CS347m Operating Systems
Spring 2020-21
Lab 1b (Ungraded)

1. This is the second (and probably the more interesting) part of Lab1. In the previous part of the lab, we solved questions related to processes and usage of basic system calls. Now, this part of the lab builds upon that to implement a real world application - a simple **interactive shell**.
2. Skeleton code is provided that takes care of parsing the user input, so you don't have to worry about implementing a custom parser for the shell (Phew!). Ofcourse, you are free to implement your own parser that takes care of all possible scenarios, as well.
3. This lab will not be graded, and hence is optional. You do not need to submit this lab (and there is no deadline). **This is a practice lab with lots to learn.**

Building a (simple) shell

Write a program that implements a simple command shell for Linux. Sample code inside `shell.c` is provided to tokenize strings (to deal with user commands).
You can assume that the command to run and its arguments are separated by one or more spaces in the input, so that you can "tokenize" the input stream using spaces as the delimiters. For this part, you can assume that the Linux built-in commands are invoked with simple command-line arguments, and without any special modes of execution like background execution, I/O redirection, or pipes. You need not parse any other special characters in the input stream. Please do not worry about corner cases or overly complicated command inputs for now; just focus on getting the basics right.
Do not use the `system()` library function.

Few syscalls that you'll find helpful for this part of the lab: **`fork, execvp, wait, signal/sigaction`**

A few points to note:

- Write all your code inside **`shell.c`** (You're also free to split it into multiple source files, just make sure all the source files can be compiled to a single executable binary, i.e., there must be a single **`main()`** function or simply, a single entry point to your program. Do whichever works best for you, just make sure it compiles and runs properly.
- A **Makefile** is provided for this part of the lab, for ease of compilation. In the terminal, while being in the same directory as your code for the shell, do a
  `make all` to compile your code (all source files in the **same directory**) into a single executable binary.
  `make run` to launch your shell application (after a successful compilation)

`make clean` to remove the object files and the executable (Do this when you make changes to the code, before compiling again)

Use any creative message as a prompt of the shell (or just leave it as it is) waiting for user commands (The examples below use "**myshell>**" as the prompt, which is the default prompt). Below are the commands you need to implement in the shell, and the expected behavior of the shell for that command.

- All simple standalone built-in commands of Linux e.g., (ls, cat, echo, sleep) should be executed,as they would be in a regular shell. All such commands should execute in the foreground, and the shell should wait for the command to finish before prompting the user for the next command. Any errors returned during the execution of these commands must be displayed in the shell.
  Hint: You'll need to fork a process for this part.

- In this part, you'll be implementing three simple custom commands:

  1. `hello` that simply prints the message "Hello there !!". Make sure you terminate the string with a newline Here's an example:

```
myshell> hello
Hello there !!
myshell>
```

  2. `tracker` that prints the message "This command has been called *x* number of times". Here *x* is a strictly monotonically increasing counter (initially set to **1**) that must be incremented every time this command is executed by the user inside your shell Make sure you terminate the string with a newline.
  Here's an example:

```
myshell> tracker
This command has been called 1 times
myshell> tracker
This command has been called 2 times
myshell> tracker
This command has been called 3 times
```

  3. `quit` that will exit your shell application silently, without printing anything.

- Your shell should be able to handle Unix signals. Signals mentioned below should be handled by your shell:

SIGINT: Whenever the user presses **Ctrl-C** (displayed as ^C), then your shell program should show the message "The program is interrupted, do you want to exit ? [Y/N]". If Y is pressed then terminate otherwise keep the shell running. Here's an example of how it may look:

```
myshell> ^C
The program is interrupted, do you want to exit ? [Y/N]
N
myshell>
```

(Notice that the user entered N and the shell is back waiting for user input)
You can be assured that SIGINT will not be sent while your shell is busy executing some long running command.
Hint: Check out `signal()` or `sigaction()`

Your shell must gracefully handle errors. An empty command (hitting Enter key without entering anything) should simply cause the shell to display a prompt again without any error messages. **For all incorrect commands or any other erroneous input, the shell itself should not crash**. It must simply notify the error and move on to prompt the user for the next command.

For all commands, you must take care to terminate and **carefully reap any child process the shell may have spawned**. Verify this property using the `ps` command during testing (more on it below). When not running any command, there should only be the one main shell process running in your system, and no other children.

To check if there are any zombie processes, open a different tab/window in the terminal you're using and enter the following

```
ps aux | grep Z
```

Under the STAT column, if you see a 'Z' in the entry corresponding to a command that you recently executed in your shell application (for example `ls`), then it means you haven't reaped the process.

Tip: After executing a command (that involved forking a new process) from your shell, check whether that process was properly reaped or it became a zombie. Do this few times for every command executed during the initial phase of writing your shell, until you're sure that no zombies are being created.

More shell

A typical shell not all creates processes and loads executable, but also uses file descriptor based mechanisms to handle IO redirection. Try to implemented the following types of operators for IO redirection,

- ";;" the separator operator to execute a list of command one after another
- "&" put a process to execute in the background
- "<" input redirection
- ">" output redirection
- "|" the pipe operator (to chain output of command as input to the other)

Note that several of these operators can be chained together for an interesting sequence of commands for the shell. You will have to lookup for variations in the usage of fork, exec, wait and also never system calls like dup, pipe etc.

Examples:

```
cat filename.txt | wc -l
cat "helloworld" > hw.txt
cat < hw.txt
cat filename.txt | grep "abc" | grep "xyz"
ls -l ;; data ;; hostname
find / -name filename.txt &
```