

oshell

Implementation of a Unix Inspired Shell in C11

Project Report

Author: Ole Wortmann

Date: December 19, 2025

Abstract

The topic of the following project report covers the development of a simple shell written in C11. It covers planning, essential terminology and concepts, such as the implementation and testing of the software.

Keywords: C11, BASH, SHELL, SYSTEMS PROGRAMMING, I/O REDIRECTION, UNIX,
COMMAND PARSING, CHILD MANAGEMENT, JOB MANAGEMENT

Contents

1	Introduction	1
2	Planning	1
3	Essential Terminology and Concepts	1
4	Design	2
4.0.1	Memory Management	2
4.0.2	Error Handling	3
5	The Implementation	4
5.1	Getting the input	4
5.2	Parsing the user input	5
5.2.1	Removing quotes and building a new string	7
5.3	Executing Commands	7
5.4	Builtin Commands and functionalities	7
5.4.1	CD - Change Directory	7
5.4.2	Kill - Killing a process by name or ID	8
5.4.3	Clear - Clearing the terminal	9
5.4.4	Piping	10
5.4.5	Input Redirection (<)	11
5.4.6	Output Redirection (>, >>)	11
5.4.7	Background Execution &	12
5.4.8	Command chaining (&&)	12
5.5	Environment Variables	13
5.6	Input loop	13
6	Testing	14
7	Evaluation	15
	Bibliographie	16
	Appendix	
	Parsing - Remove quotes function	
	Execute Command - Change Directory	
	Execute Command - Kill	
	Execute Command - Input Redirection	
	Execute Command - Output Redirection	
	Utils - Find a shell operator	
	Parsing Utils - Build a new string	
	Piping	

1 Introduction

The following is a project report for a small Unix-based shell, it was not created in any official context and should be treated as such.

This project came to life because because of the search for a new project that was not game development, had something to do with systems but was not as complicated as making an Operating system. Tho this may very well be a step in the direction of starting developing one, there are no plans as of right now.

This is neither something new nor outstanding but just a fun project that took way longer then planned, because alot of edgecases or “obvious” failure points, that came to light when testing.

It also does not implement any new ideas, but builds on a specification sheet. It also mostly follows the bash shell way of commands, tho keep in mind that this is really simple, and advanced bash commands might not work. **OSHELL** only supports command chaining by `&&`.

2 Planning

Since building a (*small*) shell is not unusual for a University project, the assumption was that there would be some article or specification or anything that could give a sense of direction and ease the planning. Luckily a professor at Florida State University (FSU) had posted a specification for one of his classes (<https://www.cs.fsu.edu/~cop4610t/assignments/project1/writeup/specification.pdf>). A copy is provided in the github repository. The specification provided a clear roadmap to follow, outlining features and functionalities to implement.

This is most likely not the most efficient or effective way this could have been implemented, nontheless this is a medium advanced project.

3 Essential Terminology and Concepts

This will explain some basic terminology like **STDIN/STDOUT**, file descriptors and other terms that will come up in this report.

- **STDIN**: Standard input (integer value 0), comes from a “standard input device” which is usually (but not limited to) the keyboard, but could also be a `file.txt`.
- **STDOUT**: Standard output (integer value 1), either goes to the terminal or an X terminal, this entirely depends on where the process originated.
- **File Descriptors**: “A file descriptor is an unsigned integer used by a process to identify an open file” [1]. An open file could be a normal (text) file or **STDIN/STDOUT** [2].

- Pipes: Pipes are point-to-point, half duplex interprocess communication files (in linux everything is a file see [3]). They connect one write end to another read end.

4 Design

OSHELL follows this really simple **read-parse-execute** loop, Figure 1 displays this nicely. Input is parsed, and then passed to the `execute_command()` function. This is really the most obvious way to implement the high level behavior of a shell, since it is always running, and always needs to do these three things.

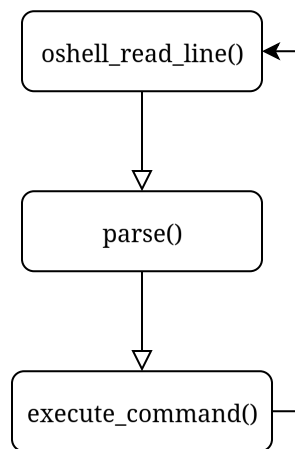


Figure 1: Figure visualising the main flow

The **input handler** passes the string without modification to the **parser**, which parses the string, tokenizes it, and removes excess quotes. The command execution function then checks for different character “operators” in the parsed string, to then trigger different functions based on the input.

Figure 2 shows a simplified flowchart of what the execution looks like.

The specification sheet, specified the commands which were to be implemented as inbuilt (although not all have been).

4.0.1 Memory Management

Dynamic memory allocation in C always bears risk. If one can it should be avoided, but especially with something like a shell where you never know how big the input is, dynamic allocation is needed. One extreme example would be the 3d dynamic array in `pipng.c`. In this case one has to free every pointer to the array `[cmd_id][cmd_arg]` every array `[cmd_id]` and every pointer to the “string” commands. Figure 3 visualises the data structure.

A more common way that was used in the `kill_process()` function (See Listing 13) is using a `goto` statement, that just checks what files or folders are open to close them,

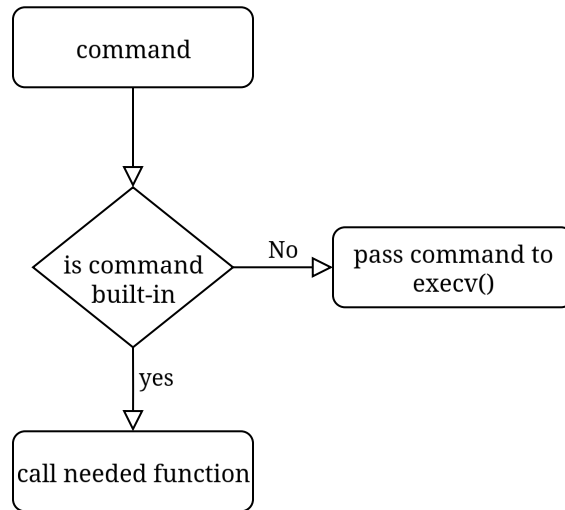


Figure 2: Figure visualising the flow of command execution (simplified)

