
*      @param token The token to check
*      @return int 1 if the token is a chaining operator, 0 otherwise  */
#define IS_CHAINING_OPERATOR(token) (strcmp(token, "&") == 0 || strcmp(token,
";") == 0)

/**
*      @brief Checks if the given token is a background operator.
 *
*      The background operator is `&`, which indicates that the command should
be executed in the background. This macro returns 1 if the token matches `&`,
otherwise 0.
 *
*      @param token The token to check
*      @return int 1 if the token is a background operator, 0 otherwise  */
#define IS_BACKGROUND(token) (token && strcmp(token, "&") == 0)

/**
*      @brief Checks if the given token is a pipe operator.
 *
*      The pipe operator `|` is used to pass the output of one command as input
to the next command. This macro returns 1 if the token matches `|`, otherwise 0.
 *
*      @param token The token to check
*      @return int 1 if the token is a pipe, 0 otherwise
 */
#define IS_PIPE(token) (strcmp(token, "|") == 0)
```

```
/**
*      @brief Checks if the given token is a file output redirection operator.
 *
*      File output redirection operators are `>` (overwrite) and `>>` (append).
This macro returns 1 if the token matches one of these operators, otherwise 0.
 *
*      @param token The token to check
*      @return int 1 if the token is a file output redirection operator, 0
otherwise
 */
#define IS_FILE_OUT_REDIR(token) (strcmp(token, ">") == 0 || strcmp(token, ">>")
== 0)

/**
*      @brief Checks if the given token is a file input redirection operator.
 *
*      The file input redirection operator is `<`. This macro returns 1 if the
token matches `<`, otherwise 0.
 *
*      @param token The token to check
*      @return int 1 if the token is a file input redirection operator, 0
otherwise
 */
#define IS_FILE_IN_REDIR(token) (strcmp(token, "<") == 0)

/**
*      @brief Checks if the given token is a standard error redirection operator.
 *
*      The standard error redirection operator is `2>`. This macro returns 1 if
the token matches `2>`, otherwise 0.
 *
*      @param token The token to check
*      @return int 1 if the token is a standard error redirection operator, 0
otherwise
 */
#define IS_STDERR_REDIR(token) (strcmp(token, "2>") == 0)

/**
*      @brief Checks if the given token is NULL.
 *
*      This macro returns 1 if the token is NULL, which typically indicates an
invalid or uninitialized token.
 *
*      @param token The token to check
*      @return int 1 if the token is NULL, 0 otherwise
 */
#define IS_NULL(token) (!token)

/**
*      @brief Checks if the given token is ignorable.
 *
*      Ignorable tokens include spaces, tabs, newlines, and empty strings. This
macro returns 1 if the token is one of these ignorable values, otherwise 0.
 *
*      @param token The token to check
*      @return int 1 if the token is ignorable, 0 otherwise
 */
#define IGNORE(token) (token && (strcmp(token, " ") == 0 || strcmp(token, "\t")
== 0 || strcmp(token, "\n") == 0 || strcmp(token, "") == 0))

/**
*      @brief Checks if the given token is an append operator.
 *
```

```
*       The append operator is `>>`, used for appending output to a file. This
macro returns 1 if the token matches `>>`, otherwise 0.
 *
*       @param token The token to check
*       @return int 1 if the token is an append operator, 0 otherwise  */
#define IS_APPEND(token) (strcmp(token, ">>") == 0)
```

```
/**
* @brief Parses an array of tokens into a command chain.
 *
* The `parseTokens` function takes an array of tokens (strings) and processes
  them into a structured command chain. The tokens array is assumed to be
  nullterminated. It is the caller's responsibility to free the allocated memory
  for the returned CommandChain.
 *
* @param tokens The array of tokens to parse. Should be null-terminated.  *
  @return CommandChain* Pointer to the constructed CommandChain. Returns NULL if
  parsing fails.
 */
CommandChain* parseTokens(char** tokens);

#endif // PARSER_H
```

Listing 4: Shell/include/parser.h

```
/**
*      @file builtins.h
*      @brief Contains structures and utilities for managing built-in commands
and the internal state of the shell.
*      @version 0.1
 *
*      This file includes definitions for shell history management, shell state
representation, and built-in command functions.
 *
 */

#ifndef BUILTINS_H
#define BUILTINS_H

#include "command.h"

// Maximum path length for file operations
#define MAX_PATH_LENGTH 1024

// Environment variable for home directory
#define HOME_DIR getenv("HOME")

// Structure to represent a single command in the shell history
typedef struct HistoryNode {
    char* command;                /**< The command string */
    struct HistoryNode* next;     /**< Pointer to the next node in the history
list */ } HistoryNode;

// Structure to represent a list of commands in shell history
typedef struct HistoryList {
    HistoryNode* head;             /**< Pointer to the first node in the history
list */
    HistoryNode* tail;            /**< Pointer to the last node in the history
list for quick insertions */
    size_t size;                  /**< The number of commands in the history list
*/
} HistoryList;

/**
*      @brief Adds a command to the history list.
 *
*      This function creates a new HistoryNode and appends it to the history
list. The command string is copied to the new node.
 *
*      @param list The history list to which the command will be added.
*      @param command The command string to add.
*      @return int Returns 0 on success, -1 on failure.
 */
int add_to_history(HistoryList* list, char* command);
```

```
/**
*      @brief Retrieves a command at a particular index from the history list.
 *
*      This function returns the command string at the specified index in the
history list.
 *
*      @param list The history list from which to retrieve the command.
*      @param index The index of the command to retrieve.
*      @return char* The command string at the specified index, or NULL if the
index is out of range.
 */
char* get_command(HistoryList* list, unsigned int index);

/**
*      @brief Cleans up and frees memory used by the history list.
 *
*      This function frees all nodes in the history list and the memory
associated with them.
 *
*      @param list The history list to be cleaned up.
 */
void clean_history(HistoryList* list);

/**
*      @brief Finds the last command in the history that starts with the given
prefix.
 *
*      This function searches through the history list and returns the most
recent command that starts with the specified prefix.
 *
*      @param list The history list to search.
*      @param prefix The prefix to search for.
*      @return char* The last command that starts with the prefix, or NULL if no
match is found.
 */
char* find_last_command_with_prefix(HistoryList* list, const char* prefix);
// Structure to represent the state of the shell
typedef struct ShellState {

    // File descriptors for the original standard input, output, and
error     int originalStdoutFD;  /**< File descriptor for original stdout
*/     int originalStdinFD;   /**< File descriptor for original stdin */
int originalStderrFD;  /**< File descriptor for original stderr */
    // Buffer to hold the current prompt string
    char prompt_buffer[MAX_STRING_LENGTH]; /**< Current shell prompt string */
    // History list to store commands entered by the user
    HistoryList history;  /**< The shell command history list */
} ShellState;

/**
*      @brief Initializes the shell state.
 *
*      This function allocates and initializes a ShellState structure, setting up
the original file descriptors and prompt buffer.
 *
*      @return ShellState* Pointer to the initialized ShellState structure, or
NULL on failure.
 */
ShellState* init_shell_state();

/**
*      @brief Cleans up and frees memory used by the shell state.  *
```

```
*      This function frees all resources associated with the ShellState
structure, including the history list.
 *
*      @param stateObj The ShellState structure to be cleaned up.  * @return int
Returns 0 on success, -1 on failure.
 */
int clear_shell_state(ShellState* stateObj);

/**  * @brief Type definition for a function that executes a simple
command.
 *
*      This type is used for function pointers that refer to built-in command
execution functions.
 *
*      @param command The SimpleCommand structure representing the command to
execute.
*      @return int Returns 0 on success, or a non-zero status on failure.
 */
typedef int (*ExecutionFunction)(SimpleCommand*);

