Developer documentation

Áron Pákozdi

December 2024

1 Structure

My program has a modular structure, each source file has its reason and application. Because my program only works with strings and integers, I only used dynamic arrays to store data. I also used a macro, to store the delimeter, for parsing. The source files:

- main.c: decides wether the shell should run in interactive or script mode and calls that function.
- shell_interactive.c: includes the main loop of the REPL environment. As a REPL, it operates the reading, executing and outputing in a loop.
- prompt.c: prints the prompt to indicate to the user, that the shell is waiting for input.
- read_line.c: reads and stores the input of the user.
- parse.c: Parses the line to arguments.
- handle_input.c: operates the execution of the given commands. Also contains the built-in functions.
- execute_command.c: executes the non-built-in commands.
- redirect.c: operates the I/O redirections of the input, if there is any. Also contains the argument shifting function.
- stdout.c: Includes the stdout write and stdout append redirector functions.
- stdin.c: Includes the two stdin redirector functions
- stderr.c: Includes the two stderr redirector functions.
- pipe.c: Includes the piping function.
- shell_non_interactive.c: Operates the scripting mode of the shell. Calls the appropriate functions. Behaves very similarly to the shell_interactive.c file.

- read_stream.c: reads the input in non-interactive mode.
- shell.h: My only header file. Connects all the files and contains all the prototypes and imports.

2 Functions

- int main(): calls shell_interactive or shell_non_interactive, depending on the use of the program. Returns 0 if everything went well.
- void shell_interactive(): handling the shell, while running in interactive mode. Enter a loop, which runs until exited or an error occurs.
- void shell_non_interactive(): handling the shell, while running in non-interactive mode.
- void get_prompt(): prints the prompt for the user.
- char* read_line(): Reads the input using the getline() function, returns a dynamically allocated string. If something went wrong, it prints it to the terminal and exits the program.
- char* read_stream(): Similarly to the read_line it reads the input, but its a bit different, because of the difference between interactive and non-interactive mode. It uses the getchar() function. Also returns a line, which is a dynamically allocated string.
- char** parse_line(char* line): gets a string as input (the inputted line), and tokenizes it by the delimeter. It uses the strtok() function. Returns a dynamically allocated array of strings.
- int handle_input(char** args): Gets the tokenized line as input and returns the status of execution. 0 if success, else if failure. I used function pointers for stroring built-in commands, instead of if else statements, because of scalability. Calls redirect_checker and the corresponding built-in function or execute_commands()
- int redirect_checker(char** args): Gets the argument list as input, checks if there are any redirectors in the given input. If there is, then calls the corresponding function. returns the status of execution (0 if success). I also used function pointers for scalabilty.
- void shift_args_left(char **args, int index): deletes from the args input at the given index and the next one. Used to delete redirectors.
- The redirector functions work quite similarly. They recieve a list of strings and an integer. The file handling is done by the file descriptor. They only open it differently. The functions returns the status of success. I used dup2() function for redirection. They also use shift_args_left() function.

There are some mysterious problems with them, some I could not figure out.

- int pipe_function(char** args, int index): also quite similar, but a little bit more complicated, because of the different processes. I used the fork() function for forking, and the shift_args_left() function.
- int execute_command(char** args): executes the non-built in commands. The ones, which are already implemented in the unix based systems root directory. It uses the execvp() function to execute those commands. Gets the line as input (array of strings) and returns the status of execution. (0 on success)
- int my_exit(char** args): exits the program. input and output doesn't matter. it is only there because of consistency with the function pointer.
- int my_echo(char** args): my implementation of echo function. Prints the given input to the stdout. Gets the arguments as input and returns status.
- my_cd(char** args): my implementation of the cd command. It changes directory. It may not work on every device, because of getenv() function. Some devices my not have HOME set. Gets the args as input and returns status of execution.
- int my_pwd(char** args): my implementation of the pwd command. It prints the current working directory to the stdout. It uses the getcwd() function.