Assignment 2 - Shell

Contents

1 Description **2**

2 List of files **2**

3 Self diagnosis and evaluation **2**

4 Discussion of Solution **3**

4.1 Modular design and abstractions . . . 3

4.2 Data structure and algorithmic choices . . . 3

4.3 Ease of feature addition . . . 3

4.4 Debugging . . . 3

5 Test Evidence **4**

5.1 Compilation with make . . . 4

5.2 Simple commands . . . 4

5.3 Tokenisation . . . 5

5.4 Wildcard expansion . . . 7

5.5 Hanlding of interruputs . . . 8

5.6 Advanced functionality . . . 8

5.7 History . . . 9

5.8 Autograding . . . 10

6 Code listings **12**

Listings

1 Shell/Makefile . . . 12
2 Shell/src/shell_builtins.c . . . 14
3 Shell/src/command.c . . . 24
4 Shell/src/utils.c . . . 31
5 Shell/src/main.c . . . 33
6 Shell/src/parser.c . . . 37
7 Shell/include/command.h . . . 43
8 Shell/include/parser.h . . . 48
9 Shell/include/shell_builtins.h . . . 48
10 Shell/include/log.h . . . 51
11 Shell/include/utils.h . . . 52

12 Shell/test/test.py 54

1 Description

We designed and implemented a minimal command-line interpreter, aka. shell that mimics the functionality of a real UNIX shell (e.g. BASH, ZSH etc). The shell program will not be commercial shell equivalent, but will have the core functionality down. Like the shells you use daily (or rarely), ours will issue a prompt (when running interactively), at which it reads input commands from the user and executes them.

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/
|-- Report/
|   |-- report.pdf
```

3 Self diagnosis and evaluation

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

1. A reconfigurable shell prompt has been implemented. Shell starts off with the default prompt of `%` and can configure the prompt by issuing the command `prompt` with a single argument.
2. A Builtin `pwd` is implemented instead of using the default binary included in `/usr/bin`.
3. We have enabled directory walking. User can issue the `cd` command with a single argument to go to a specific directory, or with no argument to return to the home directory.
4. We have added wildcard characters support. Wildcard characters in a command get expanded by shell to filenames.
5. We have also enabled redirection of standard input, output and error streams to different files.
6. Unix pipelines are also supported by our shell. A pipeline can be arbitrarily long, and can connect multiple commands.
7. We have also enabled jobs that can run in the background via the use of the `&` operator. The shell starts a background job, and immediately returns with the prompt, executing the job in the

background. Background jobs can also be connected with other background jobs, and foreground jobs.

8. Sequential job execution is also enabled, and the user can run multiple pipelines sequentially via the use of the `;` operator.

9. Bash like history is also enabled. Each command is stored in shell's history. `history` command prints the previous commands, in bash like format. The `!` operator is also implemented that can either run the command at that index in history, or the last command matching that prefix.

10. Shell inherits environment from parent process.

11. Shell also has a bash like builtin `exit`, which can also take an argument to specify the exit code, which by default is 0.

4 Discussion of Solution

A key focus was placed on creating a modular and extensible design through the implementation of abstractions. This discussion explores design choices made, and some of their advantages.

4.1 Modular design and abstractions

I adopted a modular design approach to enhance maintainability and extensibility of the shell. By representing simple commands, pipelines, and chains as distinct structures, or objects, with their own methods, I ensured a clear separation of concerns. This choice facilitates the addition of new features independently at each level. For instance, the seamless integration of logical chaining became straightforward due to this modular approach.

4.2 Data structure and algorithmic choices

Array Usage for Improved Cache Locality

- Command Arguments and Pipelines: Arrays were employed to store command arguments and individual commands in pipelines. This decision was grounded in the desire for improved cache locality, as these elements are frequently accessed during execution. - Chains: While sequential chains are accessed less frequently, linked lists were chosen for their faster addition. However, a nuanced approach was taken by incorporating a tail pointer for O(1) write times, considering the infrequent reads.

History Management

- Current Implementation: The history, being accessed less frequently and written to more often, is currently implemented as a linked list. - Potential Improvement: Recognizing a potential enhancement, an exploration into using arrays for history storage is considered, aiming to optimize the trade-off between read and write operations.

4.3 Ease of feature addition

The decision to adopt distinct structures for commands, pipelines, and chains not only improves maintainability but also facilitates the seamless addition of new features. Each level can be extended independently, making the shell more adaptable to evolving requirements.

4.4 Debugging

I also want to discuss the choices made for debugging that really helped with the project's development. I added a header only logging library to have logs of different level. This way I was able to have two different types of builds, release and debug. Debug build shows more output logs, but I don't have to remove them

in the release build. Release build also builds an optimized binary. Valgrind was used excessively (added to the makefile) to test for memory leaks, and to remove segfaults.

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 by running the shown commands and collecting their output and showing it verbaitm.

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 successfuly and shows user the prompt. (Note that the compilation was done inside of our own shell, indicating it works perfectly.)

5.2 Simple commands

Shell is able to run simple commands easily as specified in the grammar.

```
% ls
Makefile  build  include  src
% touch tempfile
% ls
Makefile  build  include  src  tempfile
% rm -f tempfile
% mkdir -p random_ahh_dir
% rm -rf random_ahh_dir
% ls
Makefile  build  include  src
% prompt testuser$
testuser$ pwd
/home/Shell
testuser$ cd include
testuser$ ls
command.h  parser.h          utils.h
log.h      shell_builtins.h
testuser$ cd ..
testuser$ cd src
testuser$ ls
command.c  parser.c          utils.c
main.c     shell_builtins.c
```

```
testuser$ exit
exit
#
```

The test shows that all the simple commands as well as simple builtins including pwd and exit are working correctly.

5.3 Tokenisation

We can use the debug build to show the internal workings of the shell.

```
$ 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,88) Starting shell
% ls
[DEBUG]:  (main,138) 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,423) Waiting for child process, with command name ls
Makefile  build  include  random_ahh_dir  src
[DEBUG]:  (executeProcess,440) Finished executing command ls
[DEBUG]:  (executeCommand,230) Command executing with pid: 60710
[DEBUG]:  (main,149) Command executed with status 0
[DEBUG]:  (cleanUpSimpleCommand,257) Cleaning up simple command: ls
% sleep 1 & ps -l ; echo hello ; ls | grep b
[DEBUG]:  (main,138) Token 0: [sleep]
[DEBUG]:  (main,138) Token 1: [1]
[DEBUG]:  (main,138) Token 2: [&]
[DEBUG]:  (main,138) Token 3: [ps]
[DEBUG]:  (main,138) Token 4: [-l]
[DEBUG]:  (main,138) Token 5: [;]
[DEBUG]:  (main,138) Token 6: [echo]
[DEBUG]:  (main,138) Token 7: [hello]
[DEBUG]:  (main,138) Token 8: [;]
[DEBUG]:  (main,138) Token 9: [ls]
[DEBUG]:  (main,138) Token 10: [|]
[DEBUG]:  (main,138) Token 11: [grep]
[DEBUG]:  (main,138) 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,440) Finished executing command sleep
[DEBUG]:  (executeCommand,230) Command executing with pid: 60873
[DEBUG]:  (executeCommand,214) Executing command : ps
[DEBUG]:  (executeProcess,423) Waiting for child process, with command name ps
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S  1000     322     321  0  80   0 -  1585 do_wai pts/0    00:00:00 bash
0 S  1000   60697     322  0  80   0 -   700 do_wai pts/0    00:00:00 Shell
0 S  1000   60873   60697  0  80   0 -   802 hrtime pts/0    00:00:00 sleep
0 R  1000   60874   60697  0  80   0 -  1870 -      pts/0    00:00:00 ps
[DEBUG]:  (executeProcess,440) Finished executing command ps
[DEBUG]:  (executeCommand,230) Command executing with pid: 60874
[DEBUG]:  (executeCommand,214) Executing command : echo
[DEBUG]:  (executeProcess,423) Waiting for child process, with command name echo
hello
[DEBUG]:  (executeProcess,440) Finished executing command echo
[DEBUG]:  (executeCommand,230) Command executing with pid: 60875
```

```
[DEBUG]:  (executeCommand,214) Executing command : ls
[DEBUG]:  (executeProcess,423) Waiting for child process, with command name ls
[DEBUG]:  (executeProcess,440) Finished executing command ls
[DEBUG]:  (executeCommand,230) Command executing with pid: 60876
[DEBUG]:  (executeCommand,214) Executing command : grep
[DEBUG]:  (executeProcess,423) Waiting for child process, with command name grep
build
[DEBUG]:  (executeProcess,440) Finished executing command grep
[DEBUG]:  (executeCommand,230) Command executing with pid: 60877
[DEBUG]:  (main,149) 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 tried to execute two different commands, a simple command, and a complex command with multiple shell operators to look at the debug output. As we can see shell tokenizes them correctly, parses them as expected and then runs them.

5.4 Wildcard expansion

Following session illustrates the wildcard expansion capabilities of the shell.

```
% ls *
Makefile

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
    164 src/main.c
    309 src/parser.c
    497 src/shell_builtins.c
    111 src/utils.c
    196 include/command.h
    77 include/log.h
    43 include/parser.h
    113 include/shell_builtins.h
    70 include/utils.h
    1956 total
% wc -l src/*.c include/*.h | grep total | awk '{print $1}'
1956
% wc -l src/*.c include/*.h | grep total | awk '{print $1}' | xargs echo "Total LOC: "
```

```
Total LOC:  1956
% touch file.txt_ez.c log.pop_ab.d
% echo *.txt_*.?
file.txt_ez.c   log.pop_ab.d
%
```

5.5 Hanlding of interruputs

Shell correctly handles interrupts generated by the CTRL-Z, CTRL-C, CTRL-\ keys. We can use the debug build to see how shell handles these signals.

```
[DEBUG]:  (main,88) Starting shell
% ^C
[DEBUG]:  (sigint_handler,53) CTRL-C pressed. signo: 2

% ^Z
[DEBUG]:  (sigtstp_handler,58) CTRL-Z pressed. signo: 20

% ^\
[DEBUG]:  (sigquit_handler,63) CTRL-\ pressed. signo: 3

%
```

As you can see, the program exhibits the exact same behaviour as the BASH's wildcard expansion.

