Lab 2

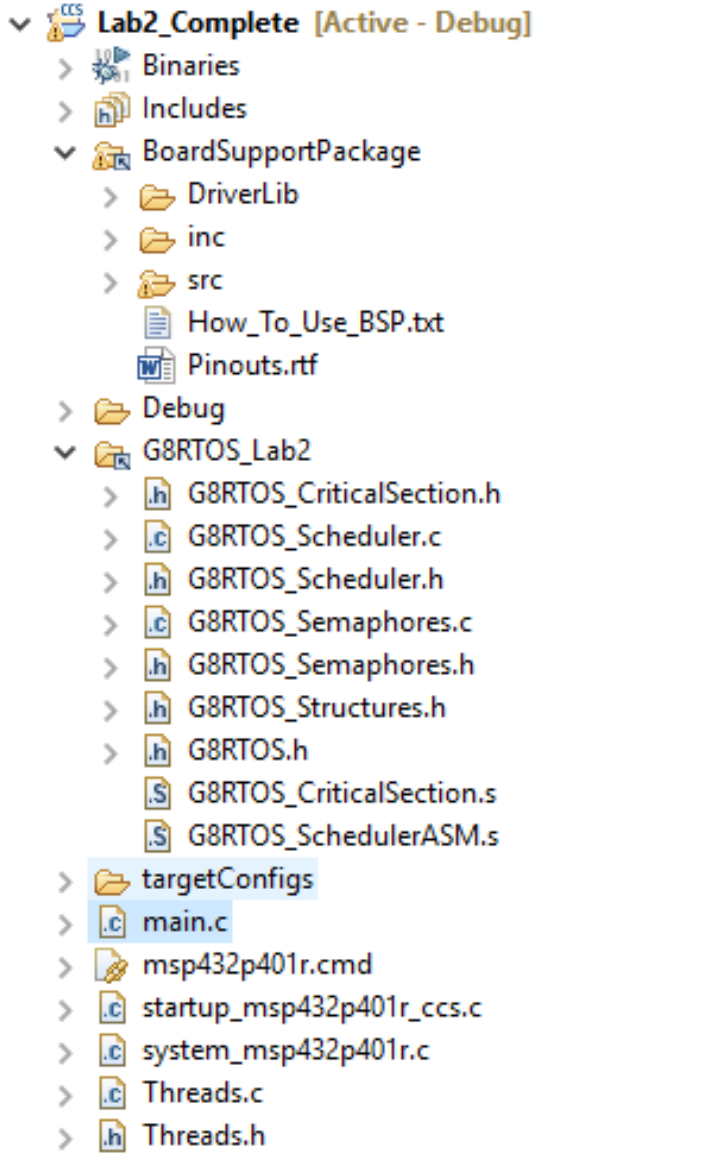
G8RTOS IMPLEMENTATION

Overview

- Implement a basic RTOS on your board.
- You have to implement:
  - OS Structures (Thread Control Block)
  - OS Initialization functions (SysTick and Structures)
  - A Round-Robin Scheduler
  - Context Switching (PendSV handler)
  - Thread related function (Add thread)
  - Launch and start your RTOS

Board Support Package

- You will be given a boardsupportpackage with a bunch of useful functionality
- Create a new CCS Project
- Follow the instructions in Lab 1 and add the new BSP

