CG2271 Real-Time Operating Systems

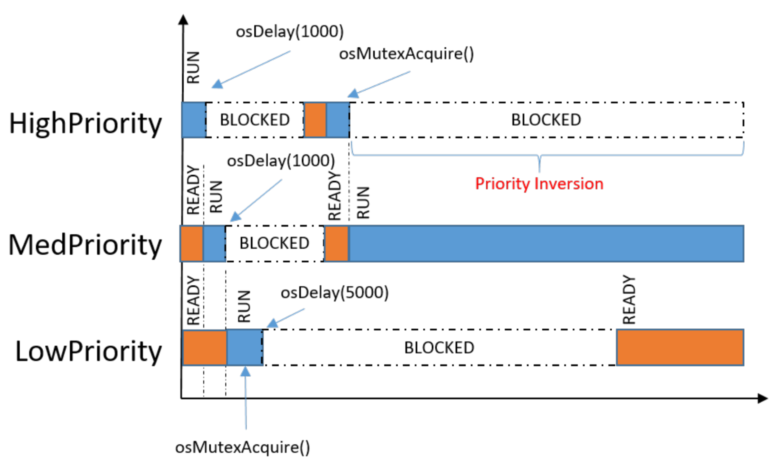
Tutorial 6 Suggested Solutions

Q1. The following code shows THREE threads.

```
#include "cmsis_os2.h"
osMutexId_t mutex_id;

void HighPrioThread(void *argument) {
  osDelay(1000U); // wait 1s until start actual work
  while(1) {
    osMutexAcquire(mutex_id, osWaitForever); // try to acquire mutex
    // do stuff
    osMutexRelease(mutex_id);
  }
}

void MidPrioThread(void *argument) {
  osDelay(1000U); // wait 1s until start actual work
  while(1) {
    // do non blocking stuff
  }
}

void LowPrioThread(void *argument) {
  while(1) {
    osMutexAcquire(mutex_id, osWaitForever);
    osDelay(5000U); // block mutex for 5s
    osMutexRelease(mutex_id);
    osDelay(5000U); // sleep for 5s
  }
}
```

a) Can you describe what happens once the system starts? Which task will run first and at which instances does context switching occur? Draw a TimeLine to describe what happens.

Answer:



b) What is this phenomenon called?

Answer:

This is called a Priority Inversion problem, where a high priority thread is blocked by a low priority thread due to a shared resource.

Q2. In RTX RTOS, we can enable Priority Inheritance when using a Mutex. Explain the concept of Priority Inheritance.

Answer:

Priority Inheritance allows a lower priority thread to be raised to the same level of a higher priority thread, the moment the higher priority thread becomes blocked due to the shared resource already being acquired by a lower priority thread.

