Computer Systems Engineering Technology
CST 347 – Real-Time Operating Systems

Lab 04 – Scheduling in FreeRTOS Name_____

Possible Points: 10

Instructions

Complete the following procedures. Zip up your final project and upload to Blackboard. Answer the questions in the Blackboard **Lab 4 Actions** "Test".

Procedure

Lab 3 introduced intertask (interprocess) communication through the messaging queue. Queues were independently created and assigned a handle. Tasks sent and received messages to queues using the queue's handle.

This lab will focus on scheduling behavior by observing FreeRTOS' kernel scheduling. You will look at two scheduling schemes implemented by the FreeRTOS kernel.

> FreeRTOS is implemented to provide **fixed-priority preemptive scheduling** (i.e. priority assigned at creation; the kernel does reassign priorities and only tasks can do so). The Running task is *preempted* when a higher priority task becomes Ready or if a lower priority Ready task has its priority elevated higher than that of the Running task.
>
> FreeRTOS also provides a **cooperative scheduling** scheme where the Running task is *never preempted* and stays in the Running state until it voluntarily surrenders the CPU either by blocking (e.g. `vTaskDelay()`) or by returning itself to the Ready state (in FreeRTOS, this is the `taskYIELD()` call). In this case, another Ready task of the same priority will be scheduled and dispatched Run. If no other task of the same priority is Ready, the YIELDing task will again Run immediately.

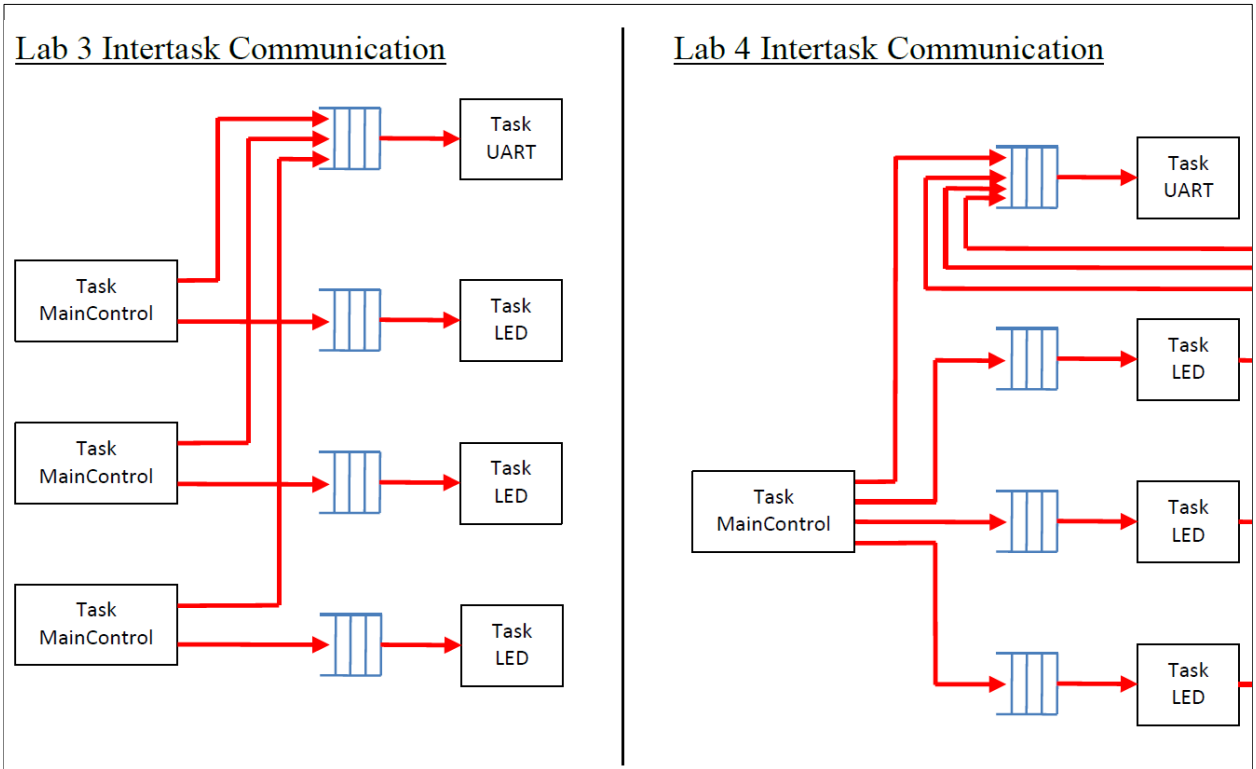

Figure 1: Intertask Communications

The application function of Lab 4 will be a modified version of Lab 3's function. **SW1-SW3** will serve the exact same function as they did in Lab 3. That is, **SW1** will increase the flash rate of the active **LED**. **SW2** will decrease the flash rate of the active **LED**, and **SW3** will rotate the active **LED** for control by **SW1** and **SW2**. The Lab 3 timing parameters apply. In Lab 4, however, there will only be one *MainControl* task and it will select the active **LED** based on **SW3** and send messages to the active **LED's** queue. This is actually a simplification over Lab 3 as *MainControl* tasks no longer `taskSuspend()` and `taskResume()` other *MainControl* tasks. In addition, each **LED** task will now send messages to the **UART** queue under certain specified circumstances.

