Programming Assignment #2

- **Due:** 9 AM on Saturday 17 February, 2018.
- **Team Based: You may work in groups of 4.**
- **Required Artifacts:**
  - The source code (.c) file. Only one student needs to upload the file to Titanium for the entire group. Write the following information as comments at the beginning of the file.
    - All the student names with the associated CWIDs
      - Ex: Gina Ackerman 135798
    - Assignment #
  - Large screenshot(s) of your program output.
- **Note:** No late assignment will be accepted after 24 hours of the due date, i.e., your group will get 0 point.
- **Grading Rubric:**
  - Build cleanly: 10 points
  - Correct Output: 10 points
  - Functions:
    - long read_command(char commandBuffer[]); // 10 points
    - void parse_command(char commandBuffer[], char *args[], long length); // 20 points
    - int main(); // 20 points

Requirement

In this exercise, you will write a simple shell program. When the program is running, the parent process will display the prompt as follows:

```
shell>
```

The parent process will wait for the user to enter a command such as ls –al or any Linux/Unix command:

shell> ls –al

The parent process will perform the following actions:

- Read in the command that user enters
- Parse the command line to create the arguments for ***execlp(…)***

- Call ***folk()*** to create a child
- Call ***wait()*** to wait for the child to terminate

The child process will use execlp() to replace its program with the program entered at the command line. After the child produces the output, the parent will prompt the user for another command. For example:

shell> cat HowToCompile

```
gcc –o shell shell.c
Child Complete
shell>
```

The parent will repeat the sequence above until the user types the exit or quit command, which will cause the parent process to exit.

NOTE: Please make sure to error-check all system calls in this assignment. This is very important in practice and can also save you hours of debugging frustration. fork(), execlp(…), and wait(…) will return -1 on error. Hence, you need to always check their return values and terminate your program if the return value is -1. For example:

```
pid_t pid = fork();
if (pid < 0)
{
    perror("Could not fork a child process");
    exit(-1);
}
```

The ***perror(…)*** function above will print out fork followed by the explanation of the error.