/**
*      @brief Returns the execution function for a given built-in command.
 *
*      This function returns a pointer to the function that handles the
execution of the specified command.
 *
*      @param commandName The name of the command.
*      @return ExecutionFunction The function pointer for executing the command.
 */
ExecutionFunction getExecutionFunction(char* commandName);

/**
*      @brief Built-in function to change the current directory.
 *
*      This function changes the current working directory to the path specified
in the command.
 *
*      @param command The command to be executed, which should include the
target directory.
*      @return int Returns 0 on success, -1 on failure.
 */
int cd(SimpleCommand* command);

/**
*      @brief Built-in function to exit the shell.
 *
*      This function terminates the shell process.
*       * @param command The command to be executed, which should be the exit
command.  * @return int Returns 0 on success, -1 on failure.
 */
int exitShell(SimpleCommand* command);

/**
*      @brief Built-in function to print the current working directory.
 *
*      This function outputs the current working directory to the standard
output.
 *
*      @param command The command to be executed, which should be the pwd
command.  * @return int Returns 0 on success, -1 on failure.
 */
int pwd(SimpleCommand* command);

/**
```

```
*       @brief Built-in function to print the command history.
 *
*       This function outputs the command history stored in the shell's history
list.  *
```

```
* @param command The command to be executed, which should be the history command.
* @return int Returns 0 on success, -1 on failure.
 */
int history(SimpleCommand* command);

/**
* @brief Executes a process by forking and executing the command.
 *
* This function creates a child process, executes the command in the child
  process, and returns the execution status.
 *
* @param command The command to be executed.
* @return int Returns 0 on success, non-zero on failure.
 */
int executeProcess(SimpleCommand* command);

#endif // BUILTINS_H
```

Listing 5: Shell/include/shell_builtins.h

```
/**
*      @file utils.h
*      @brief Contains useful macros and utility functions for the shell.
*      @version 0.1
 *
*      This header file includes macros for logging, utility functions for
handling strings and file descriptors,
*      and other utility functions useful for shell operations.
 *
 *
 */

#ifndef UTILS_H
#define UTILS_H

#include "log.h"

#include <string.h>
#include <stdlib.h>

// Macros for logging with different levels and colors
#define LOG_ERROR(...) LOG(LOG_ERR, "[ERROR]", LOG_COLOR_ERR, __VA_ARGS__)   /**<
Macro for logging error messages */
#define LOG_DEBUG(...) LOG(LOG_DBG, "[DEBUG]", LOG_COLOR_DBG, __VA_ARGS__)   /**<
Macro for logging debug messages */
#define LOG_PRINT(...) LOG(LOG_PRI, "[PRINT]", LOG_COLOR_PRI, __VA_ARGS__)   /**<
Macro for logging general print messages */

// Defines the maximum length for strings used in the program
#define MAX_STRING_LENGTH 1024   /**< Maximum length of strings (including null
terminator) */

// Macro for safely copying a string up to MAX_STRING_LENGTH
#define COPY(str) (str ? strndup(str, MAX_STRING_LENGTH) : NULL)  /**< Macro
to duplicate a string up to a maximum length, returns NULL if input is NULL */
// File descriptor constants for standard input, output, and error, and for pipe
ends
#define STDIN_FD 0                /**< File descriptor for standard input */
#define STDOUT_FD 1               /**< File descriptor for standard output */
#define STDERR_FD 2               /**< File descriptor for standard error */
#define PIPE_READ_END 0          /**< Pipe end for reading data */
#define PIPE_WRITE_END 1         /**< Pipe end for writing data */

/**
 * @brief Tokenizes a string based on a delimiter.
```

*

```
*      This function splits a string into an array of tokens, using the specified
delimiter. It handles quoted strings properly,
*      ignoring delimiters within quotes. The resulting array is NULL-terminated.
 *
*      @param str The string to tokenize.
*      @param delimiter The character used to delimit tokens.
*      @return char** An array of tokens (NULL terminated). The caller is
responsible for freeing this memory using freeTokens().
 */
char** tokenizeString(const char* str, const char delimiter);

/**
*      @brief Frees the memory allocated for tokens.
 *
*      This function frees the memory used by the array of tokens returned by
tokenizeString().
 *
*      @param tokens The array of tokens to free. This should be a NULL-
terminated array of strings.
 */
void freeTokens(char** tokens);

/**
*      @brief Removes quotes from a string.
*       * This function removes surrounding quotes from a string, if they are
present.
If the string is not quoted, it returns the original string.
*      If the string is quoted, it allocates a new string without the quotes and
returns it. The original string is freed if quotes are removed.
 *
*      @param str The string from which quotes should be removed.
*      @return char* A pointer to the new string without quotes. The caller is
responsible for freeing this memory.
 */
char* removeQuotes(char* str);

/**
*      @brief Gets the number of tokens in an array.
 *
*      This function counts the number of tokens in a NULL-terminated array of
strings.
 *
*      @param tokens The array of tokens (NULL-terminated).  * @return int The
number of tokens in the array.
 */
int getTokenCount(char** tokens);

#endif // UTILS_H
```

Listing 6: Shell/include/utils.h

```
/**
* @file command.c
* @brief Contains the function implementations for handling commands in the
  shell.  * @version 0.1
 *
* This file defines functions for initializing, managing, executing, and cleaning
  up commands and command chains.
 *
 */

#include "command.h"

// Macro to check if the previous command is chained with a specific operator
#define CHAINED_WITH(opr) (prevCommand ? (prevCommand->chainingOperator ?
```

```
(strcmp(prevCommand->chainingOperator, opr) == 0) : 0) : 0)
```

```
/*-------------------------------------------------------------------------
--
---------*/

/*-----------------------------Initializers--------------------------------
--
--------*/

// Initializes a SimpleCommand structure with default values
SimpleCommand* initSimpleCommand()
{
    SimpleCommand* simpleCommand = (SimpleCommand*)
malloc(sizeof(SimpleCommand));

    if (!simpleCommand)
        return NULL;  // Return NULL if memory allocation fails

    // Set default values for the SimpleCommand fields
simpleCommand->commandName = NULL;
simpleCommand->args        = NULL;
simpleCommand->argc        = 0;
simpleCommand->inputFD     = STDIN_FD;
simpleCommand->outputFD    = STDOUT_FD;
simpleCommand->stderrFD    = STDERR_FD;
simpleCommand->noWait      = 0;
simpleCommand->execute     = NULL;
simpleCommand->pid         = -1;

    return simpleCommand;
}

// Initializes a Command structure with default values
Command* initCommand()
{
    Command* command = (Command*)malloc(sizeof(Command));

    if (!command)
        return NULL;  // Return NULL if memory allocation fails

    // Set default values for the Command fields
command->simpleCommands   = NULL;
command->nSimpleCommands  = 0;
command->background       = false;
command->chainingOperator = NULL;
command->next             = NULL;

    return command;
}

// Initializes a CommandChain structure with default values
CommandChain* initCommandChain()
{
    CommandChain* chain = (CommandChain*)malloc(sizeof(CommandChain));
     if
(!chain)
        return NULL;  // Return NULL if memory allocation fails

    // Set default values for the CommandChain
fields     chain->head = NULL;     chain->tail =
NULL;

    return chain;
}
```

```
/*------------------------------Setters (push functions)----------------------
--
--------*/

// Adds a Command to the end of the CommandChain
int addCommandToChain(CommandChain* chain, Command* command)
{
    if (!chain)
    {
        LOG_DEBUG("Invalid command chain passed\n");
return -1;  // Return error code if chain is NULL
    }