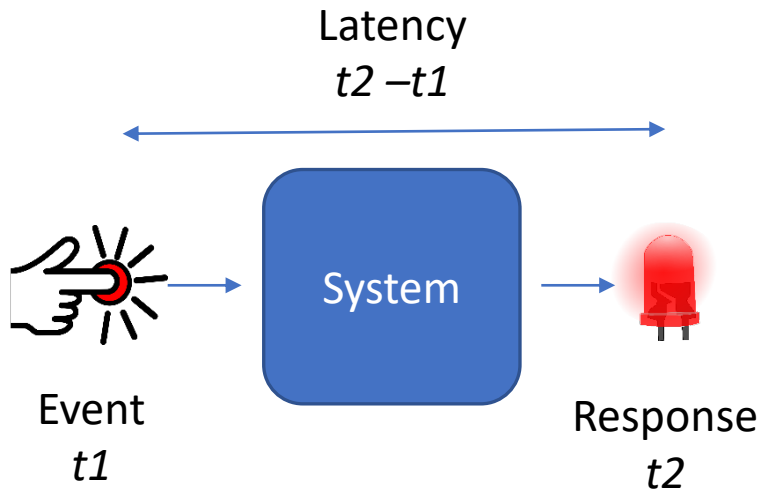


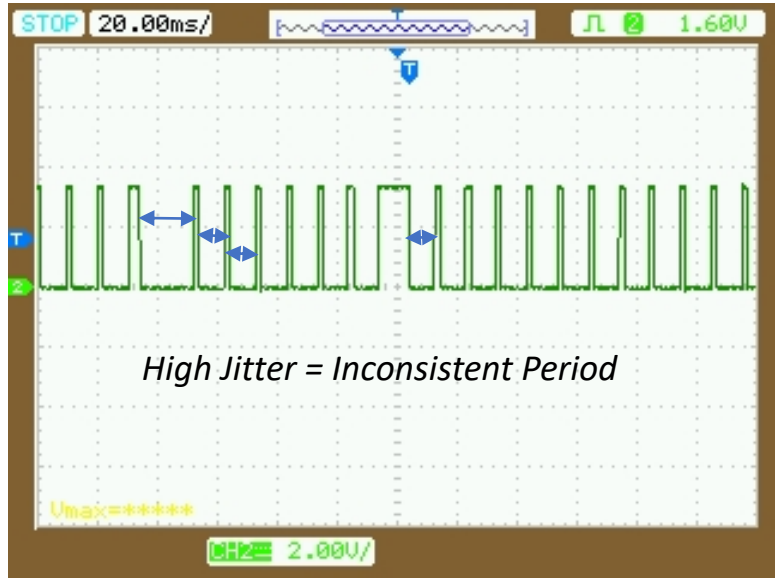
Integrate the LED Library

- Integrate the LED library you created in Lab1 into the BSP
- Add the LED .c file into `src` and the LED library .h file into `inc` folders of the BSP
- Add an #include “yourLEDLibrary.h” in `BSP.h`
- Add a call to the LEDinit function in your library to the `_initBoard` function in the `BSP.c` file.