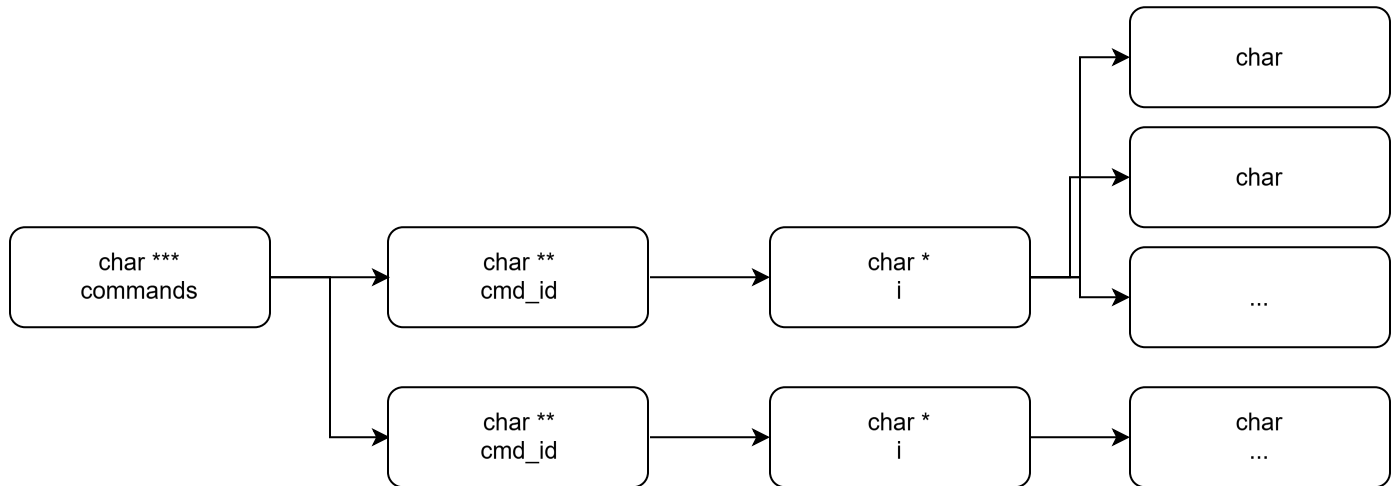


Figure 3: visualisation of 3d dynamic array

or what memory is allocated and can be freed. This is really nice since the kill function has many points where it can fail.

4.0.2 Error Handling

The approach for the error handling is to never exit the shell if it is not absolutely necessary. The cases where the `exit()` function would be called are absolute edgecases that should never happen, except the exit command. Normally you will just be sent back to the shell to re-enter the command. It also tries to provide user-friendly error messages, especially if the user might be new to the commandline / shell usage. One

example of good error handling would be the `cd` function.

```
oshell> cd ~/noExist
change_directory() error: No such file or directory
To avoid this error make sure this directory you want to navigate to exists.
    directory and Directory are not the same.
```

5 The Implementation

5.1 Getting the input

C provides several functions for capturing the user input (STDIN) from the commandline, including `fgets()` (<https://linux.die.net/man/3/fgets>), and `getline()` (<https://linux.die.net/man/3/getline>). While both are valid options to capture user input, `getline()` is (in this case) superior.

While initially `fgets()` seems like a valid approach, it stops when reading a new-line character (`'\n'`) and also does not handle memory management by itself [4]. The `getline()` function does all this, it reads line-by-line, includes the newline character, null-terminates the buffer and handles the memory management for the buffer dynamically. Which makes the shell less prone to memory-related bugs.

```
char *oshell_read_line(void)
{
    char *line = NULL;
    size_t buffsize = 0;

    if(getline(&line, &buffsize, stdin) == -1)
    {
        if(feof(stdin))
        {
            free(line);
        } else {
            perror("oshell: readline");
            free(line);
            exit(EXIT_FAILURE);
        }
    }
    return line;
}
```

Listing 1: Implementation of the `oshell_read_line()` function

5.2 Parsing the user input

Parsing the input for a small/basic shell might not seem like a huge endeavour, just look at the first part of the input, as the command and the second as the argument, pass that to `execv()` (<https://linux.die.net/man/3/execv>) and you are done. This works for something like `echo "hello world"`, but this will fall apart quite fast for anything even slightly more complex e.g.

```
echo "foo bar baz" | wc -w
```

Expected Output:

3

Because now the pipe operator (`|`) needs to be dealt with (see Section 5.4.4). To successfully parse the user input we need to:

1. Tokenize

- What are tokens? Tokens are the smallest “unit” that the shell can work with. When looking at a command like `ls -l /home/user` the tokens would be `[ls]` `[-l]` `[/home/user]`.
- Why is tokenisation necessary? Tokenization is needed because the `execv()` function that **OSHELL** uses to execute the commands, takes the `path` and the `argv` separate. So in the `parse()` function the string is tokenised, and prepared for execution.
- How does the tokenisation work? When tokenising **OSHELL** loops through the input string and checks for every character if it is a space or a null-terminator (`'\0'`). They act as so called **delimiters**. “A delimiter is one or more characters that separate text strings.” [5]. If the character to be replaced is not the first character, it is replaced with the (`'\0'`), to terminate the string (See Listing 2 for example). The address of the beginning of the new string is then saved to a `tokens` array, and if needed the memory will be reallocated to fit the string.

```
oshell> echo test
test
```

Listing 2: example shell input

2. If the last character is a newline, replace it with null-terminator

```
char *mod_str = args[i];
if (strlen(mod_str) > 0 && mod_str[strlen(mod_str) - 1] == '\n') {
    mod_str[strlen(mod_str) - 1] = '\0';
}
```

Listing 3: Removing the trailing newline during tokenization

If the last character of the last token (`mod_str[strlen(mod_str) - 1]`) is a newline (`'\n'`), it gets replaced with a null-terminator (`'\0'`). This can effectively “shorten” the string. Which would look something like this:

```
foobar\n\0 → foobar\0\0 → foobar\0
```

This sanitization is necessary because some programs may behave incorrectly or crash if they receive strings with embedded newline characters.

