CS 392
Spring 2024 Systems Programming

Homework 4

1 Objective

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, certain commands as built-ins, signal handling, and fork/exec for all other commands.

2 Main Task

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.

Unlike other assignments where there are multiple smaller tasks, this assignment contains only one task. We describe different components of this assignment in this section.

2.1 Prompt: Color Printing

The prompt always displays the current working directory in square brackets followed immediately by a `>` and a space. The directory should be printed in **blue**. Place these color definitions beneath the includes of your `minishell.c` file:

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

These strings will allow you to print to the terminal in blue or the default color for text, based on the terminal's properties. For example, to print just the string `"Hello, world!"` in blue, you would write:

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

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:

```
printf("%sHello, world! in blue%s\n", BLUE, DEFAULT);
printf("I'm back!\n");
```

2.2 Commands

2.2.1 Manually Implemented Commands

You will have to manually implement the following five commands by using related C functions, instead of invoking the binary from the system.

- `cd`
  If `cd` is called with no arguments or the one argument `~`, it should change the current working directory

to the user's home directory. Do **NOT** hard code any specific folder as the home directory. Instead work with `getuid()`, `getpwuid()`, and `chdir()`.

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

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

- `pwd`
  This is the same as the built-in command to print current working directory.

- `lf`
  This is similar to the `ls` command, except that you don't need to consider any command-line arguments and flags. Just typing `lf` command and it'll list all the files under the current directory, one file a line. Do not list `.` and `..`, but do include hidden files.

- `lp`
  The `lp` (***l**ist **p**rocess*) is a new command in our minishell that doesn't exist in the original Linux system. What this command does is to simply list all the current processes in the system in the following format:

```
<PID> <USER> <COMMAND>
```

where `<COMMAND>` is the command that invokes that process. It is a simplified version of `top` command: the output of `lp` only contains the first, second, and the last column of the output from `top`. The following is a segment of an example output:

```
    1 root systemd
    2 root kthreadd
    3 root rcu_gp
    4 root rcu_par_gp
    6 root kworker/0:0H-kblockd
  520 root loop7
97036 ubuntu sshd
```

To find out all the processes, you must visit `/proc/` directory. Under that directory, each of the subdirectories whose name is an integer number represents a process, and that number is its process ID. The owner of the subdirectory is also the user of the process, and the command that invoked that process can be read from `/proc/<PID>/cmdline`.

Note that the output has to be sorted based on their PIDs, from smallest to the largest. As a system programmer, you should know how to use `qsort()` provided by the C library by now.

2.2.2 Other Commands

All other commands will be executed using `exec()`. In a command is not one of the built-ins (as listed in the section above), 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.

You can use any flavor of `exec()` you like, but do NOT use `system()` function, or `popen()` related functions.

2.3 Signal Handling

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.

One suggestion would be to use a single `volatile sig_atomic_t` variable, say called `interrupted` that is set to true inside the signal handler. Then, inside the program's main loop that displays the prompt, read the input, and execute 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.

To declare a volatile `sig_atomic_t` variable, simply use:

```
volatile sig_atomic_t interrupted;
```

2.4 Error Handling

Errors for system/function calls should be handled with printing error messages to `stderr`. 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.

3 Sample Run

You don't need to worry about empty lines or spaces.

Sample Run

```
$ ./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/]> 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
$
```

4 Grading

We will manually test your program with multiple command inputs. Each command input will have the same amount of points, accumulating to 100 points in total. You need to read the document fully to make sure you have completed everything mentioned in this document.

After accumulating points from the testing above, we will inspect your code and apply deductibles listed below. The lowest score is 0, so no negative scores.

- **-100:** the code does not compile, or executes with run-time error;
- **-100:** the code is generated by AI, and/or through reverse engineering on the tester;
- **-100:** the code used `system()` or `popen()`-related functions;
- **-100:** the code invoked native shell programs such as `sh` and `bash`;
- **-50:** error messages are not printed through `stderr`;
- **-50:** hard coded specific folder as home directory, or user name, *etc*;
- **-50:** used `signal()` function instead of `sigaction()`;
- **-30:** memory leak through valgrind (only "definitely lost" category);
- **-10:** no pledge and/or name in C file.

Note: if you need to link some libraries during linking phase, such as `-lm` for `math` library, you have to clearly state it at the top of your C file, next to the pledge. Failing to do so will be considered as failure to compile, and will result in a 0 for the homework. Absolutely no exceptions/negotiation.

Earlybird Extra Credit: 2% of extra credit will be given if the homework is finished two days before the deadline. For specific policy, see syllabus.

Deliverable

Submit a single `minishell.c`.