COMP 4511- System and Kernel Programming in Linux
Programming Assignment #2

Assignment 2 - Simple Linux Shell (Part I)

A shell is a user-level program to launch, suspend, and kill other programs. It provides an interface for a user to send commands to the underlying operating system, parse the commands, invoke corresponding system functions, and finally respond to the user.

In the current Linux system, there is a variety of shell programs, such as

- Bourne shell (sh)
- Bourne-again shell (bash)
- C shell (csh)

Shell Program Lifecycle

A shell program is running in an infinite loop (until an `exit` command to terminate the shell program), taking the following steps:

1. **Print a prompt:** A default prompt can be hardcoded into a shell. The shell can be designed to print the current directory as part of the prompt. For example, each time you use `cd` to change to a different directory, the prompt will change accordingly.
2. **Read a command line:** The shell performs a blocking read operation until the user types the `NEWLINE'\n'`. Then the command line string is returned to the shell.
3. **Parse a command line:** Parse the command line string into command name and a list of its parameters.
4. **Find a file:** The program specified by the command line could locate in `/bin` and `/usr/bin`. In bash, it is determined by the environment variable `$PATH`. Otherwise, the shell notifies the user that the command is unable to find.
5. **Execute a command:** To execute a command, the shell uses multiple child processes to accomplish this, availing itself of the system calls: `fork()`, `execve()`, and `wait()`. You are assumed to have knowledge on the three syscalls. If not, check any useful material available.
6. **Repeat:** Once the child (the launched program) finishes, the shell goes to step 1, waiting for other commands from users.

Problem Statement

In Simple Linux Shell (Part I), your task is to implement a primitive shell that knows how to launch new programs in both foreground and background. There is **no skeleton code** provided in this programming assignment (and the next programming assignment), but you can use the lab exercise as a starting point to develop this simple Linux shell.

Feature 1 – Customize the prompt output format:

The shell prompt should be in the following format:

```
[DIR] myshell>
```

For example, if the current directory is `/home/cs/test`, the shell prompt should be as follows:

```
[test] myshell>
```

Feature 2 – Input arguments handling

Your shell program needs to support the following input argument formats:

Format 1:

```
$ <program name> <arg1> <arg2> ... <argN>
```

In this example, Hello is the first argument:

```
$> echo Hello
```

Format 2:

Each argument is written in the format of `-[character]`, where the character part can be any alphabetical letter (i.e. A-Z or a-z) or digits (i.e. 0-9). The order is not important.

```
$ <program name> -a -b ... -n
```

Here is an example of short arguments. '`a`' and '`l`' are short arguments (order can be reversed)

```
$> ls -a -l
```

Format 3:

Packed argument list based on the format 2. The following example should produce the same result as the above example.

```
$> ls -al
```

Format 4:

Mixing the format 1, 2 and 3 in different order. For example, the following command will list out all files from the root directory and display the file size using human readable format

```
$> ls / -al -h
```

One or more empty spaces and/or tab characters separate each argument. In other words, you **CANNOT** assume that each argument is separated by exactly one space.

Feature 3 –Background process support

The command follows the basic form (see feature 2):

```
$ <program name> <arg1> <arg2> ... <argN> [&]
```

```
$ <program name> -a -b ... -n [&]
```

The last character `'&'` is **OPTIONAL**. You can assume that the program will be executed in the foreground if `'&'` is missing.

If `'&'` is provided, the shell will execute the program in background. When a program executing in background, the shell does not have to wait for the command to finish before it prompts the user for another command.

We can assume that the `'&'` token is separated from the last argument with one or more spaces or tab characters. There won't be extra letters after the `'&'` token.

Feature 4 – Search directories

The shell should first search the program in the current directory. If the target program is not found, the shell should search for the target program from the directories defined by the $PATH environment variable. If the target program is not found, print the following error message:

```
<program name>: Command not found.
```

For example, `program_not_exist` should not be a valid program in the current directory and should not be found in any directory in $PATH. You should see the following output:

```
[test] myshell> program_not_exist
program_not_exist: Command not found.
```

Feature 5 – Recognize the exit command

The `exit` command **MUST** be implemented using `exit()` defined in `<stdlib.h>`. For example:

```
[test] myshell> exit
```

After running the exit command, the default shell of your Linux distribution is returned. Please note that you should clean up all allocated memory using `free()` before you exit your shell program.

Feature 6 – Recognize the cd command

The `cd` command **MUST** be implemented using `chdir()` defined in `<unistd.h>` to change to a new directory. If it has no argument, it should change to the `$HOME` directory. For example, suppose my home director is `/home/cspeter`, the expected output will be:

```
[test] myshell> cd
[cspeter] myshell>
```

In addition, you should support the followings:

- Change to the current directory: `cd .`
- Change to the upper level directory: `cd ..`
- Change to the root directory: `cd /`

Feature 7 – Recognize the child command

The child command MUST be implemented using `fork()`, `wait()`, and `sleep()`. The command `child [n]` creates a child process that runs for `n` seconds. After that, it prints out the PID of the child process and the status code when the child process terminates. The expected output will be (please note that the PID will be automatically assigned by the system):

```
[test] myshell> child 5
child pid 1955 is started
child pid 1955 is terminated with status 0
[test] myshell> child 12
child pid 1956 is started
child pid 1956 is terminated with status 0
```