Improving Your G8RTOS

EEL 4745C: Microprocessor Applications II

Fall 2022

Md Jahidul Islam

Lecture 5

ECE Department of Electrical & Computer Engineering
UF UNIVERSITY of FLORIDA

Overview

- Discussions on: improving your G8RTOS implementation in Lab-2

- In Lab-3, you will do the following:
  - 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

- You have to demonstrate:
  - Periodic threads are working alongside background threads
  - Consistent IPC is happening based on FIFO principles
  - Joystick, temperature sensor, light sensors values are manipulated correctly
    - See the lab-3 manual for details

Improved Semaphore

Blocked Semaphore

- Previously, we implemented a simple spinlock semaphore; the continuous spin-locks wasted CPU memory
- We will improve this by adding a blocked flag in TCB structure. If the blocked flag was set, the blocked thread will yield the CPU control to next thread during the SysTick handler.

```
typedef struct tcb_t {
    int32_t *stackPointer;
    struct tcb_t *nextTCB;
    struct tcb_t *previousTCB;
    semaphore_t *blocked;
    uint32_t sleepCount;
    bool asleep;
} tcb_t;
```

Improved Semaphore: Wait

```
wait (S) {
    while S <= 0
        ; // no-op
    S--;
}
```

```
/*
 * No longer waits for semaphore
 *  - Decrements semaphore
 *  - Blocks thread if semaphore is unavailable
 * Param "s": Pointer to semaphore to wait on
 * THIS IS A CRITICAL SECTION
 */
void G8RTOS_WaitSemaphore(semaphore_t *s)
{
    // your code
}
```

Improved "Semaphore Wait"

- If the semaphore is not available
  - The blocked semaphore of running thread should be initialized
  - Then yield control to the next available thread

Improved Semaphore: Signal

```
signal (S) {
    S++;
}
```

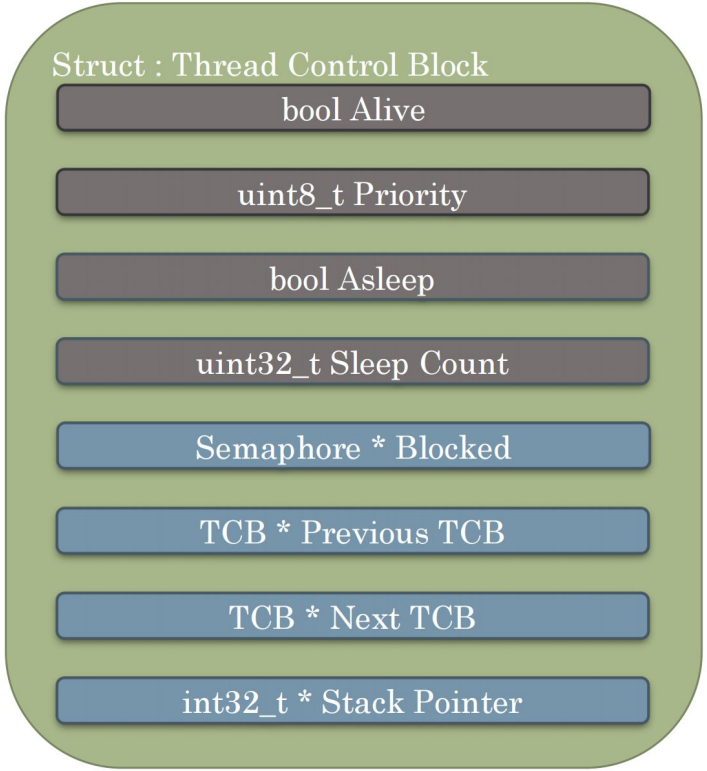
```
37    // Increment semaphore
38    (*s)++;
39
40    if((*s) <= 0)
41    {
42        tcb_t *pt = CurrentlyRunningThread->nextTCB;
43        while(pt->blocked != s)
44        {
45            pt = pt->nextTCB;
46        }
47
48        pt->blocked = 0;
49    }
```

Improved "Signal Semaphore"

- If the semaphore is not available
  - Go through the TCB list and unblock the first thread blocked on the same semaphore.\
  - Move that unlocked thread to the next thread to be executed

Improved TCB

```
typedef struct tcb_t {
    int32_t *stackPointer;
    struct tcb_t *nextTCB;
    struct tcb_t *previousTCB;
    semaphore_t *blocked;
    uint32_t sleepCount;
    bool asleep;
} tcb_t;
```