    // If the chain is empty, set both head and tail to the new command
if (!chain->head)
    {
        chain->head = command;
chain->tail = command;
    }
    // Otherwise, append the command to the end of the
chain     else     {
        chain->tail->next = command;
chain->tail = command;
    }
    return 0;  // Return success code
}

// Adds a SimpleCommand to the current Command's array of SimpleCommands
int addSimpleCommand(Command* command, SimpleCommand* simpleCommand)
{
    if (!command)
    {
        LOG_DEBUG("Invalid command passed. It's NULL\n");
return -1;  // Return error code if command is NULL
    }
    if (!simpleCommand)
    {
        LOG_DEBUG("Invalid simpleCommand passed. It's NULL\n");
return -1;  // Return error code if simpleCommand is NULL
    }

    // Reallocate memory to accommodate the new SimpleCommand
    SimpleCommand** temp = (SimpleCommand**)realloc(command->simpleCommands,
(command->nSimpleCommands + 1) * sizeof(SimpleCommand*));

    if (!temp)
    {
        LOG_DEBUG("Realloc error. Failed to reallocate memory for the
array.\n");         return -1;  // Return error code if reallocation fails
    }
    command->simpleCommands = temp;
temp = NULL;

    // Add the new SimpleCommand to the end of the array
command->simpleCommands[command->nSimpleCommands] = simpleCommand;
command->nSimpleCommands++;  // Increment the count of SimpleCommands

    return 0;  // Return success code
}

// Adds an argument to the SimpleCommand's argument array, ensuring it's
NULLterminated
```

```
int pushArgs(char* arg, SimpleCommand* simpleCommand)
{
    if (!simpleCommand)
    {
        LOG_DEBUG("Invalid simpleCommand passed. It's NULL\n");
return -1;  // Return error code if simpleCommand is NULL
    }

    // Reallocate memory to accommodate the new argument
    char** temp = (char**)realloc(simpleCommand->args, (simpleCommand->argc +
2)
* sizeof(char*));

    if (!temp)
    {
        LOG_DEBUG("Realloc error. Failed to reallocate memory for the
array.\n");         return -1;  // Return error code if reallocation fails
    }
    simpleCommand->args = temp;
temp = NULL;

    // Add the new argument to the end of the array and ensure the array is
NULLterminated
    simpleCommand->args[simpleCommand->argc] = COPY(arg);
simpleCommand->args[simpleCommand->argc + 1] = NULL;
simpleCommand->argc++;

    // Set the command name if this is the first argument
if (simpleCommand->argc == 1)
    {
        simpleCommand->commandName = COPY(arg);
    }
    return 0;  // Return success code
}

/*------------------------------Command Execution functions------------------
--
---------*/

// Executes a CommandChain, processing each Command in sequence
int executeCommandChain(CommandChain* chain)
{
    if (!chain)
    {
        LOG_DEBUG("Invalid command chain passed\n");
return -1;  // Return error code if chain is NULL
    }

    Command* command = chain->head;

    // Variable to hold the exit status of the last command
int lastStatus = 0;

    // Process each Command in the
chain     while (command)     {
        lastStatus = executeCommand(command);  // Execute the current command

        command = command->next;  // Move to the next command in the chain
    }
    return lastStatus;  // Return the exit status of the last command
}
```

```
// Executes a single Command, including I/O redirections
int executeCommand(Command* command)
{
    if (!command)
    {
        LOG_DEBUG("Invalid command passed\n");
        return -1;  // Return error code if command is NULL
    }

    // Check if the command is empty and return an error if so
if (!command->simpleCommands || command->nSimpleCommands == 0)
    {
        LOG_DEBUG("Invalid command. It's empty\n");
        return -1;  // Return error code if command is empty
    }
    for (int i = 0; i < command->nSimpleCommands; i++)
    {
        LOG_DEBUG("Executing command : %s\n", command->simpleCommands[i]-
>commandName);
        SimpleCommand* simpleCommand = command->simpleCommands[i];

        // If the command is to be run in the background, set noWait
flag         if (command->background)             simpleCommand->noWait
= 1;

        // Check if the command name is empty and return an error if so
if (!simpleCommand->commandName)
        {
            LOG_DEBUG("Invalid command name. It's empty\n");
return -1;  // Return error code if command name is empty
        }

        // Execute the SimpleCommand and get the status
int status = simpleCommand->execute(simpleCommand);
        LOG_DEBUG("Command executing with pid: %d\n", simpleCommand->pid);

        // If the command execution failed, return the
status         if (status)         {
            return status;
        }

        // Close file descriptors if they were redirected
if (simpleCommand->inputFD != STDIN_FD)
close(simpleCommand->inputFD);

        if (simpleCommand->outputFD != STDOUT_FD)
close(simpleCommand->outputFD);
    }
    return 0;  // Return success code
}

/*------------------------------Clean up functions----------------------------
--
---------*/

// Frees memory allocated for a SimpleCommand void
cleanUpSimpleCommand(SimpleCommand* simpleCommand)
{
    if (!simpleCommand)
        return;  // Return if the SimpleCommand is NULL

    LOG_DEBUG("Cleaning up simple command: %s\n", simpleCommand->commandName);
```

```
    // Free the commandName if it was allocated
if (simpleCommand->commandName)
    {
        free(simpleCommand->commandName);
simpleCommand->commandName = NULL;
    }

    // Free each argument in the args array
if (simpleCommand->args)
    {
        for (int i = 0; i < simpleCommand->argc; i++)
        {
            if (simpleCommand->args[i])
            {
                free(simpleCommand->args[i]);
simpleCommand->args[i] = NULL;             }
        }

        // Free the args array itself
free(simpleCommand->args);
simpleCommand->args = NULL;
    }

    // Free the SimpleCommand
structure     free(simpleCommand);
simpleCommand = NULL;
}

// Frees memory allocated for a Command
void cleanUpCommand(Command* command)
{
    if (!command)
        return;  // Return if the Command is NULL

    // Free each SimpleCommand in the Command
    if (command->simpleCommands)
    {
        for (int i = 0; i < command->nSimpleCommands; i++)
        {
            cleanUpSimpleCommand(command->simpleCommands[i]);
        }
    }

    // Free the array of SimpleCommands
free(command->simpleCommands);

    // Free the chainingOperator if it was allocated
if (command->chainingOperator)
    {
        free(command->chainingOperator);
command->chainingOperator = NULL;
    }

    // No need to free next command, as it will be handled by chain cleanup
}

// Frees memory allocated for the CommandChain and its commands
void cleanUpCommandChain(CommandChain* chain)
{     if
(!chain)
        return;  // Return if the CommandChain is NULL
```

```
    // Free each Command in the CommandChain
if (chain->head)
    {
        Command* command = chain->head;
while (command)
        {
            Command* nextCommand =
command->next;
cleanUpCommand(command);
free(command);              command =
nextCommand;
        }
    }

    // Free the CommandChain
structure     free(chain);     chain
= NULL;
}

/*--------------------------------Utility functions------------------------------
-
---------*/
```

```
// Prints the details of a CommandChain
void printCommandChain(CommandChain* chain)
{
    LOG_DEBUG("Printing command chain\n");
if (!chain)
        return;  // Return if the CommandChain is NULL
    Command* command =
chain->head;     int counter = 1;
while (command)
    {
        LOG_DEBUG("[Link %d]\n", counter);
        for (int i = 0; i < command->nSimpleCommands; i++)
        {
            printSimpleCommand(command->simpleCommands[i]);
        }
counter++;
        command = command->next;
    }
}

