Nelms Institute for the Connected World
UNIVERSITY of FLORIDA

Lab 3

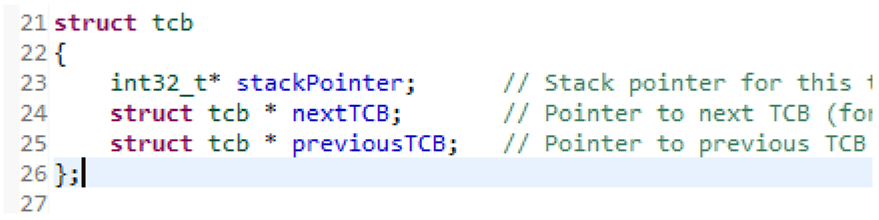
SLEEPING, PERIODIC EVENT THREADS, YIELDING

Overview

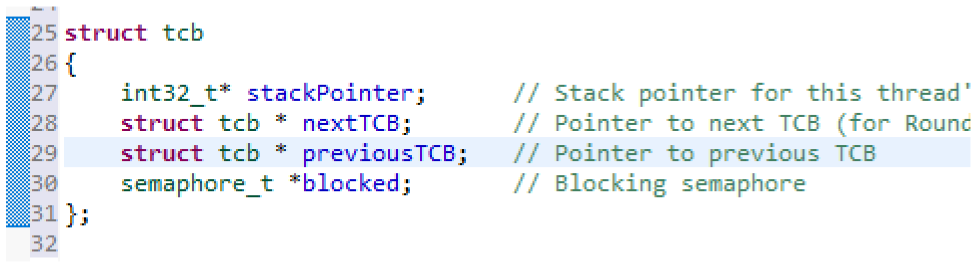
- Improve your implementation of RTOS in Lab 2.
- You have to implement:
  - Improved semaphores, blocking, and yielding.
  - Sleeping.
  - Periodic event threads.
  - FIFOs.
- You have to demonstrate:
  - Periodic threads.
  - Background communication via FIFO.
  - Joystick, temperature sensor, light sensor usage.

Improvement

- Blocked Semaphore
  - We implemented a simple spinlock semaphore in Lab 2. While checking the available flag, the CPU time was wasted.
  - We improve this by add 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.



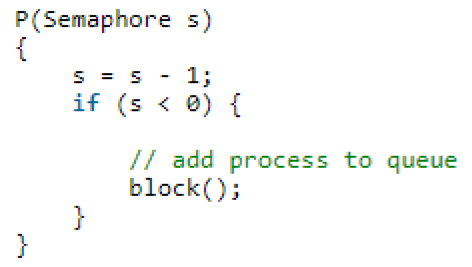
```
struct tcb
{
    int32_t* stackPointer;      // Stack pointer for this
    struct tcb * nextTCB;       // Pointer to next TCB (for
    struct tcb * previousTCB;   // Pointer to previous TCB
};
```



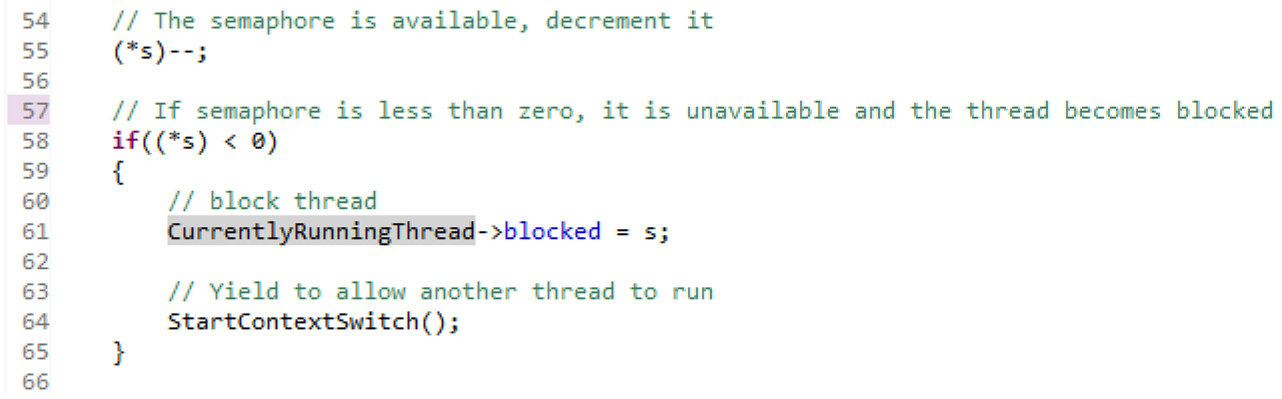
```
struct tcb
{
    int32_t* stackPointer;      // Stack pointer for this thread'
    struct tcb * nextTCB;       // Pointer to next TCB (for Round
    struct tcb * previousTCB;   // Pointer to previous TCB
    semaphore_t *blocked;       // Blocking semaphore
};
```

Improvement

- Improve “Semaphore Wait” part of the semaphore library component. If the semaphore is not available, the blocked semaphore of running thread should be initialized. Then yield the control to next available thread.



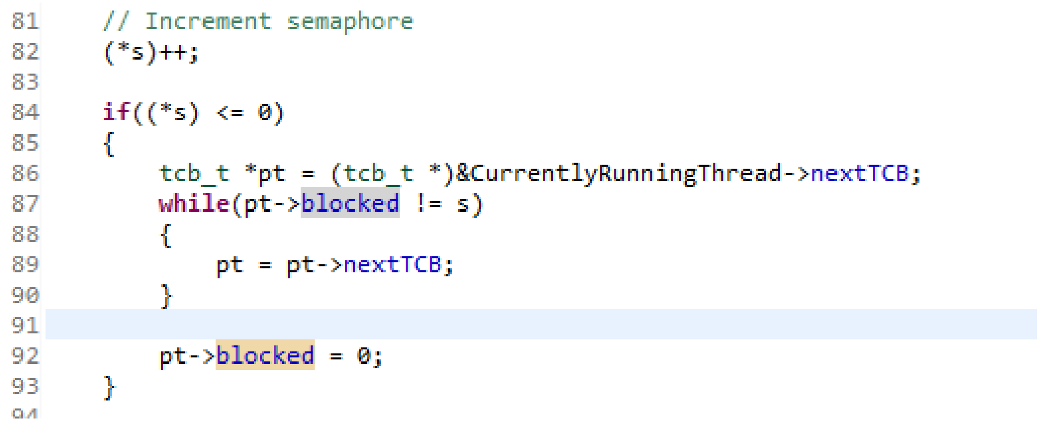
```
P(Semaphore s)
{
    s = s - 1;
    if (s < 0) {

        // add process to queue
        block();
    }
}
```



```
    // The semaphore is available, decrement it
    (*s)--;

    // If semaphore is less than zero, it is unavailable and the thread becomes blocked
    if((*s) < 0)
    {
        // block thread
        CurrentlyRunningThread->blocked = s;

        // Yield to allow another thread to run
        StartContextSwitch();
    }