3. Removing the outer quotes after tokenization is necessary so that the command and thus the output (mostly) behaves like assumed. This is a “very basic” implementation of parsing quotes and building new strings. As of writing this **OSHELL** does not behave 1:1 like **BASH** or **ZSH**, this can be seen with the parsing and handling of newline characters inside quotes. Most simple example would be `echo` and `sort` e.g.

```
echo "3\n1\n2" | sort -n
```

Expected Output

```
tuxpad% echo "3\n1\n2" | sort -n
1
2
3
```

But **OSHELL** will just parrot the string back to you, because it does not handle the newline character inside quotes as the newline character but just as a string literal.

```
echo "3\n1\n2" | sort -n
3\n1\n2
```

providing `echo` with the `-e` flag, will yield the correct output.

```
echo -e "3\n1\n2" | sort -n
1
2
3
```

Using `echo "test123" >> append.txt` as an example, if during parsing the quotes are not removed the quotes will be appended to the file so `cat append.txt` would return

```
"test123"
```

instead of

```
test123
```

5.2.1 Removing quotes and building a new string

The `remove_quotes()` function (see Listing 11), loops through every word, and letter, to find either a first single or double quote. Upon finding one, it logs the first word and letter position and the last word and letter position (See Listing 11 lines 28 - 46). If a closing quote is found, the `build_quote_string()` function is called with the following parameters provided:

1. `arg`: The `char **string` array.
2. `start_arg`: The position of the starting string.
3. `start_pos`: The position of the starting character.
4. `end_arg`: The position of the ending string.
5. `end_pos`: The position of the ending character.

This then just turns the multiple arguments into one string without the quotes, null-terminating the string (See Listing 17). It also directly shows why the string is passed to the function as an array, to be able to “delete” the quotes, one needs to access the individual characters.

5.3 Executing Commands

The `execute_command()` function uses the return value `char **args` from the `parse()` function which holds the command name, and the arguments e.g. `echo "hello world"`. It first checks for the inbuilt commands such as, `exit` which exits the shell, `help` which prints help text, `cd` which can change the directory or `kill`, which kills a process based on the PID or `process name`. `OSHELL` can also do piping, output and input redirection, this is all handled internally before spawning any external processes.

If the command or the arguments do not call any of the internal functions, `OSHELL` builds the command with the proper structure, and creates a fork to try and execute the command.

The following subsections will go into more detail about how each function/system works.

5.4 Builtin Commands and functionalities

5.4.1 CD - Change Directory

The inbuilt commands all follow the same pattern, `strcmp()` checks if the command (`args[0]`) is equal to any of the inbuilt commands. Should this be the case it calls the correct function.

E.g. “cd” has two cases:

1. User wants to navigate to directory X. `args[1]` is not NULL meaning there is something behind `cd` ideally a path to a directory

2. User wants to take the shortcut to `$HOME`. `args[1]` is `NULL`.

```
if (strcmp("cd", command) == 0)
{
    if(args[1] != NULL) {
        int status = change_directory(args[1]);
        if (status != 0) {
            perror("oshell: change_directory() error");
            return 1;
        }
    } else {
        chdir(getenv("HOME"));
    }
    return 0;
}
```

Listing 4: sourcecode showing the 'cd' implementation of OSHELL

Depending on if `args[1]` is `NULL` or not the `change_directory()` (see Listing 11) function is called and the wanted directory name is passed to it. If the user enters a directory name that does not exists in the directory the user is currently in, an error is thrown and the user is sent back to the commandline. Depending on if the first character after the `cd` command is a tilde (`~`), a new path is constructed with `$HOME/user/some/new/path` because the `chdir()` function, **OSHELL** uses to actually change the directories, does not use relative paths with tilde, but only absolute paths.

5.4.2 Kill - Killing a process by name or ID

Killing a process by ID or NAME is done through the same function (`kill_process()`). This function takes a `char *process_name_or_id`, and utilizes a helper function called `string_to_int()` that can parse the `process_name_or_id` into an integer if the string represents a valid number. Based on the return value of the `string_to_int()` function there can be two cases:

1. User submitted a valid integer
2. User submitted a string

In the event that the user submitted an `int` it will be cast to `pid_t` and passed to the `kill()` (<https://linux.die.net/man/1/kill>) function.

In the event that the user submitted a string, the function will try and get the PID it does this the following way.

On unix systems there is a pseudo-filesystem called `/proc/` which acts like an **API** to kernel data structures. It holds a subdirectory to each process that is currently running. Each of those holds process specific files, like the `comm` file [6] [7]. The `comm` file stores the process name e.g. `cat/<PID>/comm` could return "zsh", and has the PID of the running

process, as the filename. When looping through `/proc/<PID>` the program checks whether the directory name is all integers, if so it constructs a path to said folder.

```
snprintf(full_proc_path, full_proc_pathLength, "%s%s", proc_dir_name,
↪ dirent->d_name);
```

Else it just takes the PID provided and tries to kill the process.

```
if(strcmp(strip_non_alpha(buffer), strip_non_alpha(process_name_or_id)) == 0) {
    if(kill(pid, 9) == 0) {
        fprintf(stderr, "killed %s with PID: %i", process_name_or_id, pid);
        fflush(stderr);
        process_found = true;
    }
    goto cleanup;
}
```

Listing 5: sourcecode showing comparison between file contents (buffer) and the user input

Before validating if this is the correct process the PID (directory name) is saved, so that it can be later used to kill said process if it is the correct one. A new path to the `comm` file is then created to be able to read the file contents. If the contents and the user input match the process is killed.

5.4.3 Clear - Clearing the terminal

The `clear()` function does not actually erase the content. Instead, this function prints enough newline characters to move the existing text beyond the visible screen area. The previous content remains in memory and can be accessed by scrolling up.

It works by calling the `ioctl()` function, with the open file descriptor, the request type and in this case to store the output in the `winsize` structure (`w`). The 0 (STDIN) tells the `ioctl()` function that the terminal it wants information about is the one currently being used.

The `TIOCGWINSZ` is a request that returns the current window size [8]. It then loops over the amount of rows the terminal currently has, and prints a newline character each time, after that the cursor is being moved to the top left.

```
void clear(void)
{
    struct winsize w;
    ioctl(0, TIOCGWINSZ, &w);

    for (int i = 0; i < w.ws_row; i++) {
        fprintf(stderr, "\n");
    }
}
```

```

    }
    fprintf(stderr, "\033[0;0H");
}

```

Listing 6: sourcecode showing implementation of the `clear()` function

5.4.4 Piping

The implementation of piping (See Listing 18) was one of the hardest functionalities to implement. A pipe is a point-to-point connection that has one read-end and one write-end. It allows for the standard output of one process to become the standard input of another process, without needing to make any temporary files which makes it extremely efficient.

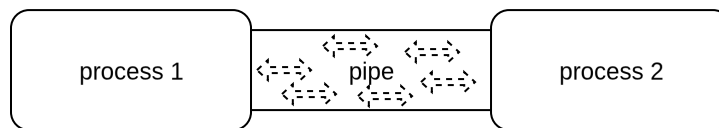


Figure 4: Figure visualising the pipe

The string is divided into multiple strings whenever the pipe operator (`|`) is identified, and the position of it is saved. The pipe positions are used to be able to construct commands out of the separated strings e.g. `ls -la | grep txt | wc -l` is represented as three different commands, always split at the pipe symbol.