// Prints the details of a SimpleCommand
void printSimpleCommand(SimpleCommand* simpleCommand)
{
    if (!simpleCommand)
        return;  // Return if the SimpleCommand is NULL
    LOG_DEBUG("-- name: %s\n", simpleCommand->commandName);
LOG_DEBUG("-- args:\n");
    for (int i = 0; i < simpleCommand->argc; i++)
    {
        LOG_DEBUG("-- -- %s \n", simpleCommand->args[i]);
    }

    LOG_DEBUG("-- Input FD: %d\n", simpleCommand->inputFD);
    LOG_DEBUG("-- Output FD: %d\n", simpleCommand->outputFD);
    LOG_DEBUG("--------------------\n");
}

/*-------------------------------------------------------------------------
---------*/
```

Listing 7: Shell/src/command.c

```
#include "utils.h"
#include "command.h"
#include "parser.h"
#include "shell_builtins.h"

#include <errno.h>
#include <signal.h>
#include <fcntl.h>
#include <sys/wait.h>

// Global variables
int lastExitStatus = 0;  ///< Stores the exit status of the last executed command

ShellState* globalShellState; ///< Holds the state of the shell, including
history and prompt settings

FILE* scriptFile; ///< File pointer for reading script files in non-
interactive mode
/**
*      @brief Reads input from either the terminal or a script file.
 *
*      @param interactive Indicates if the shell is running in interactive mode.
```

```
*      @return char* Pointer to the input string or NULL on failure or end-of-
file.  */
```

```
char* getInput(int interactive)
{
    char* input = malloc(MAX_STRING_LENGTH);  ///< Allocate memory for input
buffer
    if (input == NULL) {
        LOG_ERROR("Memory allocation failed");
        exit(EXIT_FAILURE);  ///< Exit if memory allocation fails
    }
    if (interactive)
    {
        int again = 1;
        char *linept;  ///< Pointer to the line buffer

        while (again)
{             again = 0;
            printf("%s ", globalShellState->prompt_buffer);  ///< Print prompt
linept = fgets(input, MAX_STRING_LENGTH, stdin);  ///< Read input from stdin
            if (linept == NULL)
{                 if (feof(stdin))
{                     free(input);
                  return NULL;  ///< End of file (Ctrl-D)
                } else if (errno == EINTR) {
                  again = 1;  ///< Signal interruption, read again
                } else {
                  LOG_ERROR("Error reading input: %s\n", strerror(errno));
free(input);
                  exit(EXIT_FAILURE);  ///< Exit on read error
                }
            }
        }
```

```
        // Remove the trailing newline character from
input          size_t ln = strlen(input);          if (ln >
0 && input[ln - 1] == '\n') {              input[ln - 1] =
'\0';
```

```
        }
}     else
{
        size_t len = 0;
        ssize_t read = getline(&input, &len, scriptFile);  ///< Read input
from script file         if (read == -1) {             free(input);
            return NULL;  ///< End of file (script ended)
        }

        // Remove trailing newline character
if (read > 0 && input[read - 1] == '\n')
{             input[read - 1] = '\0';
        }
    }
    return input;
}

/**
* @brief Handles SIGINT signal (Ctrl-C).
 *
* @param signo Signal number.
 */
void sigint_handler(int signo) {
    LOG_DEBUG("\nCTRL-C pressed. signo: %d\n", signo);  ///< Log SIGINT signal
}

/**
```

```
* @brief Handles SIGTSTP signal (Ctrl-Z).
```

```
 *
* @param signo Signal number.
 */
void sigtstp_handler(int signo) {
    LOG_DEBUG("\nCTRL-Z pressed. signo: %d\n", signo);  ///< Log SIGTSTP signal }

/**
* @brief Handles SIGQUIT signal (Ctrl-\).
 *
* @param signo Signal number.
 */
void sigquit_handler(int signo) {
    LOG_DEBUG("\nCTRL-\\ pressed. signo: %d\n", signo);  ///< Log SIGQUIT signal }

/**
* @brief Handles SIGCHLD signal, reaping child processes.
 *
* @param signo Signal number.
 */
void sigchld_handler(int signo)
{     (void) signo;  ///< Unused
parameter
    int more = 1;  ///< Flag to check if more zombies need to be reaped
pid_t pid;  ///< PID of the zombie process
    int status;  ///< Termination status of the zombie process
    // Reap all zombie processes
while (more) {
        pid = waitpid(-1, &status, WNOHANG);   ///< Non-blocking wait for child
processes
        if (pid <= 0) {
            more = 0;  ///< No more zombies or error
        }
    }
}

/**
* @brief Main function of the shell.
 *
* @param argc Argument count
* @param argv Argument vector
* @return int Exit status of the shell
 */
int main(int argc, char** argv)
{
    // Default to interactive mode
int interactive = 1;
scriptFile = NULL;

    // Check for correct number of arguments
if (argc > 2)
    {
        LOG_ERROR("Usage: %s [script]\n", argv[0]);
exit(1);  ///< Exit if arguments are incorrect     }

    // If a script is provided, open it and set non-interactive mode
if (argc == 2)
    {
        interactive = 0;
        LOG_DEBUG("Running script %s\n",
argv[1]);         scriptFile = fopen(argv[1],
"r");         if (!scriptFile)
        {
            LOG_ERROR("Error opening script %s: %s\n", argv[1], strerror(errno));
exit(1);  ///< Exit if script file cannot be opened
```

```
        }
    }

    // Initialize global shell state
globalShellState = init_shell_state();

    LOG_DEBUG("Starting shell\n");

    char delimiter = ' ';  ///< Tokenization delimiter

    // Set up signal handlers
    if (signal(SIGINT, sigint_handler) == SIG_ERR)
{         LOG_ERROR("Unable to register SIGINT handler");
        exit(EXIT_FAILURE);  ///< Exit if SIGINT handler cannot be set
    }
    if (signal(SIGTSTP, sigtstp_handler) == SIG_ERR)
{         LOG_ERROR("Unable to register SIGTSTP handler");
        exit(EXIT_FAILURE);  ///< Exit if SIGTSTP handler cannot be set
    }
    if (signal(SIGQUIT, sigquit_handler) == SIG_ERR)
{         LOG_ERROR("Unable to register SIGQUIT handler");
        exit(EXIT_FAILURE);  ///< Exit if SIGQUIT handler cannot be set
    }
    if (signal(SIGCHLD, sigchld_handler) == SIG_ERR)
{         LOG_ERROR("Unable to register SIGCHLD handler");
        exit(EXIT_FAILURE);  ///< Exit if SIGCHLD handler cannot be set
    }
    while (1)
{
        char* input = getInput(interactive);

        // Handle end-of-file (Ctrl-D) or errors
if (input == NULL)
        {
            if (feof(stdin)) {
                printf("\nEOF detected. Exiting shell.\n");
            } else {
                LOG_ERROR("Error reading input: %s\n", strerror(errno));
            }
            break;  ///< Exit the shell loop
        }

        // Skip empty input
if (strcmp(input, "") == 0)
```

```
{             free(inpu
t);
continue;
        }

        // Exit the shell if "exit" command is entered
if (strcmp(input, "exit") == 0)

{             free(inpu
t);
            break;  ///< Exit the shell loop
        }

        // Add input to command history
        add_to_history(&globalShellState->history, input);

        // Tokenize the input string
        char** tokens = tokenizeString(input, delimiter);

        // Log each token for debugging
for (int i = 0; tokens[i] != NULL; i++) {
            LOG_DEBUG("Token %d: [%s]\n", i, tokens[i]);
        }

        // Parse tokens into a CommandChain
        CommandChain* commandChain = parseTokens(tokens);
        // Display the command chain for debugging
        printCommandChain(commandChain);

