EEL 4745C: Microprocessor Applications 2
Lab ③: G8RTOS Periodic Threads and Queueing
Fall 2022

OBJECTIVES
In the previous lab (#2), you implemented the basic structures and operational components of G8RTOS. There were many inefficient functionalities, which we will address in this lab. Specifically, you will:

- Improve semaphores using the blocking and yielding features;
- Add sleeping feature to background threads to free up CPU time as opposed to a *delay;*
- Integrate periodic threads in conjunction with multiple background threads; and
- Implement IPC: Inter-Process Communication using FIFOs.

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 3

Documentations
- TM4C123GH6PM manual and datasheet
- LP3943 manual and datasheet
- Sensors Booster Pack datasheet
- Some of the Lecture 5 contents
- Feel free to use books or browse the internet!

IN-LAB REQUIREMENTS
Lab 3 spans weeks 6-7. The demo is due in week 7; we will not grade you till then, but we expect you to complete Part A-B by week 6 and Part B-C by week 7.

PRE-LAB QUESTIONS
- What is a jitter? Can you calculate the jitter for periodic thread 0, and background thread 1 and 2 (when you are done implementing)?
- What are the differences between: a periodic vs an aperiodic vs a sporadic process?

Part A.1: Improved Semaphores, Blocking, & Yielding

In Lab-2, you implemented a spinlock semaphore check to synchronize threads and provide exclusive access to peripherals. However, this approach wastes CPU time by continuously checking a flag, where we could be running another thread in the meantime until the semaphore becomes available. Now, you will update the semaphore library components to use ***blocking*** to improve CPU utilization.

You can follow the these steps for this part:
- Begin by modifying the Thread Control Block (TCB) structure to include a pointer to a blocked semaphore. This semaphore will either contain a 0 (not blocked), or the semaphore that the thread is currently waiting on. The definitions are already created for you (see G8RTOS_Lab3/G8RTOS_Structures.h).

- Next, modify the semaphore library so that we are no longer *polling* the semaphore in the semaphore wait (G8RTOS_WaitSemaphore) function. If the semaphore is not available, the blocked semaphore for that thread should be initialized; then ***yield*** control to allow another thread to run in the meantime.

- For the G8RTOS_SignalSemaphore function, you will add a process that will go through the linked list of TCBs and unblock the first thread that is blocked on that semaphore. *Note: that this process should only be executed if the semaphore value is less than or equal to zero, signifying that a thread has been waiting on that semaphore to be released.*

The last thing you need to modify is the G8RTOS scheduler. It must verify that the next TCB is not currently blocked. If it is blocked, keep going through the linked list until a thread that is not blocked is found. *Note: It is important, as the programmer, to ensure that a deadlock does not occur, in which case thread A is waiting on a semaphore to be released by thread B, but thread B is waiting on a semaphore to be released by thread A, and neither thread will be able to continue running.*

Part A.2: Sleeping

In microprocessors, you may have used empty loops to serve as a simple delay or for some other purpose. Alternatively, a better solution is to use a timer to perform such a task to increase the accuracy of the delay. However, in a multithreaded system, CPU time can't be evenly distributed by using this method.

To solve this problem, you will incorporate the ***sleeping*** feature; that is, when a thread needs to wait for a prescribed amount of time, other threads should be able to run in the meantime. It is important to note that sleeping is an appropriate solution when the accuracy of time is not important, but CPU usage is. When CPU usage and timing accuracy are important, periodic threads are more appropriate.

Moreover, the thread status must have a way to keep track of *sleep duration* and its *sleep status*. Since the `SysTick` runs at a rate of 1ms, this variable should be in terms of milliseconds so that it can be easily handled within the `SysTick` handler. *Note: that this means a thread can sleep for a minimum of 1ms, with increments of 1ms. Check and use the sleep function implemented for you in the scheduler.*

Lastly, you need to make sure that sleeping threads will be checked in the scheduler just like blocked threads. That is, instead of just running the next thread (as in Lab-2), we will run the next thread in the linked list that is neither *sleeping* nor *blocked*.

Part B.1: Periodic Event Threads

As mentioned earlier, ***periodic threads*** are useful when CPU usage and timing accuracy is of great importance to the user. A periodic event will consist of a doubly linked list with the following parameters:

- Function pointer to periodic event handler
- Period and execution time
- Pointer to the previous periodic event
- Pointer to the next periodic event

See the definition of ptcb_t (Periodic Thread Control Block) in the G8RTOS_Structure.

The maximum number of periodic events should be defined by the OS (allow up to 6 periodic events); however, it is the user's job to add a periodic event to the linked list. Therefore, much like adding a regular thread, you will have a function that will initialize a new periodic event as well as handle the doubly linked list. In the scheduler, you will need to implement a function named G8RTOS_AddPeriodicEvent, which should be operationally very similar to the G8RTOS_AddThread function you implemented in Lab-2.