Each command is executed as its own process. Using `fork()` and `execv()` while they are “connected” to the pipe. Depending on the commands position the content is redirected different.

- First command: **STDOUT** is redirected to the **STDIN** of the following command.
- Middle command(s): **STDIN** will receive the **STDOUT** of the previous command, and will redirect the output to the **STDIN** of the following command.
- Last command: Gets its **STDIN** from the previous command it points its **STDOUT** to the current terminal.

Figure 5 shows a visualisation of the data flow. The parsed commands are represented as `char ***commands`, which is simply a pointer to an array of pointers that point to pointers of characters (a string). Which is a typical way to represent arguments for a shell since this can store multiple arguments while still being able to easily access individual strings if needed. Figure 6 gives a simplified view of how `echo "foo bar baz" | wc -w` is represented in memory.

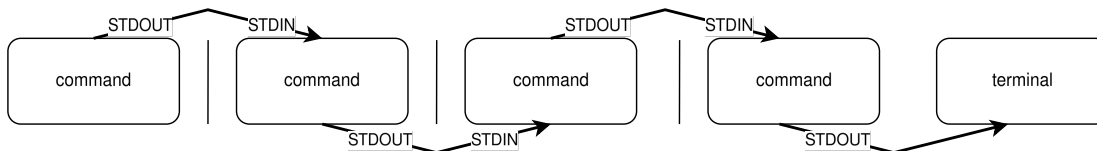


Figure 5: Figure visualising the data flow of multiple pipes chained together

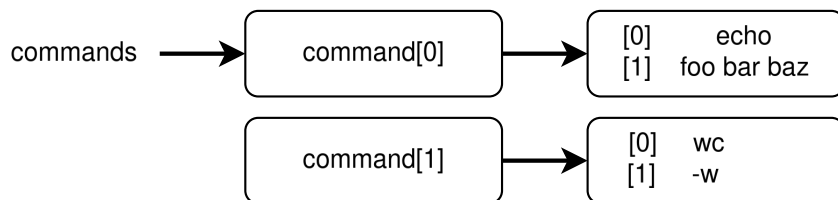


Figure 6: Memory layout example of `commands` (simplified)

5.4.5 Input Redirection (<)

Input redirection redirects the input for the command. It allows the command to read the input from a file instead of the standard input. This is essentially a toned down version of piping (see Section 5.4.4). Opposed to piping it does not work with live processes, only with files, this makes it useful for simple operations.

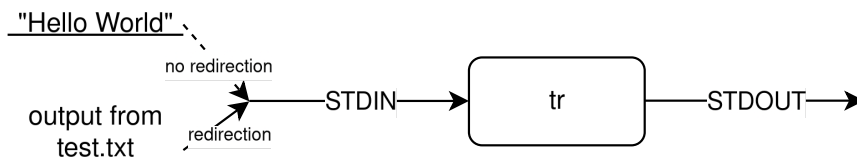


Figure 7: Figure to help visualize input redirection

So instead of feeding the information in the commandline like so: `echo "Hello World" | tr 'a-z' 'A-Z'`, it can be read from a file e.g. `tr 'a-z' 'A-Z' < test.txt`. This will open `test.txt` and redirect its contents to the `STDIN` of `tr`.

The following happens, `<` basically creates a pipe between the `STDOUT` from `echo` to the `STDIN` from `tr`

For the implementation see Listing 14.

5.4.6 Output Redirection (>, >>)

Output redirection basically works the same. A user can redirect output from a command like `ls` and either truncate (`>`), or append (`>>`) to the file.

`ls -la > file_list.txt` will create the file if needed, and print the `STDOUT` from `ls -la` into `file_list.txt`.

Figure 8 shows a simple visualisation of how the output redirection might look like.

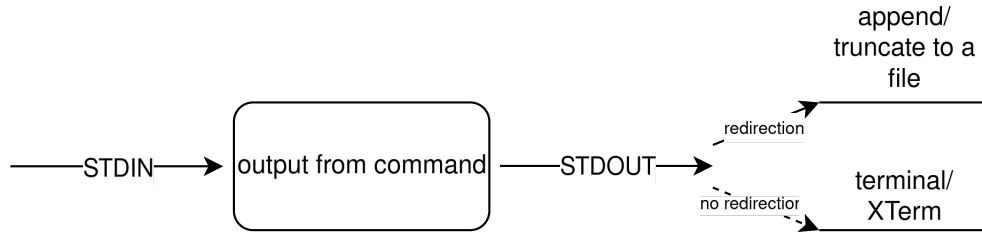


Figure 8: Figure to help visualize output redirection

For the implementation of this see Listing 15

5.4.7 Background Execution &

Background execution basically implements itself. There is a check for a single & indicating background execution. If so the function that executes commands skips `waitpid()` and just continues. This means that the shell then shares the `STDOUT/STDERR` with the background process. But the user can use the shell again.

5.4.8 Command chaining (&&)

Command chaining allows the user, to execute multiple commands, as “one big piece”. A following command will only be executed, it chains commands, only executes if the previous command was successful e.g.

```
echo start && false && echo end
```

Would only print “start” and return immediately, since false, does not return 0, but 1.

This works by treating every command in between the ampersands as its own command and retrieving the exit status.

If there are two ampersands &&, it calls the `split_on_chain()` function, otherwise it proceeds normally (See Listing 7).

```

if(strstr(tmp_user_input, "&&") != NULL) {
    user_input = split_on_chain(tmp_user_input);
} else {
    user_input[0] = my_strdup(tmp_user_input);
}
for(int i = 0; user_input[i] != NULL; i++) {
    args = parse(user_input[i]);
    if(args == NULL) perror("oshell: parsing failed ");
    if (execute_command(args) != 0) break;
}
  
```

Listing 7: “normal” case vs chaining case

5.5 Environment Variables

An environment variable is a variable, with a name and value. They can hold all sorts values, for example the `HOME` path, things like the `DESKTOP_SESSION` or your current `SHELL`. On Linux they can be listed with `printenv` or `env`. `OSHELL` uses the `secure_getenv()` (https://linux.die.net/man/3/secure_getenv) function to retrieve the value behind the variable if valid.

```
int dollar_pos = find_shell_operator("$", args);
if(dollar_pos >= 0 && args[dollar_pos][1]) {
    char envChar = args[dollar_pos][0];
    strncpy(envVar, &args[dollar_pos][1], sizeof(args[dollar_pos] - 1));
    char envCharStr[2] = {envChar, '\0'};
    if(strncmp("$", envCharStr) == 0) {
        fprintf(stderr, "%s", secure_getenv(envVar));
        free(envVar);
        return 0;
    }
}
```

Listing 8: sourcecode showing implementation of `OSHELL` handling environment variables.