        // Execute the command chain and get the exit status
int status = executeCommandChain(commandChain);
        LOG_DEBUG("Command executed with status %d\n", status);
        // Free memory allocated for tokens
freeTokens(tokens);

        // Free memory allocated for command chain
cleanUpCommandChain(commandChain);

        // Free memory allocated for input buffer
free(input);
    }

    // Clean up command history
    clean_history(&globalShellState->history);

    return 0;  ///< Return success
status }
```

Listing 8: Shell/src/main.c

```
#include "parser.h"
#include "shell_builtins.h"

#include <fcntl.h>
#include <glob.h>

#define COMPARE_TOKEN(token, string) (token && strcmp(token, string) == 0)

/**
*      @brief Parses an array of tokens and generates a command chain.
 *
*      The function processes tokens to build a command chain. Each command in
the chain may consist of multiple simple commands.
*      It handles various operators like pipes, redirections, and chaining
operators.
 *
*      @param tokens Array of tokens to parse.
*      @return CommandChain* Pointer to the generated command chain or NULL on
failure.
 */
CommandChain* parseTokens(char** tokens)
{
    // Initialize a new command chain
    CommandChain* chain = initCommandChain();
if (!chain)
    {
        LOG_DEBUG("Failed to allocate memory for command chain\n");
return NULL; // Memory allocation failed
    }
    int currentIndexInTokens = 0; // Index to traverse the tokens array

    while (tokens[currentIndexInTokens] != NULL)
    {
        // Initialize a new command
Command* command = initCommand();
if (!command)
        {
            LOG_DEBUG("Failed to allocate memory for command\n");
            cleanUpCommandChain(chain);
            return NULL; // Memory allocation failed
        }

        // Initialize a new simple command
        SimpleCommand* simpleCommand = initSimpleCommand();
if (!simpleCommand)
        {
            LOG_DEBUG("Failed to allocate memory for simple command\n");
            cleanUpCommandChain(chain);
cleanUpCommand(command);
            return NULL; // Memory allocation failed
        }

        // Process tokens until we encounter a chaining operator or end of
tokens         for (; !IS_NULL(tokens[currentIndexInTokens]) &&
!IS_CHAINING_OPERATOR(tokens[currentIndexInTokens]); currentIndexInTokens++)
        {
            if (IS_NULL(tokens[currentIndexInTokens]))
            {
                // Push the simpleCommand to the command's simple commands
if (!simpleCommand->commandName)
                {
                    LOG_DEBUG("Parse error. Null command
encountered\n");                     cleanUpCommandChain(chain);
cleanUpCommand(command);
```

```
cleanUpSimpleCommand(simpleCommand);                    return NULL;
// No command name found
                }
                simpleCommand->execute = getExecutionFunction(simpleCommand-
>commandName);
                addSimpleCommand(command, simpleCommand);
simpleCommand = NULL; // No more simple commands
break;             }
            else if (IS_PIPE(tokens[currentIndexInTokens]))
            {
                // Handle pipe operator
if (!simpleCommand->commandName)
                {
                    LOG_DEBUG("Parse error near \'%s\'\n",
tokens[currentIndexInTokens]);
                    cleanUpCommandChain(chain);
cleanUpCommand(command);
cleanUpSimpleCommand(simpleCommand);
                    return NULL; // Grammar error: no command before pipe
                }

                if (simpleCommand->outputFD != STDOUT_FD)
                {
                    LOG_DEBUG("Parse error. Cannot pipe to multiple
commands\n");                     cleanUpCommandChain(chain);
cleanUpCommand(command);
cleanUpSimpleCommand(simpleCommand);
                    return NULL; // Error: multiple output redirections
                }

                int pipeFD[2];
if (pipe(pipeFD) == -1)
                {
                    LOG_DEBUG("Failed to create pipe\n");
cleanUpCommandChain(chain);
cleanUpCommand(command);
cleanUpSimpleCommand(simpleCommand);
return NULL; // Pipe creation failed
                }

                simpleCommand->outputFD = pipeFD[PIPE_WRITE_END];
```

```
                simpleCommand->execute = getExecutionFunction(simpleCommand-
>commandName);
                addSimpleCommand(command, simpleCommand);

                // Start a new simple command for the next
segment                 simpleCommand = initSimpleCommand();
if (!simpleCommand)
                {
                    LOG_DEBUG("Failed to allocate memory for simple
command\n");                     cleanUpCommandChain(chain);
cleanUpCommand(command);
                    return NULL; // Memory allocation failed
                }

                // Set the new simple command's inputFD to connect via pipe
simpleCommand->inputFD = pipeFD[PIPE_READ_END];
            }
            else if (IS_FILE_OUT_REDIR(tokens[currentIndexInTokens]))
            {
                // Handle output redirection
if (!simpleCommand->commandName)
                {
                    LOG_DEBUG("Parse error. Output redirection encountered
before command\n");
                    cleanUpCommandChain(chain);
cleanUpCommand(command);
cleanUpSimpleCommand(simpleCommand);
                    return NULL; // Output redirection without command
                }

                if (simpleCommand->outputFD != STDOUT_FD)
                {
                    LOG_DEBUG("Cannot redirect output to multiple
files\n");                     cleanUpCommandChain(chain);
cleanUpCommand(command);
cleanUpSimpleCommand(simpleCommand);                    return NULL; //
Multiple output redirections
                }

                int fileFD = -1;
int isAppend = 0;
                if (IS_APPEND(tokens[currentIndexInTokens]))
isAppend = 1;

                char* fileNameToken = NULL;
do {
                    fileNameToken = tokens[++currentIndexInTokens];
                } while (IGNORE(fileNameToken));

                if (isAppend)
                    fileFD = open(fileNameToken, O_WRONLY | O_CREAT | O_APPEND,
0644);
else
                    fileFD = open(fileNameToken, O_WRONLY | O_CREAT | O_TRUNC,
0644);
                if (fileFD == -1)
                {
                    LOG_DEBUG("Failed to open file for output
redirection\n");                     cleanUpCommandChain(chain);
cleanUpCommand(command);
cleanUpSimpleCommand(simpleCommand);                    return NULL; //
File open failed
                }
```

```
                simpleCommand->outputFD = fileFD;
            }
            else if (IS_FILE_IN_REDIR(tokens[currentIndexInTokens]))
            {
                // Handle input redirection
                if (simpleCommand->inputFD != STDIN_FD)
                {
                    LOG_DEBUG("Cannot redirect input from multiple
files\n");                     cleanUpCommandChain(chain);
cleanUpCommand(command);
cleanUpSimpleCommand(simpleCommand);                     return NULL; //
Multiple input redirections
                }

                char* fileNameToken = NULL;
do {
                    fileNameToken = tokens[++currentIndexInTokens];
                } while (IGNORE(fileNameToken));

                int fileFD = open(fileNameToken, O_RDONLY);
if (fileFD == -1)
                {
                    LOG_DEBUG("Failed to open file for input
redirection\n");                     cleanUpCommandChain(chain);
return NULL; // File open failed
                }

                simpleCommand->inputFD = fileFD;
            }
            else if (IS_STDERR_REDIR(tokens[currentIndexInTokens]))
            {
                // Handle stderr redirection
                if (simpleCommand->stderrFD != STDERR_FD)
                {
                    LOG_DEBUG("Cannot redirect stderr to multiple
files\n");                     cleanUpCommandChain(chain);
cleanUpCommand(command);
cleanUpSimpleCommand(simpleCommand);                     return NULL; //
Multiple stderr redirections
                }
                char* fileNameToken = NULL;
do {
                    fileNameToken = tokens[++currentIndexInTokens];
                } while (IGNORE(fileNameToken));

                int fileFD = open(fileNameToken, O_WRONLY | O_CREAT | O_TRUNC,
0644);
                if (fileFD == -1)
                {
                    LOG_DEBUG("Failed to open file for stderr
redirection\n");                     cleanUpCommandChain(chain);
return NULL; // File open failed
                }

                simpleCommand->stderrFD = fileFD;
            }
            else if (IGNORE(tokens[currentIndexInTokens]))
            {
                // Ignore irrelevant tokens (e.g., empty tokens)
continue;
            }
            else if (!simpleCommand->commandName &&
```