5.6 Advanced functionality

All the features specified in advanced functionality have been implemented as the following session demonstrates.

As visible, the shell doesn't kill itself, it just gracefully prints what happened, and continues execution.

```
------------------ Sequential commands ------------------------
% ls
Makefile  build  include  src
% echo hello
hello
% echo hello ; echo world ; echo 1 ; ps -l ; pwd ; ls ; cd src ; ls
hello
world
echo 1
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S  1000     322     321  0  80   0 -  1585 do_wai pts/0    00:00:00 bash
0 S  1000   65972     322  0  80   0 -   695 do_wai pts/0    00:00:00 Shell
0 R  1000   66184   65972  0  80   0 -  1870 -      pts/0    00:00:00 ps
/home/Shell
Makefile  build  include  src
command.c  main.c  parser.c  shell_builtins.c  utils.c
% cd ..
% ls
Makefile  build  include  src
---- Background and concurrent execution (no zombie processes) ----
% 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
4 S  1000     322     321  0  80   0 -  1585 do_wai pts/0    00:00:00 bash
0 S  1000   65972     322  0  80   0 -   695 do_wai pts/0    00:00:00 Shell
0 S  1000   66316   65972  0  80   0 -   802 hrtime pts/0    00:00:00 sleep
0 R  1000   66329   65972  0  80   0 -  1870 -      pts/0    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
4 S  1000     322     321  0  80   0 -  1585 do_wai pts/0    00:00:00 bash
0 S  1000   65972     322  0  80   0 -   695 do_wai pts/0    00:00:00 Shell
0 S  1000   68474   65972  0  80   0 -   802 hrtime pts/0    00:00:00 sleep
0 S  1000   68475   65972  0  80   0 -   802 hrtime pts/0    00:00:00 sleep
0 R  1000   68476   65972  0  80   0 -  1870 -      pts/0    00:00:00 ps
done
% sleep 5 & ps -l | grep sleep
0 S  1000   66947   65972  0  80   0 -   802 hrtime pts/0    00:00:00 sleep
--------------------------- IO redirection -----------------------------
% 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 xaxaxa 2> error.txt
% cat error.txt
ls: cannot access 'xaxaxa': No such file or directory
% ls xaxax > error.txt
ls: cannot access 'xaxax': No such file or directory
% ls *.txt
concat.txt  error.txt  hello.txt  world.txt
% rm *.txt
% echo helloo | wc > wc.txt ; cat wc.txt ; rm wc.txt
      1       1       7
--------------------------- Pipelines ----------------------------------
% 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
```

The above shell session demonstrates the advanced functionality correctly. All the required features work correctly with each features combination with others.

5.7 History

```
% ls
Makefile  build  include  src
% pwd
/home/Shell
% history
1 ls
2 pwd
3 history
% !2
/home/Shell
% !3
1 ls
2 pwd
3 history
4 !2
5 !3
% !pw
/home/Shell
% echo hello
hello
% !ec
hello
% echo a
a
% echo b
b
% echo c
c
% !echo
c
%
```

5.8 Autograding

A testing framework is also added to enable testing of a large amount of commands. This is enabled by building a script mode in the shell, in which it reads input from a file instead of a user and then runs the command in the same way. The testing framework then accepts a series of test files and runs the test commands on those script files. The framework compares the output of these ones with a reference shell, in this case, bash. Here are the results of the testing.

```
$ make test

SHELL TEST SUITE

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

...
```

```
13
14 Running tests : simple
15     Running file: simple.test               PASSED
16     Running file: builtins.test             PASSED
17     Running file: exec_only.hidden          PASSED
18     Running file: pipeline.test             PASSED
19     Running file: ioredir.test              PASSED
20     Running file: ioredir_one.hidden        PASSED
21     Running file: pipeline_one.hidden       PASSED
22     Running file: pipeline_two.hidden       PASSED
23     Running file: pipeline_ioredir.hidden   PASSED
24 Running tests : advanced
25     Running file: chaining.test             PASSED
26     Running file: wildcards.test            PASSED
27     Running file: quotes.test               PASSED
28     Running file: wild_chaining.test        PASSED
29     Running file: wild_chaining.hidden      PASSED
30     Running file: wildcards_one.hidden      PASSED
31
32 ...
33
34 SUMMARY
35
36 Total : 15, Passed : 15, Failed : 0
37 Finished in 2.11s.
```

Our implementation passes all the tests. The testing framework is also included with the submission, as well as all the files. These files provide a significant test coverage, and test all the features of the shell that we possibly can. The other features have been tested manually and been included in the report in the above sections. Note the usage of `.test and .hidden` files. During development, only `.test` files were used, and the final shell was tested on all the files including the hidden ones. This was done to ensure that no hardcoding occurs. All the script files are very descriptive, and contain combinations of commands, as well as simple commands.

6 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=
```

```
51
52    CLEAN=
53    MK_INIT_ERROR=
54  else
55
56    TRACE_CC       = @echo "$(CYAN)  CC      $(RESET)" $<
57    TRACE_LD       = @echo "$(CYAN)  LD      $(RESET)" $@
58    TRACE_MKDIR    = @echo "$(CYAN)  MKDIR   $(RESET)" $@
59    TRACE_CP       = @echo "$(CYAN)  CP      $(RESET)" $< "-->" $@
60    Q ?= @
61
62    BUILD_SUCCESS  =@echo "-- $(GREEN)Build successful in $(BUILD_DEFAULT) mode.$(RESET)"
63    BUILD_FAILURE  =echo  "-- $(RED)Build failed.$(RESET)"; exit 1
64    LINK_FAILURE   =echo  "-- $(RED)Linking failed.$(RESET)"; exit 1
65    INIT_MAIN      =@echo "-- $(CYAN)Creating main.c$(RESET)"
66    INIT_SUCCESS   =@echo "-- $(GREEN)Initialized the project structure$(RESET)"
67    RUN            =@echo "-- $(CYAN)Executing$(RESET): $(TARGET_NAME)"
68    VALGRIND_RUN   =@echo "-- $(CYAN)Running Valgrind on$(RESET): $(TARGET_NAME)"
69    CLEAN          =@echo "-- $(GREEN)Cleaned$(RESET): $(BUILD_DIR)/*"
70
71    MK_INIT_ERROR  =@echo "$(RED)Error: $(SRC_DIR) directory doesn't exist. Please run make
          init to initialize the project.$(RESET)"
72  endif
73
74  # phony targets
75  .PHONY: all run valgrind clean test
76
77  # Sets flags based on the build mode.
78  ifeq ($(BUILD_DEFAULT), release)
79    CFLAGS += $(RELEASE_FLAGS)
80  else
81    CFLAGS += $(DEBUG_FLAGS)
82  endif
83
84  # Find all the source files and corresponding objects
85  SRCS := $(wildcard $(SRC_DIR)/*.c)
86  OBJS := $(patsubst $(SRC_DIR)/%.c, $(BUILD_DIR)/%.o, $(SRCS))
87
88  # The all target, builds shell
89  all: $(TARGET)
90          $(Q) echo "$(BUILD_DEFAULT)" > $(BUILD_DIR)/build_mode
91
92  # The TARGET target depends on the generated object files.
93  $(TARGET): $(OBJS)
94          $(TRACE_LD)
95          $(Q) $(CC) $(CFLAGS) -I$(INCLUDE_DIR) $^ -o $@ $(LINKER_FLAGS) || ($(LINK_FAILURE
        ))
96          $(BUILD_SUCCESS)
97
98  # The object files' targets, depend on their corresponding source files.
99  $(BUILD_DIR)/%.o: $(SRC_DIR)/%.c
100         $(TRACE_CC)
101         $(Q) $(CC) $(CFLAGS) -I$(INCLUDE_DIR) -c $< -o $@ || ($(BUILD_FAILURE))
102
```

```
# 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 builtitns.c
 * @brief Contains the function definitions for the builtin shell functions.
 * @version 0.1
 *
 * @copyright Copyright (c) 2023
 *
 */

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

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

// stores the original stdin and stdout fds
extern ShellState* globalShellState;

// adds a command to the history list
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;
}

// retrieves a command at a particular index from the list
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;
}

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;
}

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;
}

// initializes the shell state
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;
}

// cleans things up and frees memory
int clear_shell_state(ShellState* stateObj)
{
    if (!stateObj)
    {
        LOG_DEBUG("Can't clear NULL shell state object.\n");
        return -1;
```

```
138     }
139
140     free(stateObj);
141     return 0;
142 }
143
144 /*------------------------------File Desc Manipulators-----------------------------------
        */
145
146 /**
147  * @brief Sets up the file descriptors for a command. Duplicates the file descriptors to
        stdin, and stdout, and if we are in the parent process, we also save the original
        stdin and stdout file descriptors.
148  *
149  * Uses dup2 system call to set up the file descriptors. Returns 0 on success, -1 on
        failure. Only dups if the file descriptors are not the default ones.
150  *
151  * @param inputFD The input file descriptor
152  * @param outputFD The output file descriptor
153  * @return int Status code (0 on success, -1 on failure)
154  */
155 static int setUpFD(int inputFD, int outputFD, int stderrFD)
156 {
157     if (inputFD != STDIN_FD)
158     {
159         globalShellState->originalStdinFD = dup(STDIN_FD);
160
161         if (dup2(inputFD, STDIN_FD) == -1)
162         {
163             LOG_DEBUG("dup2: %s\n", strerror(errno));
164             return -1;
165         }
166
167         close(inputFD);
168     }
169
170     if (outputFD != STDOUT_FD)
171     {
172         globalShellState->originalStdoutFD = dup(STDOUT_FD);
173
174         if (dup2(outputFD, STDOUT_FD) == -1)
175         {
176             LOG_DEBUG("dup2: %s\n", strerror(errno));
177             return -1;
178         }
179
180         close(outputFD);
181     }
182
183     if (stderrFD != STDERR_FD)
184     {
185         globalShellState->originalStderrFD = dup(STDERR_FD);
186
187         if (dup2(stderrFD, STDERR_FD) == -1)
```

