EEL 4745C: Microprocessor Applications 2
Lab ②: G8RTOS Scheduler and Synchronizers
Fall 2022

OBJECTIVES
In this lab, you will implement the basic structures and functionalities necessary for all the major operational components of your G8RTOS. You will implement multiple threads for specific tasks, a thread scheduler, and simple spin-lock semaphores to allow thread synchronization. The main goal is to get a hands-on understanding on how to ensure mutually exclusive access of shared resources in an RTOS.

REQUIRED MATERIALS

Hardware
- Tiva TM4C123GH6PM Launchpad
- LP3943 LED array modules (that are already) integrated to your daughterboard
- Sensors Booster Pack
- Laptop workstation with CCS setup

Software
- TivaWare and Board Support Package (BSP)
- Skeleton code provided for Lab 2

Documentations
- TM4C123GH6PM manual and datasheet
- LP3943 manual and datasheet
- Sensors Booster Pack datasheet
- Lecture 3 contents
- Feel free to use books or browse the internet!

MOTIVATIONS
This laboratory is the heart of this uP2 course: implementing your own RTOS (aka G8RTOS) and its computational components. These concepts are going to help you in many ways while working with multithreaded embedded systems. You will be able to write your threads for specific tasks, without messing up the existing ones. This lab will enable your understanding of how shared resources need to be regulated, what sorts of issues may arise, and how to solve them. The assembly linking and I2C/UART communication exercises you have done on Lab 1 will come in handy as well. You will get to dive deeper into the TIVA TM4C123GH6PM programming and learn how to properly manage sensory integration via multiple threads and semaphores in G8RTOS.

IN-LAB REQUIREMENTS
Lab 2 spans weeks 3-5. The demo is due in the first hour of week 5; then there will be an in-lab quiz. We will not grade you till week 5, but we expect you to complete: Part A by week 3, Part B-C by week 4, and the rest by week 5.

PRE-LAB QUESTIONS
- What are deadlocks in a multithreaded system?
- What are the differences: *i)* thread vs semaphore; *ii)* thread vs process; *iii)* semaphore vs mutex?

Part A: Setting up BSP, drivers, and OS structure

The first thing to do after creating your empty project is to set up the given BSP (Board Support package) and *G8RTOS_Lab2* libraries. Moreover, the LED Drivers you created in your previous lab will need to be integrated in BSP. To do this, follow these steps:
- Add your C and H files to the BoardSupport *src* and *inc* folders in your project.
- In BSP.h add an *#include* to your H file
- In BSP.c add your initialization function into the InitializeBoard function

***Note**: these steps are probably already done in the provided code template for lab 2. If you are not confident about your LED drives, please check the Lab 1 solutions (when out) to update the LedModeSet and PWMColorSet functions. Same applies for the I2C driver C and H files. See the additional functionalities (I2C for LED and sensor) in the BSP>src.*

Create OS structures
File: G8RTOS_Lab2/G8RTOS_Structures.h

The Thread Control Block (TCB) is responsible for holding all relevant information regarding the status of a given thread. For this lab, it should contain the following fields:
- Next TCB pointer
- Previous TCB pointer
- Stack pointer for the thread's stack

Create the TCB structure in the appropriate header file. See comments for supporting details. You will add additional fields in subsequent labs.

Complete the initialization
File: G8RTOS_Lab2/G8RTOS_Scheduler.c

Implement these two functions:
- **InitSysTick:** this initializes the SysTick and SysTick Interrupt, which will be responsible for starting context switching of threads. Hints:
  - Initialize SysTick to overflow every 1ms. You may use SysCtlClockGet() to get the current clock speed in Hz.
  - You can also use the BSP systick functions to do this.

- **G8RTOS_Init:** this will be the first function called in your RTOS project. The function needs to accomplish the following:
  - Initialize system time to zero
  - Set the number of threads to zero
  - Initialize all hardware on the board

Part B: Threads, exception handlers, and schedulers

Implement G8RTOS_AddThread
File: G8RTOS_Lab2/G8RTOS_Scheduler.c

The G8RTOS_AddThread function will take in a void/void function pointer to insert the thread into the scheduler. It should accomplish the following:

- Initialize a TCB for the given thread
- Add TCB to the Round-Robin scheduler list
- Initialize the thread stack to hold a default thread register context

Implement Exception Handlers
File: G8RTOS_Lab2/G8RTOS_Scheduler.c and G8RTOS_Lab2/G8RTOS_SchedulerASM.s

To accomplish multithreading, you need to enable the exception handler to begin context switching.

