Lab 3: Periodic Threads, Blocking, Sleeping, and FIFOs

OBJECTIVES

- Improve semaphores using blocking and yielding
- Add sleeping to background threads to free up CPU time as opposed to a delay
- Integrate periodic threads in conjunction with multiple background threads
- Implement inter-process communication using FIFOs

REQUIRED MATERIALS

Hardware

- MSP432 Launchpad
- Sensors Booster Pack
- LED Array Module

Software

- BSP
- Lab 2 G8RTOS

PART A: Improved Semaphores, Blocking, and Yielding

In lab 2, we implemented a simple spinlock semaphore check to synchronize threads and provide exclusive access to peripherals. However, this approach wastes CPU time by checking a flag, where we could otherwise be running another thread in the meantime until the semaphore becomes available. In this section of the lab, you will update your semaphore library component to use blocking to improve CPU utilization.

Begin by modifying the thread control block struct to include a pointer to a blocked semaphore. The semaphore will either contain a 0 (not blocked), or the semaphore that the thread is currently waiting on.

Next, modify the semaphore library component so that we are no longer polling the semaphore in the "semaphore wait" 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.

For the "Signal Semaphore" 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 will only execute 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 OS's scheduler. It must verify that the next TCB is not currently blocked. If it is, 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 B: 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 increasing the accuracy of the delay. However, in a multithreaded system, CPU time can't be evenly distributed using this method. To solve this problem, you will incorporate "sleeping" when a thread needs to wait for a prescribed amount of time before continuing its process, while other threads may 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.

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.

When a thread wants to sleep, it will simply call an OS_Sleep function that initializes its sleep count, puts the thread to sleep, and yields control to allow other threads to run.

Lastly, sleeping threads will be checked in the scheduler just like blocking (run the next thread in the linked list that is neither asleep *nor* blocked.

PART C: Periodic Event Threads

As previously stated, periodic threads are appropriate when CPU usage and timing accuracy is of great importance to the user.

You will add a new data structure that will define the parameters of a periodic event. A periodic event will consist of a doubly linked list with the following parameters:

- Function pointer to periodic event handler
- Period
- Execute Time
- Pointer to the previous periodic event
- Pointer to the next periodic event

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. You should return an error should you exceed the maximum number of periodic threads defined by the OS.

Within the scheduler, you will check every periodic event's execute time and run the thread after the amount of prescribed time has passed. Important 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 execute time to stagger their run times.

PART D: FIFOs

A FIFO (First In, First Out) data structure can be used for safe asynchronous communication between threads. **In this part of the lab, you will create a new**

Lab 3: Periodic Threads, Blocking, Sleeping, and FIFOs

G8RTOS library component for inter-process communication (IPC) that will be responsible for:

- Initializing a FIFO
- Reading from the FIFO
- Writing to the FIFO

(Note: the FIFOs are circular buffers, so we must wrap the head and tail pointers if necessary when we read/write to a FIFO).

When initializing a FIFO, the function should take in a uint32_t variable, which is the index of the array of FIFOs provided by the OS. The OS should allow the user to use up to 4 FIFOs (max number of FIFOS). Should the user try to initialize more than 4 FIFOs, the initializing function should return an error. You also need to determine a max FIFO size for the buffer. You can use 16 for now, but can be changed later if desired. The function should initialize the FIFO struct, which will contain:

- Buffer array (int32_t)
- Head pointer
- Tail Pointer
- Lost Data count
- Current Size semaphore
- Mutex semaphore

The read function will take in an integer value that will determine which FIFO is to be read from. Before reading from the FIFO, we must first 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 reading to be read. Because the current size and mutex are semaphores, a thread can become blocked waiting for the FIFO to obtain data that can be read. Once we have read from the FIFO and updated the head pointer, we can signal the mutex semaphore and return the data.

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.

NOTE: Use BITBAND_PERI(Px->OUT, n) to avoid affected other bits when toggling a pin. This eliminates the need for a semaphore!

PART E: Using Your New and Improved OS

Periodic Thread 0 (Period: 100ms):

a. Read X-coordinate from joystick
b. Write data to Joystick FIFO
c. Toggle an available GPIO pin (don't forget to initialize it in your main)

Background Thread 0:

a. Read the BME280's temperature sensor
b. Send data to temperature FIFO
c. Toggle an available GPIO pin (don't forget to initialize it in your main)
d. Sleep for 500ms

Background Thread 1:

a. Read light sensor
b. Send data to light FIFO
c. Toggle an available GPIO pin (don't forget to initialize it in your main)
d. Sleep for 200ms

Background Thread 2:

a. Read light FIFO
b. Calculate RMS value (See appendix for details)
c. You should write a static function to calculate the square root of a value using Newton's method
d. If RMS < 5000, set global variable to true, otherwise keep it false

Background Thread 3:

a. Read temperature FIFO
b. Output data to LEDs as shown in Figure B

Background Thread 4

a. Read Joystick FIFO
b. Calculate decayed average for X-Coordinate (See appendix for details)
c. Output data to LEDs as shown in Figure A

Periodic Thread 1 (Period: 1s)

a. If global variable for light sensor is true, do b & c; otherwise, do nothing
b. Print out the temperature (in degrees Fahrenheit) via UART
c. Print out decayed average value of the Joystick's X-coordinate via UART

Background Thread 5

a. Idle thread "while(1)"

All the required drivers for this lab are included in the BSP, along with detailed documentation for usage.