```
188         {
189             LOG_DEBUG("dup2: %s\n", strerror(errno));
190             return -1;
191         }
192 
193         close(stderrFD);
194     }
195 
196     return 0;
197 }
198 
199 /**
200  * @brief Resets the file descriptors to the default ones (stdin and stdout).
201  *
202  * @return void
203  */
204 static void resetFD()
205 {
206     if (globalShellState->originalStdinFD != STDIN_FD)
207     {
208         if (dup2(globalShellState->originalStdinFD, STDIN_FD) == -1)
209         {
210             LOG_ERROR("dup2: %s\n", strerror(errno));
211             exit(1);
212         }
213     }
214 
215     if (globalShellState->originalStdoutFD != STDOUT_FD)
216     {
217         if (dup2(globalShellState->originalStdoutFD, STDOUT_FD) == -1)
218         {
219             LOG_ERROR("dup2: %s\n", strerror(errno));
220             exit(1);
221         }
222     }
223 
224     if (globalShellState->originalStderrFD != STDERR_FD)
225     {
226         if (dup2(globalShellState->originalStderrFD, STDERR_FD) == -1)
227         {
228             LOG_ERROR("dup2: %s\n", strerror(errno));
229             exit(1);
230         }
231     }
232 }
233 
234 /*------------------------------Builtins-----------------------------------------------
    */
235 
236 int cd(SimpleCommand* simpleCommand)
237 {
238     if (simpleCommand->argc > 2)
239     {
240         LOG_ERROR("cd: Too many arguments\n");
```

```
241         return -1;
242     }
243 
244     // Don't think cd ever needs any input from stdin, neither it puts anything to stdout
    , so dont need to modify file descriptors
245     const char* path = NULL;
246     if (simpleCommand->argc == 1)
247     {
248         // No path specified, go to home directory
249         path = HOME_DIR;
250     }
251     else
252     {
253         path = simpleCommand->args[1];
254     }
255 
256     if (chdir(path) == -1)
257     {
258         LOG_ERROR("cd: %s\n", strerror(errno));
259         return -1;
260     }
261 
262     return 0;
263 }
264 
265 int pwd(SimpleCommand* simpleCommand)
266 {
267     if (simpleCommand->argc > 1)
268     {
269         LOG_ERROR("pwd: Too many arguments\n");
270         return -1;
271     }
272 
273     char cwd[MAX_PATH_LENGTH];
274 
275     if (getcwd(cwd, sizeof(cwd)) == NULL)
276     {
277         LOG_ERROR("pwd: %s\n", strerror(errno));
278         return -1;
279     }
280 
281     if (setUpFD(simpleCommand->inputFD, simpleCommand->outputFD, simpleCommand->stderrFD)
    )
282     {
283         return -1;
284     }
285 
286     LOG_PRINT("%s\n", cwd);
287 
288     resetFD();
289     return 0;
290 }
291 
292 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);
}

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;
```

```
344         if(strspn(simpleCommand->args[1], "0123456789") == strlen(simpleCommand->args[1])
    )
345         {
346             // execute the commmand at that index
347             unsigned int idx = (unsigned int)atoi(simpleCommand->args[1]);
348             input = COPY(get_command(&globalShellState->history, idx));
349 
350             if (!input)
351             {
352                 LOG_ERROR("history: invalid index\n");
353                 return -1;
354             }
355         }
356         else
357         {
358             char* last = find_last_command_with_prefix(&globalShellState->history,
    simpleCommand->args[1]);
359             input = COPY(last);
360 
361             if (!input)
362             {
363                 LOG_ERROR("history: no matching command found\n");
364                 return -1;
365             }
366         }
367 
368         // simple whitespace tokenizer
369         char** tokens = tokenizeString(input, ' ');
370 
371         // generate the command from tokens
372         CommandChain* commandChain = parseTokens(tokens);
373 
374         // execute the command
375         int status = executeCommandChain(commandChain);
376         (void)status;
377         // Free tokens
378         freeTokens(tokens);
379 
380         // free the command chain
381         cleanUpCommandChain(commandChain);
382 
383         // Free buffer that was allocated for input
384         free(input);
385     }
386 
387     return 0;
388 }
389 
390 int executeProcess(SimpleCommand* simpleCommand)
391 {
392     int pid = fork();
393 
394     if (pid == -1)
395     {
```

```
396         LOG_DEBUG("fork: %s\n", strerror(errno));
397         return -1;
398     }
399     else if (pid == 0)
400     {
401         // Duplicate the FDs. Default FDs are STDIN AND STDOUT but, if pipes or  < > are
    used, the FDs are updated in the parsing step, by opening the relevant file or
    creating relevant pipes
402         setUpFD(simpleCommand->inputFD, simpleCommand->outputFD, simpleCommand->stderrFD)
    ;
403 
404         // Execute the command
405         if (execvp(simpleCommand->commandName, simpleCommand->args) == -1)
406         {
407             LOG_ERROR("%s: %s\n", simpleCommand->commandName, strerror(errno));
408             exit(1);
409         }
410 
411         // This should never be reached
412         LOG_ERROR("This should never be reached\n");
413         exit(0);
414     }
415     else
416     {
417         // Parent process
418         simpleCommand->pid = pid;
419 
420         if (!simpleCommand->noWait) {
421             // waiting for the child process to finish
422             int status;
423             LOG_DEBUG("Waiting for child process, with command name %s\n", simpleCommand
    ->commandName);
424             if (waitpid(pid, &status, 0) == -1)
425             {
426                 LOG_ERROR("waitpid: %s\n", strerror(errno));
427                 return -1;
428             }
429 
430             // print the error (if any) from errno
431             if (WEXITSTATUS(status) != 0)
432             {
433                 // LOG_ERROR("%s: %s\n", simpleCommand->commandName, strerror(errno));
434                 LOG_DEBUG("Non zero exit status : %d\n", WEXITSTATUS(status));
435                 return WEXITSTATUS(status);
436             }
437         }
438     }
439 
440     LOG_DEBUG("Finished executing command %s\n", simpleCommand->commandName);
441     return 0;
442 }
443 
444 // Changes the current prompt of the shell
445 int prompt(SimpleCommand* simpleCommand)
```

```
446 {
447     if (simpleCommand->argc == 1)
448     {
449         LOG_ERROR("prompt: Too few arguments\n");
450         return -1;
451     }
452 
453     if (simpleCommand->argc > 2)
454     {
455         LOG_ERROR("prompt: Too many arguments\n");
456         return -1;
457     }
458 
459     strcpy(globalShellState->prompt_buffer, simpleCommand->args[1]);
460 
461     return 0;
462 }
463 
464 /**
465  * @brief This struct represents the builtin commands of the shell, and their
        corresponding execution functions.
466  *
467  */
468 typedef struct commandRegistry
469 {
470     char* commandName;
471     ExecutionFunction executionFunction;
472 } CommandRegistry;
473 
474 /**
475  * @brief Registry of all the commands supported by the shell, and their corresponding
        execution functions. If a command is not found in the registry, it is assumed to be a
         process to be executed and the executeProcess function is called. Add new commands
        here, with their appropriate functions.
476  *
477  */
478 static const CommandRegistry commandRegistry[] = {
479     {"cd", cd},
480     {"pwd", pwd},
481     {"exit", exitShell},
482     {"history", history},
483     {"prompt", prompt},
484     {NULL, NULL}
485 };
486 
487 ExecutionFunction getExecutionFunction(char* commandName)
488 {
489     for (int i = 0; commandRegistry[i].commandName != NULL; i++)
490     {
491         if (strcmp(commandRegistry[i].commandName, commandName) == 0)
492         {
493             return commandRegistry[i].executionFunction;
494         }
495     }
```

```
496
497     return executeProcess;
498 }
```

Listing 2: Shell/src/shell_builtins.c

```
1  /**
2   * @file command.c
3   * @brief Contains the function definitions for the functions defined in command.h
4   * @version 0.1
5   *
6   * @copyright Copyright (c) 2023
7   *
8   */
9
10 #include "command.h"
11
12 // simple macro to check if this command is chained with a certain operator  with the
        last command(just a hack for readability)
13 #define CHAINED_WITH(opr) (prevCommand ? (prevCommand->chainingOperator ? (strcmp(
        prevCommand->chainingOperator, opr) == 0) : 0) : 0)
14
15 /*
        ------------------------------------------------------------------------------------------
        */
16
17 /*--------------------------------Initializers-----------------------------------------
        */
18
19 // initializes a simple command with default values
20 SimpleCommand* initSimpleCommand()
21 {
22     SimpleCommand* simpleCommand = (SimpleCommand*) malloc(sizeof(SimpleCommand));
23
24     if (!simpleCommand)
25         return NULL;
26
27     simpleCommand->commandName = NULL;
28     simpleCommand->args        = NULL;
29     simpleCommand->argc        = 0;
30     simpleCommand->inputFD     = STDIN_FD;
31     simpleCommand->outputFD    = STDOUT_FD;
32     simpleCommand->stderrFD    = STDERR_FD;
33     simpleCommand->noWait      = 0;
34     simpleCommand->execute     = NULL;
35     simpleCommand->pid         = -1;
36
37     return simpleCommand;
38 }
39
40 // initializes a command with default values
41 Command* initCommand()
42 {
43     Command* command = (Command*)malloc(sizeof(Command));
44
```

```
45      if (!command)
46          return NULL;
47
48      command->simpleCommands   = NULL;
49      command->nSimpleCommands  = 0;
50      command->background       = false;
51      command->chainingOperator = NULL;
52      command->next             = NULL;
53
54      return command;
55  }
56
57  // initializes a command chain with default values
58  CommandChain* initCommandChain()
59  {
60      CommandChain* chain = (CommandChain*)malloc(sizeof(CommandChain));
61
62      if (!chain)
63          return NULL;
64
65      chain->head = NULL;
66      chain->tail = NULL;
67
68      return chain;
69  }
70
71  /*------------------------------Setters (push functions)--------------------------------
        */
72
73  // adds a command to the command chain
74  int addCommandToChain(CommandChain* chain, Command* command)
75  {
76      if (!chain)
77      {
78          LOG_DEBUG("Invalid command chain passed\n");
79          return -1;
80      }
81
82      // If the chain is empty, add the command as the head
83      if (!chain->head)
84      {
85          chain->head = command;
86          chain->tail = command;
87      }
88      // Otherwise, add the command to the tail
89      else
90      {
91          chain->tail->next = command;
92          chain->tail = command;
93      }
94
95      return 0;
96  }
97
```

