

Salih Özyurt 150117855
Muhammed Bera Koç 150116062

CSE3033 Project 2

Overview

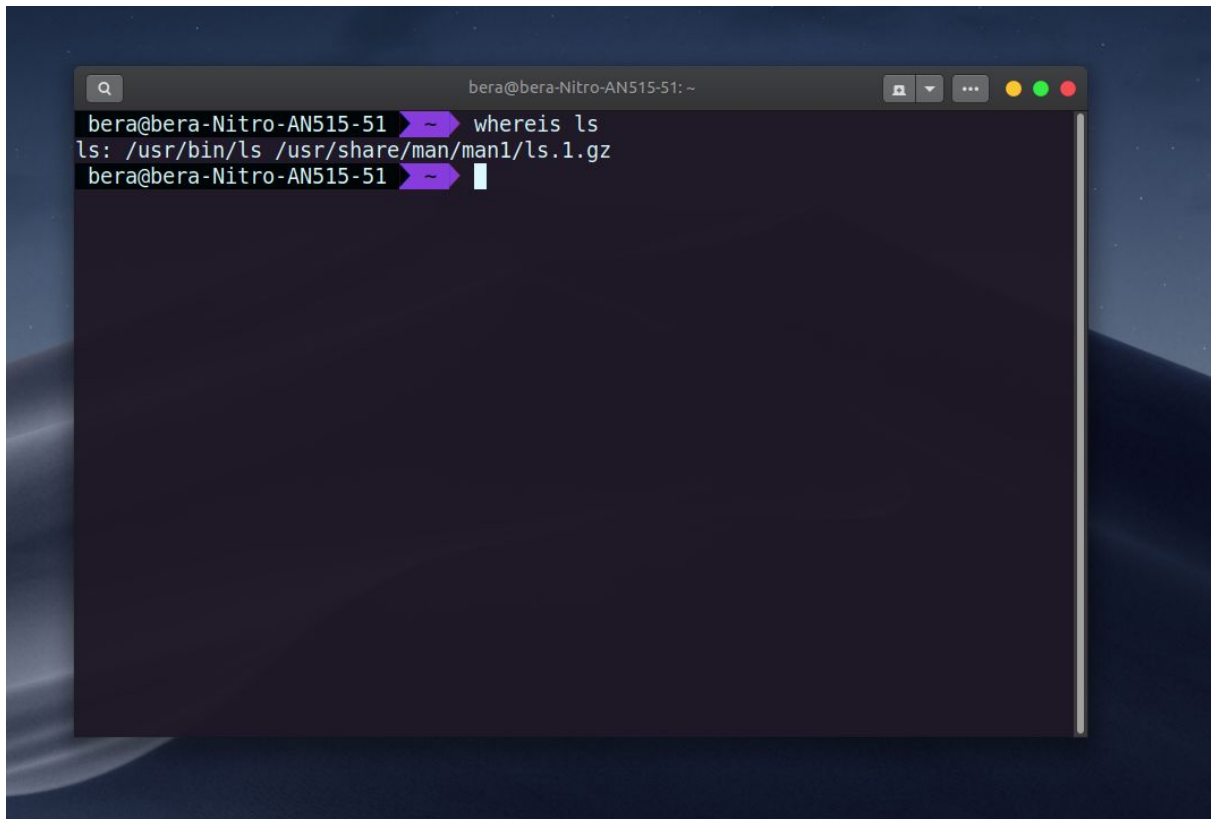
The main goal of the project is to create a mini-terminal pretending the bash CLI. To achieve this goal we have to use C language as a tool. The range of commands is not so complicated. Only basic commands like builtin shell commands (e.g.: `ls`, `cd`, `echo`, etc.), and some simulated commands for the given project (e.g.: `history`, `path`).

Fundamental Architecture

To apply all of these commands in our terminal first we need some tools in C. Like splitting which is one of the most useful tools for this project, Queue structure and Boolean enum. We created an additional library for terminal -console.h- and called `setup` function in an infinite while loop. Setup function is consisting of dozens of miscellaneous functions to provide enormous amount of abstraction.

Making of Builtin Commands

It may seem like an easy task to implement builtin commands. Spoiler alert: It is not! First problem is commands are not following a monolithic pattern. This makes them hard to implement. However let's start with our game plan. First task is we have to find the given command path. We have to use `whereis` command. Unfortunately there is a little problem.