Sleeping

- **Active State:** Thread is ready to run but waiting for its turn
- **You Sleep State:** Thread is waiting for a fixed amount of time before it enters the active state again
- **Blocked State:** Thread is waiting on some external or temporal event
- Blocking and sleeping help to free up the processor to perform other tasks as opposed to just "spinning" (wasting its entire time slice checking if the event condition is met)

```
/*
 * Puts the current thread into a sleep state.
 *  param durationMS: Duration of sleep time in ms
 */
void sleep(uint32_t durationMS)
{
    CurrentlyRunningThread->sleepCount = durationMS + SystemTime;
    CurrentlyRunningThread->asleep = 1;
    HWREG(NVIC_INT_CTRL) |= NVIC_INT_CTRL_PEND_SV;
}
```

Using the Sleep Function

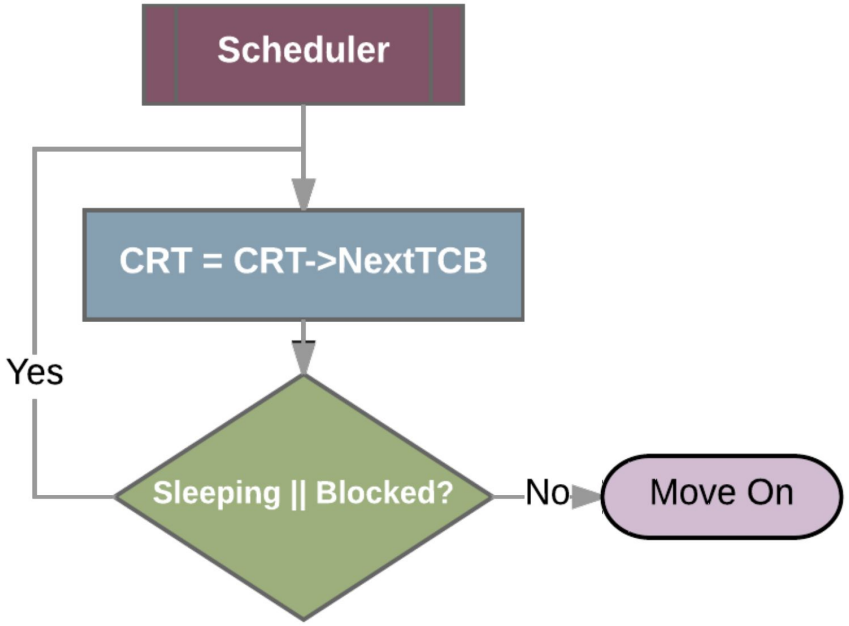
- In the SysTick handler, check every sleeping thread's sleep count
- If the thread's sleep count is equal to the current SystemTime
  - Then that thread is to be wake up
  - Otherwise, it remains sleeping

```
/*
 * Puts the current thread into a sleep state.
 *  param durationMS: Duration of sleep time in ms
 */
void sleep(uint32_t durationMS)
{
    CurrentlyRunningThread->sleepCount = durationMS + SystemTime;
    CurrentlyRunningThread->asleep = 1;
    HWREG(NVIC_INT_CTRL) |= NVIC_INT_CTRL_PEND_SV;
}
```

Improved Scheduler

Note: *It is possible that all threads can be either sleeping or blocked, in which case we enter an infinite loop here.*

How do we avoid this in Lab-3?



Improved SysTick Handler

```
void SysTick_Handler()
{
    SystemTime++;
    tcb_t *ptr = CurrentlyRunningThread;
    ptcb_t *Pptr = &Pthread[0];
```

Increments system time
Gets the current threads (periodic and background)

Loop through the periodic threads, and execute them appropriately (if their time is now!)

Loop through the background threads: check sleeping threads and wake them up appropriately (if their time is now!)

```
    // now lets do the context switch
    HWREG(NVIC_INT_CTRL) |= NVIC_INT_CTRL_PEND_SV;
}
```

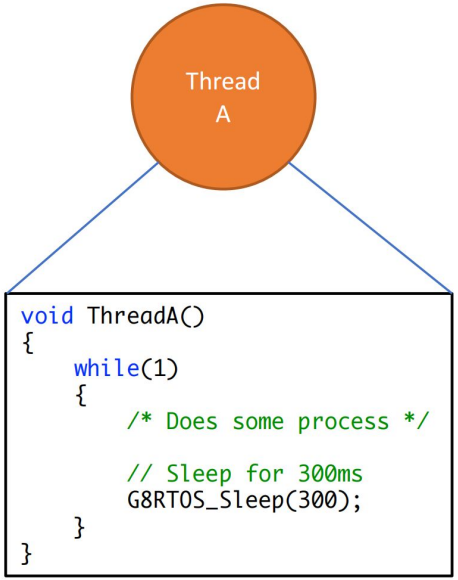
Context Switch

Alternate Sleeping Implementation

- *Another way to implement sleeping is to*
  - ***Remove** the new sleeping thread from the linked list of active threads, and*
  - ***Insert** it to a doubly linked list of sleeping threads*
- *This list of sleeping threads will be **sorted** from smallest to highest sleep count*
- *Once the thread with the lowest sleep count equals the system time, that thread is woken up*
- ***Advantage:***
  - *Now we only have to check one sleeping thread's sleep count within the SysTick handler as opposed to every initialized thread*