```
98  // adds a simpleCommand to the current command chain link
99  int addSimpleCommand(Command* command, SimpleCommand* simpleCommand)
100 {
101     if (!command)
102     {
103         LOG_DEBUG("Invalid command passed. It's NULL\n");
104         return -1;
105     }
106 
107     if (!simpleCommand)
108     {
109         LOG_DEBUG("Invalid simpleCommand passed. It's NULL\n");
110         return -1;
111     }
112 
113     // we need to realloc the array containing the commands.
114     SimpleCommand** temp = (SimpleCommand**)realloc(command->simpleCommands, (command->
        nSimpleCommands + 1) * sizeof(SimpleCommand*));
115 
116     if (!temp)
117     {
118         LOG_DEBUG("Realloc error. Failed to reallocate memory for the array.\n");
119         return -1;
120     }
121 
122     // update the pointer
123     command->simpleCommands = temp;
124     temp = NULL;
125 
126     // add the new command at the end.
127     command->simpleCommands[command->nSimpleCommands] = simpleCommand;
128 
129     // update the number of commands
130     command->nSimpleCommands++;
131 
132     return 0;
133 }
134 
135 // pushes an arg to the simpleCommand's args array. makes sure the args array is always
        null terminated.
136 int pushArgs(char* arg, SimpleCommand* simpleCommand)
137 {
138     if (!simpleCommand)
139     {
140         LOG_DEBUG("Invalid simpleCommand passed. It's NULL\n");
141         return -1;
142     }
143 
144     // when NULL ptr given, realloc behaves like malloc
145     char** temp = (char**)realloc(simpleCommand->args, (simpleCommand->argc + 2) * sizeof
        (char*));
146 
147     if (!temp)
148     {
```

```
149         LOG_DEBUG("Realloc error. Failed to reallocate memory for the array.\n");
150         return -1;
151     }
152 
153     simpleCommand->args = temp;
154     temp = NULL;
155 
156     simpleCommand->args[simpleCommand->argc] = COPY(arg);
157     simpleCommand->args[simpleCommand->argc + 1] = NULL;
158     simpleCommand->argc++;
159 
160     if (simpleCommand->argc == 1)
161     {
162         simpleCommand->commandName = COPY(arg);
163     }
164 
165     return 0;
166 }
167 
168 /*------------------------------Command Execution functions
    ------------------------------*/
169 
170 // executes a command chain
171 int executeCommandChain(CommandChain* chain)
172 {
173     if (!chain)
174     {
175         LOG_DEBUG("Invalid command chain passed\n");
176         return -1;
177     }
178 
179     Command* command = chain->head;
180 
181     // lastStatus is the exit status of the last command in the chain, used by logical
    chaining operators
182     int lastStatus = 0;
183 
184     while (command)
185     {
186         // execute the current command in the chain
187         lastStatus = executeCommand(command);
188 
189         // move on to the next one
190         command = command->next;
191     }
192 
193     return lastStatus;
194 }
195 
196 // executes a Command (with or without IO redirs)
197 int executeCommand(Command* command)
198 {
199     if (!command)
200     {
```

```
201         LOG_DEBUG("Invalid command passed\n");
202         return -1;
203     }
204 
205     // If the command is empty, return an error
206     if (!command->simpleCommands || command->nSimpleCommands == 0)
207     {
208         LOG_DEBUG("Invalid command. It's empty\n");
209         return -1;
210     }
211 
212     for (int i = 0; i < command->nSimpleCommands; i++)
213     {
214         LOG_DEBUG("Executing command : %s\n", command->simpleCommands[i]->commandName);
215         SimpleCommand* simpleCommand = command->simpleCommands[i];
216 
217         // if this is a background pipeline, we dont wait for any command
218         if (command->background)
219             simpleCommand->noWait = 1;
220 
221         // If the command name is empty, return an error
222         if (!simpleCommand->commandName)
223         {
224             LOG_DEBUG("Invalid command name. It's empty\n");
225             return -1;
226         }
227 
228         // non-zero status means the command execution failed (both for built-in and
    external commands)
229         int status = simpleCommand->execute(simpleCommand);
230         LOG_DEBUG("Command executing with pid: %d\n", simpleCommand->pid);
231 
232         // If the command failed, return the status
233         if (status)
234         {
235             return status;
236         }
237 
238         // if the command succeeded, simply close the file descriptors
239         if (simpleCommand->inputFD != STDIN_FD)
240             close(simpleCommand->inputFD);
241 
242         if (simpleCommand->outputFD != STDOUT_FD)
243             close(simpleCommand->outputFD);
244     }
245 
246     return 0;
247 }
248 
249 /*------------------------------Clean up functions
    ---------------------------------------*/
250 
251 // cleans up a simple command and frees memeory
252 void cleanUpSimpleCommand(SimpleCommand* simpleCommand)
```

```
253 {
254     if (!simpleCommand)
255         return;
256 
257     LOG_DEBUG("Cleaning up simple command: %s\n", simpleCommand->commandName);
258 
259     // free the commandName. It was allocated with strdup, so this is the only pointer to
        that string. The source for the string was the input token, which is freed in the
        main loop.
260     if (simpleCommand->commandName)
261     {
262         free(simpleCommand->commandName);
263         simpleCommand->commandName = NULL;
264     }
265 
266     // free the args. They were allocated with strdup, so this is the only pointer to
        that string. The source for the string was the input token, which is freed in the
        main loop.
267     if (simpleCommand->args)
268     {
269         for (int i = 0; i < simpleCommand->argc; i++)
270         {
271             if (simpleCommand->args[i])
272             {
273                 free(simpleCommand->args[i]);
274                 simpleCommand->args[i] = NULL;
275             }
276         }
277 
278         free(simpleCommand->args);
279         simpleCommand->args = NULL;
280     }
281 
282     // free the  simpleCommand
283     free(simpleCommand);
284     simpleCommand = NULL;
285 }
286 
287 // cleans up a command
288 void cleanUpCommand(Command* command)
289 {
290     if (!command)
291         return;
292 
293     // free the simpleCommands
294     if (command->simpleCommands)
295     {
296         for (int i = 0; i < command->nSimpleCommands; i++)
297         {
298             cleanUpSimpleCommand(command->simpleCommands[i]);
299         }
300     }
301 
302     free(command->simpleCommands);
```

```
303
304     // free the chainingOperator, it was allocated with strndup
305     if (command->chainingOperator)
306     {
307         free(command->chainingOperator);
308         command->chainingOperator = NULL;
309     }
310
311     // don't need to free the next and current command, they will be handled by the chain
         cleanup
312 }
313
314 // clean up the command chain linked list
315 void cleanUpCommandChain(CommandChain* chain)
316 {
317     if (!chain)
318         return;
319
320     // free the commands
321     if (chain->head)
322     {
323         Command* command = chain->head;
324         while (command)
325         {
326             Command* nextCommand = command->next;
327             cleanUpCommand(command);
328             free(command);
329             command = nextCommand;
330         }
331     }
332
333     // free the chain
334     free(chain);
335     chain = NULL;
336 }
337
338 /*------------------------------Utility functions
        ----------------------------------------*/
339
340 void printCommandChain(CommandChain* chain)
341 {
342     LOG_DEBUG("Printing command chain\n");
343     if (!chain)
344         return;
345
346     Command* command = chain->head;
347     int counter = 1;
348     while (command)
349     {
350         LOG_DEBUG("[Link %d]\n", counter);
351         for (int i = 0; i < command->nSimpleCommands; i++)
352         {
353             printSimpleCommand(command->simpleCommands[i]);
354         }
```

```
        counter++;
        command = command->next;
    }
}

void printSimpleCommand(SimpleCommand* simpleCommand)
{
    if (!simpleCommand)
        return;

    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 3: Shell/src/command.c

```
/**
 * @file utils.c
 * @brief Function definitions for the utility functions.
 * @version 0.1
 *
 * @copyright Copyright (c) 2023
 *
 */

#include "utils.h"

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


// tokenizes the string based on the delimiter
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')
```

```
28      {
29          if (input[i] == delimiter && !inside_quotes)
30          {
31              int token_length = i - token_start;
32              tokens[token_count] = (char *)malloc(sizeof(char) * (token_length + 1));
33              strncpy(tokens[token_count], input + token_start, token_length);
34              tokens[token_count][token_length] = '\0';
35              token_count++;
36              token_start = i + 1;
37          }
38          else if (input[i] == '"' || input[i] == '\'')
39          {
40              inside_quotes = !inside_quotes;
41          }
42          i++;
43      }
44
45      int token_length = i - token_start;
46      tokens[token_count] = (char *)malloc(sizeof(char) * (token_length + 1));
47      strncpy(tokens[token_count], input + token_start, token_length);
48      tokens[token_count][token_length] = '\0';
49      token_count++;
50
51      char** temp = (char **)realloc(tokens, sizeof(char *) * (token_count + 1));
52      if (!temp)
53          return NULL;
54
55      tokens = temp;
56      temp = NULL;
57
58      tokens[token_count] = NULL;
59
60      return tokens;
61  }
62
63  // counts the number of tokens in the token array
64  int getTokenCount(char **tokens)
65  {
66      int token_count = 0;
67      while (tokens[token_count] != NULL)
68      {
69          token_count++;
70      }
71      return token_count;
72  }
73
74  // frees the tokens
75  void freeTokens(char **tokens)
76  {
77      for (int i = 0; tokens[i] != NULL; i++)
78      {
79          free(tokens[i]);
80      }
81      free(tokens);
```

```
}

// removes quotes from the string
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 4: Shell/src/utils.c

```
/**
 * @file main.c
 * @brief This is the main file for the shell. It contains the main function and the loop
     that runs the shell.
 * @version 0.1
 *
 * @copyright Copyright (c) 2023
 *
 */

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

#include <errno.h>
#include <readline/readline.h>
#include <readline/history.h>
#include <signal.h>
#include <fcntl.h>
```

```c
#include <sys/wait.h>

// Global variables
int lastExitStatus = 0;

ShellState* globalShellState;