- The **PendSV_Handler** will be used to save a thread's context, call the scheduler, and load the next scheduled thread's context. You will need to write this handler in assembly (*ie*, in the G8RTOS_SchedulerASM.s file), so that you have direct access to the CPU registers.

- Additionally, the **SysTick_Handler** function in G8RTOS_Scheduler.c will be used to provide a constant quantum for each thread before preemption. For this lab, this handler will simply trigger the PendSV exception. This function is implemented for you (because we will do much more in Lab 3 on this).

Implement G8RTOS_Scheduler
File: G8RTOS_Lab2/G8RTOS_Scheduler.c

The scheduler will be called by the **PendSV_Handler** and will be responsible for choosing the next TCB to run. For this lab's Round-Robin scheduler, the **G8RTOS_Scheduler** function will set the pointer, CurrentlyRunningThread, to the currently running thread's 'next' (nextTCB). Hint: it can be accomplished in just one line!

Part C: Semaphores & peripheral controls

Implement G8RTOS_Launch and G8RTOS_Start
File: G8RTOS_Lab2/G8RTOS_Scheduler.c and G8RTOS_Lab2/G8RTOS_SchedulerASM.s

After successful completion of Part A-B features, your G8RTOS will be ready to launch. To start the OS, you must *arm* the SysTick and PendSV exceptions, set the CurrentlyRunningThread to the first thread in the scheduler, load the context of that thread into the CPU, and enable interrupts.

This task will be split into two functions.

- **G8RTOS_Launch**: This C function is called from main and it should do the following:
  - Set CurrentlyRunningThread
  - Initialize SysTick
  - Set the priorities of PendSV and SysTick to the lowest priority
  - Call G8RTOS_Start
  - Hint: There are many useful BSP functions (in interrupt.c, systick.c, etc.) that you can/should utilize to your advantage. *Don't reinvent the wheel!*

- **G8RTOS_Start:** This assembly function should do the following:
  - Load the currently running thread's context into the CPU
  - Enable interrupts

Testing the Scheduler
File: main.c
At this point you will write three simple threads to make sure your scheduler works before adding semaphores or peripheral controls. One simple way to do this:

- Create three functions called task0, task1, and task2 in your main.c
- Each task function will increment their own counter variable. Name them accordingly: task0 has counter0, task1 has counter1, etc.)
- Call G8RTOS_Init, add the threads to your scheduler, and launch the OS
- *Note: this step is only for testing, not for demo or any other purposes.*

Implement Semaphore Functions
File: G8RTOS_Semaphores.c

You will now add the ability to synchronize threads and control peripheral access through a naive spinlock semaphore implementation.

- **G8RTOS_InitSemaphore:** This function will assign the semaphore pointer parameter the value of the value parameter. *Note: This should all be accomplished in a critical section!*

- **G8RTOS_WaitSemaphore:** This function will check if the given semaphore parameter is greater than 0. If it is not, it will constantly check until it is. During this *"spinlock"*, the function will exit and reenter a critical section to allow other threads to run. After the spinlock, the semaphore will be decremented to indicate ownership of the semaphore.

- **G8RTOS_SignalSemaphore:** This function will increment the semaphore to indicate releasing ownership of the semaphore. *Note: This is also a critical section!*

Part D: Finally, Add Threads for Sensory Interfacing

You will need to implement three threads for three different tasks; specifically, we want to interface the accelerometer, light sensor, and gyro of the sensor booster pack. Three threads will communicate with these on-board sensors to accomplish the following.

Thread 0

- Wait for the sensor I2C semaphore.
- Read from the accelerometer's x-axis and save the value into a local variable.
- Release the sensor I2C semaphore.
- Wait for the LED I2C semaphore.
- Output data to **Red LEDS** (see Figure A).
- Release the LED I2C semaphore.