A terminal window titled 'bera@bera-Nitro-AN515-51: ~' with standard window controls. The prompt is 'bera@bera-Nitro-AN515-51 ~' followed by a purple arrow icon. The command 'whereis ls' has been entered, and the output is 'ls: /usr/bin/ls /usr/share/man/man1/ls.1.gz'. The prompt is now 'bera@bera-Nitro-AN515-51 ~' with a purple arrow icon and a cursor.

```
bera@bera-Nitro-AN515-51 ~$ whereis ls
ls: /usr/bin/ls /usr/share/man/man1/ls.1.gz
bera@bera-Nitro-AN515-51 ~$
```

It has a special syntax. So we have to first get this path data from the output string. To conduct this we first write the output to a file called "path.log" using file descriptors. This function in console lib is `get_exec_path`. It takes one argument -a char array, the bare form of the command- and returns the full path. Now since we have path, we can use `execv` function. One can simply ask now "Why we need the full path of a command?". Well answer is simple. Because `execv` takes the full path. Amazing we are making progress. We have now two aims left. Detecting irregular builtin commands and making the actual execution. Before that knowing the second argument of `execv` is an array of string containing main commands and args, we have to split the input array. Lucky us, we have a default mechanism in library laid before our service: `ParserNode`. `ParserNode` consists of two data field. An array of string and its length. Nice but how to convert a pure string to a `ParserNode`. Well we have a function for that: `parse_string`:

```

// A function which takes an array of strings, a string and a delimiter
// Splits string using delimiter and puts substrings into capsule.
// Also returns a ParserNode which holds the length of capsule
ParserNode parse_string(char *capsule[], char input[], const char delim[])
{
    int index = 0; // An index variable to hold the current substring location
    char *token; // Creates a token and stores each substring inside this token
    token = strtok(input, delim);
    while (token != NULL)
    {
        capsule[index++] = token; // puts the token inside capsule increasing index by one
        token = strtok(NULL, delim); // In each iteration splits one chunk of string
    }
    return _ParserNode(capsule, index); // Returns a ParserNode using creator function
}

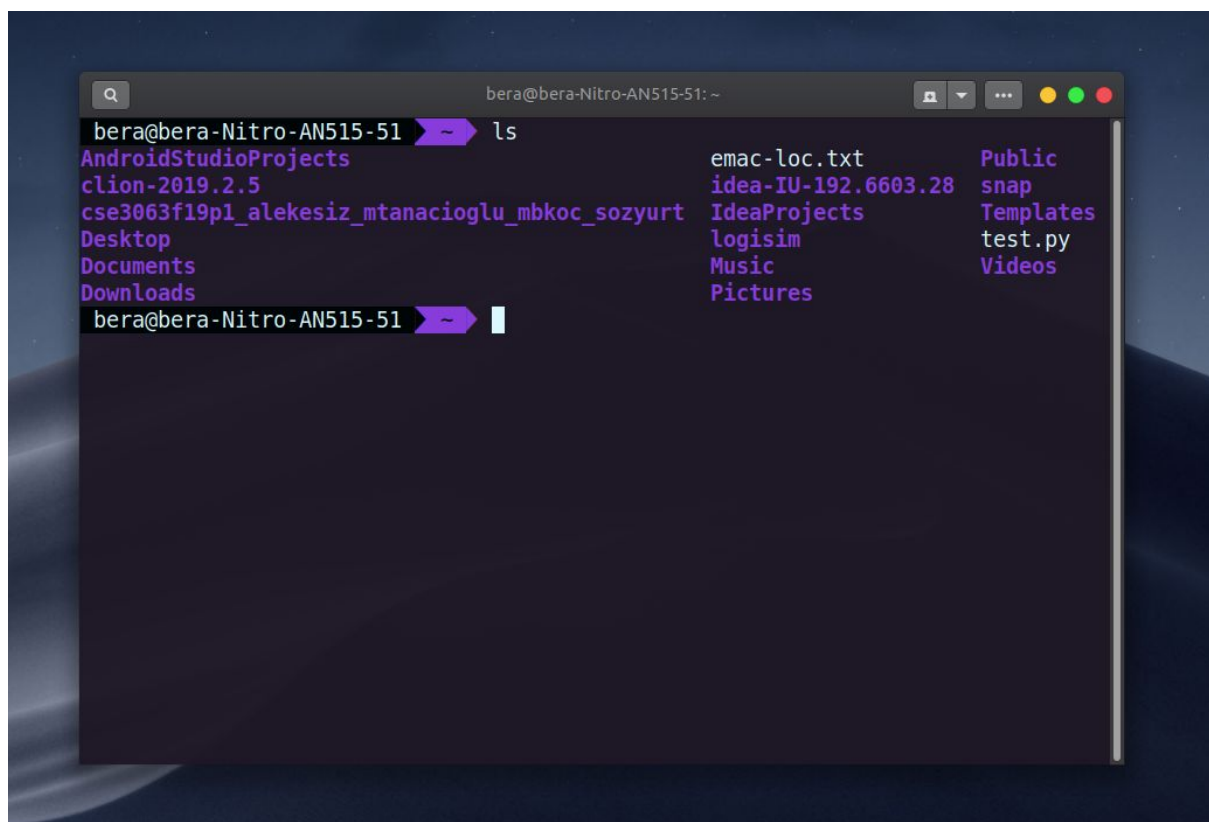
```