// shell can also support scripting, useful for testing
FILE* scriptFile;

char* getInput(int interactive)
{
    char* input = malloc(MAX_STRING_LENGTH);

    if (interactive)
    {
        int again = 1;
        char *linept;        // pointer to the line buffer

        while (again) {
            again = 0;
            printf("%s ", globalShellState->prompt_buffer);
            linept = fgets(input, MAX_STRING_LENGTH, stdin);
            if (linept == NULL)
                if (errno == EINTR)
                    again = 1;        // signal interruption, read again
        }

        // remove the trailing newline
        size_t ln = strlen(input) - 1;
        if (input[ln] == '\n')
            input[ln] = '\0';
    }
    else
    {
        size_t len = 0;
        ssize_t read;
        read = getline(&input, &len, scriptFile);
        if (read == -1)
        {
            free(input);
            return NULL;
        }
        // Remove trailing newline
        if (input[read - 1] == '\n')
            input[read - 1] = '\0';
    }

    return input;
}

void sigint_handler(int signo) {
    // Handle SIGINT (CTRL-C)
    LOG_DEBUG("\nCTRL-C pressed. signo: %d\n", signo);
```

```
74  }
75
76  void sigtstp_handler(int signo) {
77      // Handle SIGTSTP (CTRL-Z)
78      LOG_DEBUG("\nCTRL-Z pressed. signo: %d\n", signo);
79  }
80
81  void sigquit_handler(int signo) {
82      // Handle SIGQUIT (CTRL-\)
83      LOG_DEBUG("\nCTRL-\\ pressed. signo: %d\n", signo);
84  }
85
86  void sigchld_handler(int signo) {
87      (void) signo;
88      int more = 1;        // more zombies to claim
89      pid_t pid;           // pid of the zombie
90      int status;          // termination status of the zombie
91
92      while (more) {
93          pid = waitpid(-1, &status, WNOHANG);
94          if (pid <= 0)
95              more = 0;
96      }
97  }
98
99  /**
100  * @brief This is the main function for the shell. It contains the main loop that runs
     the shell.
101  *
102  * @return int
103  */
104 int main(int argc, char** argv)
105 {
106     // by default we are in interactive
107     int interactive = 1;
108     scriptFile = NULL;
109
110     if (argc > 2)
111     {
112         LOG_ERROR("Usage: %s [script]\n", argv[0]);
113         exit(1);
114     }
115
116     // If a script is provided, run it and exit
117     if (argc == 2)
118     {
119         interactive = 0;
120         LOG_DEBUG("Running script %s\n", argv[1]);
121         scriptFile = fopen(argv[1], "r");
122         if (!scriptFile)
123         {
124             LOG_ERROR("Error opening script %s: %s\n", argv[1], strerror(errno));
125             exit(1);
126         }
```

```
127     }
128     globalShellState = init_shell_state();
129 
130     LOG_DEBUG("Starting shell\n");
131 
132     char delimiter = ' ';
133 
134     if (signal(SIGINT, sigint_handler) == SIG_ERR) {
135         LOG_ERROR("Unable to register SIGINT handler");
136         exit(EXIT_FAILURE);
137     }
138 
139     if (signal(SIGTSTP, sigtstp_handler) == SIG_ERR) {
140         LOG_ERROR("Unable to register SIGTSTP handler");
141         exit(EXIT_FAILURE);
142     }
143 
144     if (signal(SIGQUIT, sigquit_handler) == SIG_ERR) {
145         LOG_ERROR("Unable to register SIGQUIT handler");
146         exit(EXIT_FAILURE);
147     }
148 
149     if (signal(SIGCHLD, sigchld_handler) == SIG_ERR) {
150         LOG_ERROR("Unable to register SIGCHLD handler");
151         exit(EXIT_FAILURE);
152     }
153 
154     while (1)
155     {
156         // read input
157         char* input = getInput(interactive);
158 
159         // Check for EOF.
160         if (!input)
161             break;
162         if (strcmp(input, "") == 0)
163         {
164             free(input);
165             continue;
166         }
167         if (strcmp(input, "exit") == 0)
168         {
169             free(input);
170             break;
171         }
172 
173         // Add input to readline history.
174         add_to_history(&globalShellState->history, input);
175 
176         // simple whitespace tokenizer
177         char** tokens = tokenizeString(input, delimiter);
178 
179         for (int i = 0; tokens[i] != NULL; i++) {
180             LOG_DEBUG("Token %d: [%s]\n", i, tokens[i]);
```

```
        }

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

        // display the command chain
        printCommandChain(commandChain);

        // execute the command
        int status = executeCommandChain(commandChain);
        LOG_DEBUG("Command executed with status %d\n", status);

        // Free tokens
        freeTokens(tokens);

        // free the command chain
        cleanUpCommandChain(commandChain);

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

    // clean up history before we leave
    clean_history(&globalShellState->history);

    return 0;
}
```

Listing 5: Shell/src/main.c

```
#ifndef PARSER_H_
#define PARSER_H_

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

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

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

// Parses an array of tokens and generates a command chain, where each link is a table of commands to be executed.
CommandChain* parseTokens(char** tokens)
{
    CommandChain* chain = initCommandChain();
    if (!chain)
    {
        LOG_DEBUG("Failed to allocate memory for command chain\n");
        return NULL;
    }

    int currentIndexInTokens = 0;

    while (tokens[currentIndexInTokens] != NULL)
```

```
25      {
26          // the main loop adds commands to the chain
27          Command* command = initCommand();
28          if (!command)
29          {
30              LOG_DEBUG("Failed to allocate memory for command\n");
31              cleanUpCommandChain(chain);
32              return NULL;
33          }
34
35          // the simple commands are added to the command using this temporary
36          SimpleCommand* simpleCommand = initSimpleCommand();
37          if (!simpleCommand)
38          {
39              LOG_DEBUG("Failed to allocate memory for simple command\n");
40              cleanUpCommandChain(chain);
41              cleanUpCommand(command);
42              return NULL;
43          }
44
45          // processing the tokens, until we have a chaining operator
46          for (; !IS_NULL(tokens[currentIndexInTokens]) && !IS_CHAINING_OPERATOR(tokens[
    currentIndexInTokens]); currentIndexInTokens++)
47          {
48              if (IS_NULL(tokens[currentIndexInTokens]))
49              {
50                  // push the simpleCommand to the command's simple commands
51                  if (!simpleCommand->commandName)
52                  {
53                      LOG_DEBUG("Parse error. Null command encountered\n");
54                      cleanUpCommandChain(chain);
55                      cleanUpCommand(command);
56                      cleanUpSimpleCommand(simpleCommand);
57                      return NULL;
58                  }
59                  simpleCommand->execute = getExecutionFunction(simpleCommand->commandName)
    ;
60                  addSimpleCommand(command, simpleCommand);
61                  simpleCommand = NULL; // no more simple commands
62                  break;
63              }
64              else if (IS_PIPE(tokens[currentIndexInTokens]))
65              {
66                  // create a pipe, and update the current simple command's outputFD. push
    the simple command to the command's simple commands, and then create a new simple
    command, setting its inputFD to the pipe's read end
67
68                  // if there's two pipes in a row, or no command before the pipe, that is
    a grammar error
69                  // if there's two pipes, the current simple command will be empty
70                  if (!simpleCommand->commandName)
71                  {
72                      LOG_DEBUG("Parse error near \'%s\'\n", tokens[currentIndexInTokens]);
```

```
73                  cleanUpCommandChain(chain);           // cleans up the chain built so
    far. note that this chain does not contain the current command, and simple command
    temporaries, so we can clean them up separately
74                  cleanUpCommand(command);              // command chain link which
    hasn't been added to the chain yet, so we can clean it up separately
75                  cleanUpSimpleCommand(simpleCommand);  // probably a newly initialized
    simple command, so it hasn't been added to the command yet, so we can clean it up
    separately
76                  return NULL;
77              }
78
79              // only update the outputFD if it is not stdout, if its not stdout, that
    indicates that the simple command already has an outputFD, and we cannot pipe to
    multiple commands
80              if (simpleCommand->outputFD != STDOUT_FD)
81              {
82                  LOG_DEBUG("Parse error. Cannot pipe to multiple commands\n");
83                  cleanUpCommandChain(chain);
84                  cleanUpCommand(command);
85                  cleanUpSimpleCommand(simpleCommand);
86                  return NULL;
87              }
88
89              int pipeFD[2];
90              if (pipe(pipeFD) == -1)
91              {
92                  LOG_DEBUG("Failed to create pipe\n");
93                  cleanUpCommandChain(chain);
94                  cleanUpCommand(command);
95                  cleanUpSimpleCommand(simpleCommand);
96                  return NULL;
97              }
98
99              simpleCommand->outputFD = pipeFD[PIPE_WRITE_END];
100             simpleCommand->execute = getExecutionFunction(simpleCommand->commandName)
    ;
101             addSimpleCommand(command, simpleCommand);
102
103             // start with a new simple command
104             simpleCommand = initSimpleCommand();
105             if (!simpleCommand)
106             {
107                 LOG_DEBUG("Failed to allocate memory for simple command\n");
108                 cleanUpCommandChain(chain);
109                 cleanUpCommand(command);
110                 return NULL;
111             }
112
113             // update the new simple command's inputFD, to connect the previous
    simpleCommand and the new simpleCommand via pipe
114             simpleCommand->inputFD = pipeFD[PIPE_READ_END];
115         }
116         else if (IS_FILE_OUT_REDIR(tokens[currentIndexInTokens]))
117         {
```