The `find_shell_operator()` function loops through the string array, provided to it and checks against the operator given (see Listing 16).

Since `find_shell_operator()` will only return a positive non zero value on success it is safe to assume that if the program goes inside the `if` statement, there is at least one char behind the dollar sign (`$`). This means that we can copy everything but the dollar sign and construct an environment variable to pass to the `secure_getenv()` function.

5.6 Input loop

The shell is something that needs to continuously run, to read user input, parse it manage different processes and execute commands.

`OSHELL` implements this using the `oshell_loop()` as its main function. This basically runs the whole shell.

1. **Read:** Reads the user input line by line.
2. **Parse:** Tokenizes the string and parses it e.g. for operators like `<`, `<<`, `~` or `$`
3. **Execute:** Based on the input of the string performs different actions to execute the command, inbuilt commands are handled differently then commands that have a pipe or input redirection.

Listing 9 shows the implementation of the main loop.

```

void oshell_loop(void)
{
    char *user_input = oshell_read_line();
    char **args = parse(user_input);

    execute_command(args);
    free(user_input);
    free(args);
}

```

Listing 9: sourcecode showing the main loop

6 Testing

Two attempts have been made at writing the tests¹. The initial idea was to use Golang, as it seemed like a simple enough language to write the tests in, and it was not Python. However, not all tests behaved as expected because Go was not interacting with the shell through the TTY, but through pipes.

The Rust crate `expectrl` (<https://github.com/zhiburt/expectrl>) solved this issue by interacting with the shell via a *pseudoterminal* (pty). A pty is a pair of virtual character devices. A character device could be a keyboard or mouse; a character device file could be `/dev/ttyS0`, for example [9].

So `expectrl` “emulates” a `tty` so that they would behave like expected.

There are ten tests that simulate the core functionalities

1. echo “hello”
2. cd (~)
3. cd (~ /Dokumente)
4. env (USER, PWD, DISPLAY)
5. kill (by Name, by PID)
6. input redirection (appending)
7. piping (single | , | multi |)
8. command chaining (logical AND &&)

All the tests are built according to this really simple scheme.

¹Golang tests: `/go_shell_test/shell_test.go`, Rust tests: `rust_shell_test/src/lib.rs`

A new command is spawned and the output is saved. The output is then converted to a proper string and the whitespaces are trimmed. This is the 'proper' expected output from Bash.

Then **OSHELL** is spawned and the same command is executed.

```
#[test]
fn test_cd_home() {
    let tmp_home = Command::new("bash")
        .arg("-c")
        .arg("cd ~ && pwd")
        .output()
        .unwrap();
    let home = String::from_utf8_lossy(&tmp_home.stdout).trim().to_string();

    let mut shell = spawn_shell().unwrap();
    exec(&mut shell, &format!("cd {}", home));
    let output = exec(&mut shell, "pwd").trim().to_string();

    assert_eq!(output, home);
}
```

Listing 10: sourcecode showing cd home test in rust

Some functions (both `kill_by_*`() and the `test_append()`) require additional checks, to confirm the output of the test. Both kill tests need to check if the process is still alive, and the append test needs to read a file to compare the output.

7 Evaluation

Even though this project took way longer then planned, the developer was able to successfully implement a basic bash inspired shell in C11. Although not everything was implemented by the developer himself, since knowledge was missing or just general inexperience, he has learned a lot about unix parsing and c in general. The shell is missing specs there were given on the specification sheet, because he is done with this project. If he feels like coming back and implementing features like background exection (&) there would be nothing stopping him. Most of the core functionalities could be implemented, although the shell is not a replacement for bash, if the user is not advanced this probably could be used day to day.

Bibliographie

References

- [1] “AIX,” Accessed: Aug. 20, 2025. [Online]. Available: <https://www.ibm.com/docs/en/aix/7.3.0?topic=volumes-using-file-descriptors#:~:text=A%20file%20descriptor%20is%20an%20unsigned%20integer%20used%20by%20a%20process%20to%20identify%20an%20open%20file%2E>.
- [2] “Stdin, stdout, stderr,” Accessed: Aug. 20, 2025. [Online]. Available: <https://www.learnlinux.org.za/courses/build/shell-scripting/ch01s04>.
- [3] “Linux basic concepts,” What The Hack, Accessed: Oct. 31, 2025. [Online]. Available: <https://microsoft.github.io/WhatTheHack/020-LinuxFundamentals/Student/resources/concepts.html>.
- [4] “Z/OS,” Accessed: Aug. 14, 2025. [Online]. Available: <https://www.ibm.com/docs/en/zos/2.5.0?topic=functions-fgets-read-string-from-stream>.
- [5] “What is a Delimiter?” Computer Hope, Accessed: Sep. 6, 2025. [Online]. Available: <https://www.computerhope.com/jargon/d/delimited.htm>.
- [6] “Proc(5) - Linux manual page,” Accessed: Aug. 18, 2025. [Online]. Available: <https://www.man7.org/linux/man-pages/man5/proc.5.html>.
- [7] “Proc pseudo file system,” Accessed: Aug. 18, 2025. [Online]. Available: <https://tldp.org/HOWTO/SCSI-2.4-HOWTO/procfs.html>.
- [8] “Ubuntu Manpage: Ioctl_list - list of ioctl calls in Linux/i386 kernel,” Accessed: Aug. 19, 2025. [Online]. Available: https://manpages.ubuntu.com/manpages/focal/en/man2/ioctl_list.2.html.
- [9] “Pty(7) - Linux manual page,” Accessed: Oct. 23, 2025. [Online]. Available: <https://man7.org/linux/man-pages/man7/pty.7.html>.

