Chapter II

Introduction

The existence of shells is linked to the very existence of IT. At the time, all coders agreed that `communicating with a computer using aligned 1/0 switches was seriously irritating`. It was only logical that they came up with the idea `to communicate with a computer using an interactive lines of commands in a language somewhat close to english`.

With `Minishell`, you'll be able to travel through time and come back to problems people faced when `Windows` didn't exist. If this doesn't make you a better coder, nothing will.

Chapter III

Objectives

Through the `Minishell` project, you will get to the core of the `Unix` system and explore an important part of this system's API: process creation and synchronisation. Executing a command inside a shell implies creating a new process, which execution and final state will be monitored by its parent's process. This set of functions will be the key to success for your `Minishell`, so be sure to code the cleanest, simplest program possible. Otherwise, you'll probably have to start from scratch for your `21sh` project and that would be a real shame.

Be rigorous and methodical in your `C` coding, take the necessary time to read and understand the `mans`, but most importantly, test your code!

Chapter IV

General Instructions

- This project will only be corrected by actual human beings. You are therefore free to organize and name your files as you wish, although you need to respect some requirements listed below.
- The executable file must be named `minishell`.
- You must submit a `Makefile`. That `Makefile` needs to compile the project and must contain the usual rules. It can only recompile the program if necessary.
- If you are clever, you will use your library for your `minishell`. Also submit your folder `libft` including its own `Makefile` at the root of your repository. Your `Makefile` will have to compile the library, and then compile your project.
- Your project must be written in `C` in accordance with the Norm.
- You have to handle errors in a sensitive manner. In no way can your program quit in an unexpected manner (Segmentation fault, bus error, double free, etc).
- Your program cannot have memory leaks.
- You'll have to submit at the root of your folder, a file called `author` containing your username followed by a '\n'

```
$>cat -e author
xlogin$
```

- Within your mandatory part you are allowed to use the following functions:
    - `malloc`, `free`
    - `access`
    - `open`, `close`, `read`, `write`
    - `opendir`, `readdir`, `closedir`
    - `getcwd`, `chdir`
    - `stat`, `lstat`, `fstat`

- ◦ `fork`, `execve`
- ◦ `wait`, `waitpid`, `wait3`, `wait4`
- ◦ `signal`, `kill`
- ◦ `exit`

- You are allowed to use other functions to carry out the bonus part as long as their use is justified during your defence. For example, to use `tcgetattr` is justified in certain case, to use `printf` because you are lazy isn't. Be smart!

- You can ask questions on the forum & Slack.

Chapter V

Mandatory part

- You must program a mini `UNIX` command interpreter.
- This interpreter must display a prompt (a simple `"$> "` for example) and wait till you type a command line, validated by pressing enter.
- The prompt is shown again only once the command has been completely executed.
- The command lines are simple, no pipes, no redirections or any other advanced functions.
- The executable are those you can find in the paths indicated in the `PATH` variable.
- In cases where the executable cannot be found, it has to show an error message and display the prompt again.
- You must manage the errors without using `errno`, by displaying a message adapted to the error output.
- You must deal correctly with the `PATH` and the environment (copy of system `char **environ`).
- You must implement a series of builtins: `echo`, `cd`, `setenv`, `unsetenv`, `env`, `exit`.
- You can choose as a reference whatever shell you prefer.
- You must manage expansions $ and ~

`Read the man carefully.`

Here is a use example of your `minishell` :

```
$> cd /dev
$> pwd
/dev
$> ls -l
total 0
crw-rw----  1 root   video     10, 175 dec 19 09:50 agpgart
lrwxrwxrwx  1 root   root            3 dec 19 09:50 cdrom -> hdc
lrwxrwxrwx  1 root   root            3 dec 19 09:50 cdrom0 -> hdc
drwxr-xr-x  2 root   root           60 dec 19 09:50 cdroms/
lrwxrwxrwx  1 root   root            3 dec 19 09:50 cdrw -> hdc
lrwxrwxrwx  1 root   root           11 dec 19 09:50 core -> /proc/kcore
drwxr-xr-x  3 root   root           60 dec 19 09:50 cpu/
drwxr-xr-x  3 root   root           60 dec 19 09:50 discs/
lrwxrwxrwx  1 root   root            3 dec 19 09:50 disk -> hda
lrwxrwxrwx  1 root   root            3 dec 19 09:50 dvd -> hdc
lrwxrwxrwx  1 root   root            3 dec 19 09:50 dvdrw -> hdc
crw-------  1 root   root      29,   0 dec 19 09:50 fb0
lrwxrwxrwx  1 root   root           13 dec 19 09:50 fd -> /proc/self/fd/
brw-rw----  1 root   floppy     2,   0 dec 19 09:50 fd0
brw-rw----  1 root   floppy     2,   1 dec 19 09:50 fd1
crw-rw-rw-  1 root   root       1,   7 dec 19 09:50 full
brw-rw----  1 root   root       3,   0 dec 19 09:50 hda
brw-rw----  1 root   root       3,   1 dec 19 09:50 hda1
brw-rw----  1 root   root       3,   2 dec 19 09:50 hda2
brw-rw----  1 root   root       3,   3 dec 19 09:50 hda3
brw-rw----  1 root   root       3,   5 dec 19 09:50 hda5
brw-rw----  1 root   root       3,   6 dec 19 09:50 hda6
$> kwame
kwame: command not found
$>
```

Chapter VI

Bonus part

Quite a few features will be on the menu of the 21sh and 42sh projects. Below are some bonuses that you can start implementing immediately. Only if you wish to do so!

- Management of signals and in particular `Ctrl-C`. The use of global variables is allowed as part of this bonus.
- Management of execution rights in the PATH.
- Auto completion.
- The separation of commands with ";".
- Other bonuses that you will think to be useful.