ECE 2220: System Programming Concepts **Fall 2016**
Machine Problem 7: Mission to Mars. **Due: 11:59 pm, Thursday, November 17**

In this lab, each student is to write a **single** program called `lab7.c`. The program utilizes a parent process to create and communicate with child processes. The student specifically explores:

- Handling processes by use of `fork()` and `wait()`
- Handling signals by use of `signal()`, `alarm()`, `pause()`, and `kill()`
- Opening and writing to terminals.

The objective of the program is to model Mission Control guiding three space craft to Mars and back to Earth. Each space craft caries a Martian Rover, and when the space craft reaches Mars it enters an orbit until the Rover has been launched. Once the Rover has been launched the space craft returns to Earth. Mission control has two primary duties. First, is must transmit updated instructions to the navigation computer on each space craft as the craft travels to Mars and back to Earth. Due to the complex requirements of space travel, the navigation computer cannot be pre-loaded with all way points for the mission, but instead they need to be updated as algorithms at Mission Control make corrections to the flight plan. So, Mission Control must make sure it sends new way point information to each space craft before the navigation computer runs out of way points. Mission Control must also signal each space craft when to launch its Rover, and make sure you send the signal only when a space craft has entered into orbit at Mars.

Operation

Before executing the program, the user should open four console (terminal) windows. Type `tty` in each terminal window to learn that terminal's `pts` number. Then start lab7 and include the four terminal numbers on the command line (e.g., `./lab7 4 7 9 14`). The first terminal window represents Mission Control, and the other three represent the three space craft. You can assume that four terminal windows are open when you start your program and that the user lists the correct terminal numbers.

The main function of the parent is to use `fork()` to spawn three child processes. There is a child process for each space craft, and each child process executes exactly the same code. A fourth child is needed to collect input from the Mission Control terminal window and to send signals (via `kill()`) to the three space craft. The parent waits for the three space craft to either return to Earth or crash (the parent collects the exit codes using `wait()`). After the parent collects the return values from `wait()` it prints a message about the status of the mission for each space craft.

Each space craft is to be initialized to contain **10 way points**, **one Martian Rover**, and to be a distance of **zero clicks** from Earth. Each space craft will take a different path to Mars, so a space craft randomly generates the number of clicks on its path to Mars to be in the range [30, 60). A space craft learns what its number is and where to write its output when the space craft function is initially called. A space craft must print the following message in its terminal when it is launched.

```
Space Craft %d initiating mission.  Distance to Mars %d
```

A space craft also installs three signals (via `signal()`). One signal is for alarms (`SIGALRM`) for use with `alarm(1)` to set a timer every second that does:

- Decrements the way point count by **one every two seconds**.
- Increments the distance from mission control by **one click every second** if going toward Mars, and decrements the distance from mission control by **one click every second** if returning to Earth.

- Once the space craft reaches Mars it enters orbit and stays in orbit until Mission Control directs it to launch its Rover. So, the distance does not change, but the way point count continues to be decremented by one every two seconds. When the space craft enters orbit it reports its status in its space craft terminal. e.g.,

`Space craft` ***`id`*** **`to Mission Control: entered orbit.  Please signal when to launch Rover.`**

- Each second print some message to indicate how far away the space craft is from Mars. You can design any output you like, such as simply printing the distance or printing some symbols to visually show the distance. Also show the number of way points and if the Rover has been launched or not.

The other two signals (**`SIGUSR1`** and **`SIGUSR2`**) are used to send commands to the space craft and are described below.

If a space craft ever runs out of way points it should be terminated by exiting with an *unsuccessful* code. If a space craft launches its Rover, it should begin returning to Earth. If it makes it back to Earth without running out of way points, it should terminate with a *successful* code.

Input

The program should accept the following input from the terminal associated with the Mission Control (where invalid commands should produce a suitable error message):

- **`ln`** – Instructs space craft **`n`** to launch its Rover. **`n`** is one of 1, 2, or 3.
- **`kn`** – Instructs space craft **`n`** to cancel its mission (the process is killed). **`n`** is 1, 2, or 3.
- **`tn`** – Transmit 10 new way points to space craft **`n`**.
- **`q`** – Exits the program.

Given the **`ln`** command, Mission Control should send the signal **`SIGUSR1`** to space craft **`n`**. Upon receiving the signal **`SIGUSR1`**, the space craft should print out that it has launched the Rover, and is leaving Mars orbit to return to Earth. However, if the space craft is not in orbit at Mars when the launch command is received, or if the Rover has already been launched and a second launch command is received, then the space craft self-destructs due to a bug in the launch control software. (Clearly a NASA software designer forgot to check for this error condition!) You should print a suitable message in the space craft's terminal window expressing its distress and exit with an unsuccessful code

Given the **`tn`** command, Mission Control process should send the signal **`SIGUSR2`** to space craft **`n`**. Upon receiving the signal **`SIGUSR2`**, a space craft should increment its count of way points by 10. Whenever a space crafts' way points drops below **5**, it should print the message "**`Houston, we have a problem!`**" If a space craft uses all its way points before a new transmission, the space craft should print the message "**`Space craft n. Lost in Space.`**" and then exit with an unsuccessful exit code.

The parent process collects the exit codes from the three space craft (via the argument value of **`wait()`**) and prints "**`Vaya Con Dios, Space craft n`**" in its terminal window if space craft **`n`** has become lost or "**`Welcome home, Space craft n`**" if space craft **`n`** returned to Earth (and exited with a successful code).

Given the **`kn`** command, Mission Control should terminate the selected child process and print "**`Terminated Space Craft n`**" if space craft **`n`** was canceled with the **`kn`** command. In all of these prints **`n`** is the space craft number.

After the status of all three missions has been determined, Mission Control prints one of these two messages in its terminal: "**`Congratulations team: Mission successful`**" if all three space craft successfully returned home, or "**`Mission failed`**" if any space craft was lost in space.

Given the **`q`** command, Mission Control should terminate all child processes and close the program.

Notes

1. Examples of **all** the techniques needed for this machine problem are found in the lecture notes. See the **`wait.c`**, **`kill.c`**, **`alarm.c`**, and **`signal.c`** series of files. Do **not** use the **`sleep`** function as it interferes with the **`alarm`** function.

2. Use **`ps -ef | grep lab7`** to verify that all of your child processes have terminated. Your final program must not leave any processes hanging. During debugging, if you find a process that is hung, use **`kill pid`** to end the process, where **`pid`** is the process number.

3. Environment comments: Older versions of VirtualBox do not correctly implement alarms, so if you normally use VirtualBox you must ssh to another machine such as the CES apollo computers. VMware works correctly.

4. Submission requirements

The code you submit must compile using the `-Wall` flag and **no** compiler errors or warnings should be printed. To receive credit for this assignment your code must correctly print to multiple terminals and demonstrate handing of processes and signals. Code that does not compile or fails to pass the minimum tests will not be accepted or graded.

5. Submit your file **`lab7.c`** to the ECE assign server. You submit by email to ece_assign@clemson.edu. Use as subject header ECE222-1,#7. When you submit to the assign server, verify that you get an automatically generated confirmation email within a few minutes.

See the ECE 222 Programming Guide for additional requirements that apply to all programming assignments.

Work must be completed by each individual student. See the course syllabus for additional policies.