The application function shall perform per the following considerations. Initially, all tasks should be created with the same priority = 1.

1. *MainControl* Task – Poll button switches and perform the following actions. As with Lab 3, the button switches need to be debounced and a lockout scheme should be implemented. All the hardware considerations from the previous labs apply.

a) **SW1** – Increase the active **LED** flash rate. Lab 3 parameters apply.

b) **SW2** – Decrease the active **LED** flash rate. Lab 3 parameters apply.

c) **SW3** – Rotate the active **LED** between **LED1-LED2-LED3-LED1-…**

d) OTHER MODIFICATIONS AND CONSIDERATIONS – *MainControl* will no longer send a message to the *UART* task announcing which LED is active based on **SW3**. Instead, it will send a message only to announce that it (*MainControl* task) is running. To do this, a `vQueueSendtoBack()` message is sent to the **UART** queue at the top of the `while(1)` loop to announce "MainControl Starting" and right before it does the end-of-loop `vTaskDelay()` with "MainControl Blocking".

2. *LED* Task – The LED task will function in the same way as it did in Lab 3. This means the task will use **`uxQueueMessagesWaiting()`** and **`xQueueReceive()`** to receive timing change messages. The one change here is that the *LED* task will now announce that it is active versus *MainControl* task as in Lab 3. As in *MainControl*, at the top of the **`while(1)`** and right before the **`vTaskDelay()`**, *LED* task will call **`vQueueSendtoBack()`** to announce "LED N Starting" and "LED N Blocking".

3. *UART* Task – The *UART* task will remains identical to Lab 3. **`initUART()`**, **`UARTPutC()`** and **`UARTPutStr()`** remain the same.

4. Queue Create – All of the queues created in Lab 3 are again created in Lab 4. Each queue's purpose, as described in the Intertask Communication Figure 1, remains the same as for Lab 3. However, the queue depth for the UART queue should be increased from 5 to 20.

Lab Observations

1. After you have created the project per the description above, observe the behavior and verify that it performs correctly. The terminal emulator (RealTerm, HyperTerminal, puTTY, etc.) will scroll off messages extremely fast. To look at task interaction as displayed in the terminal emulator, you will have to pause your debug session. Realize that you are running in preemptive scheduling mode.

Lab 4 Actions – 1: Have the instructor checkoff this function.

Observe and make note of the behavior of this first part. From here, you will make certain changes to your project and then debug/execute and observe behavior in comparison to this part. In the change scenarios, make sure to manipulate the switches to exercise the full functionality. You will record your observations and comments in the Blackboard **Lab 4 Actions** "Test".

2. Initially create tasks with the following priority scheme (still in **preemptive** mode):

   a. *UART* task = 1
   b. *MainControl* Task = 2
   c. All *LED* Tasks = 3

Lab 4 Actions – 2: Debug and observe the behavior. Record responses to the following questions.

   i. Does the behavior change from the original Part 1 behavior?

   ii. Describe the changes, if any.

   iii. If the behavior changes, provide an explanation in relation to the scheduling of tasks in this scenario and considering preemptive and cooperative task scheduling, and, prioritization.

3. Initially create tasks with the following priority scheme (still in **preemptive** mode):

   a. *UART* task = 1
   b. *MainControl* Task = 2
   c. *LED*3 Task = 3
   d. *LED*2 Task = 4
   e. *LED*1 Task = 5

Lab 4 Actions – 3: Debug and observe the behavior. Record responses to the following questions.

   i. Does the behavior change from the original Part 1 behavior?

   ii. Describe the changes, if any.

   iii. If the behavior changes, provide an explanation in relation to the scheduling of tasks in this scenario and considering preemptive and cooperative task scheduling, and, prioritization.

4. Up until now, you have been running in preemptive scheduling mode. Now, switch to **cooperative scheduling** mode by changing the `#define configUSE_PREEMPTION` statement value to **0** in the **FreeRTOSConfig.h** file. Then, create the tasks with the following priority scheme:

   a. *UART* task = 1
   b. *MainControl* Task = 1
   c. All *LED* Tasks = 1

Lab 4 Actions – 4: Debug and observe the behavior. Record responses to the following questions.

   i. Does the behavior change from the original Part 1 behavior?

   ii. Describe the changes, if any.

   iii. If the behavior changes, provide an explanation in relation to the scheduling of tasks in this scenario and considering preemptive and cooperative task scheduling, and, prioritization.