Appendix

Parsing - Remove quotes function

```
char **remove_quotes(char **arg) {
    if (arg == NULL) {
        perror("remove_quotes failed");
    }

    size_t arg_count = 0;
    size_t new_arg_count = 0;
    int quote_string_pos = 0;
    while(arg && arg[arg_count]) arg_count++;
    // in this case calloc is being used, because there are cases where it is
    ↪ being
    // accessed without having a value there causing it to crash.
    char **tmp_args = calloc(arg_count + 1, sizeof(char*));
    if(tmp_args == NULL) perror("tmp_args memory allocation failed ");

    bool *processed = calloc(arg_count, sizeof(bool));

    int end_arg = -1;
    int end_pos = -1;

    for(int i = 0; i < arg_count; i++) {
        if (processed[i]) {
            continue;
        }
        if (arg[i] == NULL) {
            i++;
        }
        bool found_quote = false;

        for(int j = 0; j < strlen(arg[i]); j++) {
            if ((arg[i][j] == '"' || arg[i][j] == '\\')) {
                char quote = arg[i][j];
                int start_arg = i;
                int start_pos = j;

                // Search for closing quote
                int current_i = i;
                int current_j = j + 1;

                bool found_closing = false;
                while (current_i < arg_count && arg[current_i] != NULL) {
                    while (current_j < strlen(arg[current_i])) {
```

```

        if (arg[current_i][current_j] == quote) {
            end_arg = current_i;
            end_pos = current_j;
            found_closing = true;
            break;
        }
        current_j++;
    }
    if (found_closing) break;
    current_i++;
    current_j = 0;
}

if (found_closing) {
    tmp_args[i] = build_quote_string(arg, start_arg, start_pos,
        ↪ end_arg, end_pos);

    for(int k = start_arg; k <= end_arg; k++) {
        processed[k] = true;
    }

    found_quote = true;
    break;
} else {
    fprintf(stderr, "Warning: No closing quote found for quote
        ↪ starting at arg[%d] [%d]\n", start_arg, start_pos);
    break;
}
}

if (!found_quote) {
    tmp_args[i] = my_strdup(arg[i]);
}
}

int n = arg_count + 1;

int j = 0;
for (int i = 0; i < n; i++) {
    if (tmp_args[i] != NULL) {
        if (i != j) tmp_args[j] = tmp_args[i];
        j++;
    }
}

// Set leftover slots to NULL

```

```
    for ( ; j < n; j++) tmp_args[j] = NULL;

    free(processed);
    free(arg);
    return tmp_args;
}
```

Listing 11: Removes quotes from argument strings.

Execute Command - Change Directory

```
int change_directory(char *directory)
{
    // add /home/user/ if ~ is the first char
    char* new_path = NULL;
    bool use_home_path = false;
    if(directory[0] == '~') {
        char *user_path = getenv("HOME");
        int new_path_length = strlen(user_path) + strlen(directory) + 1;
        // first char is ~ replace dir with /home/user/dir
        remove_char(directory, '~');
        new_path = malloc(new_path_length);
        if(new_path == NULL) {
            perror("oshell: change_directory() memory allocation failed");
        }
        snprintf(new_path, new_path_length, "%s%s", user_path, directory);
        use_home_path = true;
    }

    int res = 0;
    if(use_home_path) {
        res = chdir(new_path);
    } else {
        res = chdir(directory);
    }

    if (res == -1) {
        free(new_path);
        return -1;
    }
    free(new_path);
    return 0;
}
```

Listing 12: sourcecode showing the implementation of the change_directory function

Execute Command - Kill

```
int kill_process(char *process_name_or_id)
{
    if(string_to_int(process_name_or_id) == -1) {
        struct dirent *dirent;
        pid_t pid = 0;
        char *proc_dir_name = "/proc/";
        DIR *dir = opendir(proc_dir_name);
        if(dir == NULL) {
            exit(EXIT_FAILURE);
        }

        char *buffer = NULL;
        char *comm_file_path = NULL;
        char *full_proc_path = NULL;

        FILE *fp = NULL;

        bool process_found = false;

        /*
         * We need to open every /proc/pid/comm file (opendir() and readdir())
         * ↪ and check if the content matches
         * the process name if yes, we have to corresponding pid and we can kill
         * ↪ the
         * process
         * */
        // use opendir() to open proc/ and readdir() to go through */comm
        // /proc/*/comm

        int count = 0;
        while((dirent = readdir(dir)) != NULL) {
            if(dirent->d_type == DT_DIR) {
                // we dont need /proc/. and /proc/..
                count++;
                if(strcmp(dirent->d_name, ".") == 0 || strcmp(dirent->d_name,
                    ↪ "..") == 0) {
                    continue;
                }

                // if the directory name is not all integers we skip
                if(!is_numeric(dirent->d_name)) continue;

                // string length for proc_dir_name + dirent->d_name
```

```

size_t full_proc_pathLength = strlen(proc_dir_name) +
    ↪ strlen(dirent->d_name) + 1;
full_proc_path = malloc(full_proc_pathLength);
if(full_proc_path == NULL) {
    goto cleanup;
    ↪ //perror("oshell: not able to allocate enough Memory for
    ↪ variable: 'full_proc_path'");
}
snprintf(full_proc_path, full_proc_pathLength ,"%s%s",
    ↪ proc_dir_name, dirent->d_name);
if((opendir(full_proc_path)) != NULL ) {

    // this saves the dir name of the comm file we are about to
    ↪ visit
    // we can "save" the dir name and convert + cast it to a pid
    ↪ so
    // that if the entry is the correct one we have access to
    ↪ the pid
    // and can kill the process via kill()
    pid = (pid_t)atoi(dirent->d_name);

    size_t comm_file_pathLength = strlen(proc_dir_name) +
        ↪ strlen(dirent->d_name) + strlen("/comm") + 1;
    comm_file_path = malloc(comm_file_pathLength);
    if(comm_file_path == NULL) {
        goto cleanup;
    }
    snprintf(comm_file_path, comm_file_pathLength,"%s%s/comm",
        ↪ proc_dir_name, dirent->d_name);

    // ---- CLOSE FILE FROM HERE ----

    fp = fopen(comm_file_path, "r");
    if(fp == NULL) {
        goto cleanup;
    }

    // there is something with the /proc/*/comm files that
    ↪ causes issues with
    // using ftell. so this is seems to be a way to do it.

    long filesize = 0;
    while (getc(fp) != EOF)
        filesize++;
    fseek(fp, 0, SEEK_SET);

```

```

const size_t          BUFFER_SIZE = ((sizeof(char) *
↪ filesize) +1);

buffer = malloc(BUFFER_SIZE);
if(buffer == NULL) {
    goto cleanup;
}

fgets(buffer, BUFFER_SIZE, fp);

if(ferror(fp)) {
    goto cleanup;
    return -2;
}

if(strcmp(strip_non_alpha(buffer),
↪ strip_non_alpha(process_name_or_id)) == 0) {
    if(kill(pid, 9) == 0) {
        sleep(1);
        waitpid(pid, NULL, WNOHANG);
        fprintf(stderr, "killed %s with PID: %i\n",
↪ process_name_or_id, pid);
        fflush(stderr);
        process_found = true;
    }
    goto cleanup;
}
}
}
}
if(!process_found) {
    fprintf(stderr, "Could not find process '%s'\n",
↪ process_name_or_id);
}
cleanup:
if(buffer) free(buffer);
if(comm_file_path) free(comm_file_path);
if(full_proc_path) free(full_proc_path);
if(fp) fclose(fp);
if(dir) closedir(dir);
return 0;
}
const pid_t pid = string_to_int(process_name_or_id);
int status = 0;

```

```

if(kill(pid, 9) == 0) {
    int t_s = 0;
    if((t_s = waitpid(pid, &status, WNOHANG)) > 0) {
        fprintf(stderr, "killed %i\n", pid);
    } else {
        perror("waitpid");
    }
} else {
    perror("kill");
    fprintf(stderr, "waitpid status: %p", &status);
}
fflush(stderr);
return 0;
}

```

Listing 13: sourcecode implementation of the `kill_process()` function.