RTOS

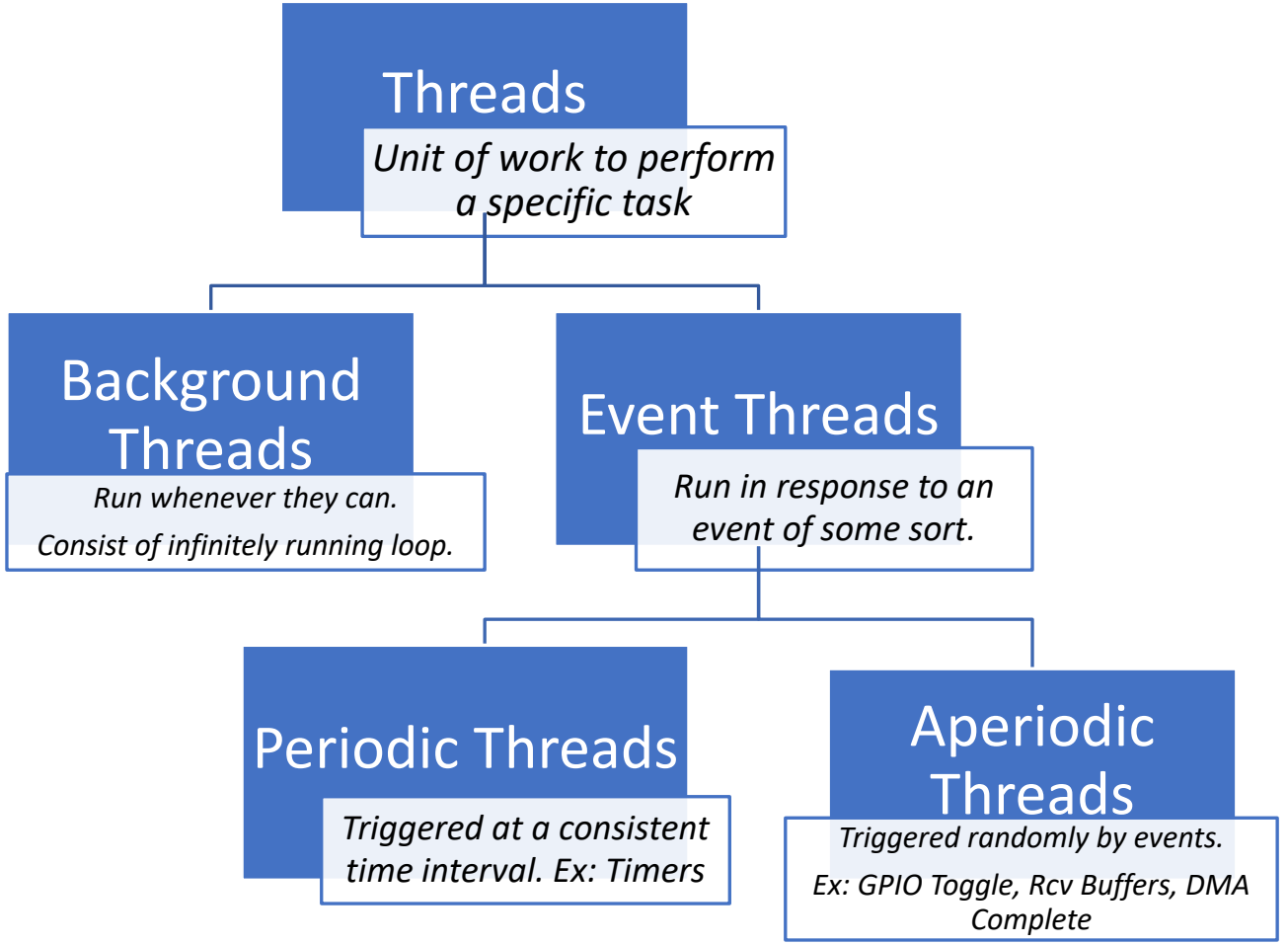
- Why an OS? Multi-**tasking** and resource sharing
- **Real-Time** OS: OS where tasks have deadlines. If the deadline is missed the plant blows up.
- We want to reduce **latency**
- We want to reduce **Jitter**





RTOS: Threads

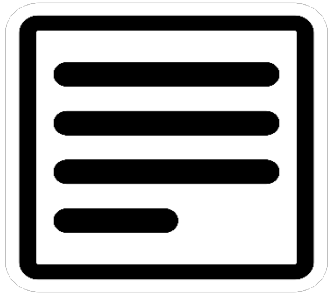
- We have Multiple tasks one CPU -> Time-multiplexed system
- Tasks -> **Threads** . We run one thread at the time.
- The **scheduler** will decide which task gets to run at the moment.



RTOS: Threads

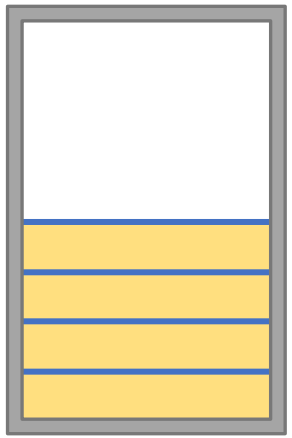
- Implementing threads?
- The components of a thread