PRE-LAB REQUIREMENTS

1. Calculate the jitter for periodic thread 0, and background thread 0 & 1.

NOTE: To toggle an LED in a thread, you must use bit banding to access the individual bits to avoid a read-modify-write instruction. Another alternative is to use a semaphore for the port, however it is not as fast.

Lab 3: Periodic Threads, Blocking, Sleeping, and FIFOs

APPENDIX

Newton's Method For calculating RMS Value

Since we are not using the FPU for the OS yet, you will use Newton's method for calculating the RMS value, which uses fixed point arithmetic.

The RMS value of X is defined as:

$$x_{RMS} = \sqrt{\frac{1}{N}(x_1^2 + x_2^2 + \cdots + x_N^2)}$$

To calculate the square root of *n* using Newton's method, we use the iterative formula

$$x_{k+1} = \frac{1}{2}\left(x_k + \frac{n}{x_k}\right), \quad k \geq 0, \; x_0 = n$$

where x_{k+1} is equal to the integer value of $\sqrt{n}$, once $|x_{k+1} - x_k| < 1$.

Calculating Decaying Average (50% Newest)

To calculate a 50% decaying average, you will have a int32_t Avg variable. After getting a new value, Avg will be updated, such that

Avg = (Avg + value) >> 1.

EEL 4930 – Fall 2017

Revision **0**

Lab 3: Periodic Threads, Blocking, Sleeping, and FIFOs

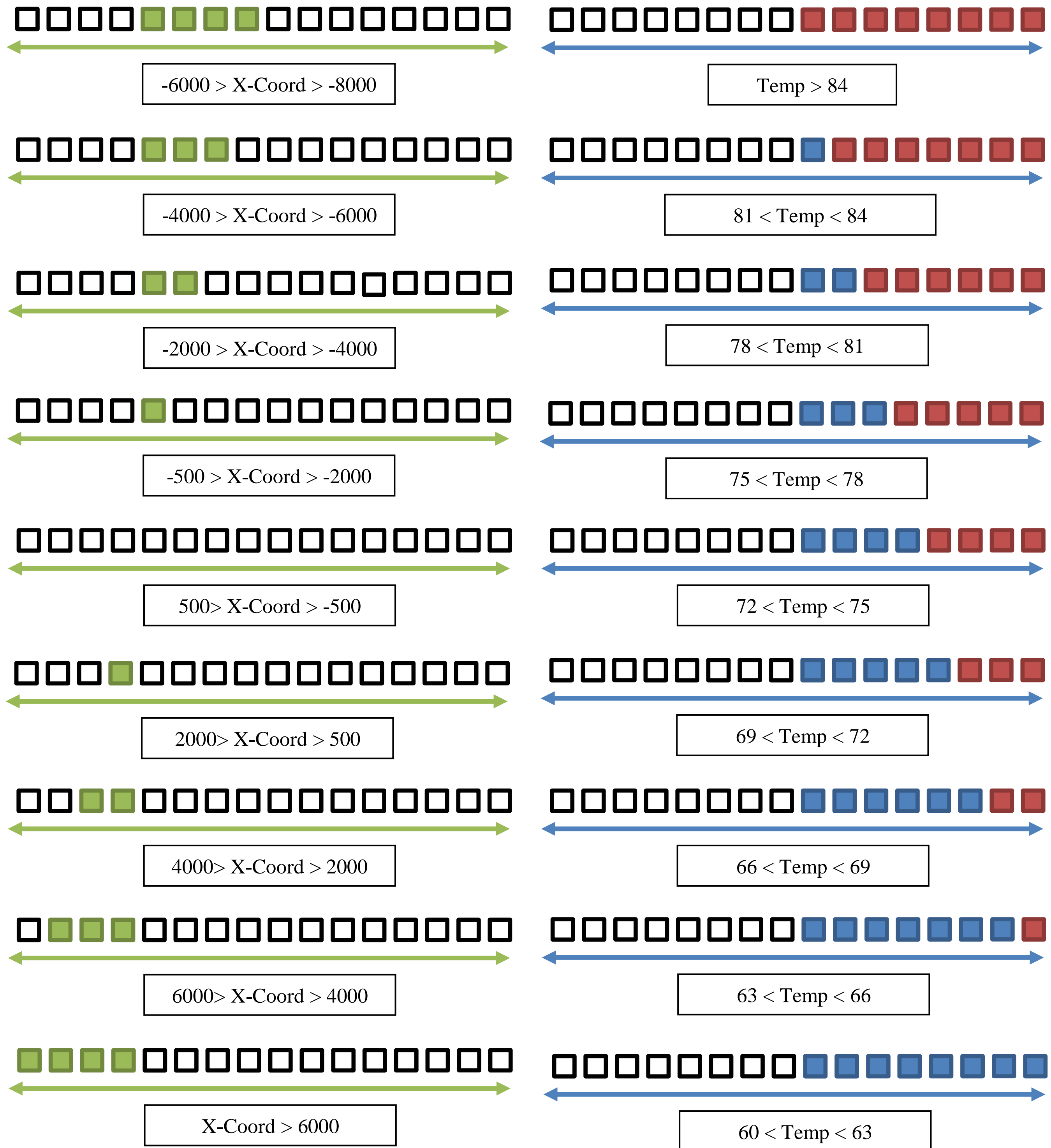

Figure A.

Figure B.