Now we have everything we want except that one below:

```
argsNode.capsule[argsNode.length++] = NULL;
```

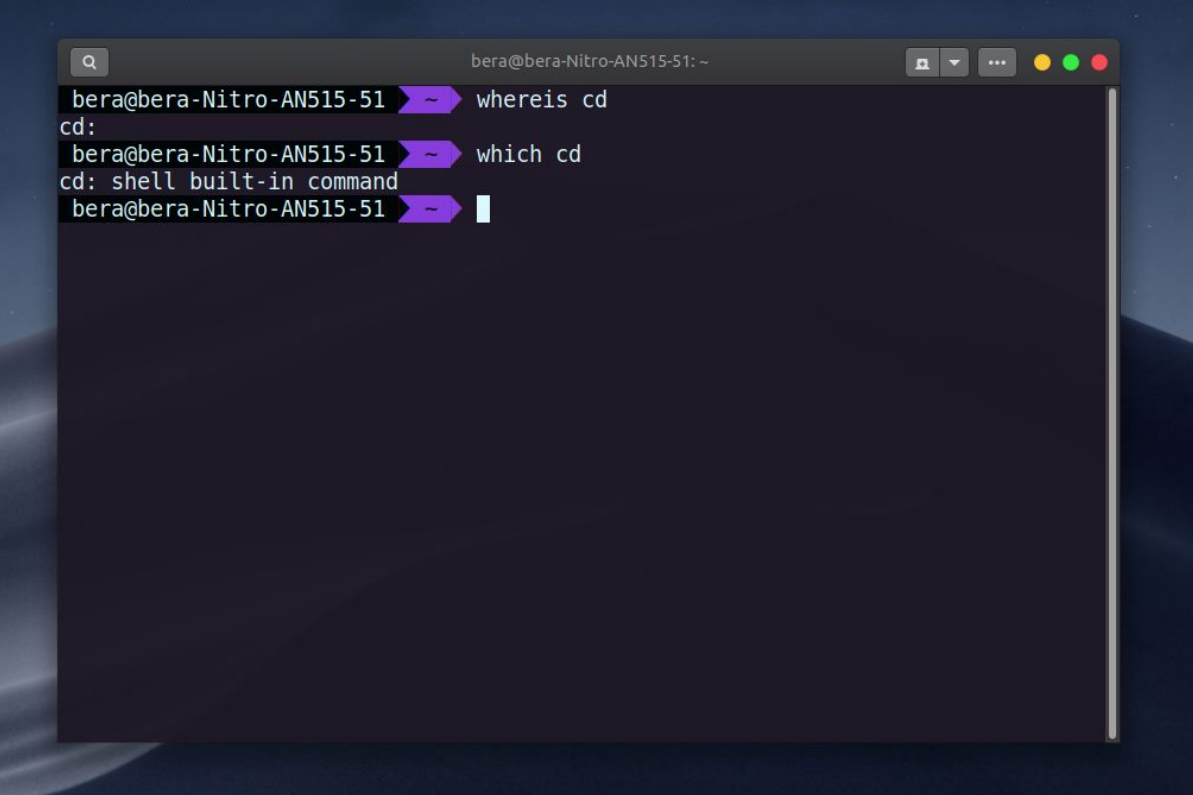
One line may seem even redundant yet it is essential since `execv` always takes an array of strings ending with `NULL`. Otherwise it will not work.

