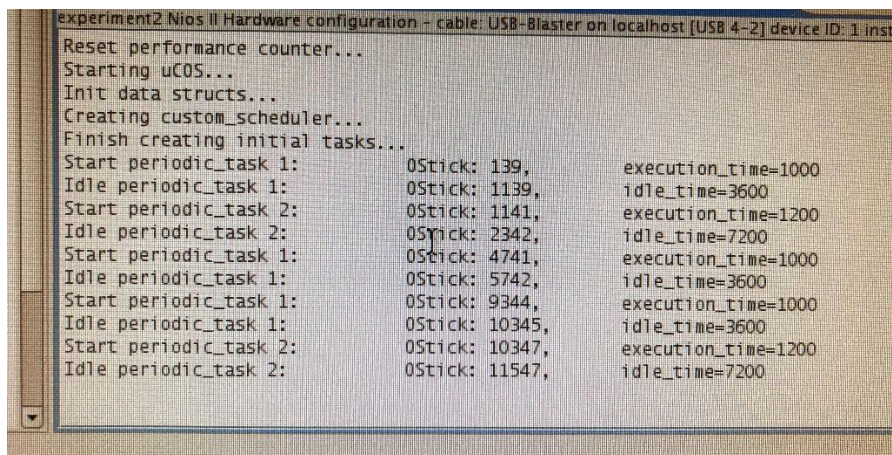


Exercise 1:

| Task Number: | Execution Time (200 X letter) | Idle time (1200 X letter) |
|----------------------|-------------------------------|---------------------------|
| #1: Task 0 (in code) | Fahim = 1000 | Gao = 3600 |
| #2: Task 1(in code) | Ridwan = 1200 | Audrey = 7200 |

Above is the table of the custom periodic tasks build starting from experiment 2 setup where Task 0 in code has a higher priority (lower integer number) than Task 1. As asked for, custom_delay was used for execution time and OSTimeDly for idle time while an enable flag (as in experiment 1) within the if condition was used to lock the scheduler (during non-pre-emptive). The difference is, for non-pre-emptive, the task that started execution would complete fully before beginning the next task since the scheduler was locked before starting the task as during our demo. Whereas in Pre-emptive if task 1 is running and task 0 also wants to start, then task 1 is stopped until task 0 is done after which task 1 continues. The difference is more apparent during Pre-emptive where the idle time for Task 1 is double that of Task 2, also when we experimented with higher exec time we noticed the differences.



```
experiment2 Nios II Hardware configuration - cable: USB-Blaster on localhost [USB 4-2] device ID: 1 insta
Reset performance counter...
Starting uCOS...
Init data structs...
Creating custom_scheduler...
Finish creating initial tasks...
Start periodic_task 1: 0Stick: 139, execution_time=1000
Idle periodic_task 1: 0Stick: 1139, idle_time=3600
Start periodic_task 2: 0Stick: 1141, execution_time=1200
Idle periodic_task 2: 0Stick: 2342, idle_time=7200
Start periodic_task 1: 0Stick: 4741, execution_time=1000
Idle periodic_task 1: 0Stick: 5742, idle_time=3600
Start periodic_task 1: 0Stick: 9344, execution_time=1000
Idle periodic_task 1: 0Stick: 10345, idle_time=3600
Start periodic_task 2: 0Stick: 10347, execution_time=1200
Idle periodic_task 2: 0Stick: 11547, idle_time=7200
```

Figure 1 Console for pre-emptive scheduling displaying OS_tick and exec and idle time