```

Improvement

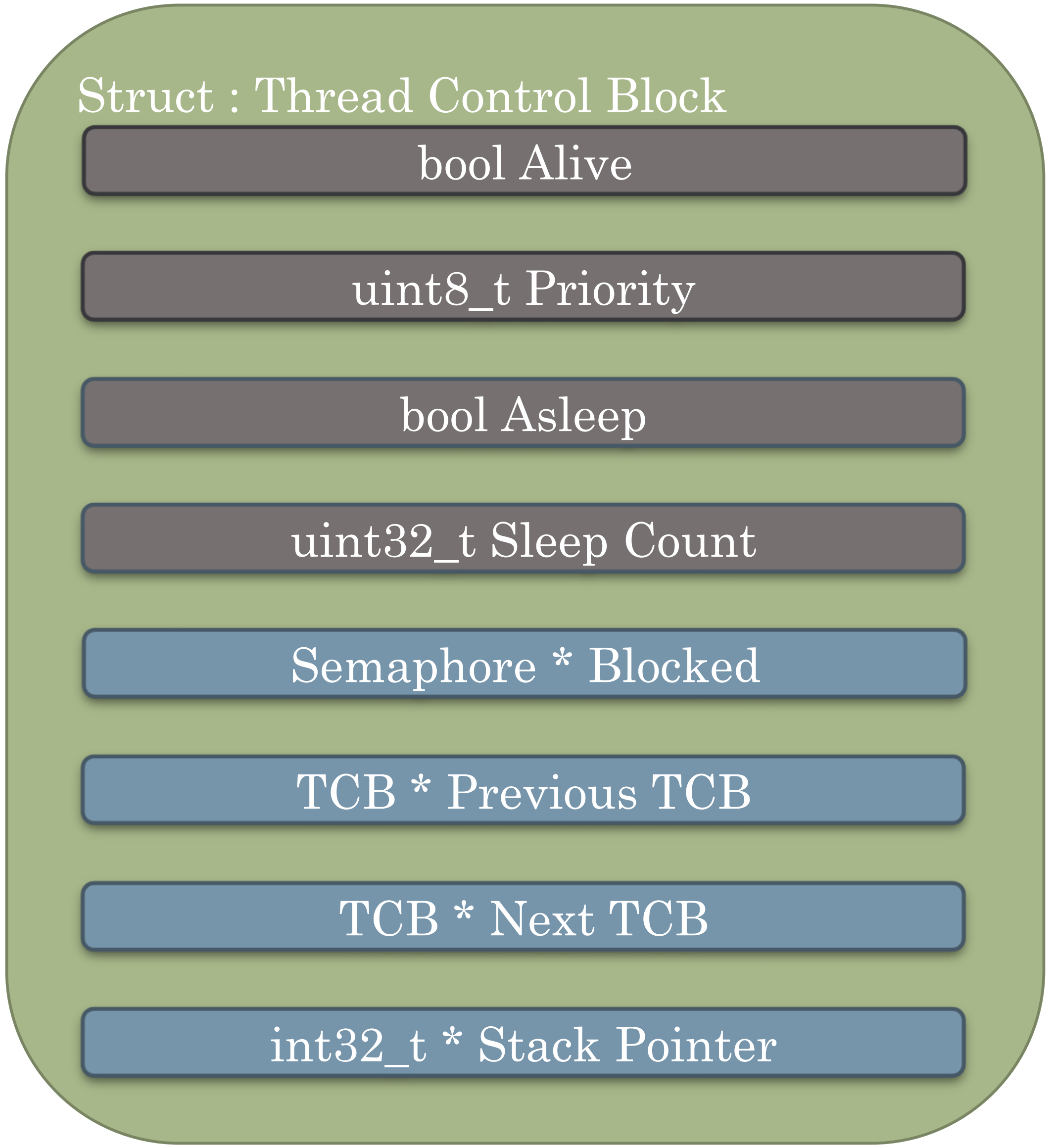
- Improve "Signal Semaphore" part of the semaphore library component. Go through the TCB list and unblock the first thread that is blocked on the exact same semaphore. Move the unlocked thread to the next thread to be executed.



```
    // Increment semaphore
    (*s)++;

    if((*s) <= 0)
    {
        tcb_t *pt = (tcb_t *)&CurrentlyRunningThread->nextTCB;
        while(pt->blocked != s)
        {
            pt = pt->nextTCB;
        }

        pt->blocked = 0;
    }
```

Improvement

- Improve the scheduler of RTOS. It must verify the next TCB is not blocked. If it is blocked, keep iterating the next TCB through the linked list.

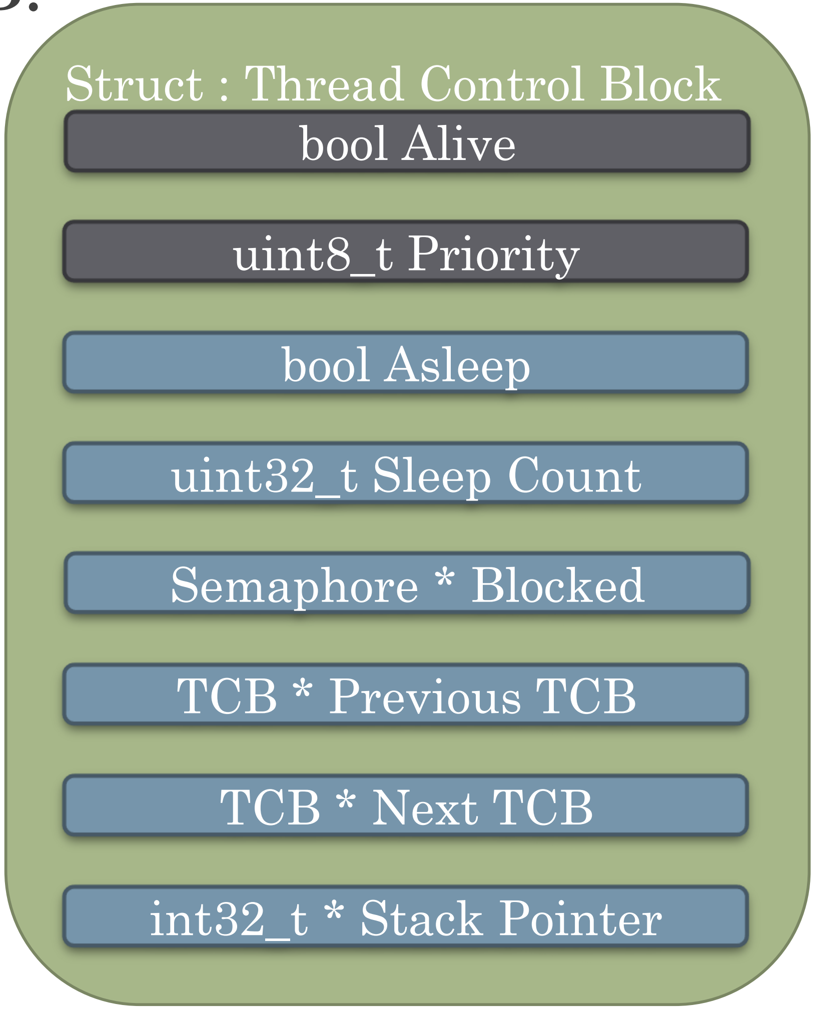


Sleeping

- **Active State**: Thread is ready to run but waiting for its turn
- **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)