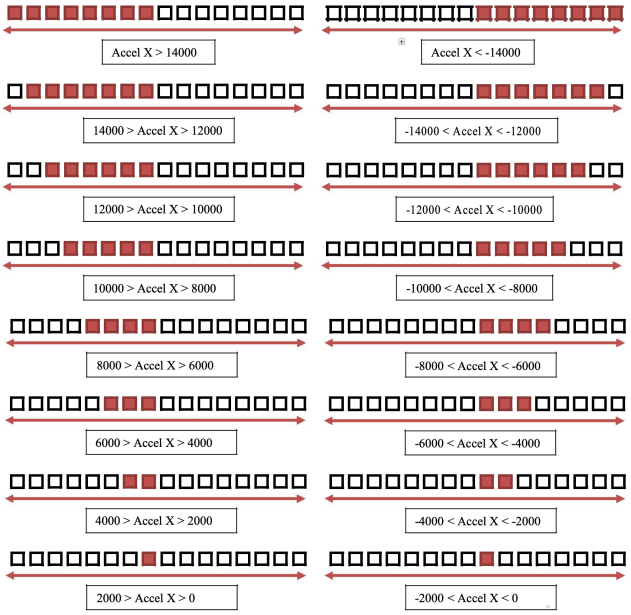
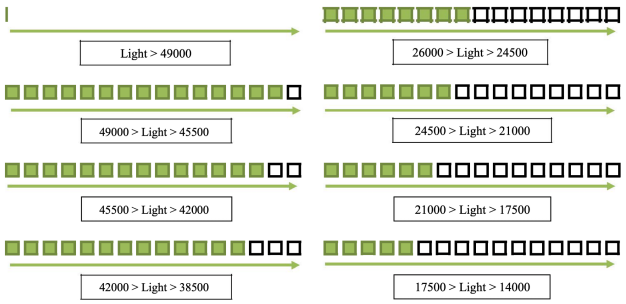


Figure A: Sample output for the Thread0 based on the accelerometer's x-axis values; remember, specific values do not matter much, but the red LED patterns are changing proportionately with *Accel X* changes. (best at 200% zoom)

Thread 1

- Wait for the sensor I2C semaphore.
- Read from the light sensor and save the value into a local variable.
- Release the sensor I2C semaphore.
- Wait for the LED I2C semaphore.
- Output data to **Green LEDS** (see Figure B).
- Release the LED I2C semaphore.



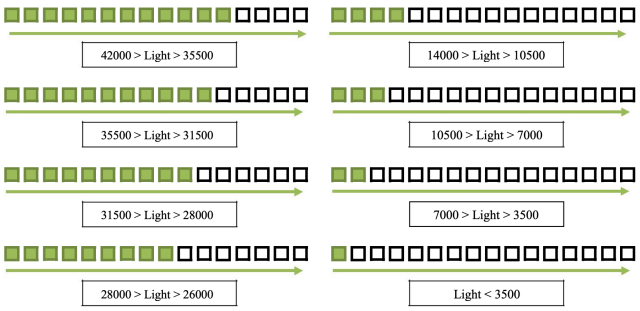


Figure B: Sample output for the Thread1 based on the light sensor values; again the specifics does not matter; you can cover the light sensor with hand, turn-off lights, or use cell-phone flashlights on the sensors - to check whether your green LEDs change accordingly. (best at 200% zoom)

Thread 2

- Wait for the sensor I2C semaphore.
- Read from the gyro's z-axis and save the value into a local variable.
- Release the sensor I2C semaphore.
- Wait for the LED I2C semaphore.
- Output data to **Blue LEDS** (see Figure C).
- Release the LED I2C semaphore.

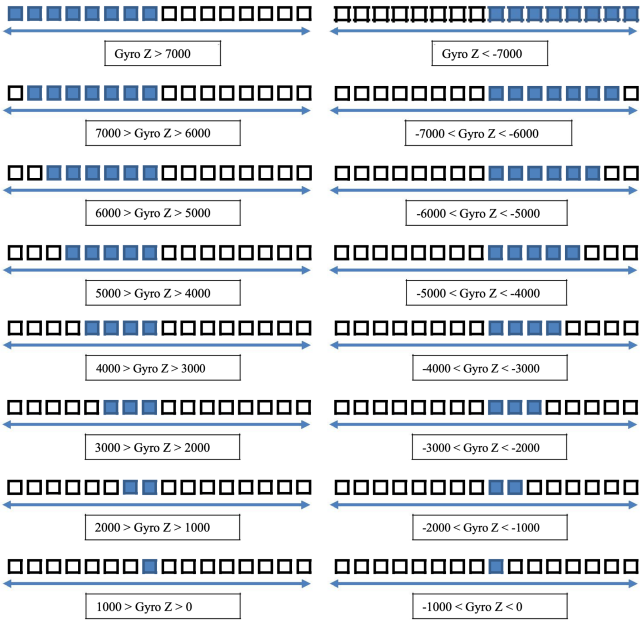


Figure C: Sample output for the Thread2 based on the gyro's z-axis values; same instructions apply. Lift your board and change its *gyro* for code validation! (best at 200% zoom)

Implementation Instructions:

- Threads.c (in the root directory) should hold all threads and semaphores.