Lab-3 Bonus point: +1

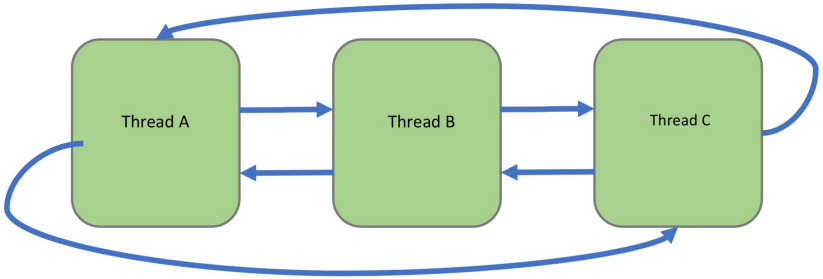
Example: Alternate Sleep Function



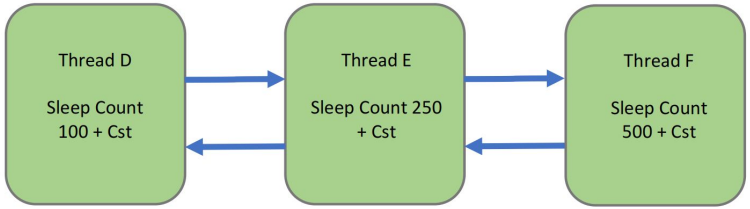

```
void ThreadA()
{
    while(1)
    {
        /* Does some process */

        // Sleep for 300ms
        G8RTOS_Sleep(300);
    }
}
```

