Assignment #4 – Implementing a “command executor” [60 pts]

You are expected to do your own work on all homework assignments. (See the statement of Academic Dishonesty on the Syllabus.)

Check the Syllabus for the late assignment policy for this course.

How to turn in?

Assignments need to be turned in via Laulima. Check the Syllabus for the late assignment policy for the course.

What to turn in?

You should turn in a tarred archive named `ics332_hw4_USERNAME.tar` that contains a single top-level directory called `ics332_hw4_USERNAME`, where `USERNAME` is your UH username. In that directory you should have all the files **named exactly** as specified in the questions below.

Expected contents of the `ics332_hw4_USERNAME` directory:

- `Makefile`: The makefile provided to you (see below)
- `command_executor.c`: The code provided to you (see below)
- `solution.c`: your code (see below)

Your program must compile without any warnings or errors.

Environment

For this assignment **you need a Linux environment** (see Assignment #0).

Exercise #1: A tiny Shell-line “command executor” [60 pts]

Preliminaries

In this assignment we implement a “command executor”, in other words, a very simplistic Shell.

You should first download the ./command_executor.tar archive, which contains the starting code for the assignment. Once you’ve downloaded the archive you can uncompress it and

```
% cd command_executor
% ls
  Makefile    README    command_executor.c      solution.c
```

The file solution.c contain an empty solution that does nothing but print "not implemented yet". This is the file you'll have to modify.

Important: Do not modify command_executor.c or Makefile

You can build and run the program right now (see the README file). It should prompt you for a command to run (e.g., "/bin/ls -la") and then ask you two Yes/No questions about how to run it (only one Yes is allowed):

- Should the output of the command be redirected to a file?
- Should the output of the command be redirected to another command?

So, we have three cases:

- Plain execution of the command: no redirection of output
- Execution of the command with redirection of output to a file
- Execution of the command with redirection of output to another command, whose output is not redirected

And you'll see in solution.c that you thus have three functions to implement:

- void execute_plain(char *cmd, char *const argv[])
- void execute_output_to_file(char *cmd, char *const argv[])
- void execute_output_to_other(char *cmd, char *const argv[])

All three function are passed the command (e.g., "/bin/ls") and its arguments (e.g., {"-la", NULL}). In each of the questions below we implement one of these functions to make our tiny Shell work.

For testing, you'll have to run various commands. Good candidates are ls, cat, wc, cp, mv, rm, sleep, and many others.

Your program must continue running until you quit it. In other words, you must always return from the function you are implementing.

WARNING:
This is a real Shell that will run real commands. So, if you ask it to run the command "/bin/rm -rf ~" it WILL delete all your files! So, when testing it, be careful!

Question #1: Implementing execute_plain() [20pts]

As we've discussed in class, this is pretty easy:

- Start a child process
- Have the child process exec the command
- Wait for the child's completion

The only requirement is that if the command fails for one reason or another, the command executor should print " ** Command failed **" (and *it's ok if there is other "error output" produced by the commands*). Otherwise, it should print " ** Command successful **".

Below is a sequence of interactions with the command executor that you should use as guidelines for implementing execute_plain() (note the user answers to the questions, which sadly I can't show in a different font color due to Jekyll's limitations with liquid tags):

```
> Enter a command you want to run:  /bin/ls
> Should the output be re-directed to a file? [Y/N] N
> Should the output be re-directed to another command? [Y/N] N
Makefile                command_executor        command_executor.o
README                  command_executor.c      solution.c
  ** Command successful **
```

The above command succeeds, and we see its output.

```
> Enter a command you want to run: bad command
> Should the output be re-directed to a file? [Y/N] N
> Should the output be re-directed to another command? [Y/N] N
  ** Command failed **
```

The above command fails because our command executor needs **fully specified absolute paths** to programs (our command executor doesn't know about the $PATH environment variable). In other terms, the call to exec will fail unless given a fully specified path.

Note that in your real Linux Shell, you can find out the path to any command by typing which and then the command. For instance, "which ls" which tell you "/bin/ls".

```
> Enter a command you want to run:  /bin/ls /stuff
> Should the output be re-directed to a file? [Y/N] N
> Should the output be re-directed to another command? [Y/N] N
  ** Command failed **
```

The above command fails because there is no /stuff file. This simply means that the /bin/ls command exited with a non-zero exit code, and the command executor, noticing this, knows that the command has failed.

```
> Enter a command you want to run:  /usr/bin/wc -l /etc/passwd
> Should the output be re-directed to a file? [Y/N] N
```

```
  ** Command successful **
```

The above command succeeds and prints the number of lines in file /etc/passwd.

That's about it. Note that our command executor does not handle commands that would require the user to type input. It also doesn't deal with Shell built-in commands like cd, echo, etc. It's very limited.

My own solution for this question consists of 15 lines of code, including error checking.

Question #2: Implementing execute_output_to_file() [25pts]

We now implement the execute_output_to_file() function, which is called when the user answers Yes to the "Should the output be re-directed to a file? [Y/N]" question. In class we have discussed output redirection quite a bit, and the same techniques are to be applied here. Here are some example interactions with the command executor:

```
> Enter a command you want to run:  /bin/ls
> Should the output be re-directed to a file? [Y/N] Y
> Enter the filepath:  output.txt
  ** Command successful **
> Enter a command you want to run:  /bin/cat output.txt
> Should the output be re-directed to a file? [Y/N] N
> Should the output be re-directed to another command? [Y/N] N
Makefile
README
command_executor
command_executor.c
command_executor.o
output.txt
solution.c
  ** Command successful **
```

The first above command (/bin/ls) succeeds and has its output redirected to file output.txt. Then, we can actually run /bin/cat output.txt, and see the file content, which is the output from /bin/ls.

```
> Enter a command you want to run:  /bin/ls
> Should the output be re-directed to a file? [Y/N] Y
> Enter the filepath:  /root/stuff.txt
  ** Command failed **
```

The above command fails because we don't have permission to open file /root/stuff.txt for writing. When failing to fopen() the file, the command executor declares the command failed.

```
> Enter a command you want to run:  ls
> Should the output be re-directed to a file? [Y/N] Y
```

The above command fails because, like in the previous question, we need full paths (i.e., /bin/ls instead of just ls)

My own solution for this question consists of 17 lines of code, including error checking.

Question #3: Implementing execute_output_to_other() [15pts]

Finally, we now implement the execute_output_to_other() function, which is called when the user answers Yes to the "Should the output be re-directed to another command? [Y/N]" question. This has to be done with popen() and pclose().

Here are interactions with the command executor:

```
> Enter a command you want to run:  /bin/ls -la
> Should the output be re-directed to a file? [Y/N] N
> Should the output be re-directed to another command? [Y/N] Y
> Enter that other command:  /usr/bin/wc -l
    11
  ** Command successful **
```

The above command succeeds, and prints the number of lines in the output of ls -la (which happens to be 11 when I ran it).

For this question, we won't test error behavior. In other words, we'll only test your code with known-to-be-successful commands combinations.

My own solution for this question consists of 26 lines of code, including error checking.

Hint: It will be very difficult to do this without using the dup() system call, which we discussed in class in the context of output redirection.