Code



CPU Registers/Context

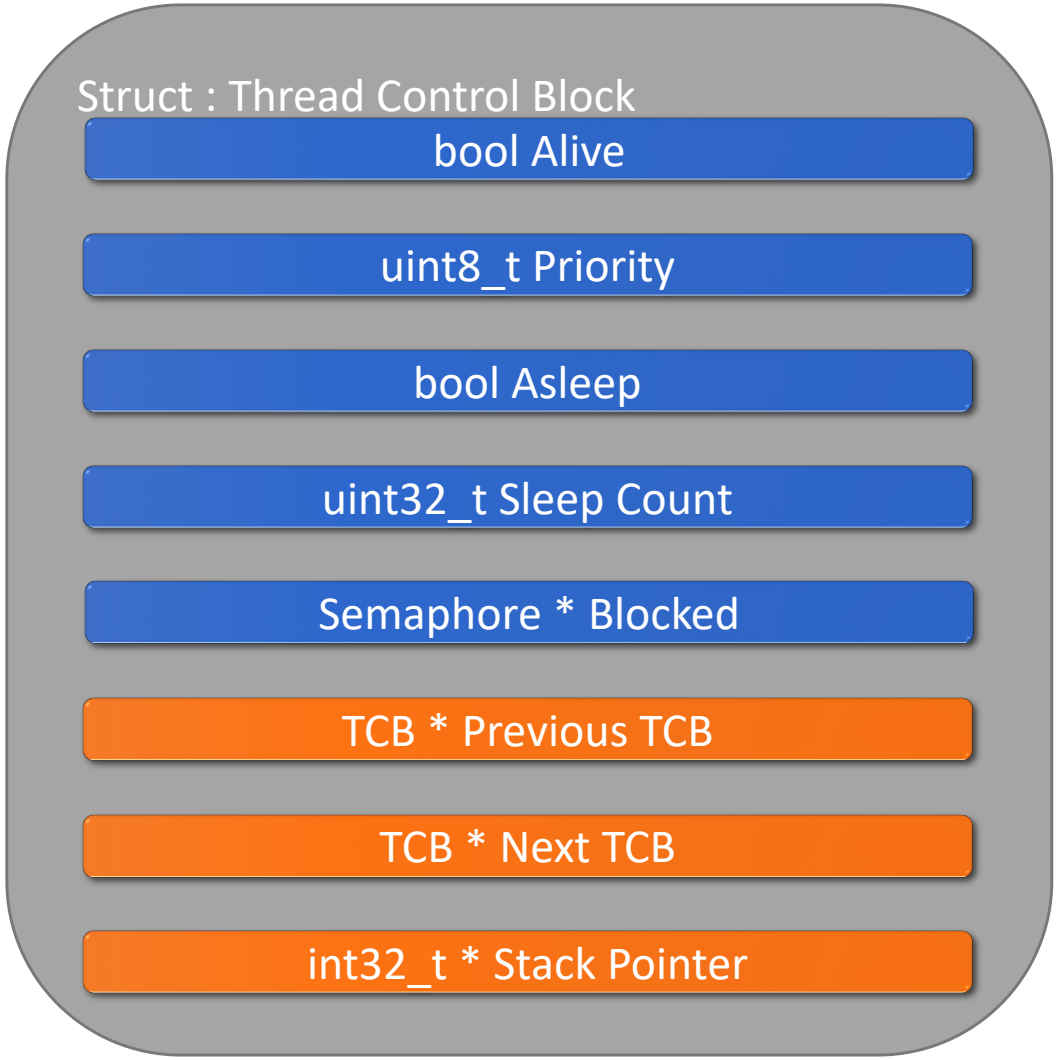
All of the Thread's memory including its stack



- We want to be able to stop a thread. Leave it aside. And then pick it up later. What is the minimum components we need for this?