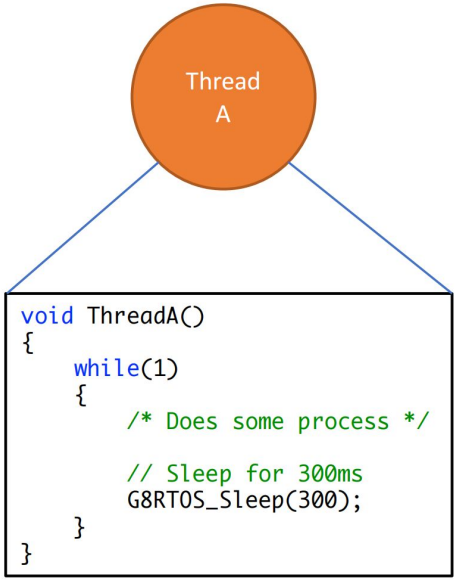
Linked List of Active Threads

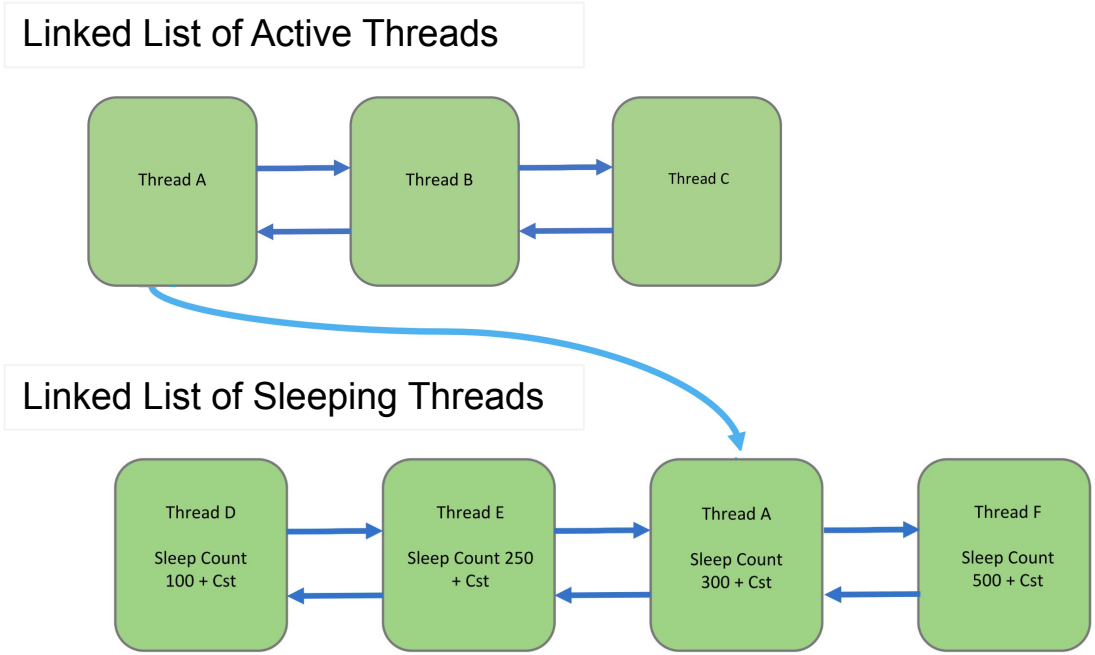


Linked List of Sleeping Threads

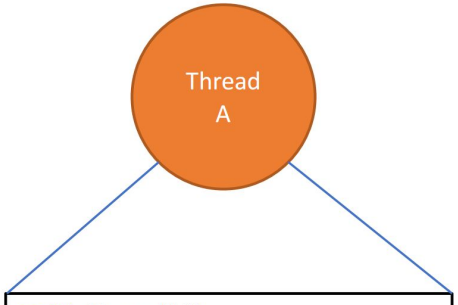


Example: Alternate Sleep Function





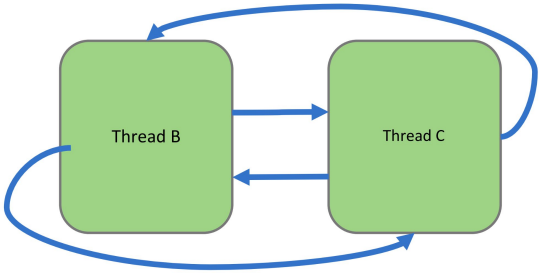
Example: Alternate Sleep Function



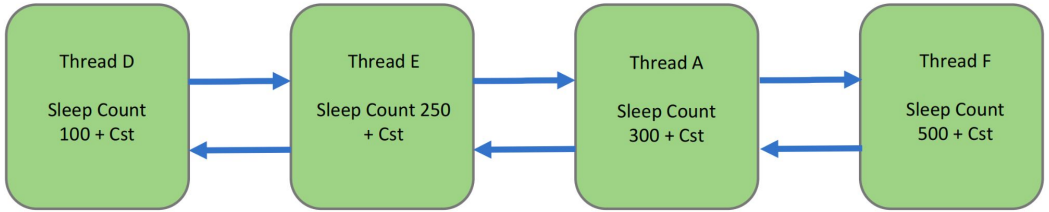

```
void ThreadA()
{
    while(1)
    {
        /* Does some process */

        // Sleep for 300ms
        G8RTOS_Sleep(300);
    }
}
```

Linked List of Active Threads



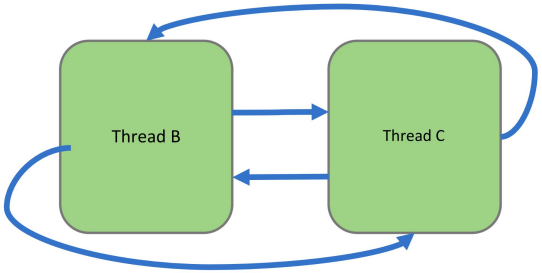
Linked List of Sleeping Threads



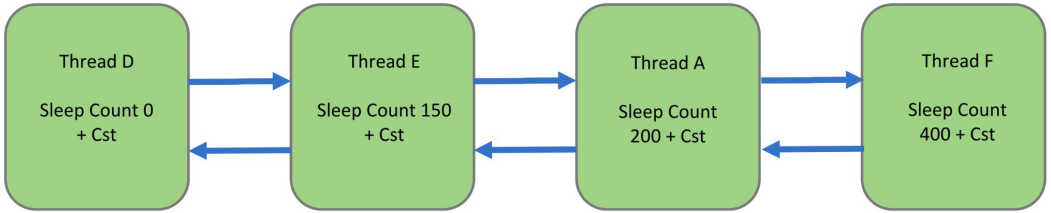
Example: Alternate Sleep Function

Now that the System Time has incremented enough times to equal Thread D's Sleep Count, it is time to add Thread D back into the Active Thread Linked List