5. Initially create tasks with the following priority scheme (still in **cooperative** mode):

   a. *UART* task = 1
   b. *MainControl* Task = 2
   c. All *LED* Tasks = 3

Lab 4 Actions – 5: Debug and observe the behavior. Record responses to the following questions.

   i. Does the behavior change from the original Part 1 behavior?

   ii. Describe the changes, if any.

iii. If the behavior changes, provide an explanation in relation to the scheduling of tasks in this scenario and considering preemptive and cooperative task scheduling, and, prioritization.

6. Initially create tasks with the following priority scheme (still in **cooperative** mode):

a. *UART* task = 1
b. *MainControl* Task = 2
c. *LED*3 Task = 3
d. *LED*2 Task = 4
e. *LED*1 Task = 5

Lab 4 Actions – 6: Debug and observe the behavior. Record responses to the following questions.

i. Does the behavior change from the original Part 1 behavior?

ii. Describe the changes, if any.

iii. If the behavior changes, provide an explanation in relation to the scheduling of tasks in this scenario and considering preemptive and cooperative task scheduling, and, prioritization.

7. Now, switch back to **preemptive** mode (***`#define configUSE_PREEMPTION 1`*** in **FreeRTOSConfig.h**). Initially create tasks with the following priority scheme. Note that this should be the same as previous step. However, create a second "modified" LED task function that uses the original busy wait delay versus the ***`vTaskDelay()`*** delay call. Also, instead of making the flash rate variable, fix the flash rate with the fastest delay (**200 ms**). *[If necessary, review your Lab 1 code where* ***`xTaskGetTickCount()`*** *was used to generate a delay]*. Create the **LED1** and **LED3** tasks using the ***vTaskDelay***-based function and create the **LED2** task using the busy-wait ***`xTaskGetTickCount()`*** based function.

a. *UART* task = 1
b. *MainControl* Task = 2
c. *LED*3 Task = 3
d. *LED*2 Task = 4
e. *LED*1 Task = 5

Lab 4 Actions – 7: Debug and observe the behavior. Record responses to the following questions.

i. Does the behavior change from the original Part 1 behavior?

ii. Describe the changes, if any.

iii. If the behavior changes, provide an explanation in relation to the scheduling of tasks in this scenario and considering preemptive and cooperative task scheduling, and, prioritization.

8. Keeping the changes made in the previous step, just change the scheduling mode back to **cooperative** mode (***#define configUSE_PREEMPTION 0*** in **FreeRTOSConfig.h**).

a. *UART* task = 1
b. *MainControl* Task = 2
c. *LED*3 Task = 3
d. *LED*2 Task = 4
e. *LED*1 Task = 5

Lab 4 Actions – 8: Debug and observe the behavior. Record responses to the following questions.

i. Does the behavior change from the original Part 1 behavior?

ii. Describe the changes, if any.

iii. If the behavior changes, provide an explanation in relation to the scheduling of tasks in this scenario and considering preemptive and cooperative task scheduling, and, prioritization.

9. Now, within the busy wait loop for **LED2**, add the kernel call ***taskYIELD()***. This will cause **LED2** task to enter the Ready state at that point. Remember, you are now in **cooperative** mode.

a. *UART* task = 1
b. *MainControl* Task = 2
c. *LED*3 Task = 3
d. *LED*2 Task = 4
e. *LED*1 Task = 5

Lab 4 Actions – 9: Debug and observe the behavior. Record responses to the following questions.

i. Does the behavior change from the Part 8 behavior above?

ii. Describe the changes, if any.

iii. If the behavior changes, provide an explanation in relation to the scheduling of tasks in this scenario and especially considering the ***taskYIELD()*** function within the **cooperative** mode scheduling scheme.

10. Leaving everything else as is, raise the priority of the *UART* task to the same priority as the *LED* 1 task.

a. *UART* task = 5
b. *MainControl* Task = 2
c. *LED*3 Task = 3
d. *LED*2 Task = 4
e. *LED*1 Task = 5

Lab 4 Actions – 10: Debug and observe the behavior. Record responses to the following questions.

i. Does the behavior change from the Part 9 behavior above?

ii. Describe the changes, if any.

iii. If the behavior changes, provide an explanation in relation to the scheduling of tasks in this scenario and especially considering prioritization within the cooperative mode scheduling scheme.

The FreeRTOS API calls you will use are:

`xTaskCreate()`

`vTaskDelay()`

`xQueueCreate()`

`xQueueSendToBack()`

`uxQueueMessagesWaiting()`

`xQueueReceive()`

`taskYIELD()`