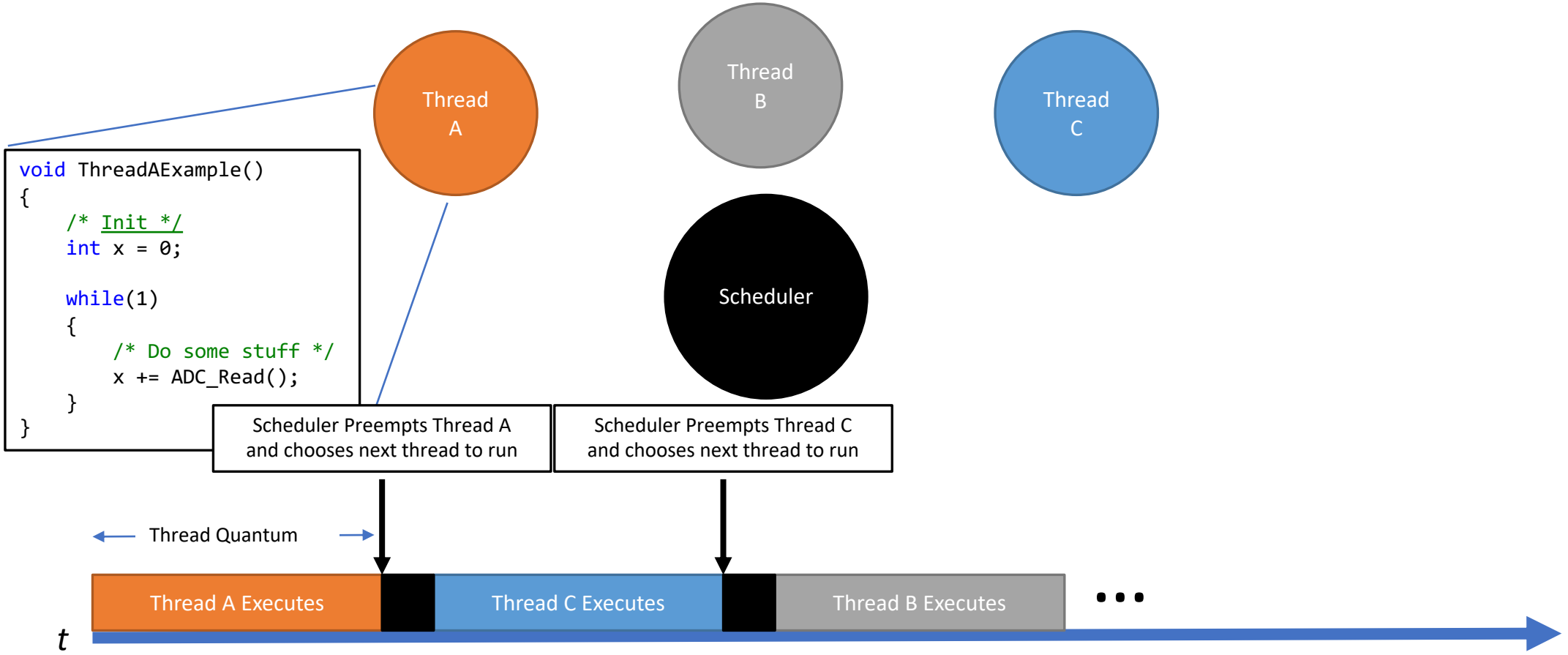
RTOS: Thread-control-block (TCB)

- Thread-control-block (TCB): a data-structure for storing the thread information. This is what you need to develop in Part1.
- For Lab2 this looks like the following.
- Create a `struct` and `typedef` it for easy access.