Linked List of Active Threads

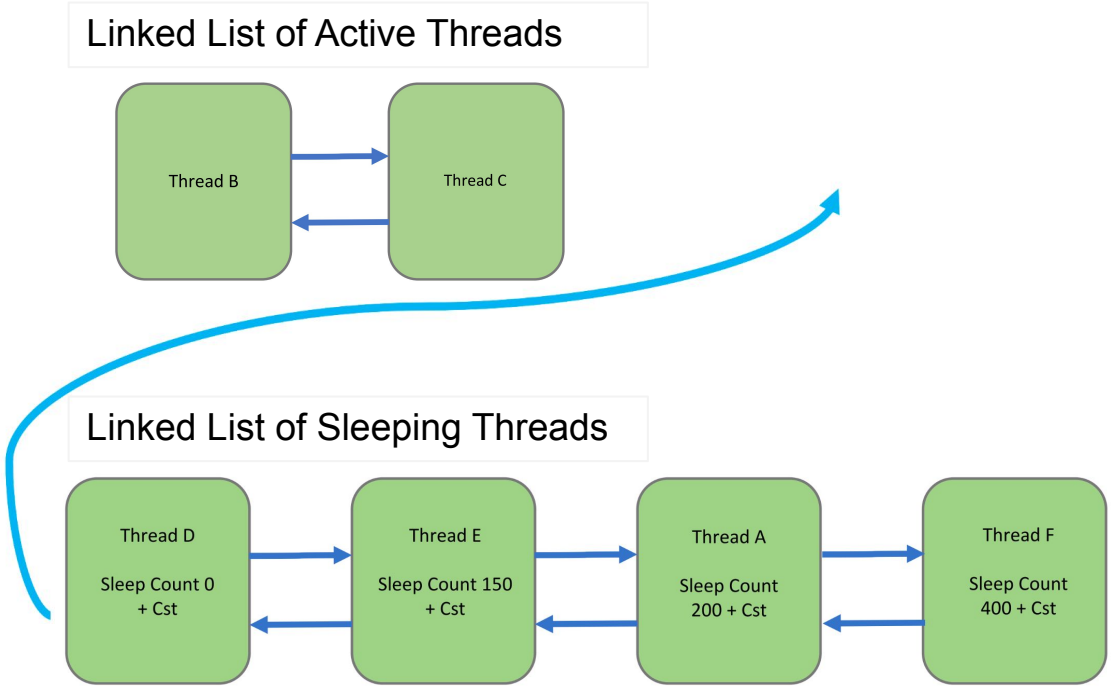


Linked List of Sleeping Threads



Example: Alternate Sleep Function

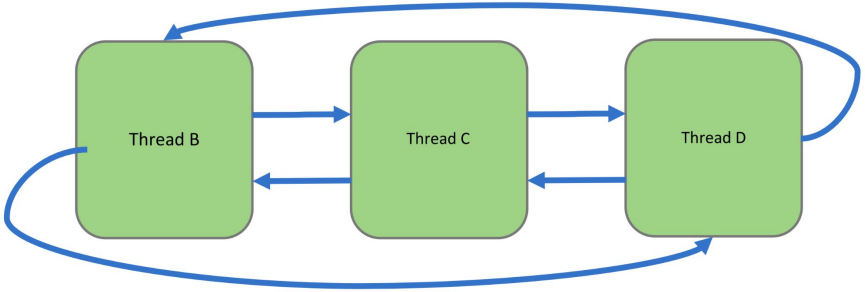
Since the list of active threads is Round-Robin (no priority), we can simply add it to the back of the linked list.