```
118                 // open the file, and update the current simple command's outputFD.
119
120                 // note that the following comparison is safe, because the last token in
    the tokens array is always NULL, and the current token is not NULL, so we can safely
    access the next memory location
121                 if (!simpleCommand->commandName)
122                 {
123                     LOG_DEBUG("Parse error. Output redirection encountered before command
    \n");
124                     cleanUpCommandChain(chain);
125                     cleanUpCommand(command);
126                     cleanUpSimpleCommand(simpleCommand);
127                     return NULL;
128                 }
129
130                 // check if the current outputFD is not stdout
131                 if (simpleCommand->outputFD != STDOUT_FD)
132                 {
133                     LOG_DEBUG("Cannot redirect output to multiple files\n");
134                     cleanUpCommandChain(chain);
135                     cleanUpCommand(command);
136                     cleanUpSimpleCommand(simpleCommand);
137                     return NULL;
138                 }
139
140                 int fileFD = -1;
141                 int isAppend = 0;
142                 if (IS_APPEND(tokens[currentIndexInTokens]))
143                     isAppend = 1;
144
145                 char* fileNameToken = NULL;
146                 do {
147                     fileNameToken = tokens[++currentIndexInTokens];
148                 } while (IGNORE(fileNameToken));
149
150                 if (isAppend)
151                     fileFD = open(fileNameToken, O_WRONLY | O_CREAT | O_APPEND, 0644);
152                 else
153                     fileFD = open(fileNameToken, O_WRONLY | O_CREAT | O_TRUNC, 0644);
154
155                 if (fileFD == -1)
156                 {
157                     LOG_DEBUG("Failed to open file for output redirection\n");
158                     cleanUpCommandChain(chain);
159                     cleanUpCommand(command);
160                     cleanUpSimpleCommand(simpleCommand);
161                     return NULL;
162                 }
163
164                 simpleCommand->outputFD = fileFD;
165             }
166             else if (IS_FILE_IN_REDIR(tokens[currentIndexInTokens]))
167             {
168                 // check if the current inputFD is not stdin
```

```
169                 if (simpleCommand->inputFD != STDIN_FD)
170                 {
171                     LOG_DEBUG("Cannot redirect input from multiple files\n");
172                     cleanUpCommandChain(chain);
173                     cleanUpCommand(command);
174                     cleanUpSimpleCommand(simpleCommand);
175                     return NULL;
176                 }
177 
178                 char* fileNameToken = NULL;
179                 do {
180                     fileNameToken = tokens[++currentIndexInTokens];
181                 } while (IGNORE(fileNameToken));
182 
183                 int fileFD = open(fileNameToken, O_RDONLY);
184                 if (fileFD == -1)
185                 {
186                     LOG_DEBUG("Failed to open file for input redirection\n");
187                     cleanUpCommandChain(chain);
188                     return NULL;
189                 }
190 
191                 simpleCommand->inputFD = fileFD;
192             }
193             else if (IS_STDERR_REDIR(tokens[currentIndexInTokens]))
194             {
195                 // check if the current errFD is not stderr
196                 if (simpleCommand->stderrFD != STDERR_FD)
197                 {
198                     LOG_DEBUG("Cannot redirect stderr to multiple files\n");
199                     cleanUpCommandChain(chain);
200                     cleanUpCommand(command);
201                     cleanUpSimpleCommand(simpleCommand);
202                     return NULL;
203                 }
204 
205                 char* fileNameToken = NULL;
206                 do {
207                     fileNameToken = tokens[++currentIndexInTokens];
208                 } while (IGNORE(fileNameToken));
209 
210                 int fileFD = open(fileNameToken, O_WRONLY | O_CREAT | O_TRUNC, 0644);
211 
212                 if (fileFD == -1)
213                 {
214                     LOG_DEBUG("Failed to open file for stderr redirection\n");
215                     cleanUpCommandChain(chain);
216                     return NULL;
217                 }
218 
219                 simpleCommand->stderrFD = fileFD;
220             }
221             else if (IGNORE(tokens[currentIndexInTokens]))
222             {
```

```
223                 continue;
224             }
225             else if (!simpleCommand->commandName && tokens[currentIndexInTokens][0] == '!
    ' && strlen(tokens[currentIndexInTokens]) > 1)
226             {
227                 // pushing the '!' as history
228                 if (pushArgs("history", simpleCommand) != 0)
229                 {
230                     LOG_DEBUG("Failed to push argument to simple command\n");
231                     cleanUpCommandChain(chain);
232                     cleanUpCommand(command);
233                     cleanUpSimpleCommand(simpleCommand);
234                     return NULL;
235                 }
236 
237                 // ignore the first character
238                 if (pushArgs(tokens[currentIndexInTokens] + 1, simpleCommand) != 0)
239                 {
240                     LOG_DEBUG("Failed to push argument to simple command\n");
241                     cleanUpCommandChain(chain);
242                     cleanUpCommand(command);
243                     cleanUpSimpleCommand(simpleCommand);
244                     return NULL;
245                 }
246             }
247             else
248             {
249                 // modify the token to remove the quotes (if any)
250                 tokens[currentIndexInTokens] = removeQuotes(tokens[currentIndexInTokens])
    ;
251                 // expand any wildcards, in case there are any, if there's none return
    the same token
252                 glob_t globbuf;
253                 int globReturn = glob(tokens[currentIndexInTokens], GLOB_NOCHECK |
    GLOB_TILDE, NULL, &globbuf);
254 
255                 if (globReturn != 0)
256                 {
257                     LOG_DEBUG("Failed to expand glob\n");
258                     globfree(&globbuf);
259                     cleanUpCommandChain(chain);
260                     cleanUpCommand(command);
261                     cleanUpSimpleCommand(simpleCommand);
262                     return NULL;
263                 }
264 
265                 // if the glob was successful, then we need to push the expanded tokens
    to the args array, note if there was no expansion, then the globbuf.gl_pathc will be
    1
266                 for (size_t i = 0; i < globbuf.gl_pathc; i++)
267                 {
268                     if (pushArgs(globbuf.gl_pathv[i], simpleCommand) != 0)
269                     {
270                         LOG_DEBUG("Failed to push argument to simple command\n");
```

```
                        globfree(&globbuf);
                        cleanUpCommandChain(chain);
                        cleanUpCommand(command);
                        cleanUpSimpleCommand(simpleCommand);
                        return NULL;
                    }
                }

                globfree(&globbuf);
            }
        }

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

        // update the chain operator
        command->chainingOperator = COPY(tokens[currentIndexInTokens]);
        if (IS_BACKGROUND(command->chainingOperator))
            command->background = true;

        // add the command to the chain
        addCommandToChain(chain, command);

        // increment the counter if current token is not NULL
        if (tokens[currentIndexInTokens])
        {
            currentIndexInTokens++;
        }
    }

    return chain;
}

#endif /* PARSER_H_ */
```

Listing 6: Shell/src/parser.c

```
/**
 * @file command.h
 * @brief This file contains the structs and function declarations for the command and
   command chain structs.
 * @version 0.1
 *
 * @copyright Copyright (c) 2023
 *
 */

#ifndef COMMAND_H
#define COMMAND_H
```

```
12
13  // Includes
14  #include "utils.h"
15  #include <stdbool.h>
16  #include <unistd.h>
17
18  /**
19   * @brief This struct represents a simple command.
20   *
21   * A simple command is a command/process with its args and its set of file descriptors.
       Different simple commands can be combined together by pipes to form a pipeline. For
       example, ‘ls -l‘ is a simple command, while ‘ls -l | grep a‘ is not a simple command.
22   *
23   * IO redirection is handled by the shell, not by the command itself. So, the command
       will just have the file descriptors, and the shell will handle the redirection.
24   *
25   */
26  typedef struct SimpleCommand {
27      char* commandName; //< cmd name, e.g. ls etc.
28
29      char** args;       //< args array, including the command name
30      int argc;          //< args count, including the command name, so it's equal to the
       length of the args array
31
32      int inputFD;       //< input file descriptor, default value is 0 (stdin)
33      int outputFD;      //< output file descriptor, default value is 1 (stdout)
34      int stderrFD;      //< stderr file descriptor, default value is 2 (stderr)
35      int pid;           //< represents the processID of the child process, in case of
       external. Default is -1.
36
37      int noWait;        //< specifies that whether dont need to wait for this simple
       command to finish. default is 0, in case of a background job, it is 1
38
39      int (*execute)(struct SimpleCommand*); //< function pointer to the function that will
        execute the simple command.
40  } SimpleCommand;
41
42  /**
43   * @brief This struct represents a command, or more precisely a pipeline.
44   *
45   * A command is a set of simple commands, and it can be a pipeline of simple commands.
       For example, ‘ls -l | grep a‘ is a command.
46   * A command's grammar can be like:
47   * ‘‘‘cmd [args]* [< file] [| cmd [args]*]* [(> OR >> OR 2>) file]‘‘‘
48   *
49   */
50  typedef struct Command {
51      struct SimpleCommand** simpleCommands;  //< array to hold simple commands
52      int nSimpleCommands;                    //< number of commands
53
54      bool background;                        //< flag for background execution
55
56      char* chainingOperator;                 //< what chaining operator is used to chain
       with the next command. can be ';'/'&' but you can add more
```

```
57      struct Command* next;                   //< pointer to the next command in the chain
58  } Command;
59
60  /**
61   * @brief This struct represents a command chain (represented via a linked list).
62   *
63   * A command chain is a set of commands, separated by ; or &.
64   *
65   * A command chain's grammar can be like:
66   * Command [(; OR &) Command]*
67   */
68  typedef struct CommandChain {
69      struct Command* head;   //< pointer to the head of the command chain
70      struct Command* tail;   //< pointer to the tail of the command chain
71  } CommandChain;
72
73
74  // Function declarations
75
76
77  // ------------------------ Initializers --------------------------------
78
79  /**
80   * @brief This function creates an empty simple command, and returns a pointer to it. It
        returns NULL on failure. The caller is responsible for freeing the memory allocated
        by this function.
81   *
82   * @return SimpleCommand* Pointer to the simple command
83   */
84  SimpleCommand* initSimpleCommand();
85
86  /**
87   * @brief This function creates an empty command, and returns a pointer to it. It returns
         NULL on failure. The caller is responsible for freeing the memory allocated by this
        function.
88   *
89   * @return Command* Pointer to the command
90   */
91  Command* initCommand();
92
93  /**
94   * @brief This function creates an empty command chain, and returns a pointer to it. It
        returns NULL on failure. The caller is responsible for freeing the memory allocated
        by this function.
95   *
96   * @return CommandChain* Pointer to the command chain
97   */
98  CommandChain* initCommandChain();
99
100
101 // ------------------------ Pushers --------------------------------
102
103 /**
```