Moreover, within the scheduler, you will check every periodic event's execution time and run the thread after the amount of prescribed time has passed. *Note: if two or more threads have a period with common multiples of each other, one way to avoid running multiple events within the same `SysTick` interrupt is to give each event a different initial value for the execution time to stagger their run times.*

Part B.2: FIFOs

A FIFO (First In, First Out) data structure can be used as a *buffer* for asynchronous communication between threads. Notice the new library in the G8RTOS folder (named IPC.c and IPC.h); with three functions for:

- Initializing the buffer (G8RTOS_InitFIFO),
- Reading from buffer (ReadFIFO), and
- Writing to the bugger (WriteFIFO).

Note: you need to implement a circular buffer, so you must wrap the head and tail pointers (if necessary) reading from or writing to the FIFO buffer.

G8RTOS_InitFIFO: When initializing a FIFO, the function should take in a `uint32_t`, which is the index of the array of FIFOs provided by the G8RTOS. We set the max number of FIFOS as 4 and the max buffer size to 16. The FIFO structure should contain the following (see their definitions in G8RTOS_IPC.c.):

- Buffer: `int32_t` array
- Head: `int32_t` pointer
- Tail: `int32_t` Pointer
- Lost Data count: `uint32_t`
- Current size: semaphore
- Mutex*:* semaphore

In addition to checking the boundary conditions, you should also Initialize the buffer pointers as well as the semaphores. Also make sure the current size starts at 0 and mutex starts at 1

ReadFIFO: The read function will take in an integer value that will determine which FIFO is to read from. Before reading from the FIFO, we must wait for the *mutex* semaphore in case the FIFO was in the middle of being read from another thread, and then wait for the *current size* semaphore to make sure there is data to be read. Because the *current size* and *mutex* are semaphores, a thread can become blocked waiting for the FIFO to obtain data. Once we have read from the FIFO and updated the head pointer, we can signal the mutex semaphore and return the data.

WriteFIFO: The write function will also take in an integer that chooses which FIFO will be written to, as well the actual data to be written. The *current size* semaphore must be compared to the size of the FIFO minus one, in case an interrupt has happened between reading the FIFO and incrementing its head pointer. Should this condition hold true, we should increment the number of lost data and return an error that the FIFO is full. Otherwise, write the data to the FIFO, update the tail pointer, signal the current size semaphore, and return that no error has occurred.

Part C: Implement Threads

BackGroundThread0:

- Empty default thread; does nothing.

BackGroundThread1:

- Read the BME280's temperature sensor.
- Sends data to temperature FIFO.
- Sleep for 500ms.

Periodic Thread 0 (Period: 100ms):

- Read X-coordinate from the joystick.
- Write data to Joystick FIFO.

Periodic Thread 1 (Period: 100ms):

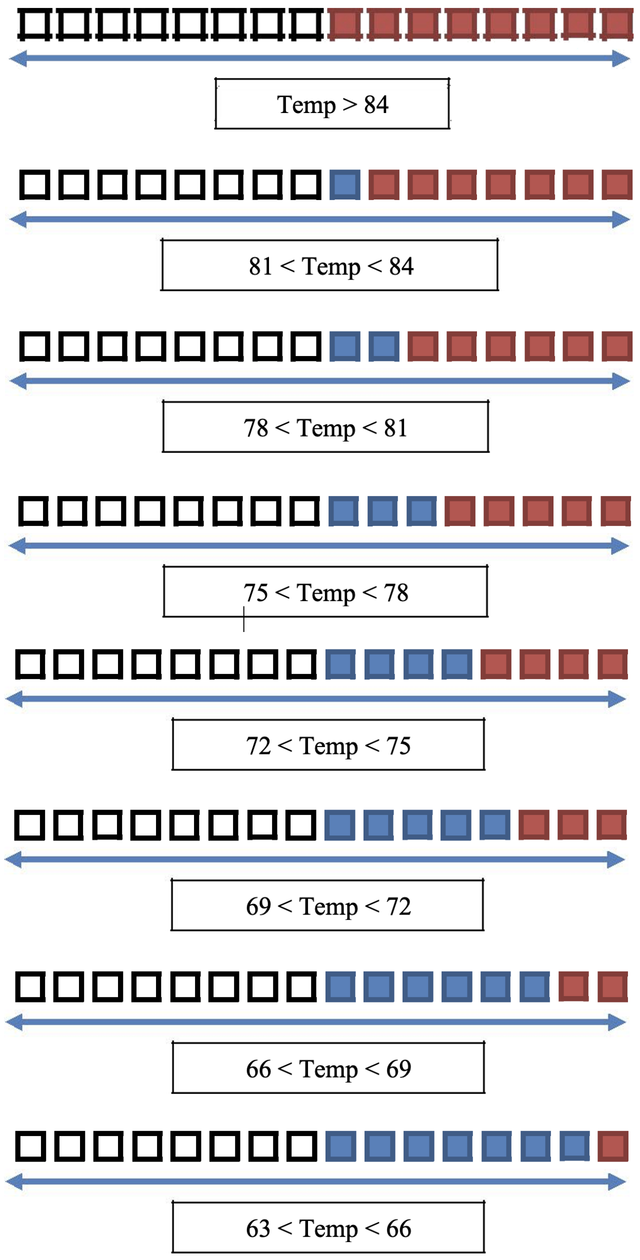
- Prints out the decayed average value of the joystick's X-coordinate in a UART console.
- Prints out the temperature value in a UART console (in degrees Fahrenheit).