```
tokens[currentIndexInTokens][0] == '!' && strlen(tokens[currentIndexInTokens]) >
1)
            {
                // Handle history expansion (!<number> or !<command>)
if (pushArgs("history", simpleCommand) != 0)
                {
                    LOG_DEBUG("Failed to push argument to simple command\n");
cleanUpCommandChain(chain);                     cleanUpCommand(command);
cleanUpSimpleCommand(simpleCommand);                        return NULL; //
Argument push failed
                }

                if (pushArgs(tokens[currentIndexInTokens] + 1, simpleCommand) !=
0)
                {
                    LOG_DEBUG("Failed to push argument to simple
command\n");                      cleanUpCommandChain(chain);
cleanUpCommand(command);
cleanUpSimpleCommand(simpleCommand);                        return NULL; //
Argument push failed
                }
   }
else
            {
                // Handle normal tokens: remove quotes and expand
wildcards                 tokens[currentIndexInTokens] =
removeQuotes(tokens[currentIndexInTokens]);                 glob_t
globbuf;
                int globReturn = glob(tokens[currentIndexInTokens], GLOB_NOCHECK
| GLOB_TILDE, NULL, &globbuf);

                if (globReturn != 0)
                {
                    LOG_DEBUG("Failed to expand glob\n");
globfree(&globbuf);
cleanUpCommandChain(chain);
cleanUpCommand(command);
cleanUpSimpleCommand(simpleCommand);
return NULL; // Glob expansion failed
```

```
                }

                // Push expanded tokens to the simple command arguments
for (size_t i = 0; i < globbuf.gl_pathc; i++)
                {
                    if (pushArgs(globbuf.gl_pathv[i], simpleCommand) != 0)
                    {
                        LOG_DEBUG("Failed to push argument to simple
command\n");                            globfree(&globbuf);
cleanUpCommandChain(chain);                             cleanUpCommand(command);
cleanUpSimpleCommand(simpleCommand);                            return NULL; //
Argument push failed
                    }
                }

                globfree(&globbuf);
            }
        }

        // Push the last simple command to the command's simple commands
if (simpleCommand && simpleCommand->commandName)
        {
            simpleCommand->execute = getExecutionFunction(simpleCommand-
>commandName);
            addSimpleCommand(command, simpleCommand);
simpleCommand = NULL; // No more simple commands
        }

        // Update the chain operator (e.g., ';', '&&', '||')
command->chainingOperator = COPY(tokens[currentIndexInTokens]);
if (IS_BACKGROUND(command->chainingOperator))
command->background = true;

        // Add the command to the chain
```

```
        addCommandToChain(chain, command);

        // Move to the next token if it's not NULL
if (tokens[currentIndexInTokens])
        {
            currentIndexInTokens++;
        }
    }
    return chain; // Return the constructed command
chain }
```

Listing 9: Shell/src/parser.c

```
/**
* @file builtins.c
* @brief Contains the function definitions for the builtin shell functions.
* @version 0.1
 *
 */

#include "shell_builtins.h"
#include "parser.h"
#include "command.h"

#include <errno.h>
#include <sys/wait.h>
#include <fcntl.h>

// Global variable to store the shell's state
extern ShellState* globalShellState;

/*------------------------------History Management-----------------------------
-
----*/

/**
*      @brief Adds a command to the history list.
 *
*      Allocates memory for a new HistoryNode, sets its command to the provided
command string, and updates the history list's head and tail pointers.
 *
*      @param list The history list to which the command should be added.
*      @param command The command string to be added to the history list.  *
@return int Status code (0 on success, -1 on failure).
 */
int add_to_history(HistoryList* list, char* command)
{
    if (!command)
return -1;

    HistoryNode* node =
malloc(sizeof(HistoryNode));     node->command =
COPY(command);     node->next = NULL;

    if (!list->head)
    {
        list->head = node;
    }
    else if (!list->head->next)
    {
        list->tail = node;
        list->head->next = list->tail;
    }
else
{
```

```
        list->tail->next = node;
list->tail = node;
    }
```

```
    list->size++;
return 0;
}

/**
* @brief Retrieves a command at a particular index from the history list.
 *
* Traverses the history list to find and return the command at the specified
  index.
 *
* @param list The history list.
```

```
*@param index The index of the command to retrieve.  * @return char* The
 command at the specified index or NULL if not found.
 */
char* get_command(HistoryList* list, unsigned int index)
{
    if (index > list->size || !list->head)
return NULL;

    HistoryNode* curr = list->head;
unsigned int ctr = 1;

    for (; curr && ctr != index; curr = curr->next, ctr++);

    return curr->command;
}

/**
*@brief Cleans up the history list by freeing all allocated memory.
 *
*Iterates through the history list, frees each node and its command, and resets
 the list.
 *
*@param list The history list to clean up.
 */
void clean_history(HistoryList* list) {
    HistoryNode* current = list->head;
    HistoryNode* next;

    while (current != NULL)
{         next =
current->next;
free(current->command);
free(current);         current
= next;
    }

    // After cleaning up all nodes, reset the
list     list->head = NULL;     list->tail =
NULL;     list->size = 0;
}

/**
*@brief Finds the last command in the history that starts with the specified
 prefix.
 *
*Searches the history list for the most recent command that starts with the
 given prefix.
 *
*@param list The history list.
*@param prefix The prefix to match commands against.
*@return char* The most recent command matching the prefix or NULL if no match
 is found.  */
char* find_last_command_with_prefix(HistoryList* list, const char* prefix)
{
    if (list == NULL || list->head == NULL || prefix == NULL)
{         return NULL; // Invalid input
    }

    HistoryNode* current = list->head;
char* lastCommand = NULL;

    while (current != NULL)
    {
```

```
        // Check if the current command starts with the given prefix
if (strncmp(current->command, prefix, strlen(prefix)) == 0)
{             // Update the lastCommand whenever a match is found
lastCommand = current->command; // Duplicate the string
        }
        current = current->next;
    }
    return lastCommand;
}

/*------------------------------Shell State Management-----------------------
--
--------*/

/**
*@brief Initializes the shell state with default values.
 *
*Allocates memory for a ShellState object, sets default file descriptors, and
 initializes the history list.
 *
*@return ShellState* Pointer to the initialized ShellState object.
 */
ShellState* init_shell_state()
{
    ShellState* stateObj = malloc(sizeof(ShellState));
if (!stateObj)
    {
        LOG_ERROR("malloc failure. Exiting.\n");
exit(-1);
    }
    stateObj->originalStdinFD = STDIN_FD;
stateObj->originalStdoutFD = STDOUT_FD;
stateObj->originalStderrFD = STDERR_FD;

    // default prompt
    strncpy(stateObj->prompt_buffer, "\%", MAX_STRING_LENGTH);

    stateObj->history.head = NULL;
stateObj->history.tail = NULL;
stateObj->history.size = 0;

    return stateObj;
}

/**
*@brief Frees memory allocated for the shell state.
 *
*Frees the ShellState object and resets its pointer.
 *
*@param stateObj The ShellState object to be cleared.
*@return int Status code (0 on success, -1 on failure).
 */
int clear_shell_state(ShellState* stateObj)
{
    if (!stateObj)
    {
        LOG_DEBUG("Can't clear NULL shell state object.\n");
return -1;
    }

    free(stateObj);
return 0;
}
```

```
/*------------------------------File Descriptor Management--------------------
--
------------*/