Now we are very nigh to our final dwelling, a fine terminal. As we stated in first lines of this section, we have irregular builtin commands. What do I mean with irregular? Simple. Look the command below:



This is a normal, lovely builtin command. When I use whereis command for ls I'll get an output like the shell example before this one.

Let's try the same command for cd.

A terminal window titled 'bera@bera-Nitro-AN515-51: ~' with a search icon in the top-left corner. The terminal shows three commands and their outputs: 1. 'whereis cd' followed by 'cd:' on the next line. 2. 'which cd' followed by 'cd: shell built-in command' on the next line. 3. A blank prompt line with a cursor. The terminal has a dark background with light-colored text and a purple prompt character. The window has standard macOS-style window controls (red, yellow, green buttons) in the top-right corner.

```
bera@bera-Nitro-AN515-51 ~$ whereis cd
cd:
bera@bera-Nitro-AN515-51 ~$ which cd
cd: shell built-in command
bera@bera-Nitro-AN515-51 ~$
```

Interesting. Whereis command prints nothing about the path. Why? Answer is in one line below. 'Cause it is a shell built-in command which means it has no path. Then how to achieve the functionality of cd. There is a secret function in c: chdir. It does everything cd do. So it is a very good alternative for our main command.

```
// Handles cd command since it is builtin exec with no full path
void handle_cd_command(char *input_path)
{
    // if nothing is given it does nothing
    if (input_path == NULL) return;
    // Otherwise changes the direction using given path
    chdir(input_path);
}
```

There is another irregular built-in command. Well to be honest it is not a command. It is a notation for running an executable. "./" it has no way to detect. So we implemented it like below:

```
// If given command is not in form ./blahblah (execute command)
if (args[0][0] != '.' && args[0][1] != '/')
```

So just putting the command is enough. We need no full path for "./".

Now the last step for executing:

```
pid_t child_id;
if ((child_id = fork()) == 0)
{
    if (*background == __True)
    {
        input_copy[strlen(input_copy) - 1] = '\0';
        setup(input_copy, args, background);
    }
    else
    {
        execv(args[0], args);
        exit(EXIT_SUCCESS);
    }
}
```

Do not mind background for now. We first create a child process and control if the program is inside child. If this is true we execute the command giving the proper parameters.

Additional Commands

We have four additional commands: path, history, fg and exit. Let's start with the easiest one: exit command.

```
// If given command is exit
else if (!(strcmp(args[0], "exit")))
{
    exit(EXIT_SUCCESS);
}
```

Cannot be easier than that. Now we will delve into path command. Path command has 2 status: no args, or two args. If there are two args, first args can be plus or minus and second one is a

directory. When it is plus, directory is concatenated to the system path variable otherwise it will be removed. Now the point is if it is plus there can be duplicate values. However for minus we have to remove all duplicates. For no args, it will print the current path. Achieving this may seem cumbersome and difficult. Nathless surprisingly it is not. Let's see it:

```
// If command is path handle it
else if (!strcmp(args[0], "path"))
{
    // Means if there is only one argument "path"
    if (storage.args.length == 2) puts(getenv("PATH"));
    else handle_path_command(args[1], args[2]);
}

// Path command is a new command. Henceforth it should be handled by us
// It takes two arguments: option can be + or - and directory which is new directory
// which is to be added to system path variable
void handle_path_command(char *option, char *directory)
{
    // __path variable holds the current system path
    char *__path = getenv("PATH");
    // A variable for modified system path
    char formatted_path[DEFAULT_SIZE] = "";
    // If option is + which mean adding a new path to system
    if (!strcmp(option, "+"))
    {
        // Combines new directory and current system path
        sprintf(formatted_path, "%s:%s", directory, __path);
        // Sets the new path variable with formatted path
        // Last argument means overwriting is true
        setenv("PATH", formatted_path, TRUE);
    }
    // If - it means given directory will be removed with all duplicates
    else if (!strcmp(option, "-"))
    {
        // Puts all path directories in global variable using : as a delimiter
        pathNode = parse_string((char*[DEFAULT_SIZE]){}, __path, ":");
        // Traverse between directories
        for (int i = 0; i < pathNode.length; ++i)
        {
            // When a system directory is not equal to given directory
            // Concatenates it to the formatted path
            if (strcmp(pathNode.capsule[i], directory))
            {
                strcat(formatted_path, pathNode.capsule[i]);
                strcat(formatted_path, ":"); // Adds it for default path format
            }
        }
        formatted_path[strlen(formatted_path) - 1] = '\0'; // To remove last : since its redundant
        char *resized_path = malloc(BIG_SIZE); // Creates new char array
        sprintf(resized_path, "%s", formatted_path); // Puts the formatted path into resized path
        setenv("PATH", resized_path, TRUE); // Sets the path environment variable as resized path
    }
}
```