BackGroundThread2:

- Read the light sensor.
- Send data to light FIFO.
- Sleep for 200ms.

BackGroundThread3:

- Read temperature FIFO.
- Output data to Red/Blue LEDs as shown in the following figure. Feel free to adjust or normalize the temperature values if needed.

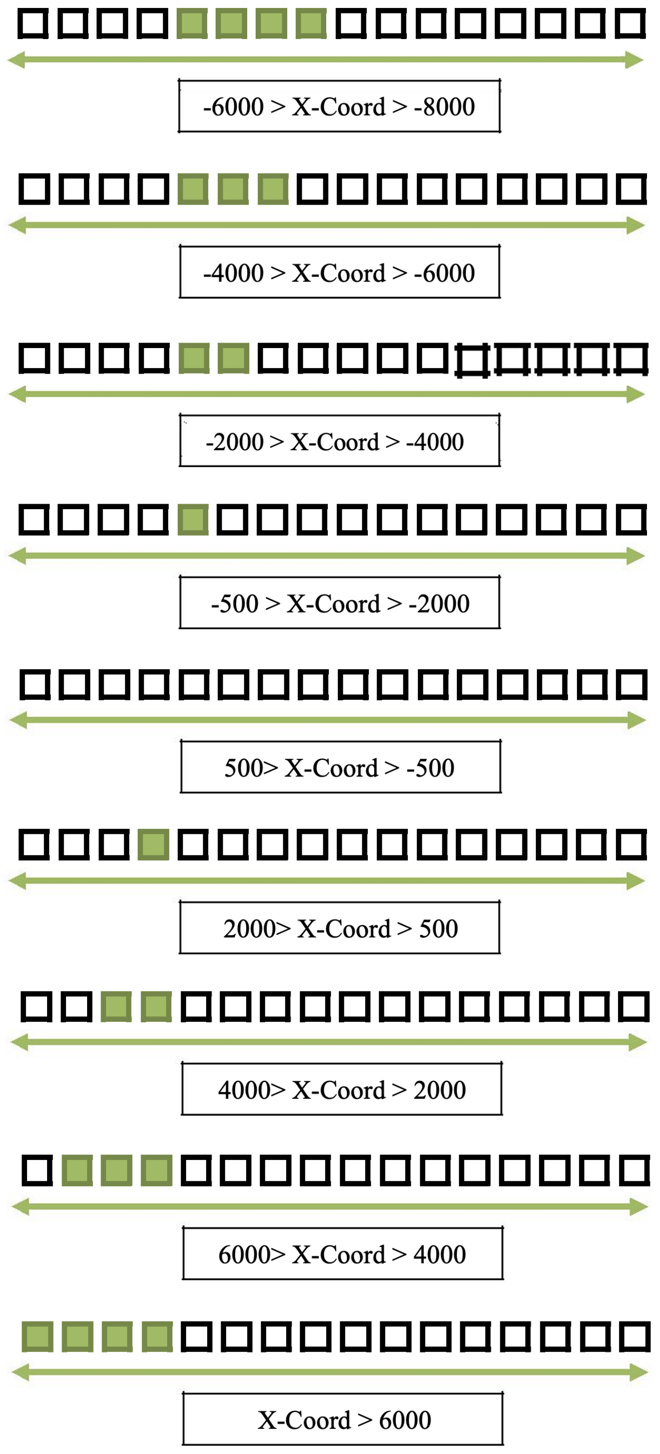


BackGroundThread4:

- Read the joystick's X-coordinate.
- Calculate decayed average: to calculate a 50% decaying average, you will have an `int32_t` variable (*eg*, named `Avg`). After getting a new value, Avg will be updated as

```
Avg = (Avg + value) >> 1.
```

- Output data to Green LEDs as shown in the following figure.


-6000 > X-Coord > -8000
-4000 > X-Coord > -6000
-2000 > X-Coord > -4000
-500 > X-Coord > -2000
500> X-Coord > -500
2000> X-Coord > 500
4000> X-Coord > 2000
6000> X-Coord > 4000
X-Coord > 6000