```
104  * @brief This function pushes an argument to the args array of a simple command. It
         returns 0 on success, -1 on failure.
105  *
106  * If the simpleCommand's name is not set, then it also sets the name of the simple
         command to the argument. Then it pushes to the args array, and increments the argc.
107  *
108  * @param arg Name or argument to push
109  * @param simpleCommand The simple command to push the argument to
110  * @return int Status code (0 on success, -1 on failure)
111  */
112 int pushArgs(char* arg, SimpleCommand* simpleCommand);
113
114 /**
115  * @brief This function adds a command to the command chain. It returns 0 on success, -1
         on failure.
116  *
117  * @param chain The command chain to add the command to
118  * @param command The command to add to the chain
119  * @return int Status code (0 on success, -1 on failure)
120  */
121 int addCommandToChain(CommandChain* chain, Command* command);
122
123 /**
124  * @brief This function adds a simpleCommand to a command.
125  *
126  * Uses realloc to increase the size of the simple commands array in the command, and
         adds the simple command to the array. Returns 0 on success, -1 on failure.
127  *
128  * @param command The command to add the simple command to
129  * @param simpleCommand The simple command to add to the command
130  * @return int Status code (0 on success, -1 on failure)
131  */
132 int addSimpleCommand(Command* command, SimpleCommand* simpleCommand);
133
134 // ------------------------ Cleaners --------------------------------
135
136 /**
137  * @brief The function is responsible for freeing up the memory allocated by a simple
         command. All the internal arrays and strings are freed, and the pointer is set to
         NULL.
138  *
139  * @param simpleCommand Pointer to the simpleCommand to be freed
140  */
141 void cleanUpSimpleCommand(SimpleCommand* simpleCommand);
142
143 /**
144  * @brief The function is responsible for freeing up the memory allocated by a command.
         All the internal arrays and strings are freed, and the pointer is set to NULL.
145  *
146  * @param command Pointer to the command to be freed
147  */
148 void cleanUpCommand(Command* command);
149
150 /**
```

```
151  * @brief The function is responsible for freeing up the memory allocated by a command
         chain. All the linked list nodes are freed, and the pointer is set to NULL.
152  *
153  * @param chain Pointer to the command chain to be freed
154  */
155 void cleanUpCommandChain(CommandChain* chain);
156
157 // ------------------------ Execute --------------------------------
158
159 /**
160  * @brief This function executes a chain of commands.
161  *
162  * Function traverses the linked list, and calls executeCommand on each command. The
         rules for executing a command chain are:
163  * 1. If the chaining operator is ';', then execute all commands in the chain, and return
          the exit status of the last command.
164  * 2. If the chaining operator is '&', then the setup was such that the pipeline was set
         to run in the background.
165  *
166  * @param chain The command chain to execute
167  * @return int Status code (exit status of the last command according to the rules above)
168  */
169 int executeCommandChain(CommandChain* chain);
170
171 /**
172  * @brief This function executes a command. The function traverses the simple commands in
          the command, and executes them one by one.
173  *
174  * @param command The command to execute
175  * @return int Status code (exit status of the last command)
176  */
177 int executeCommand(Command* command);
178
179 // ------------------------ Debug --------------------------------
180
181 /**
182  * @brief This function prints the command chain in a readable format. Purely a debug
         utility.
183  *
184  * @param chain The command chain to print
185  * @return void
186  */
187 void printCommandChain(CommandChain* chain);
188
189 /**
190  * @brief This function prints the a simpleCommand in a readable format for debugging.
         Purely a debug utility.
191  *
192  * @param simpleCommand The simple command to print
193  * @return void
194  */
195 void printSimpleCommand(SimpleCommand* simpleCommand);
196
197 #endif // COMMAND_H
```

Listing 7: Shell/include/command.h

```
/**
 * @file parser.h
 * @brief Contains useful macros and the definition for the main parser.
 * @version 0.1
 *
 * @copyright Copyright (c) 2023
 *
 */

#ifndef PARSER_H
#define PARSER_H

#include "command.h"

// Useful macros for readability

// macro to test if a token is a chaining operator. the chaining operators are &, ;.
    macro resolves to 1 if the token is a chaining operator, 0 otherwise
#define IS_CHAINING_OPERATOR(token) (strcmp(token, "&") == 0 || strcmp(token, ";") == 0)
// checks if the token is a background operator
#define IS_BACKGROUND(token) (token && strcmp(token, "&") == 0)
// check if the token is a pipe
#define IS_PIPE(token) (strcmp(token, "|") == 0)
// check if the token is file output redirection operator
#define IS_FILE_OUT_REDIR(token) (strcmp(token, ">") == 0 || strcmp(token, ">>") == 0)
// check if the token is file input redirection operator
#define IS_FILE_IN_REDIR(token) (strcmp(token, "<") == 0)
// check if the token is stderr redirection operator
#define IS_STDERR_REDIR(token) (strcmp(token, "2>") == 0)
// check if the token is NULL
#define IS_NULL(token) (!token)
// check if the token is ignorable
#define IGNORE(token) (token && (strcmp(token, " ") == 0 || strcmp(token, "\t") == 0 ||
    strcmp(token, "\n") == 0 || strcmp(token, "") == 0))
// check if the token is the append operator
#define IS_APPEND(token) (strcmp(token, ">>") == 0)

/**
 * @brief Parses the tokens and returns a command chain. It is the responsibility of the
    caller to free the memory.
 *
 * @param tokens The tokens to parse. Assumes that the tokens array is null terminated.
 * @return CommandChain* The command chain that was parsed.
 */
CommandChain* parseTokens(char** tokens);

#endif // PARSER_H
```

Listing 8: Shell/include/parser.h

```
/**
```

```
 * @file builtins.h
 * @brief Inlcudes structures and utilities to manage the builtins and the internal state
    of the shell.
 * @version 0.1
 *
 * @copyright Copyright (c) 2023
 *
 */

#ifndef BUILTINS_H
#define BUILTINS_H

#include "command.h"

#define HOME_DIR getenv("HOME")
#define MAX_PATH_LENGTH 1024

// a linked list to represent shell history
typedef struct HistoryNode {
    char* command;
    struct HistoryNode* next;
} HistoryNode;

typedef struct HistoryList {
    HistoryNode* head;
    HistoryNode* tail;
    size_t size;
} HistoryList;

// adds a command to the history list
int add_to_history(HistoryList* list, char* command);

// retrieves a command at a particular index from the list
char* get_command(HistoryList* list, unsigned int index);

// clean up history
void clean_history(HistoryList* list);

// finds the last command that starts with the prefix
char* find_last_command_with_prefix(HistoryList* list, const char* prefix);

// To represent the state of the shell.
typedef struct ShellState {

    // these holds the original shell stdin, stdout and stderr
    int originalStdoutFD;
    int originalStdinFD;
    int originalStderrFD;

    // shell variable that holds the current prompt
    char prompt_buffer[MAX_STRING_LENGTH];

    // represents the history node list, storing tail for quick insertions
    HistoryList history;
```

```
55 } ShellState;
56
57 // initializes the shell state
58 ShellState* init_shell_state();
59 // cleans things up and frees memory
60 int clear_shell_state(ShellState* stateObj);
61
62 typedef int (*ExecutionFunction)(SimpleCommand*);
63
64 /**
65  * @brief Returns the execution function for the given command.
66  *
67  * @param commandName The name of the command.
68  * @return ExecutionFunction The execution function for the given command.
69 */
70 ExecutionFunction getExecutionFunction(char* commandName);
71
72 /**
73  * @brief This function is the builtin for the cd command.
74  *
75  * @param command The command to be executed.
76  * @return int Returns 0 on success, -1 on failure.
77  */
78 int cd(SimpleCommand* command);
79
80 /**
81  * @brief This function is the builtin for the exit command.
82  *
83  * @param command The command to be executed.
84  * @return int Returns 0 on success, -1 on failure.
85  */
86 int exitShell(SimpleCommand* command);
87
88 /**
89  * @brief This function is the builtin for the pwd command.
90  *
91  * @param command The command to be executed.
92  * @return int Returns 0 on success, -1 on failure.
93  */
94 int pwd(SimpleCommand* command);
95
96 /**
97  * @brief This function is the builtin for the history command.
98  *
99  * @param command The command to be executed.
100  * @return int Returns 0 on success, -1 on failure.
101  */
102 int history(SimpleCommand* command);
103
104 /**
105  * @brief This function executes a process.
106  *
107  * The process is executed by forking a child process, and then executing the command in
     the child process.
```

```
 *
 * @param command The command to be executed.
 * @return int Returns non-zero status on failue. else returns 0 on success
 */
int executeProcess(SimpleCommand* command);

#endif // BUILTINS_H
```

Listing 9: Shell/include/shell_builtins.h

```
/**
 * @file log.h
 * @brief A collection of macros and functions to facilitate logging in a helpful manner, instead of using plain printf statements.
 * @version 0.1
 *
 * @copyright Copyright (c) 2023
 */

#ifndef LOG_H
#define LOG_H

#include <stdio.h>

// Log coloring
#define LOG_RESET   "\033[0m"
#define LOG_RED     "\033[1;31m"
#define LOG_GREEN   "\033[1;32m"
#define LOG_YELLOW  "\033[1;33m"
#define LOG_BLUE    "\033[1;34m"
#define LOG_CYAN    "\033[1;36m"
#define LOG_WHITE   "\033[1;37m"

// defines for logging modes
#define LOG_ERR     0  /* For printing critical errors, always get printed */
#define LOG_DBG     1  /* For printing debug print statements, only in debug mode */
#define LOG_PRI     2  /* For normal printing of messages, always get printed without annotations */

// Default colors for different log types
#define LOG_COLOR_ERR   LOG_RED
#define LOG_COLOR_DBG   LOG_CYAN
#define LOG_COLOR_PRI   LOG_WHITE

// debug mode, debug messages are printed only if this is set to 1 (debug is provided as a compiler flag)
#ifdef DEBUG
#define ANNOTATIONS 1      /* Change this to zero to disable all annotations even in debug mode */
#else
#define DEBUG 0
#define ANNOTATIONS 0
#endif

// Macros to change the behavior of annotations
```

```
#define ANNOTATIONS_INFO 1  /* Change this to zero to disable annotations info */

#define ANNOTATIONS_FILE 0
#define ANNOTATIONS_FUNC 1
#define ANNOTATIONS_LINE 1

// output function for printing, default is printf
#define LOG_OUT(...) printf(__VA_ARGS__)