TCB Sleeping Parameters

- Two new parameters are added to the TCB:
  - Sleep Count
  - Asleep

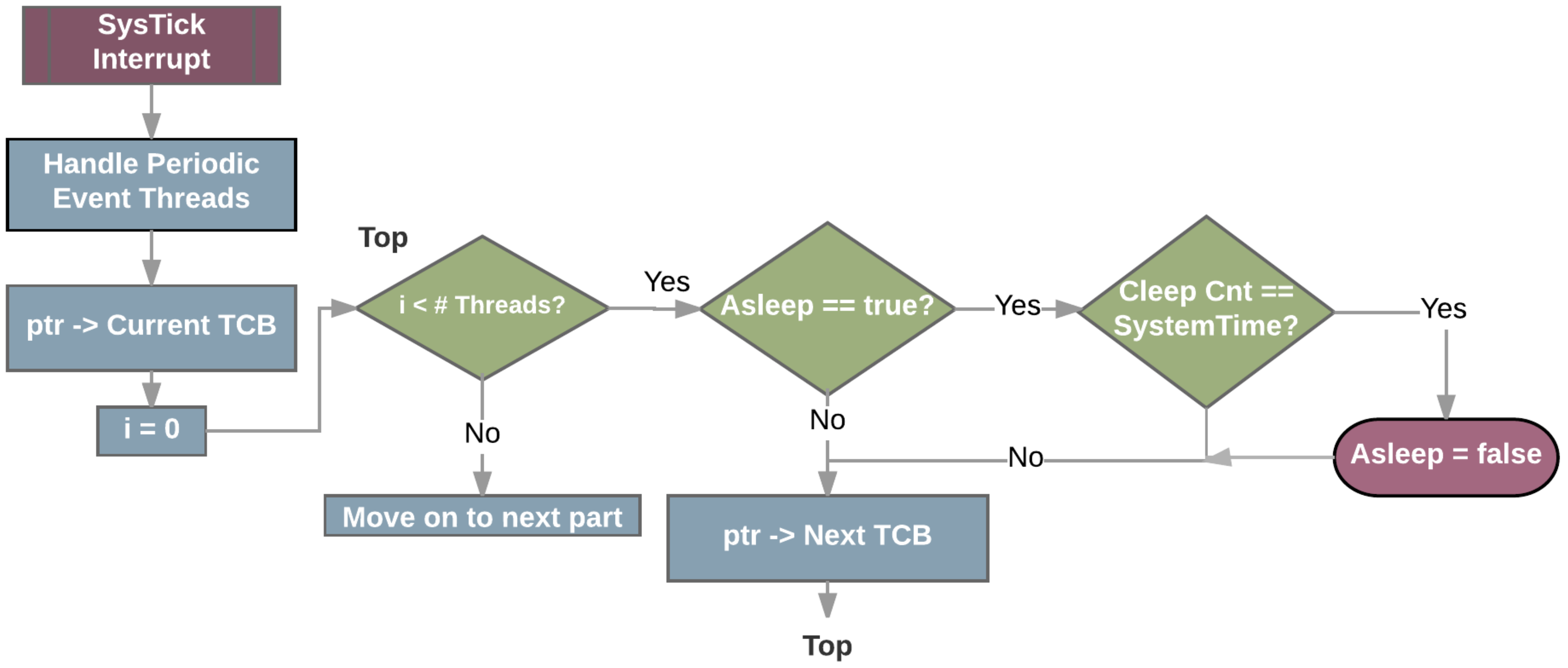


OS_Sleep Function

- When a thread needs to wait a prescribed amount of time, it will call the function **OS_Sleep**
- OS_Sleep takes in a **uint32_t Duration** (time in milliseconds)
- The function will perform the following:
  - 1. Initialize the Currently Running Thread's Sleep Count
    - Sleep Count = Duration + SystemTime
  - 2. Put thread to sleep
    - Asleep = True
  - 3. Yield control of this thread to allow other threads to run (start context switch)
- In the SysTick handler, we will 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 woken up. Otherwise, it remains sleeping.

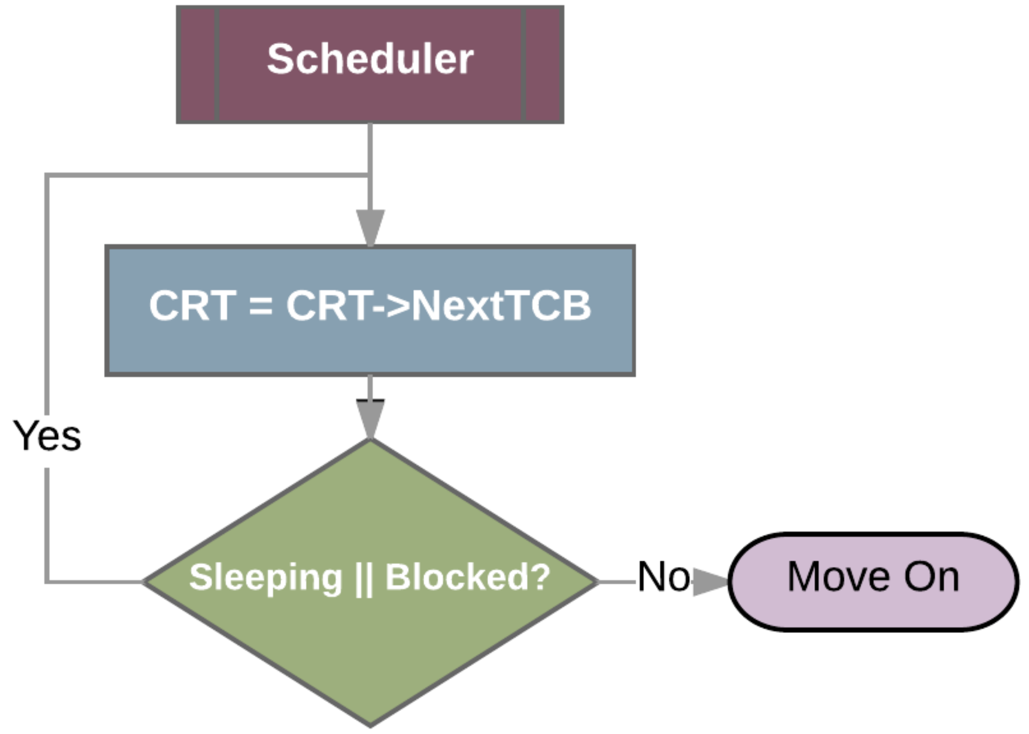
Handling Sleep Counts

- Implement this in SysTick handler



Scheduler improvement

- Check for Sleeping status and Blocking status before assigning the next thread control block as the currently running thread within the scheduler function.

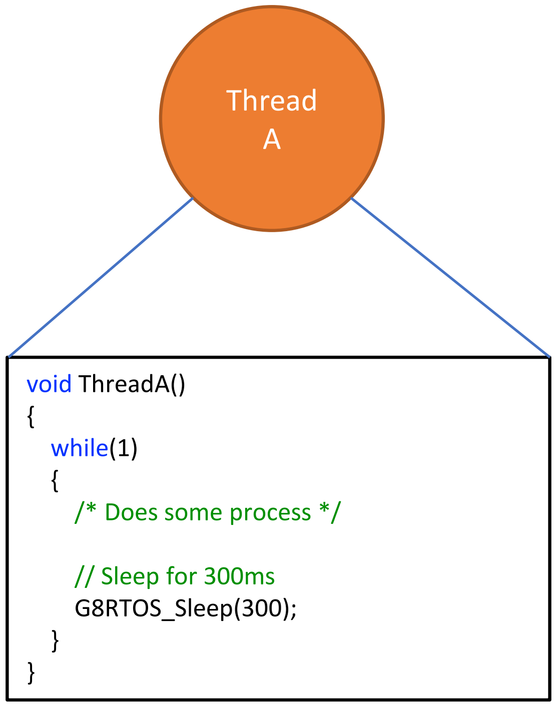


*Note: It is possible that all threads can be either sleeping or blocked, in which case we enter an infinite loop here. To avoid this for now, simply adding an idle thread will solve this problem. Later on when we implement priority, this thread will be the lowest priority thread.