Execute Command - Input Redirection

```
int saved_stdin = dup(STDIN_FILENO);
int do_input_redirection = 0;
int fd_in = 0;

int input_red_pos = find_shell_operator("<", args);

if (input_red_pos > 0 && args[input_red_pos + 1] != NULL) {
    // < does not check if the file in the argument is a valid file, it will
    ↪ redirect anything
    char      *filename = args[2];
    FILE      *file = fopen(filename, "r");

    if(file == NULL) {
        fprintf(stderr, "Error: %s is not a valid file.", filename);
    }

    if (feof(file) || ferror(file)) {
        perror("oshell: fread() failed");
        exit(EXIT_FAILURE);
    }

    // open filebased pipeline channel for file 'filename' in read only
    // 0 = stdin | 1 = stdout | 2 = stderr

    fd_in = open(filename, O_RDONLY);

    close(STDIN_FILENO);
    // we duplicate fd, into stdin
    if (dup2(fd_in, STDIN_FILENO) == -1) {
        perror("oshell: dup2 error");
        close(fd_in);
        return -1;
    }
    args[input_red_pos] = NULL; // remove '<' from the command

    do_input_redirection = 1;
}
```

Listing 14: sourcecode showing the implementation of input redirection

Execute Command - Output Redirection

```
int saved_stdout = dup(STDOUT_FILENO);
int do_output_redirection = 0;
int fd = 0;

int truncate_pos = find_shell_operator(">", args);
int append_pos = find_shell_operator(">>", args);
// redirect stdout to the file
if ((append_pos > 0 && args[append_pos + 1] != NULL) || (truncate_pos > 0 &&
↪ args[truncate_pos + 1] != NULL)) {
    // > truncate (overwrite) ; >> append

    if (args[truncate_pos + 1] == NULL || args[append_pos + 1] == NULL) {
        fprintf(stderr, "Error: missing filename after > / >> \n");
        return -1;
    }

    if (args[truncate_pos + 2] != NULL || args[append_pos + 2] != NULL) {
        fprintf(stderr, "Error: too many arguments after > / >> \n");
        return -1;
    }

    char *filename = NULL;
    // if (append_pos != -1) { append_pos } else truncate_pos
    int pos = (append_pos != -1) ? append_pos : truncate_pos;
    if (pos != -1) {
        filename = args[pos + 1];
    }

    int flags;
    if (strcmp(">>", (args[append_pos])) == 0) {
        // inplace bitwise OR (x |= y ; x = x | y)
        // add flag O_APPEND to flags
        // &= ~xyz (remove xyz)
        flags = O_APPEND | O_WRONLY | O_CREAT; // create if needed, append if
        ↪ exists
    } else {
        flags = O_TRUNC | O_WRONLY | O_CREAT; // create if needed, truncate if
        ↪ exists
    }

    fd = open(filename, flags, 0644);
    if (fd == -1) perror("oshell: open() failed");

    if (dup2(fd, STDOUT_FILENO) == -1) {
```

```
        perror("oshell: dup2 error");
        close(fd);
        return 1;
    }

    // this is to remove the operator and filename from the command
    args[pos] = NULL;
    args[pos + 1] = NULL;

    do_output_redirection = 1;
}
```

Listing 15: sourcecode showing the implementation of output redirection

Utils - Find a shell operator

```
int find_shell_operator(char* operator, char **args) {
    for (int i = 0; args[i] != NULL; i++) {
        if (strcmp(args[i], operator) == 0) {
            return i;
        }
        // this is to check for $ since it does not stand alone
        if (args[i][0] == operator[0]) {
            return i;
        }
    }
    return -1;
}
```

Listing 16: sourcecode showing the implementation of the `find_shell_operator()` function

Parsing Utils - Build a new string

```
char *build_quote_string(char **arg, const int start_arg, const int start_pos,
    ↪ const int end_arg, const int end_pos)
{
    int total_length = 0;

    for(int i = start_arg; i < end_arg; i++) {

        // this is needed otherwise, the code will try to execute the command
        ↪ provided e.g.
        // echo "foo bar baz" | wc -w just like that.
        // this would output foo bar baz | wc -w /*

        /* We only want to skip the first character in the first word,
         * since this is the outer quote that we want to remove
         */
        int start_j;
        if(i == start_arg) start_j = start_pos + 1;
        else start_j = 0;

        int end_j;
        if (i == end_arg) end_j = end_pos;
        else end_j = strlen(arg[i]);

        for(int j = start_j; j < end_j; j++) total_length++;
        if(i < end_arg) total_length++;
    }

    char *result = malloc((sizeof(char) * total_length) + 1);
    if (!result) {
        perror("malloc failed");
        exit(EXIT_FAILURE);
    }
    int index = 0;

    for(int i = start_arg; i <= end_arg; i++) {
        int start_j;
        if(i == start_arg) start_j = start_pos + 1;
        else start_j = 0;

        int end_j;
        if (i == end_arg) end_j = end_pos;
        else end_j = strlen(arg[i]);
```

```
    for(int j = start_j; j < end_j; j++) result[index++] = arg[i][j];

    if(i < end_arg) {
        result[index++] = ' ';
    }
}
return result;
}
```

Listing 17: sourcecode showing the implementation of the `build_quote_string()` function