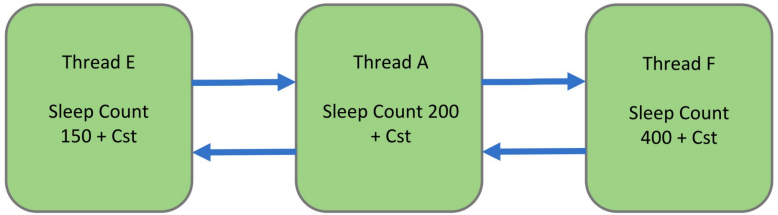
Example: Alternate Sleep Function

Previous and Next TCB pointers are assigned accordingly

Linked List of Active Threads



Linked List of Sleeping Threads

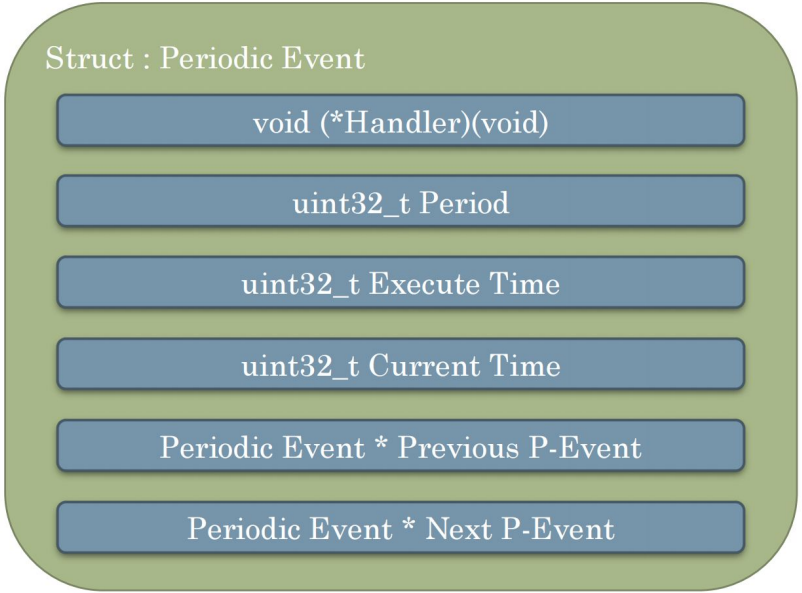


Periodic Threads

- A periodic thread is simply a function that performs a unique task after a certain amount of time has passed
- There are a few ways to trigger periodic threads:
  - **Hardware Timer**
    - If the number of periodic tasks is small, we can allocate a unique hardware timer to each task.
    - Alternatively, we could use just one timer, give each periodic thread a current time and period, and cycle through the events in round-robin fashion.
  - **SysTick Timer**
    - We can use the scheduler as the timer to call periodic events before performing a context switch.
    - This is how it will be implemented for G8RTOS!

Periodic Thread Control Block

```
typedef struct ptcb_t {
    void (*handler)(void);
    uint32_t period;
    uint32_t executeTime;
    uint32_t currentTime;
    struct ptcb_t *previousPTCB;
    struct ptcb_t *nextPTCB;
} ptcb_t;
```



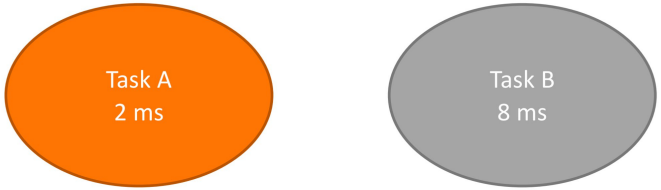
Adding a Periodic Thread in a Linked List

```
/*
 * Adds periodic threads to G8RTOS Scheduler
 * Function will initialize a periodic event struct to represent event.
 * The struct will be added to a linked list of periodic events
 * Param Pthread To Add: void-void function for P thread handler
 * Param period: period of P thread to add
 * Returns: Error code for adding threads
 */
int G8RTOS_AddPeriodicEvent(void (*PthreadToAdd)(void), uint32_t period, uint32_t execution)
{
    // your code
}
```

Recall your implementation of the G8RTOS_AddThread function!

Periods with Common Multiples

- Suppose two periodic events exist with the following periods:



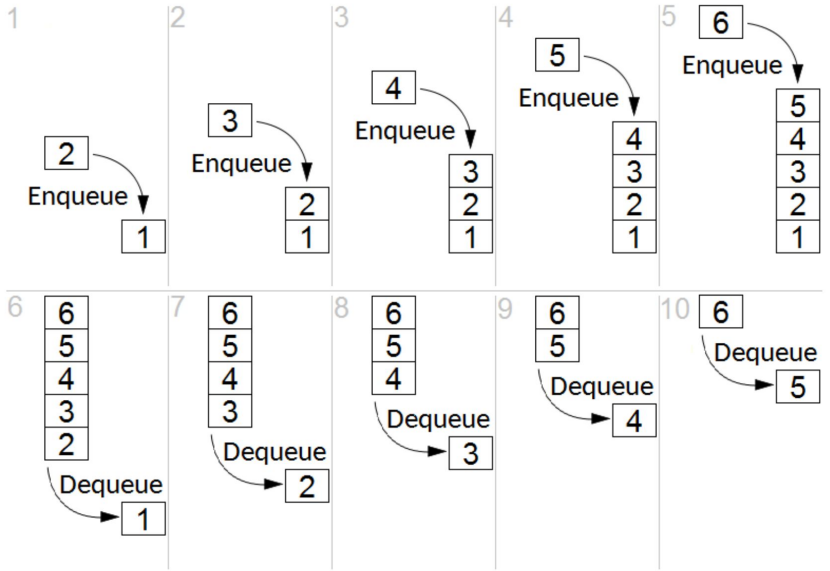
- **Task B** will always occur immediately **after** Task A, because its period is a multiple Task A's
- To combat this, we can give one P-Thread a difference initial **current time** other than 0
- *Example:*
  - Task A initial time = 0, Task B initial time = 1
  - Task A will run 3 times after 6 SysTick interrupts, and Task B will run on the 7th tick
- ***Note:*** *In order for this system to work properly, the maximum time to execute each task must be very short compared to the period of the SysTick to avoid missing interrupts*

Periodic vs Sleeping Threads

- Periodic threads are always in the active state
- Sleeping threads go between the active state and spinning in the sleep state
- Periodic threads ensure a periodic time more accurately
  - *This is because when a thread is done sleeping, it doesn't necessarily mean it is currently **running**, but simply means it is **active** and able to run when it is its turn*