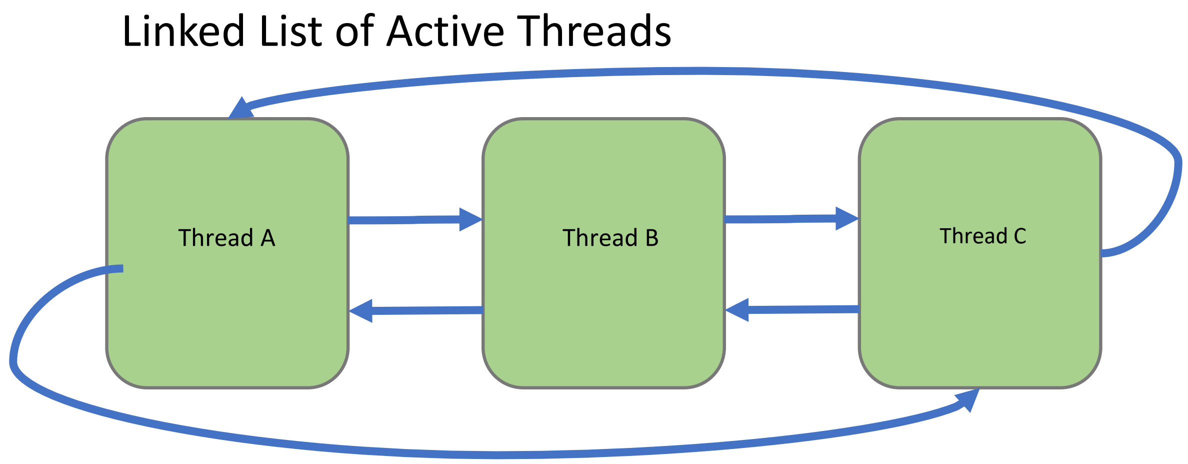
Alternate Sleeping Implementation

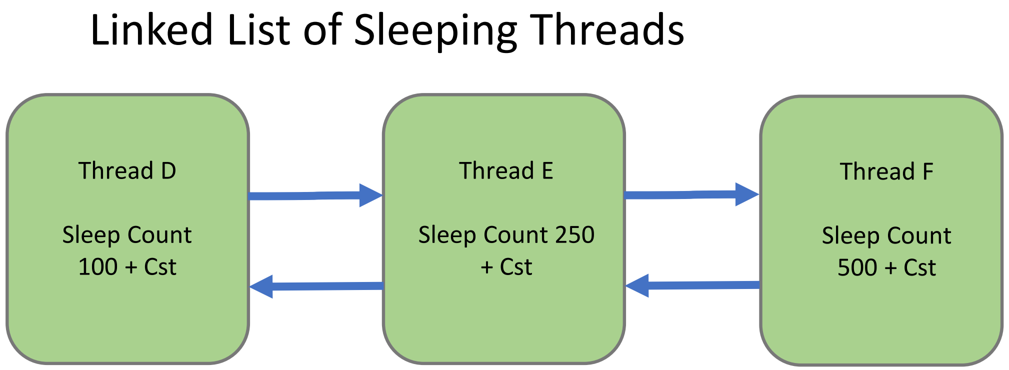
- Another way to implement sleeping is to **remove** the new sleeping thread from the linked list of active threads, and adding it to **new** doubly linked list of sleeping threads
- This new list of sleeping threads will be **sorted** from smallest to highest sleep count
- Once the thread with the lowest 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

Alternate OS_Sleep Function

