oshell

Implementation of a Unix Inspired Shell in C11

Author: Ole Wortmann

Date: September 30, 2025

Abstract

abstract

Contents

1 Introduction 1

2 Planning 1

3 Essential Terminology and Concepts 1

4 Design 2

5 The Implementation 3

5.1 Getting the input 3
5.2 Parsing the user input 4
5.2.1 Removing quotes and building a new string 6
5.3 Executing Commands 6
5.4 Builtin Commands and functionalities 6
5.4.1 CD - Change Directory 6
5.4.2 Kill - Killing a process by name or ID 7
5.4.3 Piping 8
5.4.4 Clear - Clearing the terminal 9
5.4.5 Input Redirection (<) 10
5.4.6 Output Redirection (>, >>) 10
5.5 Environment Variables 11
5.6 Input loop 11

6 Testing 12

7 ??Evaluation?? 12

Bibliographie 13

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 made by {the author} was not created in any official context and should be treated as such.

This project came to life because I was looking for a new project that was not game developement, 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. Since it had peaked my interest in the past, but I was just not ready implementing the C standard library.

This is neither something new nor outstanding but just a fun project that took me way longer then I would like to admit, because I only thought of some edgecases or obvious failure points, after I was already “done”.

It also does not implement any new ideas, but builds on a specification sheet, which you will learn more about in the planning section (see Section 2). 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. At the time of writing this command chaining (&&) aswell as background execution (&) have not (yet) been implemented.

2 Planning

Since building a *(small)* shell is not unusual for a University project, I figured there would be some article or specification or anything that could give me a sense of direction and ease the planning. I found a specification from a professor at Florida State University (FSU), but since I cannot find the article in which it was posted I can only provide a direct download link (`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. I did not follow it linear, but just worked on what I felt like.

3 Essential Terminology and Concepts

This will explain some basic 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 depens 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]

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.

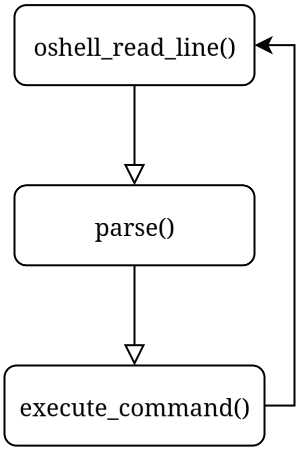

Figure 1: Figure visualising the main flow

The **input handler** passes the string without modification to the **parser**, which parses the string, tokenizes it, and removes exces 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).

memory leaks still exist so this will not be written yet.

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis

this is also a todo

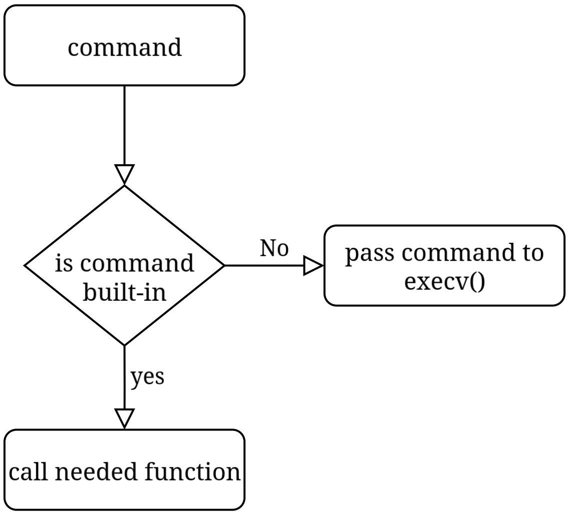

Figure 2: Figure visualising the flow of command execution (simplified)

egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

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()` seemes like a valid approach, it stops when reading a newline character (`'\n'`) and also does not handle memory management by itself [3]. Since OSHELL does not support command chaining by && this is the only way to chain commands together e.g. `command \n command1 \n command2`

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.

i feel like this is missing something, there is no text here

```c
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;
}
```

5.2 Parsing the user input

Parsing the input for a small/basic shell might not seem like a huge endevour, 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 slighty 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 5.4.3). 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 tokenization necessary? Tokenization is needed because the `execv()` function that **OSHELL** uses to execute the commands, takes the `path` and the `argv` seperate.
   - How does the tokenization work? When tokenizing **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." [4]. If the character to be replaced is not the first character, it is replaced with the (`'\0'`), to terminate the string. The adress 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.