Inter-Process Communication

- We implement the **FIFO** structure as IPC
  - First In First Out
- FIFO
  - Maintain a linked list/array as queue
  - Write to the end of queue
  - Read from the begin of queue
- You have to implement
  - FIFO Initialization
  - Read from FIFO
  - Write to FIFO



Implementing FIFO

- Hints
  - No more than 4 FIFOs are needed in your G8RTOS
  - You can static define FIFOs with array to improve performance
  - Read/Write an `int32_t` data from/into FIFO each time
- Structures
  - Buffer where data will be held: `Int32_t Buffer[FIFOSIZE]`
  - Head pointer: `int32_t *Head`
  - Tail pointer: `int32_t *Tail`
  - Lost data count: `uint32_t LostData`
  - Current Size semaphore: `semaphore_t CurrentSize`
  - Mutex semaphore: `semaphore_t Mutex`

```
22  typedef struct FIFO_t {
23      int32_t buffer[16];
24      int32_t *head;
25      int32_t *tail;
26      uint32_t lostData;
27      semaphore_t currentSize;
28      semaphore_t mutex;
29  } FIFO_t;
30
31  /* Array of FIFOS */
32  static FIFO_t FIFOs[4];
33
```

Read Function of FIFO

- Parameter: an `int32_t` value, which FIFO should be read
- Return value: an `int32_t` value from the head of FIFO
- **Mutex semaphore**
  - Wait before reading from FIFO
  - In case the FIFO is being read from another thread
- **Current Size semaphore**
  - Wait before reading from FIFO
  - In case the FIFO is empty
- When read is complete:
  - Update the head pointer
  - Signal the Mutex semaphore so other waiting threads can read

```
/*
 * Reads FIFO
 *  - Waits until CurrentSize semaphore is greater than zero
 *  - Gets data and increments the head pointer (wraps if necessary)
 * Param: "FIFOChoice": chooses which buffer we want to read from
 * Returns: uint32_t Data from FIFO
 */
int32_t readFIFO(uint32_t FIFOChoice)
{
    // your code
}
```

Write Function of FIFO

- Parameter:
  - An `int32_t` value, which FIFO should be read
  - An `int32_t` value, data to be written into the tail of FIFO
- **Current Size semaphore**
  - The value should be compared with the `FIFOSIZE-1`.
    - Provides 1 buffer cell in case an interrupt happens between reading FIFO and incrementing its head.
    - If the value is larger than `FIFOSIZE-1`, increment the lost data value **and overwrite the old data**.
- Write the data
  - Signal the Current Size semaphore and notify other waiting threads the FIFO is not empty.

```
/*
 * Writes to FIFO
 *  Writes data to Tail of the buffer if the buffer is not full
 *  Increments tail (wraps if necessary)
 *  Param "FIFOChoice": chooses which buffer we want to read from
 *       "Data': Data being put into FIFO
 *  Returns: error code for full buffer if unable to write
 */
int writeFIFO(uint32_t FIFOChoice, uint32_t Data)
{
    // your code
}
```

Implement Threads

- 2 periodic threads and 5 background threads
- 3 FIFOs and 2 Semaphores

- **BackGround Thread 0:**
  - Empty default thread; does nothing *(really?)*
- **BackGround Thread 1:**
  - 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

```
6   #ifndef THREADS_H_
7   #define THREADS_H_
8
9   #include "G8RTOS.h"
10
11  #define JOYSTICKFIFO 0
12  #define TEMPFIFO 1
13  #define LIGHTFIFO 2
14
15  semaphore_t *sensorMutex;
16  semaphore_t *LEDMutex;
17
18  void BackGroundThread0(void);
19  void BackGroundThread1(void);
20  void BackGroundThread2(void);
21  void BackGroundThread3(void);
22  void BackGroundThread4(void);
23
24  void Pthread0(void);
25  void Pthread1(void);
26
27  #endif /* THREADS_H_ */
```

Implement Threads

- 2 periodic threads and 5 background threads
- 3 FIFOs and 2 Semaphores

- **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).
  - *What if temp sensor does not work?*
    - *Use the gyro x-axis*