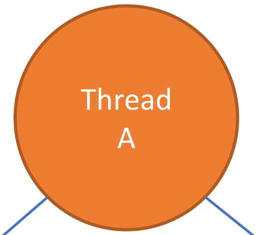


```
void ThreadA()
{
   while(1)
   {
      /* Does some process */

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

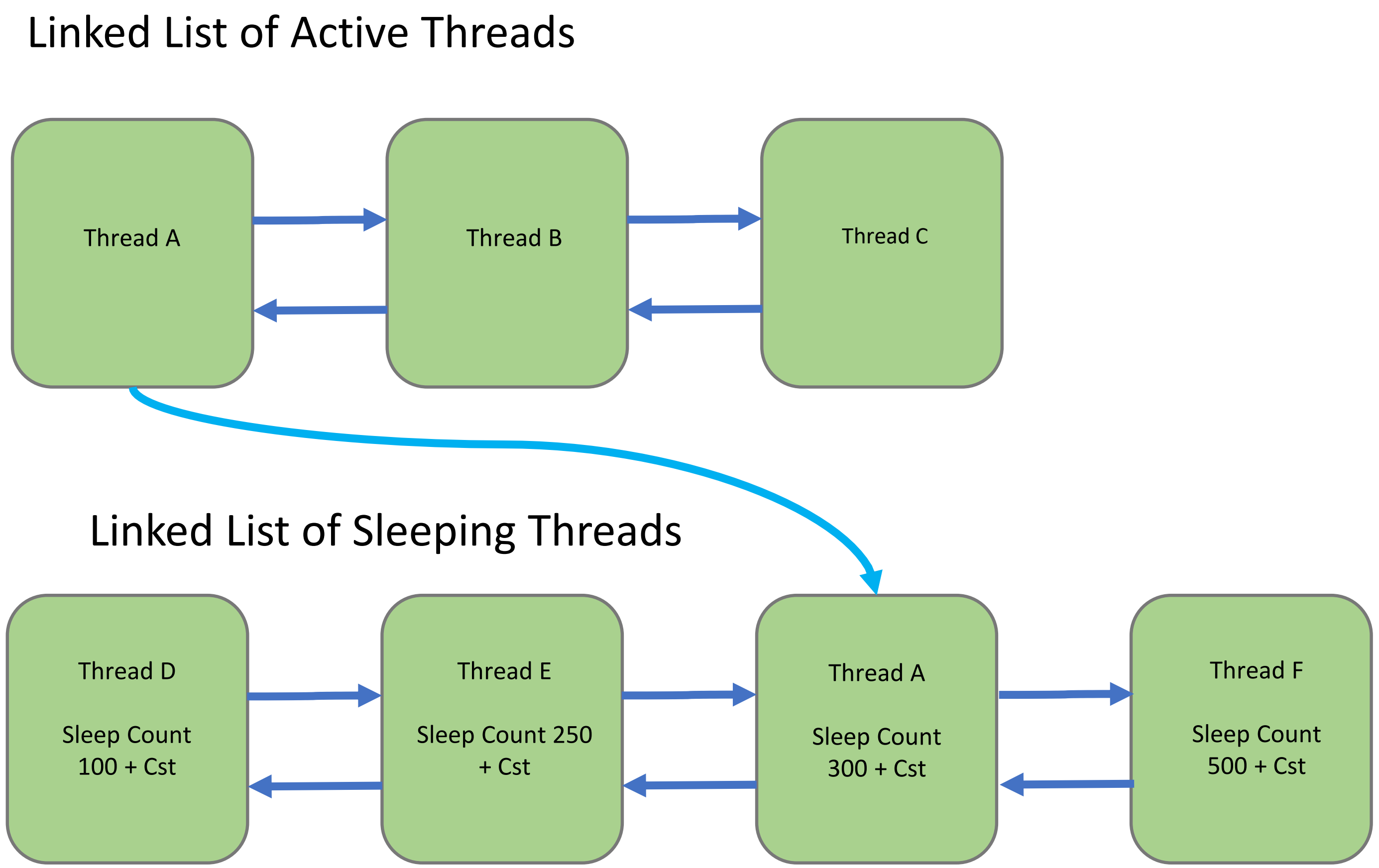




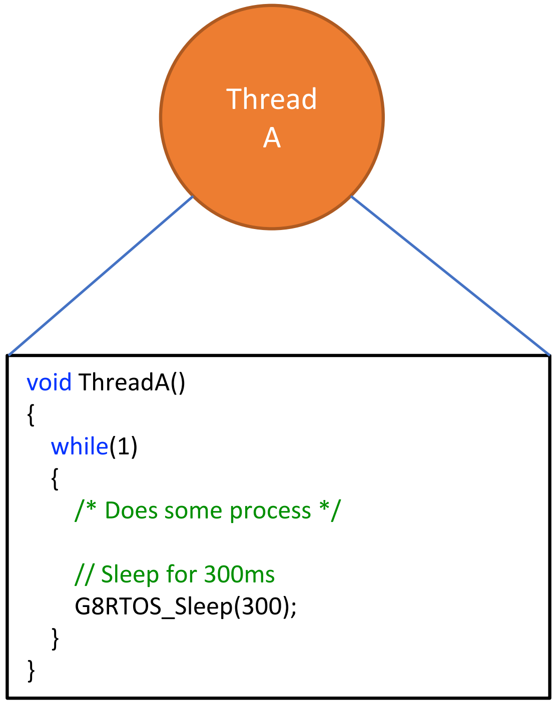
Alternate OS_Sleep Function



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

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

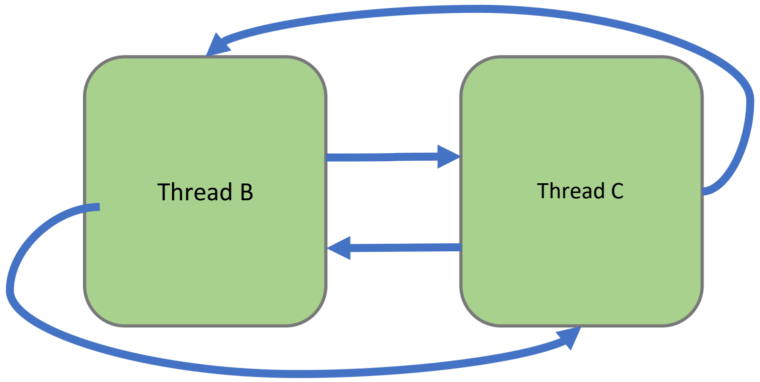


Alternate OS_Sleep Function

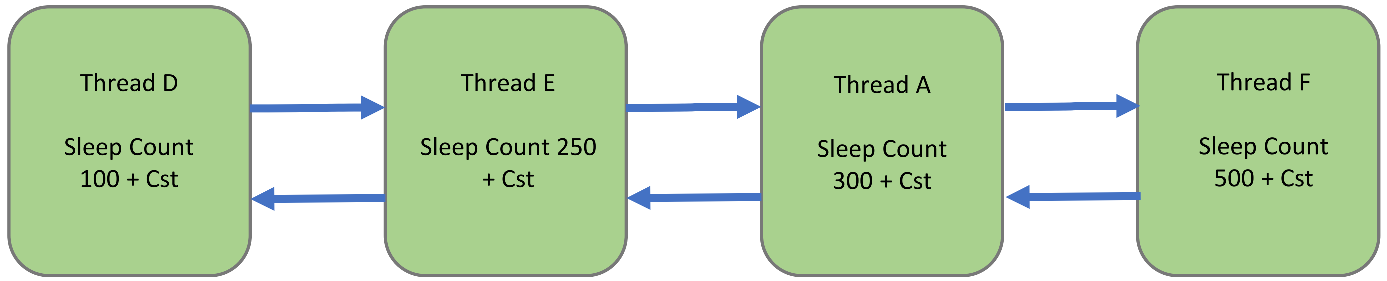


