CS 392, Systems Programming: Assignment 5 Minishell

Do not exec #!/bin/bash

Overview

For this assignment your goal is to write a basic shell program in C with a few built-in commands. This shell must implement color printing, `cd` and `exit` commands as built-ins, signal handling, and fork/exec for all other commands.

Objective

This program will involve making a shell. The main loop of this function should print the current working directory in blue, then wait for user input. When the user input has been entered, the program should act accordingly.

The program will be called from the command line and will take no arguments.

```
$ ./minishell
[/home/user/minishell]$
```

Prompt/Color Printing

The prompt always displays the current working directory (man 2 getcwd) in square brackets followed immediately by a $ and a space. The directory should be printed in bright blue. Place these color definitions beneath the includes of your minishell.c file.

```
#define BRIGHTBLUE "\x1b[34;1m"
#define DEFAULT    "\x1b[0m"
```

These strings will allow you to print to the terminal in bright blue or the default color for text, based on the terminal's properties.

For example, to print just the string "Hello, world!" in bright blue, you would write:

```
printf("%sHello, world!\n", BRIGHTBLUE);
```

Adding the escape sequence to your string will change the color of anything printed after it, so if you want to return to printing in the default terminal color, make sure you switch it back in the printf() function call.

Supported Commands

You will have to manually implement these commands:

1. `cd`

   If `cd` is called with no arguments or the one argument ~, it should change working directory to the user's home directory. Do NOT hard code any specific folder as the home directory. Instead work with the following:

   - `man 2 getuid`
   - `man 3 getpwuid`
   - `man 2 chdir`

   If `cd` is called with one argument that isn't ~, it should attempt to open the directory specified in that argument.

   Extra Credit with `cd`

   Don't worry about changing into a directory whose name contains spaces. For 10 points of extra credit, you can implement that feature. Your shell would have to support enclosing the directory name in double quotes, as in:

   ```
   cd "some folder name"
   ```

   or

   ```
   cd "some"" folder ""name"
   ```

   If a quote is missing, as in:

   ```
   cd "Desktop
   ```

   print `"Error: Malformed command.\n"` *and return to the prompt.*

   For an additional 5 points, support ~ with trailing directory names, as in:

   ```
   cd ~/Desktop
   cd ~/"some folder name"
   ```

2. `exit`

   The `exit` command should cause the shell to terminate and return EXIT_SUCCESS. Using `exit` is the only way to stop the shell's execution normally.

   All other commands will be executed using `exec`. When an unknown command is entered, the program forks. The child program will exec the given command and the parent process will wait for the child process to finish before returning to the prompt.

3. Error Handling

   Errors for system/function calls should be handled as they have been in previous assignments. At a minimum, you will need to incorporate the following error messages into your shell. The last %s in each line below is a format specifier for strerror(errno).

```
"Error: Cannot get passwd entry. %s.\n"
"Error: Cannot change directory to '%s'. %s.\n"
"Error: Too many arguments to cd.\n"
"Error: Failed to read from stdin. %s.\n"
"Error: fork() failed. %s.\n"
"Error: exec() failed. %s.\n"
"Error: wait() failed. %s.\n"
"Error: malloc() failed. %s.\n" // If you use malloc().
"Error: Cannot get current working directory. %s.\n"
"Error: Cannot register signal handler. %s.\n"
```

   If you use other system/function calls, follow a similar format for the corresponding error messages.

Signals

Your minishell needs to capture the SIGINT signal. Upon doing so, it should return the user to the prompt. Interrupt signals generated in the terminal are delivered to the active process group, which includes both parent and child processes. The child will receive the SIGINT and deal with it accordingly. Make sure you use only async-safe functions in your signal handler.

One suggestion would be to use a single volatile sig_atomic_t variable `interrupted` that is set to true inside the signal handler. Then, inside the program's main loop that displays the prompt, reads the input, and executes the command, don't do anything if that iteration of the loop was interrupted by the signal. If read fails, you need to make sure it wasn't simply interrupted before erroring out of the minishell. Finally, set `interrupted` back to false before the next iteration of the main loop.

Sample Executions

```
$ ./minishell
[/home/user/minishell]$ echo HI
HI
[/home/user/minishell]$ cd ..
[/home/user]$ cd minishell
[/home/user/minishell]$ cd /tmp
[/tmp]$ cd ~
```

```
[/home/user]$ cd minishell
[/home/user/minishell]$ ls
makefile  minishell  minishell.c
[/home/user/minishell]$ pwd
/home/user/minishell
[/home/user/minishell]$ cd
[/home/user]$ pwd
/home/user
[/home/user/minishell]$ nocommand
Error: exec() failed. No such file or directory.
[/home/user]$ cd minishell
[/home/user/minishell]$ ^C
[/home/user/minishell]$ sleep 10
^C
[/home/user/minishell]$ exit
$
```

Submission Requirements

Submit a zip file called minishell.zip containing minishell.c and a makefile. You may choose to factor out functions into separate .c/.h files. Any structure is acceptable as long as your makefile can build the project.