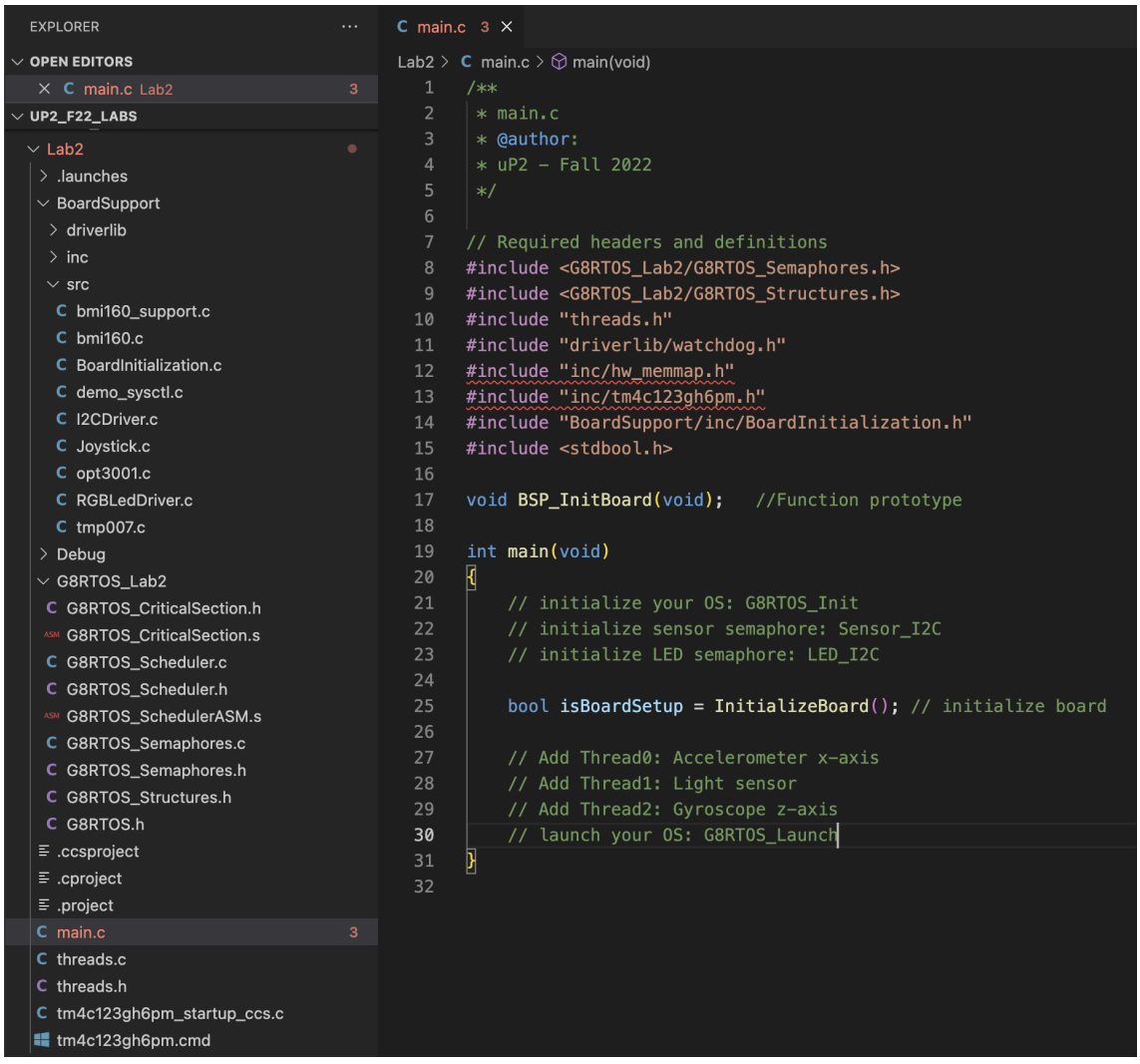
- Threads.h should reference all semaphores to be initialized and the threads for main to see. It will also hold extern declarations of all the threads as well as extern declarations of the sensor and LED semaphores.

Part E: Putting it all together!

Alrighty, if you made it thus far - it is time to put everything together and check your G8RTOS end-to-end. Complete the *main* function, *ie*, update your main.c to do the following:

- Call G8RTOS_Init to initialize the OS
- Initialize the semaphores (LED I2C and sensor I2C)
- Add the threads in Threads.h and complete the functionalities in Thread.c
- Integrate the schedulers and you are good to o
- Finally, launch the OS and test each thread

Checkout the Lab 2 template code. You should open your own project and integrate each module there, it is not the best idea to start from the given template (some machine-level issues may occur). Your project skeleton should look like the following:



Get started with this as early as possible. It is an assignment that demands time - so be patient and be thorough. All the very best!