this is also needed for parsing, and actually called in parse function

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

```c
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 1: codeblock from parse function showing way of shortening string but not always?

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 as of writing this, 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 7), loops through every word, and letter, to find either a first single or double quote. Uppon finding one, it logs the first word and letter position and the last word and letter position (See Listing 7 lines 28 - 46). If a closing quote is found, the `build_quote_string()` function is called with the following parameters provided, `arg, start_arg, start_pos, end_arg, end_pos`. This then just turns the multiple arguments into one string without the quotes, null-terminating the string (See Listing 13).

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`, `help`, `cd` or `kill`, which `exit()` the programm print some help text, change the directory or kill 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” there are 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 2: sourcecode showing the 'cd' implementation of OSHELL

Depending on if `args[1]` is NULL or not the `change_directory()` (see Listing 7) 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 `kill_process()` function. 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 `strint_to_int()` function it will the two cases:

why might you want to kill a process, and not just press the close window icon?

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 [5] [6]. 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 programm 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.

```c
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 3: 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 Piping

The implementation of piping (See Listing 14) 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 or something which makes it extremly efficient.

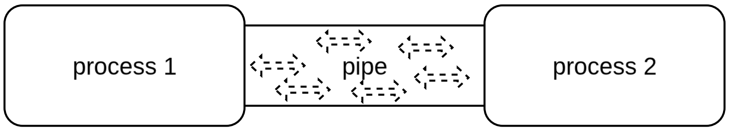

Figure 3: Figure visualising the pipe

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 seperated 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 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 4 shows a visualisation of the data flow.

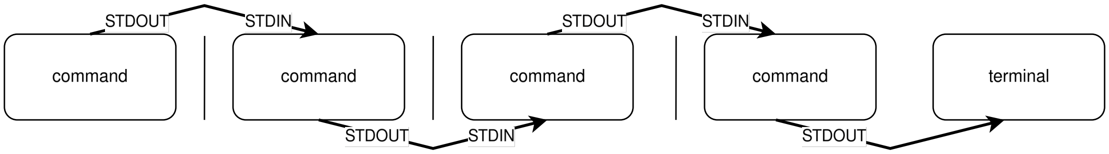

Figure 4: Figure visualising the data flow of multiple pipes chained together

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 mulitple arguments while still being able to easily access individual strings if needed. Figure 5 gives a simplified view of how `echo "foo bar baz" | wc -w` is mapped in memory.

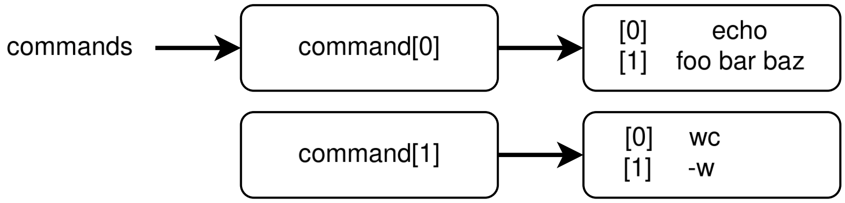

Figure 5: Memory layout example of `commands` (simplified)