```
void ThreadA()
{
   while(1)
   {
      /* Does some process */

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

Linked List of Active Threads

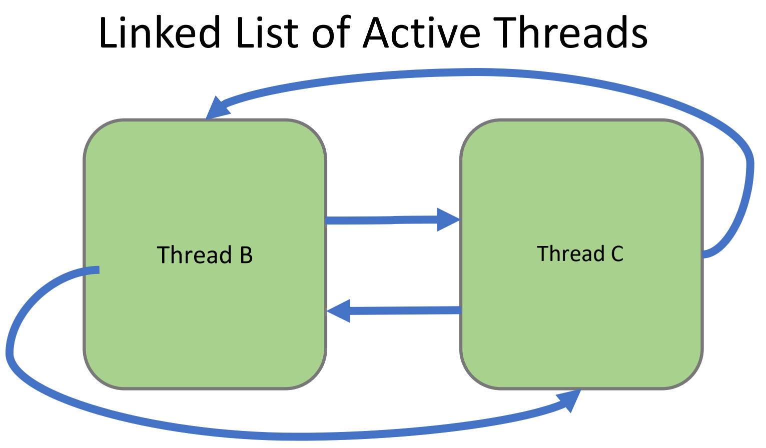


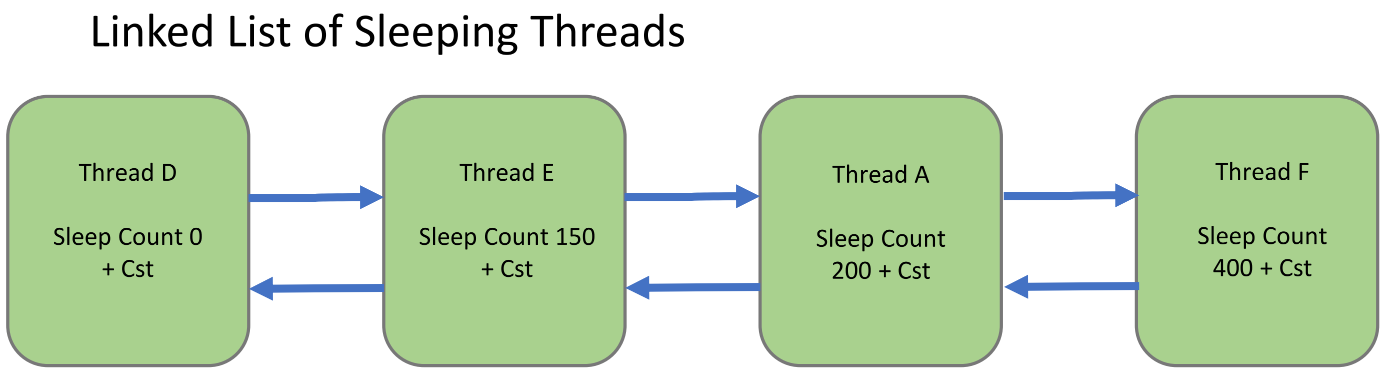
Linked List of Sleeping Threads



Alternate OS_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

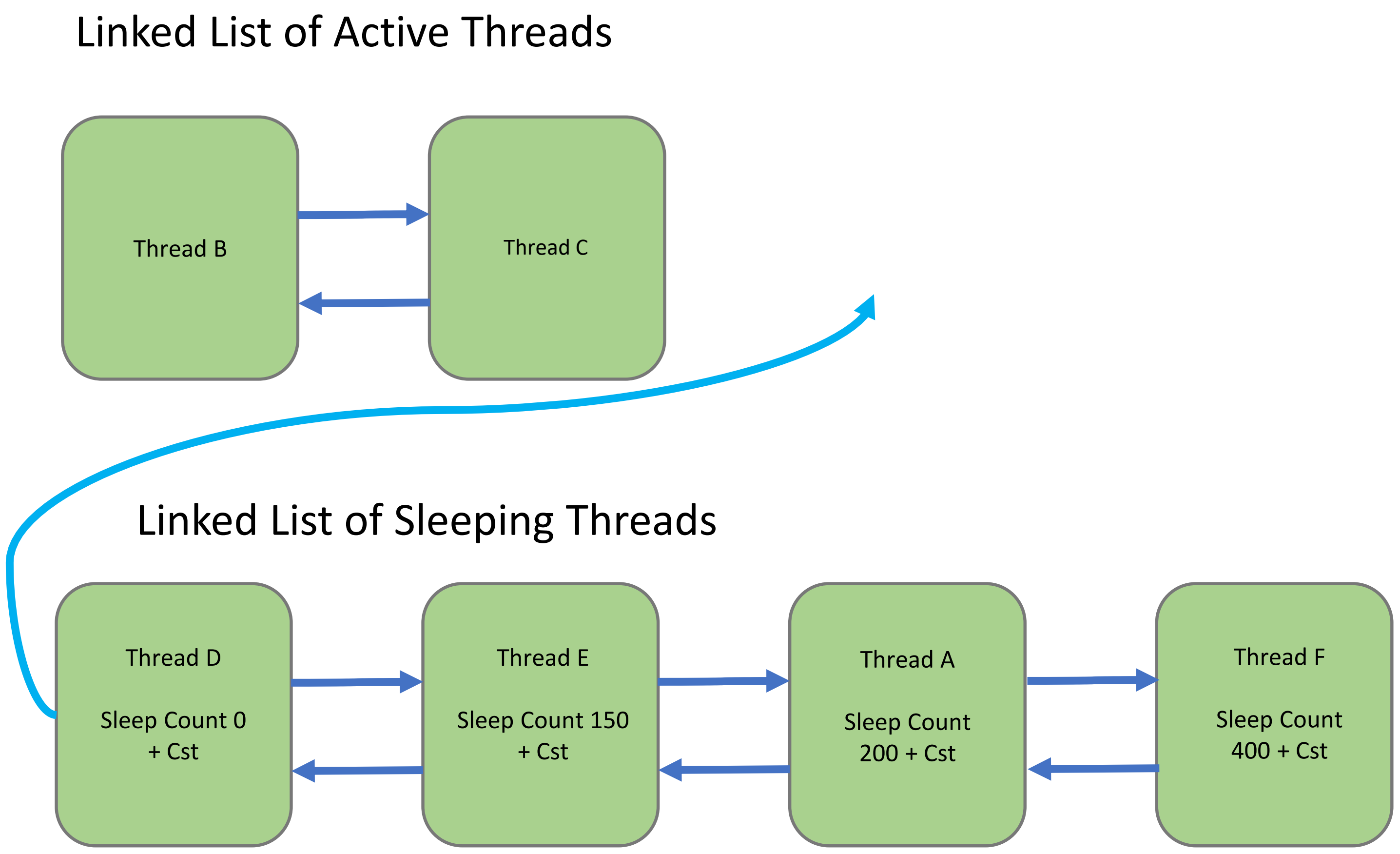




Linked List of Sleeping Threads
Thread D Sleep Count 0 + Cst
Thread E Sleep Count 150 + Cst
Thread A Sleep Count 200 + Cst
Thread F Sleep Count 400 + Cst

Alternate OS_Sleep Function

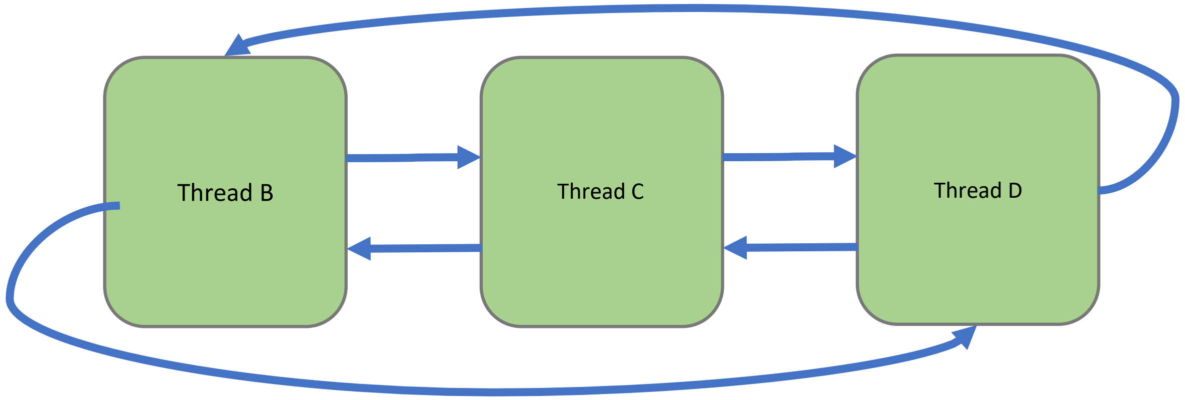
Since the list of active threads is Round-Robin (no priority), we can simply add it to the back of the linked list.



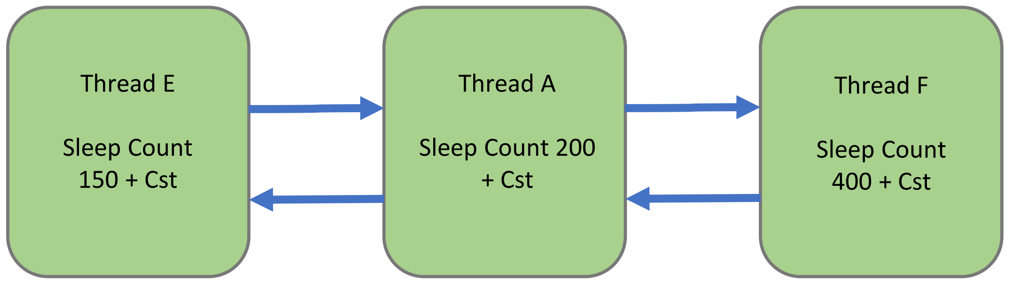
Alternate OS_Sleep Function

Previous and Next TCB pointers are assigned accordingly

Linked List of Active Threads



Linked List of Sleeping Threads



Issue of OS_Sleep

- You properly just noticed that the actual sleep time of OS_Sleep function is not accurate. Thus if you need a more accurate execution, you need time interruption and/or periodic thread.