/**
*@brief Sets up file descriptors for input, output, and stderr.
 *
*Uses the `dup2` system call to duplicate file descriptors for input, output,
 and stderr if they differ from the default values.
 *
*@param inputFD The input file descriptor.
*@param outputFD The output file descriptor.
*@param stderrFD The stderr file descriptor.
*@return int Status code (0 on success, -1 on failure).
 */
static int setUpFD(int inputFD, int outputFD, int stderrFD)
{
    if (inputFD != STDIN_FD)
    {
        globalShellState->originalStdinFD = dup(STDIN_FD);

        if (dup2(inputFD, STDIN_FD) == -1)
        {
            LOG_DEBUG("dup2: %s\n", strerror(errno));
return -1;
        }
        close(inputFD);
    }
    if (outputFD != STDOUT_FD)
    {
        globalShellState->originalStdoutFD = dup(STDOUT_FD);

        if (dup2(outputFD, STDOUT_FD) == -1)
        {
            LOG_DEBUG("dup2: %s\n", strerror(errno));
return -1;
        }
        close(outputFD);
    }
    if (stderrFD != STDERR_FD)
    {
        globalShellState->originalStderrFD = dup(STDERR_FD);

        if (dup2(stderrFD, STDERR_FD) == -1)
        {
            LOG_DEBUG("dup2: %s\n", strerror(errno));
return -1;
        }
        close(stderrFD);
    }
    return 0;
}

/**
*@brief Resets file descriptors to their original values.
 *
*Restores the original file descriptors for input, output, and stderr.
 */
static void resetFD()
{
    if (globalShellState->originalStdinFD != STDIN_FD)
    {
        if (dup2(globalShellState->originalStdinFD, STDIN_FD) == -1)
```

```
        {
            LOG_ERROR("dup2: %s\n", strerror(errno));
exit(1);
        }
    }
    if (globalShellState->originalStdoutFD != STDOUT_FD)
    {
        if (dup2(globalShellState->originalStdoutFD, STDOUT_FD) == -1)
        {
            LOG_ERROR("dup2: %s\n", strerror(errno));
exit(1);
        }
    }
    if (globalShellState->originalStderrFD != STDERR_FD)
    {
        if (dup2(globalShellState->originalStderrFD, STDERR_FD) == -1)
        {
            LOG_ERROR("dup2: %s\n", strerror(errno));
exit(1);
        }
    }
}

/*------------------------------Builtin Commands-----------------------------
--
--*/

/**
*@brief Changes the current directory.
 *
*Changes the working directory to the specified path or to the home directory
 if no path is provided.
 *
*@param simpleCommand The command to execute, containing the arguments for
 `cd`.
*@return int Status code (0 on success, -1 on failure).
 */
int cd(SimpleCommand* simpleCommand)
{
    if (simpleCommand->argc > 2)
    {
        LOG_ERROR("cd: Too many arguments\n");
return -1;
    }

    // Don't think cd ever needs any input from stdin, neither it puts
anything to stdout, so dont need to modify file descriptors     const char*
path = NULL;     if (simpleCommand->argc == 1)
    {
        // No path specified, go to home directory
path = HOME_DIR;
    }
else
{
        path = simpleCommand->args[1];
    }
    if (chdir(path) == -1)
    {
        LOG_ERROR("cd: %s\n", strerror(errno));
return -1;
    }
    return 0;
}
```

```
/**
*@brief Prints the current working directory.
 *
*Retrieves and prints the current working directory to stdout.
 *
*@param simpleCommand The command to execute, expected to have no arguments.  *
 @return int Status code (0 on success, -1 on failure).
 */
int pwd(SimpleCommand* simpleCommand)
{
    if (simpleCommand->argc > 1)
    {
        LOG_ERROR("pwd: Too many arguments\n");
return -1;
    }
    char cwd[MAX_PATH_LENGTH];

    if (getcwd(cwd, sizeof(cwd)) == NULL)
    {
        LOG_ERROR("pwd: %s\n", strerror(errno));
return -1;
    }
    if (setUpFD(simpleCommand->inputFD, simpleCommand->outputFD, simpleCommand-
>stderrFD))
{
        return -1;
    }

    LOG_PRINT("%s\n", cwd);

resetFD();
return 0;
}

/**
*@brief Exits the shell with an optional status code.
 *
*Terminates the shell process with the provided exit status or with status 0 if
 no status is provided.
 *
*@param simpleCommand The command to execute, which may include an optional
 exit status.
*@return int Status code (0 on success, -1 on failure).
 */
int exitShell(SimpleCommand* simpleCommand)
{
    if (simpleCommand->argc > 2)
    {
        printf("exit: Too many arguments\n");
return -1;
    }
    LOG_OUT("exit\n");

    if (simpleCommand->argc == 1)
exit(0);

    // check if each char in the second arg is a number or not. that was the only
standard compliant way I could think of to figure whether the argument is a
numebr or not
    if(strspn(simpleCommand->args[1], "0123456789") !=
strlen(simpleCommand>args[1]))
    {
        LOG_ERROR("exit: Expects a numerical argument\n");
return -1;
```

```
    }
    int exit_status = atoi(simpleCommand->args[1]);
    exit(exit_status);
}

/**
*@brief Displays the command history or executes a command from history.
 *
*If no arguments are provided, prints the entire history list. If an index or
 prefix is provided, executes the corresponding command from the history.
 *
*@param simpleCommand The command to execute, which may include arguments for
 history retrieval or command execution.
*@return int Status code (0 on success, -1 on failure).
 */
int history(SimpleCommand* simpleCommand)
{
    if (simpleCommand->argc > 2)
    {
        LOG_ERROR("history: Too many arguments\n");
return -1;
    }
    if (setUpFD(simpleCommand->inputFD, simpleCommand->outputFD, simpleCommand-
>stderrFD))
{
        return -1;
    }
    if (simpleCommand->argc == 1)
    {
        HistoryNode* curr =
globalShellState->history.head;         int i = 1;
while (curr)
        {
            LOG_PRINT("%d %s\n", i,
curr->command);             curr = curr->next;
i++;
        }
        resetFD();
    }
else
{
        char* input = NULL;
        if(strspn(simpleCommand->args[1], "0123456789") ==
strlen(simpleCommand-
>args[1]))
        {
            // execute the command at that index
            unsigned int idx = (unsigned int)atoi(simpleCommand->args[1]);
input = COPY(get_command(&globalShellState->history, idx));

            if (!input)
            {
                LOG_ERROR("history: invalid index\n");
return -1;
```

```
            }
}
else
{
            char* last = find_last_command_with_prefix(&globalShellState-
>history, simpleCommand->args[1]);
input = COPY(last);

            if (!input)
            {
                LOG_ERROR("history: no matching command found\n");
return -1;
            }
        }

        // simple whitespace tokenizer
        char** tokens = tokenizeString(input, ' ');

        // generate the command from tokens
        CommandChain* commandChain = parseTokens(tokens);

        // execute the command
        int status = executeCommandChain(commandChain);
        (void)status;
// Free tokens
freeTokens(tokens);

        // free the command chain
cleanUpCommandChain(commandChain);