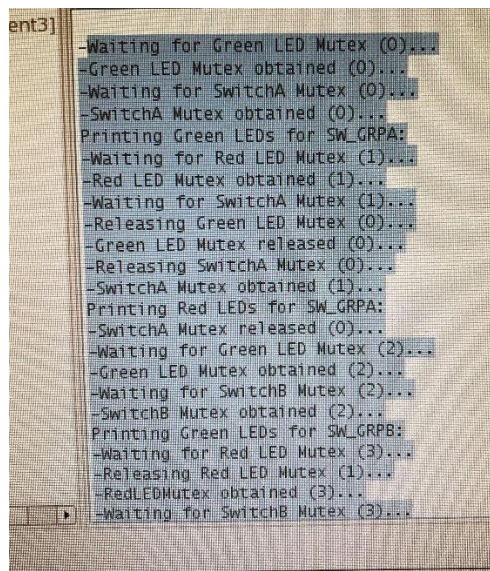
Exercise 2:

After the task was created, we save the lowest period to a temporary variable and compare the task period to the temporary variable, if it's larger than the temporary value we switch the priority of those, to make sure the lowest period has the highest priority. And every time when the new task is creating, we use the same method to update the priority so the one with the least period is displayed with the highest priority (smallest integer). **Also note even when you press a button twice simultaneously, the intended purpose (create a new task or delete the task or vise versa) occurs but the print message prints along with it which shouldn't matter.** For the delete function we keep it the same as experiment 2 as its not touched.

Exercise 3:

Using OSTimeDlyHMSM function, instead of custom delay which has same performance, we set the times as specified on the pdf for each push button after setting the priorities of mutex in define.h as specified. Also as per experiment, My_Mutex_Enable global variable flag was set to used to when the mutex was accepted and released. The results observed in the 3 experiments are detailed below:

- i) To start, as Push button 0 is pressed, the corresponding GREEN LEDs from SWITCH GROUP A are lit and printed. As PB1 activated the RED LEDs, it also waits for SWITCH GROUP A to be released by PBO before it obtains it so it can also be printed. Then when pressing PB2, it waits for GREEN LEDs and lights up and prints the corresponding SWITCH GROUP B. Since the previous button has a lower priority, printing is done at the same time as task 1 since a shared resource are not being competed for because of the mutex. Then PB3 waits and gets the RED LEDs from SWITCH GROUP B once they have been released by PB1 and PB2. Lastly the final LEDs that need to be lit for the amount of time before release is completed.
- ii) The same procedure as above is observed in reverse order, and since All our times are over 2.5seconds other than time 0, we will see an overlap in the message printed but although we are dealing with mutexes, the execution of the LEDs corresponding to the switches will need to complete for one push button operation to advance to the next (priority to the mutex that acquire task). This time more time for the execution to near finish.
- iii) This is interesting for us as when 0 is pressed, we have the whole print and green LED execution completed (turns on then off), and then when 2 is pressed, we also the whole operation with the green LED have it completed, and when 1 is pressed, that's when we still have 0.5seconds left for the lights to stay as well press 3 which starts execution while red LED is lighten up and about to turn off.



```
ent3]
-Waiting for Green LED Mutex (0)...
-Green LED Mutex obtained (0)...
-Waiting for SwitchA Mutex (0)...
-SwitchA Mutex obtained (0)...
Printing Green LEDs for SW_GRP A:
-Waiting for Red LED Mutex (1)...
-Red LED Mutex obtained (1)...
-Waiting for SwitchA Mutex (1)...
-Releasing Green LED Mutex (0)...
-Green LED Mutex released (0)...
-Releasing SwitchA Mutex (0)...
-SwitchA Mutex obtained (1)...
Printing Red LEDs for SW_GRP A:
-SwitchA Mutex released (0)...
-Waiting for Green LED Mutex (2)...
-Green LED Mutex obtained (2)...
-Waiting for SwitchB Mutex (2)...
-SwitchB Mutex obtained (2)...
Printing Green LEDs for SW_GRP B:
-Waiting for Red LED Mutex (3)...
-Releasing Red LED Mutex (1)...
-RedLED Mutex obtained (3)...
-Waiting for SwitchB Mutex (3)...
```

Figure 2 Console of print messages for experiment i)