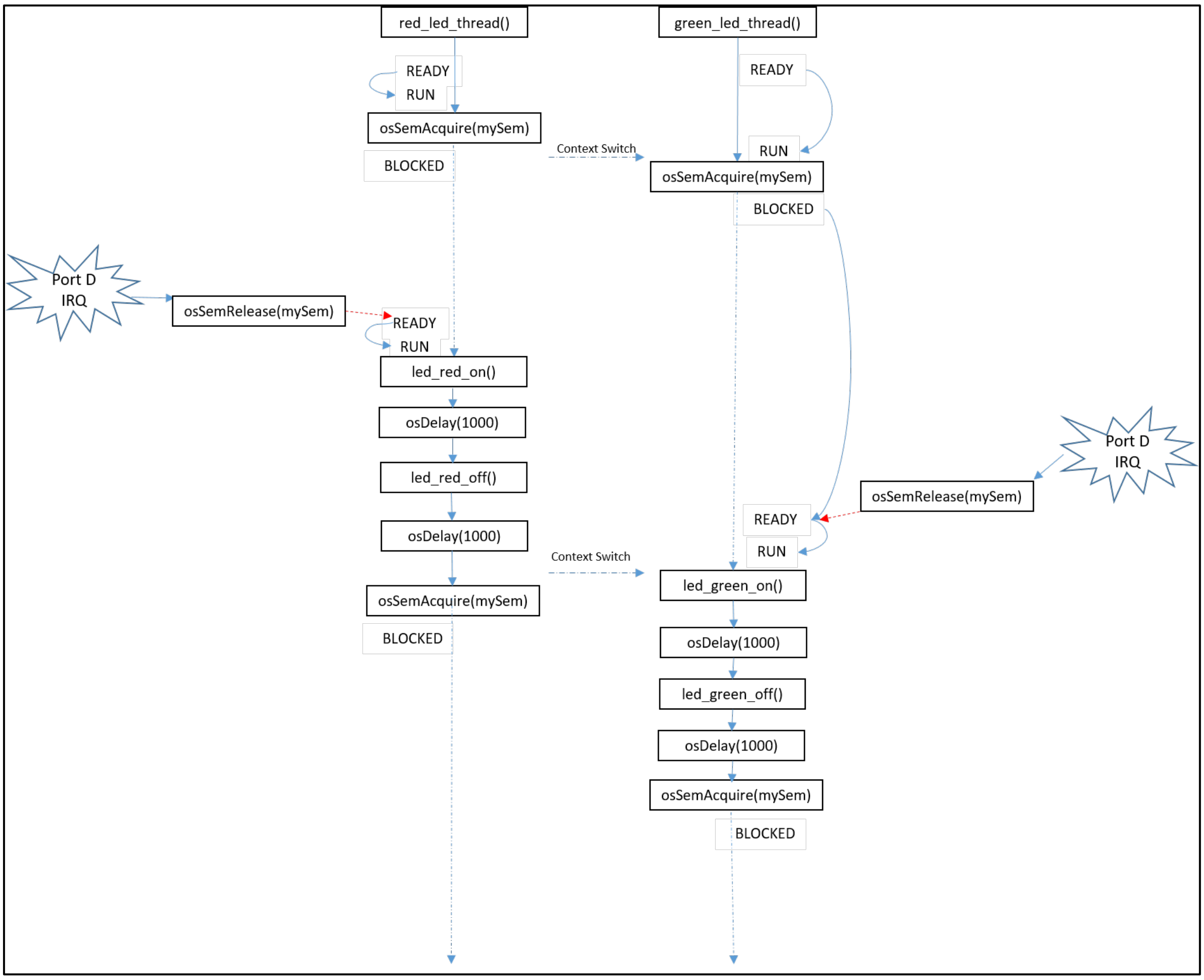
Q3. In the lab, you are going to configure Semaphores with an initial value of 0 and allow the Push Button IRQ handler to release the Semaphore. The threads will wait for the Semaphore before proceeding to control the RGB LED. The code snippets are shown below.

```
int main (void) {

  // System Initialization
  SystemCoreClockUpdate();
  initSwitch();
  InitGPIO();
  offRGB();
  // ...

  osKernelInitialize();                     // Initialize CMSIS-RTOS
  mySem = osSemaphoreNew(1,0,NULL);
  osThreadNew(led_red_thread, NULL, NULL);     // Create application led_red thread
  osThreadNew(led_green_thread, NULL, NULL);  // Create application led_green thread
  osKernelStart();                          // Start thread execution
  for (;;) {}
}
```

```
/*----------------------------------------------------------------------------
 * Application led_red thread
 *----------------------------------------------------------------------------
void led_red_thread (void *argument) {

  // ...
  for (;;) {
    osSemaphoreAcquire(mySem, osWaitForever);

    ledControl(RED_LED, led_on);
    osDelay(1000);
    ledControl(RED_LED, led_off);
    osDelay(1000);
  }
}
/*----------------------------------------------------------------------------
 * Application led_green thread
 *----------------------------------------------------------------------------
void led_green_thread (void *argument) {

  // ...
  for (;;) {
    osSemaphoreAcquire(mySem, osWaitForever);

    ledControl(GREEN_LED, led_on);
    osDelay(1000);
    ledControl(GREEN_LED, led_off);
    osDelay(1000);
  }
}
```

```
void PORTD_IRQHandler()
{
  // Clear Pending IRQ
  NVIC_ClearPendingIRQ(PORTD_IRQn);

  delay(0x80000);
  osSemaphoreRelease(mySem);

  //Clear INT Flag
  PORTD->ISFR |= MASK(SW_POS);
}
```

a) Draw a timing diagram to show how the mySem semaphore is able to control both the threads.



b) If we press the button once, it will light up the RED LED. After it goes OFF, the next press will light up the GREEN LED. The cycle repeats for subsequent presses.

What happens if the button is pressed again while the RED LED is lighted up?

Answer:

In this case, the green_led_thread gets the semaphore and transitions from BLOCKED to READY. This will allow the both the threads to run concurrently. This can happen due to the ROUND ROBIN nature of the OS or when the led_red_thread goes to the BLOCKED state after calling osDelay(). As such, you will see the RED LED turn ON first, then for a brief period of time the GREEN LED will also be ON showing a colour of YELLOW. After that, the RED LED will be OFF and only the GREEN LED will be ON for a while more.

Q4. What if we wanted the behavior such that whenever the button is pressed, the RGB LED lights up as YELLOW for 1s and then goes OFF. You must achieve this by only using the Semaphore related OS constructs.

Answer:

One possible way is to initialize the Semaphore to have a max count of 2. Subsequently, in the PORTD IRQ, we call osSemRelease() twice. This will allow both the threads to come out of the BLOCKED state and go to the READY state concurrently.

```
mySem = osSemaphoreNew(2,0,NULL);
```

```
void PORTD_IRQHandler()
{
  // Clear Pending IRQ
  NVIC_ClearPendingIRQ(PORTD_IRQn);

  delay(0x80000);
  osSemaphoreRelease(mySem);
  osSemaphoreRelease(mySem);

  //Clear INT Flag
  PORTD->ISFR |= MASK(SW_POS);
}
```

Q5. What if we wanted the behavior such that whenever the button is pressed, the RED LED lights up for 1s and goes off. When it goes off, the GREEN LED must light up for 1s and the go off. The sequence would be as such:

RED_ON -> Wait 1s-> RED_OFF & GREEN_ON -> Wait 1s

You must achieve this by only using the Semaphore related OS constructs.

Answer:

One way is to call osSemRelease() in the PortD IRQ handler to first UNBLOCK the red_led_thread. Subsequently, once the RED LED has completed 1s of ON time, it can then call osSemRelease() to UNBLOCK the green_led_thread. The Semaphore in this case is can to a max count of 1.

```
mySem = osSemaphoreNew(1,0,NULL);
```

```
void led_red_thread (void *argument) {

  // ...
  for (;;) {
    osSemaphoreAcquire(mySem, osWaitForever);

    ledControl(RED_LED, led_on);
    osDelay(1000);
    osSemaphoreRelease(mySem);
    ledControl(RED_LED, led_off);
    osDelay(1000);
  }
}
```

THE END