Modification of RTOS

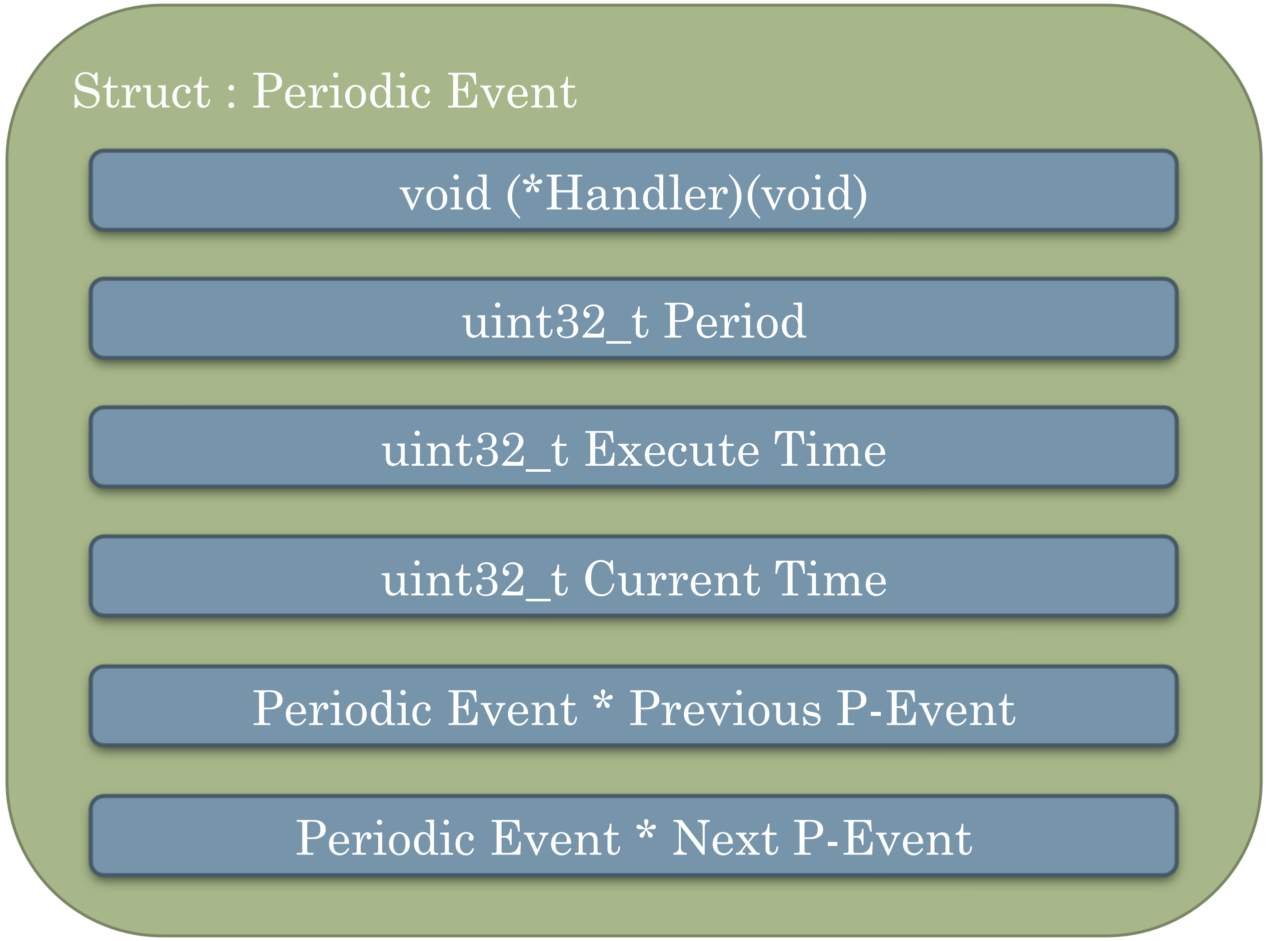
- Since we want the asleep thread yielding the control while sleeping, there are few modifications we need to do.
- Structures. We need to add two more items in TCB.
  - Sleep flag.
  - Sleep count = Duration + Current SystemTime
- Scheduler.
  - We need to continue to the next thread if the current running thread is asleep.
- SysTick Handler
  - Sets sleep counts to zero for every TCB if sleeping threads need to be woken up

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:
  1. **Hardware Timer(s)**
     a) If the number of periodic tasks is small, we can allocate a unique hardware timer to handle each task
     b) Alternatively, we could use just one timer, give each periodic thread a current time and period, and cycle through the events round-robin. After incrementing the thread's current time every interrupt, we run the task if the current time equals its period.
  2. **SysTick Timer**
     a) Much like 1.b, 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
     b) Example: Assume the SysTick is scheduled to interrupt every 1 ms, and we wish to run a periodic task every 10 ms, we could call the periodic function after entering this ISR 10 times, keeping count the same way as 1.b.

Adding a Periodic Thread in a Linked List

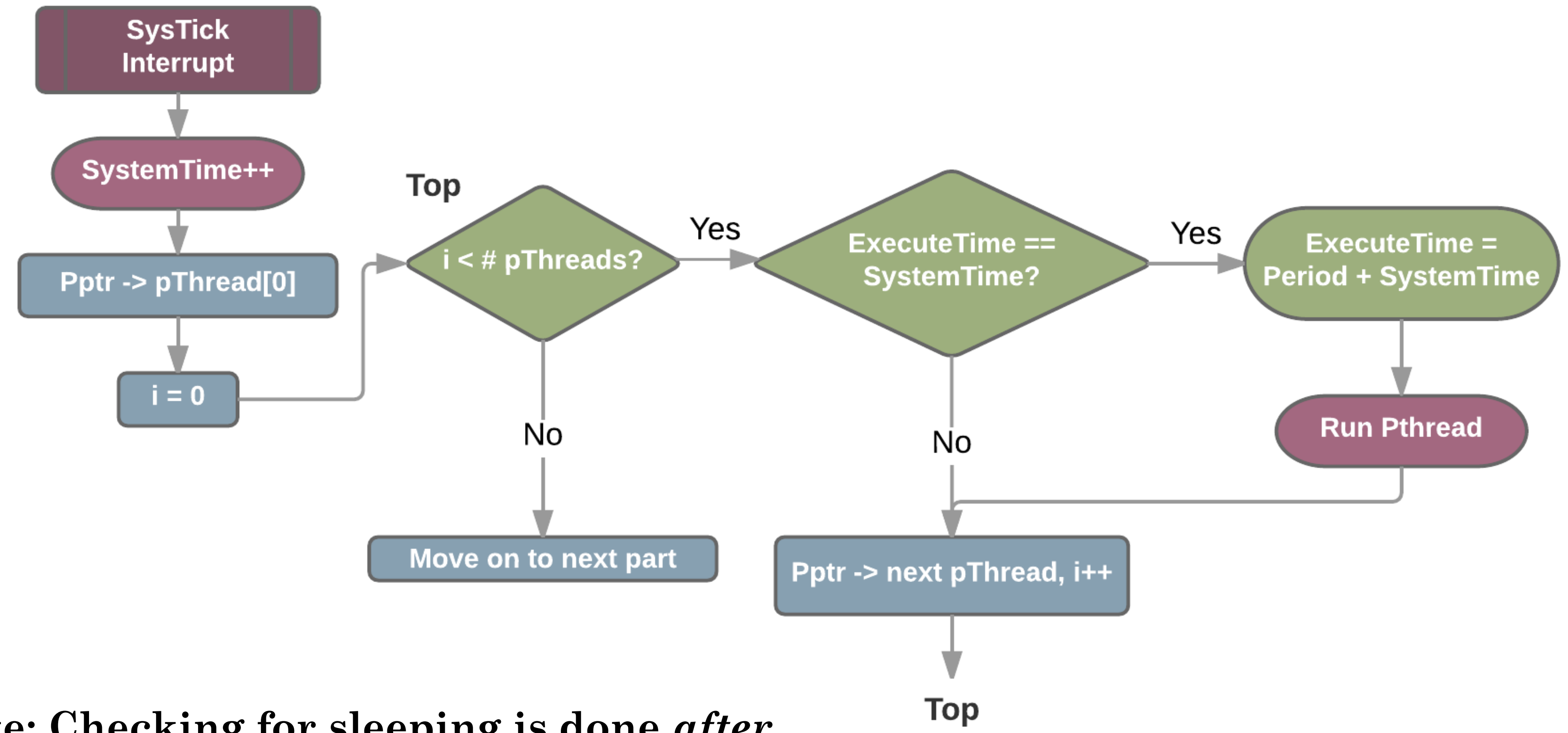
- **Periodic Threads** are responsible for holding information regarding the event's state, much like a thread control block.