5.4.4 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 windowsize [7]. 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.

```c
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 4: sourcecode showing implementation of the `clear()` function

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 esentially a toned down version of piping (see Section 5.4.3). Opposed to piping it does not work with live processes, only with files , this makes it usefull for simple operations.

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`.
Figure 6 shows a visualisation of how input redirection might look like.

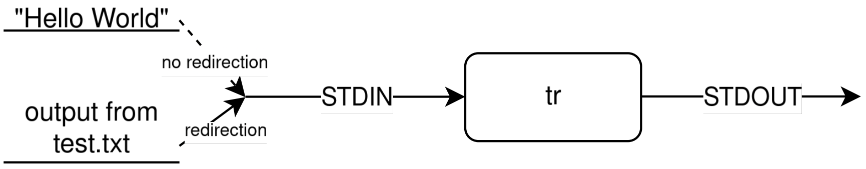

Figure 6: Figure to help visualize input redirection

For the implementation see Listing 10.

5.4.6 Output Redirection (>, >>)

A user can redirect output from a command like `ls` and either truncate (>), or append (>>) to the file. This works the same as input redirection, only for `STDOUT`.
`ls > ls.txt` will create the file if needed and truncate if exists.
`echo "line1\nline2\nline3" >> append.txt` will create the file and append if exists. Figure 7 shows a simple visualisation of how the output redirection might look like.
For the implementation of this see Listing 11

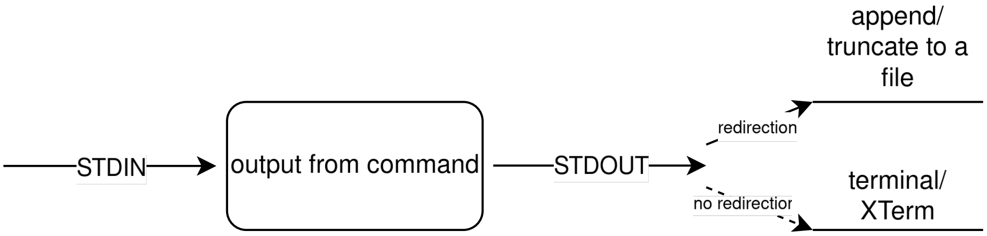

Figure 7: Figure to help visualize output redirection

5.5 Environment Variables

An environment variable is basically 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.

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

Listing 5: 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 12).

Since `find_shell_operator()` will only return a positive non zero value on success it is safe to assume that if the programm goes inside the `if` statement, there is at least once 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.

better title needed, also not final this can do better

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 6 shows the implementation of the main loop.

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

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

Listing 6: sourcecode showing the main loop

6 Testing

Testing and Validation: A critical part of any software project is demonstrating that it works correctly. A dedicated section on how you tested your shell is important. This could include:

- Unit testing: Did you test individual functions?
- Integration testing: Did you test how different parts work together?
- User acceptance testing: Did you run it with a set of common commands to verify it functions as expected?

7 ??Evaluation??

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] "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.

[4] "What is a Delimiter?" Computer Hope, Accessed: Sep. 6, 2025. [Online]. Available: https://www.computerhope.com/jargon/d/delimite.htm.

[5] "Proc(5) - Linux manual page," Accessed: Aug. 18, 2025. [Online]. Available: https://www.man7.org/linux/man-pages/man5/proc.5.html.

[6] "Proc pseudo file system," Accessed: Aug. 18, 2025. [Online]. Available: https://tldp.org/HOWTO/SCSI-2.4-HOWTO/procfs.html.

[7] "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.

Appendix

Parsing - Remove quotes function

```c
char **remove_quotes(char **arg) {
    size_t arg_count = 0;
    size_t new_arg_count = 0;
    int quote_string_pos = 0;
    while(arg && arg[arg_count]) arg_count++;
    char **tmp_args = malloc((arg_count + 1) * sizeof(char*));
    if(tmp_args == NULL) perror("tmp_args memory allocation failed ");

    // Initialize the last element to NULL
    tmp_args[arg_count] = NULL;


    bool *processed = calloc(arg_count, sizeof(bool)); // Track which args are processed

    // this is not declared with the others e.g. start_arg,start_pos
    // because it is needed later down out of
    // that scope
    int end_arg = -1;
    int end_pos = -1;

    for(int i = 0; i < arg_count; i++) {
        if (processed[i]) {
            continue; // Skip already processed arguments
        }

        bool found_quote = false;

            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) {
                    // Build the quoted string and store it in new_args[i]
                    tmp_args[i] = build_quote_string(arg, start_arg, start_pos, end_arg, end_pos);
                    // fprintf(stderr, "quoted string [%i]: %s\n", i, tmp_args[i]);

                    // Mark all arguments from start_arg to end_arg as processed
                    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 no quote was processed, copy the original string
        if (!found_quote) {
            // fprintf(stderr, "no found quote [%i]: %s\n", i, arg[i]);
            tmp_args[i] = malloc(strlen(arg[i]) + 1);
            if(tmp_args[i] == NULL) fprintf(stderr,"oshell: memory allocation for %s failed", tmp_args[i]);
            strcpy(tmp_args[i], 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);
    return tmp_args;
}
```

Listing 7: Removes quotes from argument strings.

Execute Command - Change Directory

```c
int change_directory(char *directory)
{
    // add /home/user/ if ~ is the first char
    char* new_path = NULL;
    bool use_home_path = false;
    if(strcmp(&directory[0], "~") == 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, '~');
        // fprintf(stderr, "%s%s",user_path, directory);
        new_path = malloc(new_path_length);
        snprintf(new_path, new_path_length, "%s%s", user_path, directory);
        fprintf(stderr, "%s\n", new_path);
        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 8: sourcecode showing the implementation of the change_directory function