After looking it twice, it may seem just a little hard. So in first part we control the arguments if no arguments is given we

print the system path using getenv function which is basically returns the value of the system variable given. Else we invoke a handler function for path command. Since given code is well-documented I will just over look over it. First we get the values we need. Second we control + and - cases. + case is easy just concatenate the new directory to current path. - can be a little confusing. We use the global pathNode variable and fill it with each directory of current path. Then we traverse pathNode and add the directories to formatted path unless they are equal to given directory. This is a very subtle strategy to tackle with given tricky situation. Now let's talk about the history command. Where do we store the history. In a global char array called __History. It has a size of 10. And there is a history index which holds the current index of __History.

```
// __History char array holds the past record of written terminal commands  
char *__History[10];  
// Holds the index value for the current history  
int history_index = 0;
```

For each setup call we add current command to __History using initHistory function.

```
// Initialises terminal history and sets it for each command  
void initHistory(char *input) You, a day ago • Documentation added.  
{  
    if (history_index != 10) // If total commands are less than ten it will work like a stack  
    {  
        if(history_index == 0)  
        {  
            __History[history_index] = input;  
        }  
        else  
        {  
            for (int i = history_index; i > 0; i--)  
            {  
                __History[i] = __History[i-1]; // Shift right operation  
            }  
            __History[0] = input;  
        }  
        history_index++;  
    }  
    else // Otherwise function will not increase index since stack is full  
    {  
        for (int i = history_index; i > 0; i--)  
        {  
            __History[i] = __History[i-1];  
        }  
        __History[0] = input;  
    }  
}
```

Function above behaves like a limited-sized stack. So let's talk about how to handle history. History command has two status: no args or 2 args. When it takes no args it prints the previous commands. When it has two args it is in the form history -i n. N is here is the number of command. Let's see how to handle it.

```
// If command is history
if (!(strcmp(args[0], "history")))
{
    // If history is given with -i alias
    if (storage.args.length == 4)
    {
        // Obtain the ith command parse it and put it inside args
        args = parse_string((char*[DEFAULT_SIZE]){ }, __History[atoi(args[2])], " ").capsule;
    }
    // if there is only history without no args
    else
    {
        // Print the history
        writeHistory();
        // Resets the IO devices
        reset_file_descriptors();
        // Finishes the current command
        return;
    }
}
```

First we must compare command with history and if they both equal we check if number of args 3. Why not 3? Because we have a NULL at the end. If is equal to 4 using parse we change the args using the number in arguments injecting it into __History. So we get nht command in history. And refill args for this command using parse_string. Otherwise we use writeHistory function -no worry it just a function iterates over __History and prints each command with a fancy way.- and then reset file descriptors in case of we use history with io redirection. Then we use return since we end our business at this scenario.

Handling Background Processes

As a careful reader you were probable sensible of something wrong or deficient. Yes we never mentioned any background processes. Everything happens on the main process. That was not true yet I was hiding this part from you on purpose. Now let's see which part of our code takes care of this entangled mechanism.


```

pid_t child_id;
if ((child_id = fork()) == 0)
{
    if (*background == __True)
    {
        input_copy[strlen(input_copy) - 1] = '\0';
        setup(input_copy, args, background);
    }
    else
    {
        execv(args[0], args);
        exit(EXIT_SUCCESS);
    }
}

```

That seems familiar. Because it was shown once more. Now we can focus on this `child_id`. As you can see there is a `background` and it is a Boolean value. Initially it is `False`. We create a child and control whether `background` is true or false. If false classic execution. However if it is true something strange happens. We take the input and recursively call `setup` function using the current arguments so child runs these commands not the parent `-main` terminal in our case-. But how we change the `background`?

```

input = strdup(fgets(input_buffer, MAX_INPUT_LIMIT, stdin));
*background = isBackground(input_buffer);

