Minishell

As beautiful as a shell

Summary: The objective of this project is for you to create a simple shell. Yes, your little bash or zsh. You will learn a lot about processes and file descriptors.

Version: 6

Contents

Chapter I

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 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.

Chapter II

Common Instructions

- Your project must be written in accordance with the Norm. If you have bonus files/functions, they are included in the norm check and you will receive a `0` if there is a norm error inside.
- Your functions should not quit unexpectedly (segmentation fault, bus error, double free, etc) apart from undefined behaviors. If this happens, your project will be considered non functional and will receive a `0` during the evaluation.
- All heap allocated memory space must be properly freed when necessary. No leaks will be tolerated.
- If the subject requires it, you must submit a `Makefile` which will compile your source files to the required output with the flags `-Wall`, `-Wextra` and `-Werror`, use cc, and your Makefile must not relink.
- Your `Makefile` must at least contain the rules `$(NAME)`, `all`, `clean`, `fclean` and `re`.
- To turn in bonuses to your project, you must include a rule `bonus` to your Makefile, which will add all the various headers, librairies or functions that are forbidden on the main part of the project. Bonuses must be in a different file `_bonus.{c/h}`. Mandatory and bonus part evaluation is done separately.
- If your project allows you to use your `libft`, you must copy its sources and its associated `Makefile` in a `libft` folder with its associated Makefile. Your project's `Makefile` must compile the library by using its `Makefile`, then compile the project.
- We encourage you to create test programs for your project even though this work **won't have to be submitted and won't be graded**. It will give you a chance to easily test your work and your peers' work. You will find those tests especially useful during your defence. Indeed, during defence, you are free to use your tests and/or the tests of the peer you are evaluating.
- Submit your work to your assigned git repository. Only the work in the git repository will be graded. If Deepthought is assigned to grade your work, it will be done after your peer-evaluations. If an error happens in any section of your work during Deepthought's grading, the evaluation will stop.

Chapter III

Mandatory part

Program name	`minishell`
Turn in files	
Makefile	Yes
Arguments	
External functs.	`readline, rl_clear_history, rl_on_new_line, rl_replace_line, rl_redisplay, add_history, printf, malloc, free, write, access, open, read, close, fork, wait, waitpid, wait3, wait4, signal, sigaction, kill, exit, getcwd, chdir, stat, lstat, fstat, unlink, execve, dup, dup2, pipe, opendir, readdir, closedir, strerror, perror, isatty, ttyname, ttyslot, ioctl, getenv, tcsetattr, tcgetattr, tgetent, tgetflag, tgetnum, tgetstr, tgoto, tputs`
Libft authorized	Yes
Description	Write a shell

Your shell should:

- Not interpret unclosed quotes or unspecified special characters like \ or ;.
- Not use more than one global variable, think about it and be ready to explain why you do it.
- Show a prompt when waiting for a new command.
- Have a working History.
- Search and launch the right executable (based on the PATH variable or by using relative or absolute path)
- It must implement the builtins:
  - `echo` with option `-n`
  - `cd` with only a relative or absolute path
  - `pwd` with no options

  - `export` with no options
  - `unset` with no options
  - `env` with no options or arguments
  - `exit` with no options

- `'` inhibit all interpretation of a sequence of characters.
- `"` inhibit all interpretation of a sequence of characters except for $.
- Redirections:
  - `<` should redirect input.
  - `>` should redirect output.
  - “`<<`” read input from the current source until a line containing only the delimiter is seen. it doesn’t need to update history!
  - “`>>`” should redirect output with append mode.
- Pipes `|` The output of each command in the pipeline is connected via a pipe to the input of the next command.
- Environment variables ($ followed by characters) should expand to their values.
- `$?` should expand to the exit status of the most recently executed foreground pipeline.
- `ctrl-C ctrl-D ctrl-\` should work like in bash.
- When interactive:
  - `ctrl-C` print a new prompt on a newline.
  - `ctrl-D` exit the shell.
  - `ctrl-\` do nothing.

readline function can produce some leak you don’t need to fix this.
But beware your own code should not produce leaks.
You should limit yourself to the subject description.
Anything not asked is not required.
For every point, if you have any doubt take bash as a reference.

Chapter IV

Bonus part

- If the Mandatory part is not perfect don't even think about bonuses
- `&&`, `||` with parenthesis for priorities.
- the wildcard `*` should work for the current working directory.