Adding a Periodic Thread in a Linked List

- An “Add Periodic Event” function (called from the Main) will take in a function pointer to the periodic handler and its corresponding period
- The function should exist within the scheduler’s source file
- It will initialize the periodic thread’s struct as well as handle the linked list data structure
- It also increments the static number of periodic threads inside of the scheduler’s source file

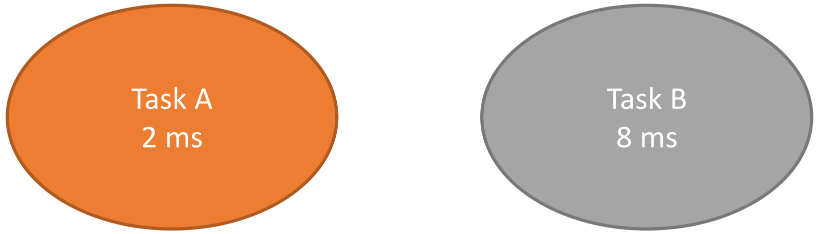
Periodic Events within SysTick Handler



Note: Checking for sleeping is done *after* checking periodic threads (next part)

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 much be very short compared to the period of the SysTick to avoid missing interrupts

Benefit of Using SysTick Timer

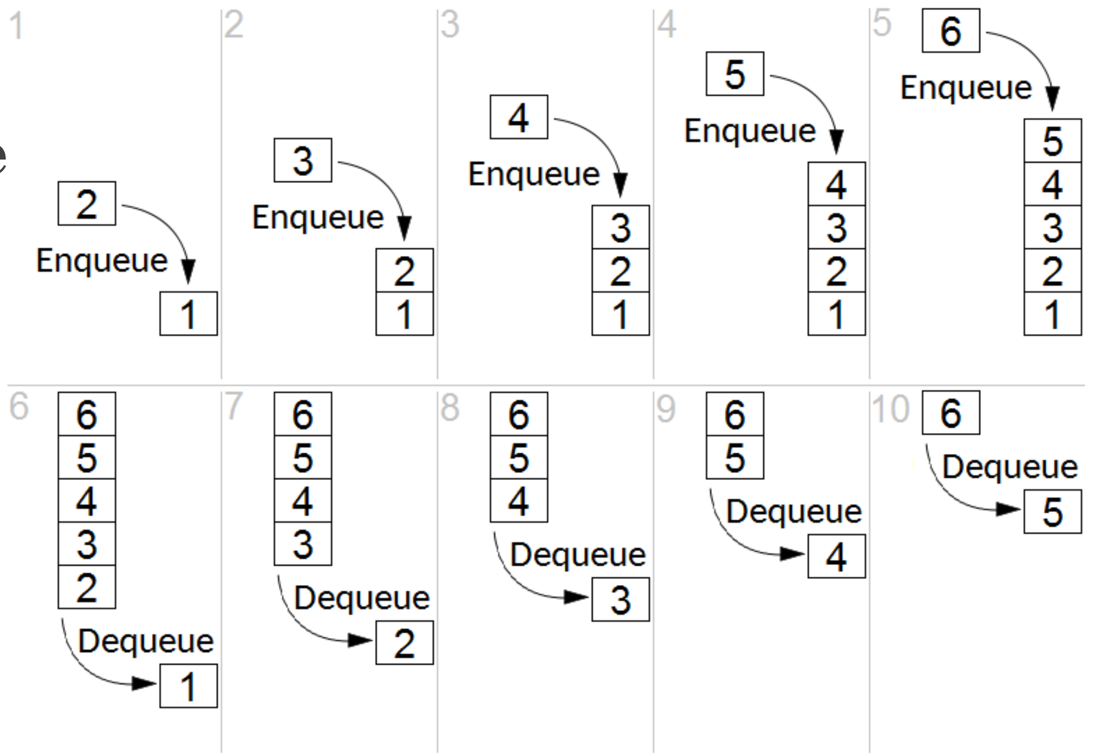
- The MSP432 has four 16-bit timers and two 32-bit timers
- By using the SysTick timer to schedule periodic events, we now have access to one more timer than can be used for other tasks
- *Example:* Timer A's interrupt period can be set to sample the ADC at a much faster rate than a minimum of the 1ms SysTick timer
  - This is considered to be an aperiodic events, that just so happens to be triggered periodically
- The down-side to using the SysTick timer to schedule periodic events is that it slightly slows down rate at which the system can perform context switches driven by the SysTick

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 than prescribing an amount of sleep time for a thread
- 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



Implementation of FIFO

- Requirement
  - No more than 4 FIFOs in 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`

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.
- Done with the read:
  - Update the head pointer.
  - Signal the Mutex semaphore so other waiting thread can read.

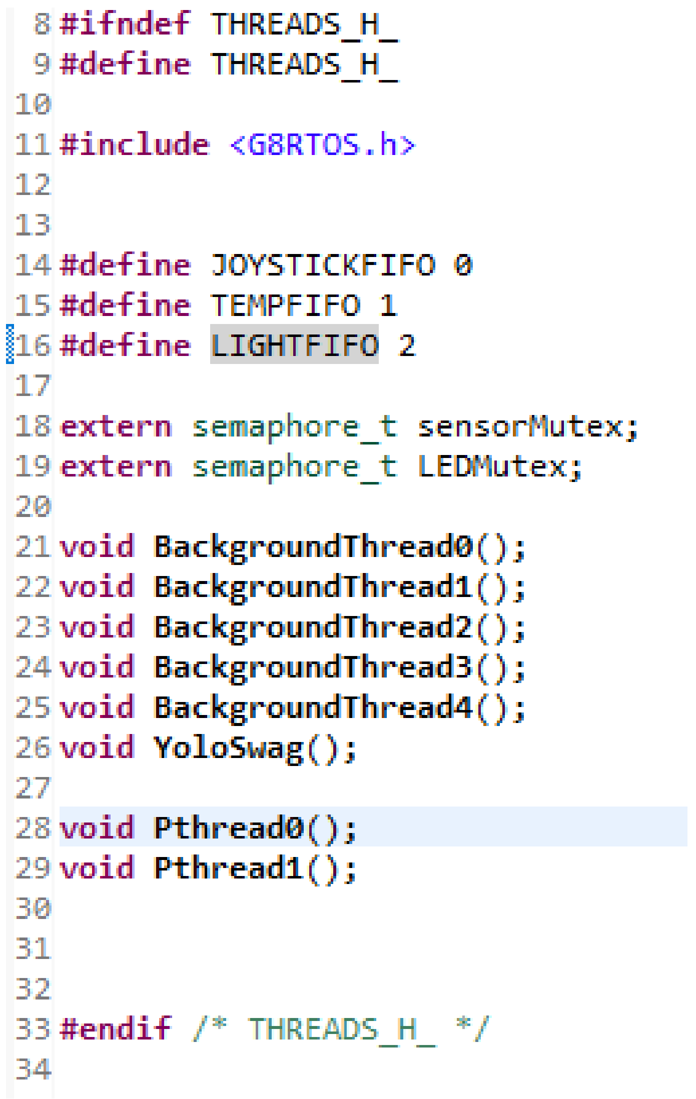
Write function of FIFO

- Parameter:
  - an `int32_t` value, which FIFO should be read.
  - an `int32_t` value, the data that should be wrote 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 `IFOSIZE-1`, increment the lost data value and return an error code.
- Write the data
  - Update the tail pointer
  - Signal the Current Size semaphore and notify other waiting threads the FIFO is not empty.

Final demonstration

- 2 periodic threads
- 6 background threads
- 3 FIFOs
- 2 Semaphores

* Refer to the lab documentation and find the details of implementation.



```c
#ifndef THREADS_H_
#define THREADS_H_

#include <G8RTOS.h>


#define JOYSTICKFIFO 0
#define TEMPFIFO 1
#define LIGHTFIFO 2

extern semaphore_t sensorMutex;
extern semaphore_t LEDMutex;

void BackgroundThread0();
void BackgroundThread1();
void BackgroundThread2();
void BackgroundThread3();
void BackgroundThread4();
void YoloSwag();

void Pthread0();
void Pthread1();



#endif /* THREADS_H_ */

```

Root Mean Square (RMS)

- Iterative formula
  - $x_{k+1} = \frac{1}{2}\left(x_k + \frac{n}{x_k}\right), \quad k \geq 0, \; x_0 = n$
  - Until $|x_{k+1} - x_k| < 1$
- Use Do-While loop
- Question:
  - How to do ½ with shift value?
  - How do we implement the $\frac{n}{x_k}$ part of formula?