```

Here we change `background` using `isBackground` function.

```

Boolean __True = TRUE, __False = FALSE;

Boolean isBackground(char input[]){
    int length = strlen(input);
    for (int i = 0; i < length; i++)
    {
        if(input[i] == '&')
        {
            return __True;
        }
    }
    return __False;
}

```

We define two Boolean instances __True and __False. isBackground function controls if input has any & character since it is the notation for background process.

```

char *input;
if (*background == __True)
{
    *background = __False;
    input = input_buffer;
}

```

This part is for the child. Notice that background is only true when a child invokes the setup. If we are in the child first make the background False then load input_buffer inside input since we don't need to read it from the terminal.

IO Redirection

To simulate the IO redirection we used file descriptors. We copied standard IO devices inside global variables to not to lose them using init_standard_io function.

```
// Initiliases standard IO devices by storing them in protected file descriptors
void init_standard_io()
{
    protected_stdout = dup(STDOUT_FILENO);
    protected_stdin = dup(STDIN_FILENO);
    protected_stderr = dup(STDERR_FILENO);
}
```

dup copies content of standard io devices. Now we have also formatted file descriptors for directed io.

```
// Boolean values of different IO operations
Boolean Write, WriteAppend, WriteError, Read;
// Standard IO devices
int protected_stdout, protected_stdin, protected_stderr;
// Directed IO devices
int formatted_stdout, formatted_stdin, formatted_stderr;
```

Our global variables for IO operations are given above. We have four signals for IO operations. We set them using set_io_signals.

```
// Resets the IO devices to standard ones
void reset_file_descriptors()
{
    if (Write || WriteAppend)
    {
        dup2(protected_stdout, STDOUT_FILENO);
        close(formatted_stdout);
    }
    if (Read)
    {
        dup2(protected_stdin, STDIN_FILENO);
        close(formatted_stdin);
    }
    if (WriteError)
    {
        dup2(protected_stderr, STDERR_FILENO);
        close(formatted_stderr);
    }
}
```

```

// Sets io signals using given command node
// Returns a storage struct
Storage set_io_signals(ParserNode argsNode)
{
    Boolean gate = TRUE; // Gate is created to detect the first IO symbol
    int io_index = 0; // The index of first io symbol => io symbols: [>, >>, 2>, <]
    // Initiliases all symbols to False
    Write = FALSE, WriteAppend = FALSE, WriteError = FALSE, Read = FALSE;
    // Loops through the commands
    for (int i = 0; i < argsNode.length; ++i)
    {
        // If current node is Null we reached the end it breaks
        if (argsNode.capsule[i] == NULL) break;
        // If current node is one of the io symbols it sets the io_index and closes the gate
        if (!strcmp(argsNode.capsule[i], ">") || !strcmp(argsNode.capsule[i], ">>") ||
            !strcmp(argsNode.capsule[i], "2>") || !strcmp(argsNode.capsule[i], "<"))
        {
            if (gate)
            {
                io_index = i;
                gate = FALSE;
            }
        }
        // Set IO signals
        if (!strcmp(argsNode.capsule[i], ">")) Write = TRUE;
        else if (!strcmp(argsNode.capsule[i], ">>")) WriteAppend = TRUE;
        else if (!strcmp(argsNode.capsule[i], "2>")) WriteError = TRUE;
        else if (!strcmp(argsNode.capsule[i], "<")) Read = TRUE;
    }
    // That means no IO symbol has been found it returns only to one ParserNode
    // Given argsNode since no IO part exists
    if (io_index == 0) return (Storage) {(ParserNode) {capsule: argsNode.capsule, length: argsNode.length}};
    // Initialises the ParserNodes
    char *current_args[DEFAULT_SIZE];
    char *current_io[DEFAULT_SIZE];
    // Until it reaches the io_index puts the strings inside the argsNode
    for (int i = 0; i < io_index; ++i)
    {
        current_args[i] = argsNode.capsule[i];
        if (i == io_index - 1) current_args[i + 1] = NULL;
    }
    // After io_index is reached. Puts all the strings in io ParserNode
    for (int i = 0; i < argsNode.length - io_index; ++i)
    {
        current_io[i] = argsNode.capsule[i + io_index];
    }
    // Returns the two ParserNodes
    return (Storage) {(ParserNode) {capsule: current_args, length: io_index + 1},
        (ParserNode) {capsule: current_io, length: argsNode.length - io_index}};
}