Piping

```
int pipe_redirection(char **args)
{
    /*
     * To support more than one, pipe for "advanced" commands, there needs to be
     * ↪ a way to track the amount of pipes we have
     * for later redirection, tracking of the multiple pipe positions, and the
     * ↪ splitting of the commands
     */
    int pipes_amount = 0;

    for (int i = 0; args[i] != NULL; i++) {
        if(strcmp("|", args[i]) == 0) pipes_amount++;
    }

    if(pipes_amount == 0) return -1; // no pipes found, however

    int pipe_pos[pipes_amount];
    int pipe_index = 0;
    for (int i = 0; args[i] != NULL; i++) {
        if (strcmp(args[i], "|") == 0) {
            pipe_pos[pipe_index++] = i;
        }
    }

    char ***commands = malloc((pipes_amount + 1) * sizeof(char**));
    char **paths = malloc((pipes_amount + 1) * sizeof(char*));

    int max_args[pipes_amount+1];
    memset(max_args, 0, sizeof(max_args));

    int end = 0;
    int cmd_id = 0;

    for(int i = 0; args[i] != NULL; i++) {
        if(strcmp(args[i], "|") == 0) {
            cmd_id++;
        } else {
            max_args[cmd_id]++;
        }
    }

    /*
```

```

* CLEANED-UP COMMAND SPLITTING LOOP - AUTHOR: ChatGPT (GPT-5 mini)
*
* This replaces the previous command splitting section in
↪ `pipe_redirection()`
* from the original version in your repository:
*
↪ https://github.com/Landixbtw/oshell/commit/f4ed2a9eec7686112a143dfd5ba2b7798d3f4426#d
*
* The goal is to reduce complexity, fix bugs with offsets, and ensure
* memory safety when splitting commands into a 2D array for piping.
*
* Just so many small errors and oversights that just lead to a big ball of
↪ shit fuck
*/

for (cmd_id = 0; cmd_id <= pipes_amount; cmd_id++) {
    int start = (cmd_id == 0) ? 0 : pipe_pos[cmd_id - 1] + 1;
    int end = (cmd_id < pipes_amount) ? pipe_pos[cmd_id] : 0;
    if (cmd_id == pipes_amount) {
        for (end = start; args[end] != NULL; end++);
    }

    int cmd_arg_count = end - start;

    commands[cmd_id] = malloc((cmd_arg_count + 1) * sizeof(char*));
    if (!commands[cmd_id]) {
        perror("oshell: memory allocation for command[cmd_id] failed");
        exit(EXIT_FAILURE);
    }

    for (int i = 0; i < cmd_arg_count; i++) {
        commands[cmd_id][i] = my_strdup(args[start + i]);
    }
    commands[cmd_id][cmd_arg_count] = NULL;

    size_t path_length = strlen("/usr/bin/") + strlen(commands[cmd_id][0]) +
↪ 1;
    paths[cmd_id] = malloc(path_length);
    if (!paths[cmd_id]) {
        perror("oshell: piping()");
        exit(EXIT_FAILURE);
    }
    snprintf(paths[cmd_id], path_length, "/usr/bin/%s",
↪ commands[cmd_id][0]);
}
/*

```

```

* pipe() returns two file descriptors, fd[0] is open for reading, fd[1] is
    ↪ open for writing
* output of fd[1] is input for fd[0].
*
* Since pipes are a point-to-point connection, one for every command, so we
    ↪ need n-1 pipes if n is the amount of commands we have
* */

int fd[pipes_amount][2];

for(int i = 0; i < pipes_amount; i++) {
    if (pipe(fd[i]) == -1) {
        perror("oshell: fd ");
    }
}

/*
* We need to track, what gets redirected where
* "This is the first command, only redirect STDOUT"
* "This is a middle command, redirect both"
* "This is the last command, only redirect STDIN"
* */

// save the stdin/stdout, to later restore
int saved_stdout = dup(STDOUT_FILENO);
int saved_stdin = dup(STDIN_FILENO);

int f = 0;
for(int i = 0; i < cmd_id; i++) {
    if ((f = fork()) == 0) {
        // child: setup the redirections
        if(i == 0) { // first command, only redirect stdout
            if(dup2(fd[i][1], STDOUT_FILENO) == -1) {
                fprintf(stderr, "Child %d: dup2 failed\n", i);
                _exit(EXIT_FAILURE);
            }
        }
        for(int j = 0; j < pipes_amount; j++) {
            close(fd[j][0]);
            close(fd[j][1]);
        }
    } else if (i == cmd_id - 1) { // last command only redirect stdin
        if(dup2(fd[i-1][0], STDIN_FILENO) == -1) {
            close(fd[i-1][0]);
            _exit(EXIT_FAILURE);
        }
    }
}

```

```

        for(int j = 0; j < pipes_amount; j++) {
            close(fd[j][0]);
            close(fd[j][1]);
        }
    } else { // not first nor last, redirect stdin and stdout
        // cmd0 ---> pipe[0] ---> cmd1 ---> pipe[1] ---> cmd2
        if(dup2(fd[i-1][0], STDIN_FILENO) == -1 || dup2(fd[i][1],
            ↪ STDOUT_FILENO) == -1) {
            _exit(EXIT_FAILURE);
        }
        for(int j = 0; j < pipes_amount; j++) {
            close(fd[j][0]);
            close(fd[j][1]);
        }
    }
    if(commands[i] == NULL) {
        fprintf(stderr, "ERROR: commands[%d] is NULL\n", i);
        _exit(EXIT_FAILURE);
    }

    if (execv(paths[i], commands[i]) == -1) {
        perror("execv() failed");
        free(paths);
        free(commands);
        for(int i = 0; i < pipes_amount; i++) {
            close(fd[i][0]);
            close(fd[i][1]);
        }
        _exit(EXIT_FAILURE);
    }
    fprintf(stderr, "Child %d: ERROR - this should never print!\n", i);
} else if (f == -1) {
    perror("oshell: fork() failed");
    return -1;
}
// parent continues to next iteration
}

for(int j = 0; j < pipes_amount; j++) {
    close(fd[j][0]);
    close(fd[j][1]);
}

for(int i = 0; i < cmd_id; i++) {
    int status;
    pid_t result = wait(&status); // Non-blocking wait

```

```

    if (result == 0) {
        continue;
    } else if (result > 0) {
        // fprintf(stderr, "Child %d finished with status %d\n", result,
        ↪ status);
    }
}

// restore stdin/stdout
dup2(saved_stdin, STDIN_FILENO);
dup2(saved_stdout, STDOUT_FILENO);

if(close(saved_stdin) != 0) perror("oshell: piping()");
if(close(saved_stdout) != 0) perror("oshell: piping()");

for (int i = 0; i <= pipes_amount; i++) {
    for (int j = 0; commands[i][j]; j++) {
        free(commands[i][j]);
    }
    free(commands[i]);
    free(paths[i]);
}

free(paths);
free(commands);

return 0;
}

```

Listing 18: sourcecode showing the implementation of the `pipe_redirection()` function