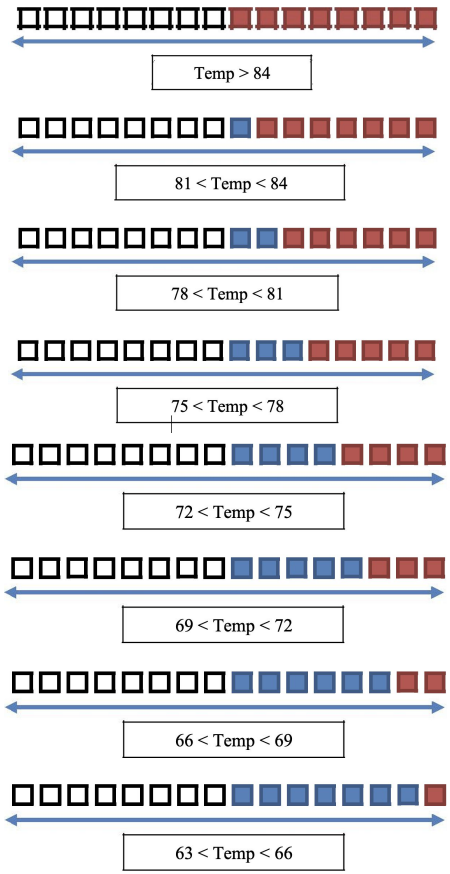
```
6   #ifndef THREADS_H_
7   #define THREADS_H_
8
9   #include "G8RTOS.h"
10
11  #define JOYSTICKFIFO 0
12  #define TEMPFIFO 1
13  #define LIGHTFIFO 2
14
15  semaphore_t *sensorMutex;
16  semaphore_t *LEDMutex;
17
18  void BackGroundThread0(void);
19  void BackGroundThread1(void);
20  void BackGroundThread2(void);
21  void BackGroundThread3(void);
22  void BackGroundThread4(void);
23
24  void Pthread0(void);
25  void Pthread1(void);
26
27  #endif /* THREADS_H_ */
```

Implement Threads

- 2 periodic threads and 5 background threads
- 3 FIFOs and 2 Semaphores

- **BackGround Thread 2:**
  - Read the light sensor.
  - Send data to light FIFO.
  - Sleep for 200ms.

- **BackGround Thread 3:**
  - Read temperature FIFO.
  - Output data to Red/Blue LEDs as shown in the figure. Feel free to adjust or normalize the temperature values if needed.



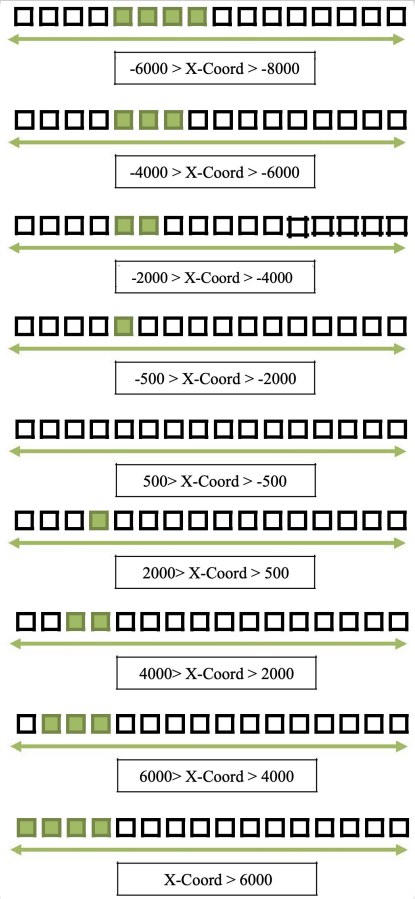
Implement Threads

- 2 periodic threads and 5 background threads
- 3 FIFOs and 2 Semaphores

- **BackGround Thread 4:**
  - Read the joystick's FIFO.
  - 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 figure.



Your Feedback Summary

⇒ Positives

- Good contents, utility for career options, good hands-on and theory balance, good slides.
- The board setup is cool, manageable workload, templates are helpful.
- The bonus points and questions in-class are great. Topics are well structured.
- The book & slide contents are consistent with lab-work, RTOS seems manageable now.
- TAs are awesome and very helpful. They try to accommodate based on given situations.

⇒ Things to improve:

- Some OS concepts should be covered at the beginning; crash-course was useful but late.
- Some Lab-2 solutions are in the book, should have more challenging problems.
- Should have more TAs, office hours are sparse.
- Lecture slides could be annotated. Lab solutions can be discussed.

⇒ Other suggestions

- *Topics: varied opinions about depth and time-spent (slow/fast) on certain topics.*
- *Labs: some theoretical/on-paper homeworks will be nice to complement the labs.*
- *In-class: Sometimes the class is too quiet (speaker is not turned on always).*
- *Others: ...*

Rating (0-10)
Count: 41

Avg: 9.42
Median: 9.4

Min: 7 (# 5)
Max: 12 (# 4)

Logistics

⇒ Lab-2 demo and quiz-1 ends this week

⇒ Lab-3 demos start from next Monday (Oct 10th)

⇒ Mid Exam

- October 14th Friday: regular class time
- Time: 40 minutes
- We will discuss question patterns and practice questions in the coming weeks