// defines the annotation string.
#define ANNOTATION_INFO_STRING do {\
    if (ANNOTATIONS_INFO) {\
        LOG_OUT(" ("); \
        if (ANNOTATIONS_FILE) printf("%s", __FILE__); \
        if (ANNOTATIONS_FILE) printf(","); \
        if (ANNOTATIONS_FUNC) printf("%s", __func__); \
        if (ANNOTATIONS_FUNC) printf(","); \
        if (ANNOTATIONS_LINE) printf("%d", __LINE__); \
        LOG_OUT(") ");\
    }\
} while (0)

// macro to log a message.
#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 10: Shell/include/log.h

```
/**
 * @file utils.h
 * @brief Contains useful macros and utilities to be used by the shell.
 * @version 0.1
 *
 * @copyright Copyright (c) 2023
 *
 */

#ifndef UTILS_H
#define UTILS_H

#include "log.h"

#include <string.h>
```

```c
#include <stdlib.h>

// Exposed macros for logging
#define LOG_ERROR(...) LOG(LOG_ERR, "[ERROR]", LOG_COLOR_ERR, __VA_ARGS__)
#define LOG_DEBUG(...) LOG(LOG_DBG, "[DEBUG]", LOG_COLOR_DBG, __VA_ARGS__)
#define LOG_PRINT(...) LOG(LOG_PRI, "[PRINT]", LOG_COLOR_PRI, __VA_ARGS__)

// we specify that the strings in our program won't exceed length of 100 characters
#define MAX_STRING_LENGTH 1024

// in order to be consistent, let's just define a macro for copying strings
#define COPY(str) (str ? strndup(str, MAX_STRING_LENGTH) : NULL)

// Useful macros for file descriptors to make the code more readable
#define STDIN_FD 0
#define STDOUT_FD 1
#define STDERR_FD 2
#define PIPE_READ_END 0
#define PIPE_WRITE_END 1

/**
 * @brief This function tokenizes a string, given a delimiter.
 *
 * It returns an array of tokens, and the number of tokens in the array. The function ignores any delimiter encountered inside quotes.
 * It is the responsibility of the caller to free the memory via the freeTokens() function.
 *
 * @param str String to tokenize
 * @param delimiter Delimiter to use for tokenization
 * @return char** Array of tokens (NULL terminated)
 */
char** tokenizeString(const char* str, const char delimiter);

/**
 * @brief This function frees the memory allocated by tokenizeString()
 *
 * @param tokens Array of tokens
 */
void freeTokens(char** tokens);

/**
 * @brief This function removes the quotes from a string. If the string is not quoted, it returns the same string. Otherwise, the current string is freed, and a pointer to the new string is returned.
 *
 * @param str String to remove quotes from
 * @return char* Pointer to the new string
 */
char* removeQuotes(char* str);

/**
 * @brief Get the Tokens count
 *
```

```
 * @param tokens The NULL terminated array of tokens
 * @return int Number of tokens in the array
 */
int getTokenCount(char** tokens);

#endif // UTILS_H
```

Listing 11: Shell/include/utils.h

```
#!/usr/bin/python3

import subprocess
import json
from rich.console import Console
from rich.table import Table
import shutil
import time
import sys
import os
import datetime

class Test:
    """
    The Test class is the main class that runs the tests.

    All the configuration params are loaded from a json file.
    """
    def __init__(self, config="config.json"):
        """Initializes the test environment with the params defined in the configuration
    json file.

        Args:
            config (str, optional): The json file with the params. Defaults to "config.
    json".
        """

        config = json.load(open(config, "r"))

        self.test_directory  = config["test_directory"]
        self.default_tests   = config["default_tests"]
        self.test_files      = config["test_files"]
        self.shell           = config["shell"]
        self.reference_shell = config["reference_shell"]
        self.default_timeout = config["default_timeout"]
        self.log_to_file     = config["log_to_file"]
        self.tests_weightage = config["weightage"]
        self.verbose         = config["verbose"]

        self.score        = 0
        self.total_tests  = 0
        self.tests_passed = 0
        self.tests_failed = 0

        self.tests = self.default_tests

```

```
45          self.temp_output_dir = os.path.join(self.test_directory, "test_output")
46
47      def print_config(self):
48          """Prints the configuration params.
49          """
50          print("Traces Directory: ", self.test_directory)
51          print("Tests: ", ", ".join(self.tests))
52          print("Shell: ", self.shell)
53          print("Reference Shell: ", self.reference_shell)
54          print("Default Timeout: ", str(self.default_timeout))
55          print("Log To File: ", str(self.log_to_file))
56          print()
57
58      def set_tests(self, tests):
59          """Sets the tests to run.
60
61          Args:
62              tests (list): A list of tests to run. Possible options: easy, medium,
    advanced
63          """
64
65          # make sure the list is correct
66          for test in tests:
67              if test not in self.default_tests:
68                  print(f"ERROR : Test {test} does not exist")
69                  print("Using default tests")
70                  return
71
72          self.tests = tests
73
74      def execute_command(self, test_file):
75          """Runs the testing "scripts" which are basically a collection of commands, on
    both shells, the student shell, as well as the reference shell, and dumps the outpus
    to their respective .out files in the outputs folder.
76
77          Args:
78              test_file (str): The name of the test file being tested right now.
79
80          Returns:
81              Boolean: True if the shells executed correctly, False otherwise.
82          """
83
84          test_shell_file = self.test_directory + "/" + test_file
85          ref_shell_file = self.test_directory + "/" + test_file
86
87          ref_specific = test_shell_file + ".custom"
88
89          if os.path.exists(ref_specific):
90              ref_shell_file = ref_specific
91
92          test_cmd = f"{self.shell} {test_shell_file} > {self.temp_output_dir}/{test_file}.
    msh.out 2> /dev/null"
93          expected_cmd = f"{self.reference_shell} {ref_shell_file} > {self.temp_output_dir
    }/{test_file}.{self.reference_shell.split('/')[-1]}.out 2> /dev/null"
```

```
94
95          try:
96              proc = subprocess.run(test_cmd, shell=True, check=True, timeout=self.
    default_timeout)
97          except subprocess.TimeoutExpired:
98              print("FAILED")
99              print(f"\t=> Your shell timed out")
100             return False
101         except subprocess.CalledProcessError:
102             print("FAILED")
103             print(f"\t=> Your shell exited with non-zero return code. Possible segfault."
    )
104             return False
105         except:
106             print("FAILED")
107             print(f"\t=> Your shell failed to execute the file")
108             return False
109
110         try:
111             proc = subprocess.run(expected_cmd, shell=True, timeout=self.default_timeout)
112         except Exception as error:
113             print("FAILED")
114             print(f"\t=> {self.reference_shell} failed to execute the file {test_cmd}")
115             print(f"\t=> {error}")
116             return False
117
118         return True
119
120     def test_validity(self, test_file):
121         """Tests whether the output generated by the student shell is correct or not.
    This is done by comparing it to the referene shell output.
122
123         Args:
124             test_file (str): The test file being tested right now.
125
126         Returns:
127             Boolean: True if the output is correct, False otherwise.
128         """
129
130         test_shell_out = self.temp_output_dir + "/" + test_file + ".msh.out"
131         ref_shell_out = f"{self.temp_output_dir}/{test_file}.{self.reference_shell.split
    ('/')[-1]}.out"
132
133         # read in lines from both files
134         try:
135             with open(test_shell_out, "r") as file:
136                 test_out_lines = file.readlines()
137         except:
138             print("FAILED")
139             print(f"\t=> Your shell failed to execute the command")
140             return False
141
142         with open(ref_shell_out, "r") as file:
143             ref_out_lines = file.readlines()
```

```
144
145         # strip everything in each list
146         test_out_lines = list(map(lambda x: x.strip(), test_out_lines))
147         ref_out_lines = list(map(lambda x: x.strip(), ref_out_lines))
148
149         for line in ref_out_lines:
150             if line not in test_out_lines:
151                 print("FAILED")
152                 print(f"\t=> \"{line}\" not found in your shell's output")
153                 return False
154
155         print("PASSED")
156         return True
157
158     def run_test(self, test_name):
159         """Runs a test.
160
161         Args:
162             test_name (str): The name of the test to run.
163         """
164
165         total_ran    = 0
166         total_passed = 0
167
168         command_files = self.test_files[test_name]
169
170         for file in command_files:
171
172             print(f"  Running file: {file:<25}", end="")
173
174             if self.execute_command(file) and self.test_validity(file):
175                 total_passed += 1
176
177             total_ran += 1
178
179         return total_ran, total_passed
180
181     def run(self):
182         """Main function of the testing class. Runs the test suite according to how its
    defined in the configuration file.
183         """
184
185         print("\n\nSHELL TEST SUITE\n")
186         self.print_config()
187
188         if os.path.exists("../build/build_mode"):
189             # open the file and see if the build_mode is release, only allow testing in
    build mode
190             with open("../build/build_mode", "r") as file:
191                 build_mode = file.read().strip()
192                 if build_mode != "release":
193                     print("ERROR : Build mode is not release")
194                     print("Please build in release mode to run the tests")
195                     return
```

```
196
197         if not os.path.exists(self.shell):
198             print(f"ERROR : Shell {self.shell} does not exist")
199             return
200
201         if os.path.exists(self.temp_output_dir):
202             shutil.rmtree(self.temp_output_dir)
203
204         os.mkdir(self.temp_output_dir)
205
206         with open(f"{self.temp_output_dir}/timestamp.log", "w") as file:
207             file.write(str(datetime.datetime.now()))
208
209         print("...\n")
210
211         for test in self.tests:
212
213             print(f"Running tests : {test}")
214
215             total, passed = self.run_test(test)
216
217             self.total_tests  += total
218             self.tests_passed += passed
219             self.tests_failed += (total - passed)
220
221             if total == 0:
222                 continue
223
224             self.score += self.tests_weightage[test] * (passed / total)
225
226
227         print("\n...\n")
228         print("SUMMARY\n")
229         print(f"Total : {self.total_tests}, Passed : {self.tests_passed}, Failed : {self.
    tests_failed}")
230
231 if __name__ == "__main__":
232
233     # the tests to run are passed via the command line args, if none is passed use the
    default tests
234
235     test = Test()
236     start = time.time()
237     test.run()
238     end = time.time()
239     print(f"Finished in {end - start:<.2f}s.")
```

Listing 12: Shell/test/test.py