```

Basically what we do here is finding the index where IO starts in arguments. For example our input is `ls -a > ls.txt`. IO starts from `>`. Hence `io_index` is 2 here. Then we start traversing the input arguments if we found any of io symbols we open the signals. If there is no io symbols (`io_symbol = 0`) we return the given arguments with an empty io arguments. Otherwise fill those two container (pure args and io args) then return them. Setting file descriptors a little bit long. We check every signal and using `dup2` we overwrite the standard device with given file.

```

// Sets the files descriptors using io ParserNode
void set_file_descriptors(ParserNode io)
{
    // Loops through all strings
    for (int i = 0; i < io.length; ++i)
    {
        // If current is NULL we reached the end break!
        if (io.capsule[i] == NULL) break;
        // If current string is equal to >, that means it is writing without append
        if (!strcmp(io.capsule[i], ">"))
        {
            ++i; // Increase index to reach the file index
            // i i + 1
            // {">", "input.txt", ..., NULL}
            // Remove if the file exists
            remove(io.capsule[i]);
            // Make the output file descriptor given file
            // 0777 means give all rights to every user
            // To prevent the need of sudo while starting the code
            if ((formatted_stdout = open(io.capsule[i], O_CREAT|O_WRONLY, 0777)) < 0)
            {
                perror("Cannot open file descriptor."); /* open failed */
                exit(1);
            }
            // Make the current standard output device current file descriptor
            if (dup2(formatted_stdout, STDOUT_FILENO) < 0)
            {
                perror("Cannot reset output device.");
                exit(1);
            }
        }
        // Same as above, >> means write with append : add O_APPEND to activate append option
        else if (!strcmp(io.capsule[i], ">>"))
        {
            ++i;
            if ((formatted_stdout = open(io.capsule[i], O_CREAT|O_WRONLY|O_APPEND, 0777)) < 0)
            {
                perror("Cannot open file descriptor.");
                exit(1);
            }
            if (dup2(formatted_stdout, STDOUT_FILENO) < 0)
            {
                perror("Cannot reset output device.");
                exit(1);
            }
        }
        // Same process 2> same as >. The only difference is for error we use standard
        // error device
        else if (!strcmp(io.capsule[i], "2>"))
        {
            ++i;
            remove(io.capsule[i]);
            if ((formatted_stdout = open(io.capsule[i], O_CREAT|O_WRONLY, 0777)) < 0)
            {
                perror("Cannot open file descriptor.");
                exit(1);
            }
            if (dup2(formatted_stdout, STDERR_FILENO) < 0)
            {
                perror("Cannot reset output device.");
                exit(1);
            }
        }
        // For input (<) we use standard input device
        else if (!strcmp(io.capsule[i], "<"))
        {
            ++i;
            if ((formatted_stdin = open(io.capsule[i], O_CREAT|O_RDONLY, 0777)) < 0)
            {
                perror("Cannot open file descriptor.");
                exit(1);
            }
            if (dup2(formatted_stdin, STDIN_FILENO) < 0)
            {
                perror("Cannot reset output device.");
                exit(1);
            }
        }
    }
}

```


There is only one last problem at the end of each command we must reset IO devices. We have a function for that which basically overwrites the protected devices with current devices.

```
// Resets the IO devices to standard ones
void reset_file_descriptors()
{
    if (Write || WriteAppend)
    {
        dup2(protected_stdout, STDOUT_FILENO);
        close(formatted_stdout);
    }
    if (Read)
    {
        dup2(protected_stdin, STDIN_FILENO);
        close(formatted_stdin);
    }
    if (WriteError)
    {
        dup2(protected_stderr, STDERR_FILENO);
        close(formatted_stderr);
    }
}
```

Conclusion

Imitating a real time terminal is a cumbersome job to handle yet basics can be established only using core functions. In this project we tried to emulate the bash terminal using number of c functions. And we see that we reached our goal. Our terminal nearly exhibits all fundamental behaviours which a linux terminal does.