        // Free buffer that was allocated for input
free(input);
    }
    return 0;
}

/**
* @brief Executes a simple command in a child process.
 *
```

```
*Forks a new process, sets up file descriptors, executes the command, and
 optionally waits for the child process to finish.
 *
*@param simpleCommand The command to execute, including its arguments and file
 descriptors.
*@return int Status code (0 on success, -1 on failure).
 */
int executeProcess(SimpleCommand* simpleCommand)
{
    int pid = fork();

    if (pid == -1)
    {
        LOG_DEBUG("fork: %s\n", strerror(errno));
return -1;
    }
    else if (pid == 0)
    {
        // Child process
        setUpFD(simpleCommand->inputFD, simpleCommand->outputFD, simpleCommand-
>stderrFD);

        // Execute the command
        if (execvp(simpleCommand->commandName, simpleCommand->args) == -1)
        {
            LOG_ERROR("%s: %s\n", simpleCommand->commandName, strerror(errno));
exit(1);
        }

        // This should never be reached
        LOG_ERROR("This should never be
reached\n");         exit(0);     }     else
    {
        // Parent process
        simpleCommand->pid = pid;

        if (!simpleCommand->noWait) {
            // Wait for the child process to finish
            int status;
            LOG_DEBUG("Waiting for child process, with command name %s\n",
simpleCommand->commandName);
            if (waitpid(pid, &status, 0) == -1)
            {
                LOG_ERROR("waitpid: %s\n", strerror(errno));
return -1;
            }

            // Print the exit status of the child process
if (WEXITSTATUS(status) != 0)
            {
                LOG_DEBUG("Non zero exit status : %d\n", WEXITSTATUS(status));
return WEXITSTATUS(status);
            }
        }
    }

    LOG_DEBUG("Finished executing command %s\n", simpleCommand->commandName);
return 0;
}

/**
*@brief Changes the current prompt of the shell.
 *
*Sets the shell prompt to the specified string.
```

```
 *
*@param simpleCommand The command to execute, including the new prompt string.
 * @return int Status code (0 on success, -1 on failure).
 */
int prompt(SimpleCommand* simpleCommand)
{
    if (simpleCommand->argc == 1)
    {
        LOG_ERROR("prompt: Too few arguments\n");
return -1;
    }
    if (simpleCommand->argc > 2)
    {
        LOG_ERROR("prompt: Too many arguments\n");
return -1;
    }
    strcpy(globalShellState->prompt_buffer, simpleCommand->args[1]);

    return 0;
}

/*------------------------------Command Registry-----------------------------
--
--*/

/**
*@brief Represents a builtin command and its corresponding execution function.
 *
*Holds the name of the command and a pointer to its execution function.
 */
typedef struct commandRegistry
{
    char* commandName;
    ExecutionFunction executionFunction;
} CommandRegistry;

/**
```

```
*      @brief Registry of all the builtin commands and their execution functions.
 *
*      Maps command names to their corresponding execution functions. If a
command is not found in the registry, the default function `executeProcess` is
used.
 *
*      @return ExecutionFunction Pointer to the function to execute for the given
command.  */
static const CommandRegistry commandRegistry[] = {
    {"cd", cd},
    {"pwd", pwd},
    {"exit", exitShell},
    {"history", history},
    {"prompt", prompt},
    {NULL, NULL}
};

/**
*      @brief Retrieves the execution function for a given command name.
 *
*      Searches the command registry for the specified command and returns its
associated execution function. If the command is not found, returns the default
process execution function.
 *
*      @param commandName The name of the command to look up.
*      @return ExecutionFunction Pointer to the execution function for the
command.
 */
ExecutionFunction getExecutionFunction(char* commandName)
{
    for (int i = 0; commandRegistry[i].commandName != NULL; i++)
    {
        if (strcmp(commandRegistry[i].commandName, commandName) == 0)
        {
            return commandRegistry[i].executionFunction;
        }
    }
    return
executeProcess; }
```

Listing 10: Shell/src/shell_builtins.c

```
/**
* @file utils.c
* @brief Function definitions for the utility functions.
* @version 0.1
 *
 */

#include "utils.h"
#include <string.h>
#include <stdlib.h>

/**
* @brief Tokenizes the input string based on a specified delimiter.
 *
* Splits the input string into an array of tokens using the provided delimiter.
Handles quoted substrings
* so that delimiters inside quotes are not considered as token separators.
 *
* @param input The string to tokenize.
* @param delimiter The character used to split the input string into tokens.
* @return char** An array of tokens, with the last element set to NULL. Returns
NULL if memory allocation fails.
 */
char **tokenizeString(const char *input, char delimiter)
{
    int input_length = strlen(input);
```

```
    char **tokens = (char **)malloc(sizeof(char *) * input_length);
int token_count = 0;

    int i = 0;     int
token_start = 0;     int
inside_quotes = 0;

    while (input[i] != '\0')
    {
        if (input[i] == delimiter && !inside_quotes)
        {
            int token_length = i - token_start;
            tokens[token_count] = (char *)malloc(sizeof(char) * (token_length +
1));
            strncpy(tokens[token_count], input + token_start,
token_length);             tokens[token_count][token_length] = '\0';
token_count++;             token_start = i + 1;
        }
        else if (input[i] == '"' || input[i] == '\'')
        {
            inside_quotes = !inside_quotes;
        }
i++;
    }
    int token_length = i - token_start;
    tokens[token_count] = (char *)malloc(sizeof(char) * (token_length + 1));
strncpy(tokens[token_count], input + token_start, token_length);
    tokens[token_count][token_length] = '\0';
token_count++;

    char** temp = (char **)realloc(tokens, sizeof(char *) * (token_count +
1));     if (!temp)         return NULL;

    tokens = temp;
temp = NULL;

    tokens[token_count] = NULL;

    return tokens;
}

/**
*      @brief Counts the number of tokens in an array of tokens.
*       * Iterates through the token array and counts the number of non-NULL
elements.
 *
*      @param tokens The array of tokens.
*      @return int The number of tokens in the array.
 */
int getTokenCount(char **tokens)
{
    int token_count = 0;
    while (tokens[token_count] != NULL)
    {
        token_count++;
    }
    return token_count;
}

/**
*      @brief Frees the memory allocated for an array of tokens.
 *
*      Deallocates memory for each token in the array and then frees the array
itself.
```

*

```
*      @param tokens The array of tokens to free.
 */
void freeTokens(char **tokens)
{
    for (int i = 0; tokens[i] != NULL; i++)
    {
        free(tokens[i]);
    }
    free(tokens);
}

/**
*      @brief Removes surrounding quotes from a string.
 *
*      If the input string is enclosed in quotes (single or double), creates a
new
string without the quotes and
*      returns it. Otherwise, returns the original string.
 *
*      @param inputString The string to process.
*      @return char* The modified string without quotes, or the original string
if not enclosed in quotes.
 */
char *removeQuotes(char *inputString)
{
    int inputLength = strlen(inputString);

    // Check if the string is long enough to contain quotes
if (inputLength < 2)
    {
        // String is too short to be enclosed in quotes
return inputString;
    }

    // Check if the string is enclosed in quotes
    if ((inputString[0] == '"' && inputString[inputLength - 1] == '"') ||
(inputString[0] == '\'' && inputString[inputLength - 1] == '\''))
{
        // Create a modified string without the quotes
size_t modifiedLength = inputLength - 2;
        char *modifiedString = malloc((modifiedLength + 1) * sizeof(char));
        strncpy(modifiedString, inputString + 1,
modifiedLength);         modifiedString[modifiedLength] = '\0';
free(inputString);          return modifiedString;
    }
else
    {
        // String is not enclosed in quotes
        return inputString; // Return a copy of the input
string     }
}
```

Listing 11: Shell/src/utils.c