Execute Command - Kill

restore some comments, but remove commented functions and clutter.

```
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;

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

                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;
                }
                snprintf(full_proc_path, full_proc_pathLength ,"%s%s",
                 ↪  proc_dir_name, dirent->d_name);
                if((opendir(full_proc_path)) != NULL ) {

                    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);

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

                    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) {
                            fprintf(stderr, "killed %s with PID: %i",
                             ↪  process_name_or_id, pid);
                            fflush(stderr);
                            process_found = true;
                        }
                        goto cleanup;
                    }
                }
            }
        }
        if(!process_found) {
```

```c
            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);
        // the directory was not closed in every secenario
        if(dir) closedir(dir);
        return 0;
    }
    const pid_t pid = string_to_int(process_name_or_id);
    kill(pid, 9);
    fprintf(stderr,"killed %i\n", pid);
    fflush(stderr);
    return 0;
}
```

Listing 9: sourcecode implementation of the `kill_process()` function.
(some comments have been removed for clarity, can be found in the repository)

Execute Command - Input Redirection

```c
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 close stdin
    // 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 10: sourcecode showing the implementation of input redirection

Execute Command - Output Redirection

```c
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 11: sourcecode showing the implementation of output redirection

Utils - Find a shell operator

```c
int find_shell_operator(char* operator, char **args) {
    for (int i = 0; args[i] != NULL; i++) {
        if (strcmp(args[i], operator) == 0) {
            return i;
        }
        if (args[i][0] == operator[0]) {
            return i;
        }
    }
    return -1;
}
```

Listing 12: sourcecode showing the implementation of the `find_shell_operator()` function

Parsing Utils - Build a new string

```c
char *build_quote_string(char **arg, int start_arg, int start_pos, int end_arg, 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++; // amount of characters
        if(i < end_arg) total_length++; // add N (amount of args) - 1 spaces for between each word
    }

    char *result = malloc((sizeof(char*) * total_length)  + 1);
    if(result == NULL) perror("oshell: memeory allocation for string result failed");
    int index = 0;

    /*
     * Building the string
     * This is basically the same as the loop before we make start_j and end_j
     * and then loop from start to end, adding the characters one by one to the
     * result string and null terminating it.
     * */

    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];

        // add a space after each word
        if(i < end_arg) {
            result[index++] = ' ';
        }
    }
    return result;
}
```

Listing 13: sourcecode showing the implementation of the `build_quote_string()` function

Piping

```c
int pipe_redirection(char **args)
{
/*
     * To support more then 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 spliting 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 = 0;
    for(int i = 0; args[i] != NULL; i++) {
        MAX_ARGS++;
    }

    int start = 0;
    int end = 0;
    int cmd_id = 0;
    for (cmd_id = 0; cmd_id <= pipes_amount; cmd_id++) {
        if(cmd_id < pipes_amount) {

            commands[cmd_id] = malloc(MAX_ARGS * sizeof(char*));
            if (!commands[cmd_id]) {
                perror("oshell: memory allocation for command[cmd_id] failed");
            }
            memset(commands[cmd_id], 0, MAX_ARGS * sizeof(char*));

            end = pipe_pos[cmd_id];
            // create the commands from [cmd_id][i] = args[i]
            for(int i = 0; i < end; i++) {
                /*
                 * Since commands[cmd_id][i] is not initialized and only
                 * command[cmd_id] we need to allocate for each string in the array
                 * */
                if(args[i] != NULL) {
                    commands[cmd_id][i] = malloc(strlen(args[i]) + 1);
                    strcpy(commands[cmd_id][i], args[i]);
                }
                                }
            commands[cmd_id][end] = NULL;
        } else {
            commands[cmd_id] = malloc(MAX_ARGS * sizeof(char*));
            if (!commands[cmd_id]) {
                perror("oshell: memory allocation for command[cmd_id] failed");
            }
            memset(commands[cmd_id], 0, MAX_ARGS * sizeof(char*));

            int j = 0;
            do {
                j++;
            }while (args[j] != NULL);
            start = end + 1; // Skip the pipe
            end = j;
            for(int i = start; i < end; i++) {
                // using strdup we dont need to manually allocate memory for
                // commands[cmd_id][xyz] since strdup handles that
                if(args[i] != NULL) {
                    // shit fuck why is i - start needed ??? DOES NOT WORK WITHOUT
                    commands[cmd_id][(i - start)] = my_strdup(args[i]);
                }
            }
            commands[cmd_id][end - start] = 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(-1);}
        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
     * ouput 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++) {
```

```c
        close(fd[j][0]);
        close(fd[j][1]);
    }

    for(int i = 0; i < cmd_id; i++) {
        int status;
        pid_t result = waitpid(-1, &status, WNOHANG);  // Non-blocking wait
        if (result == 0) {
            continue;
        } else if (result > 0) {
        }
    }

    // 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()");

    free(paths);
    free(commands);

    return 0;
}
```

Listing 14: sourcecode showing the implementation of the `pipe_redirection()` function