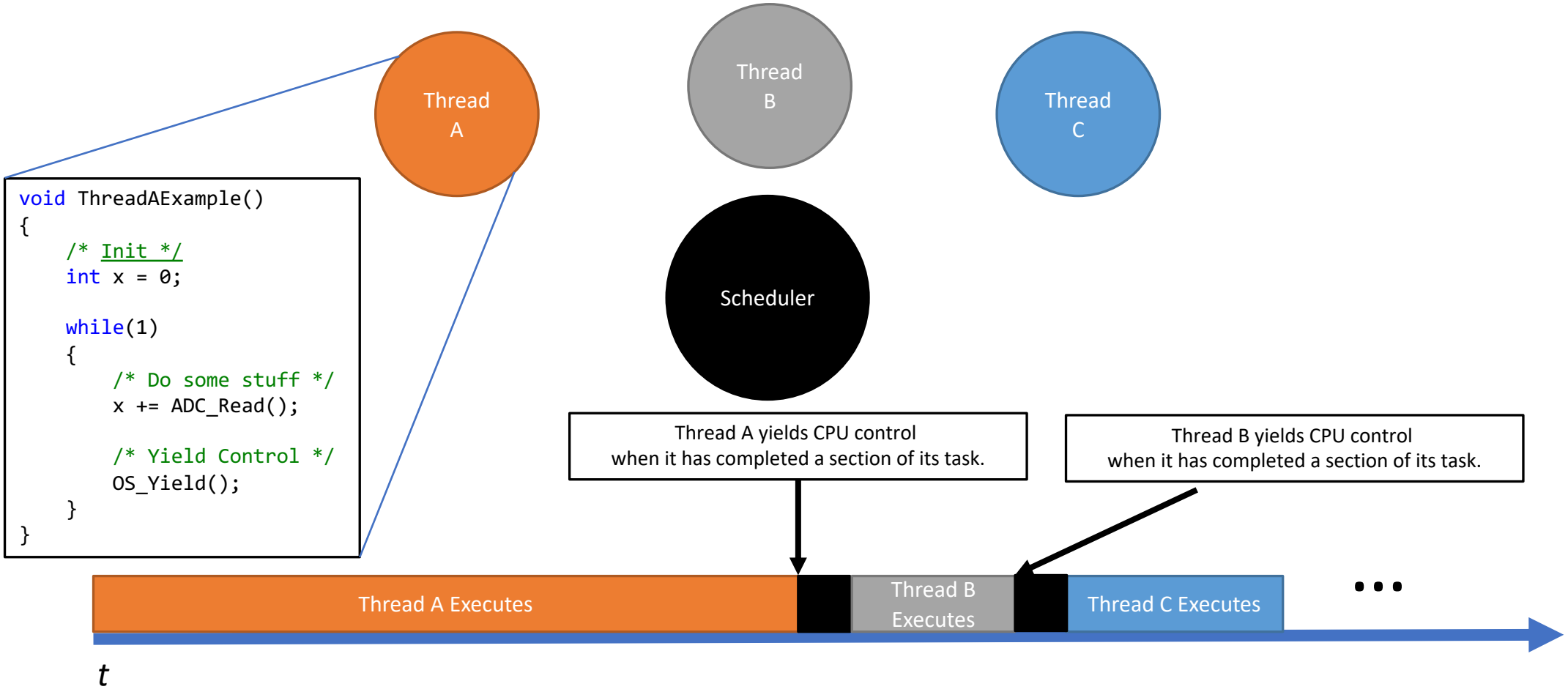
RTOS: Scheduler

- Preemptive Thread Scheduling where the scheduler jumps in and stops one thread and replaces it with another (preempts the thread).



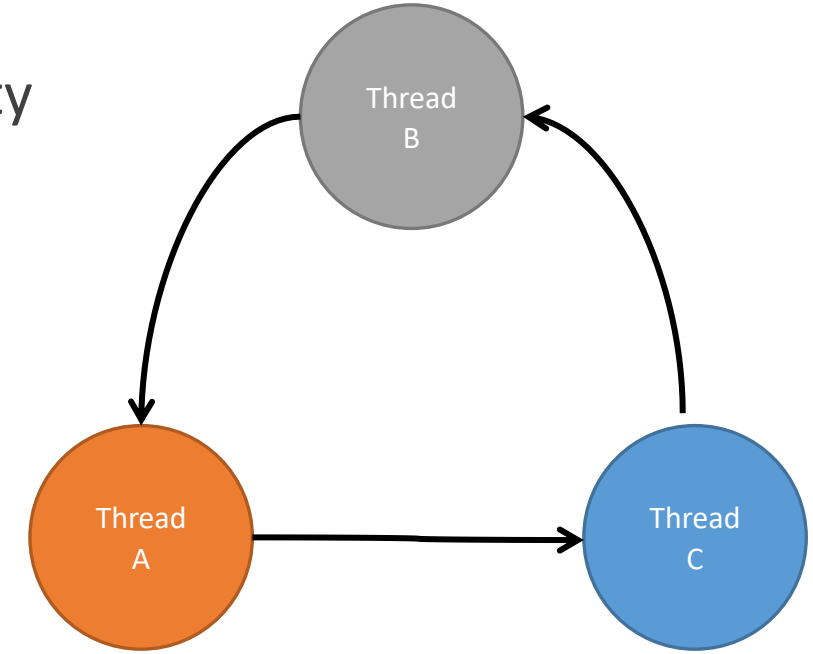
RTOS: Scheduler

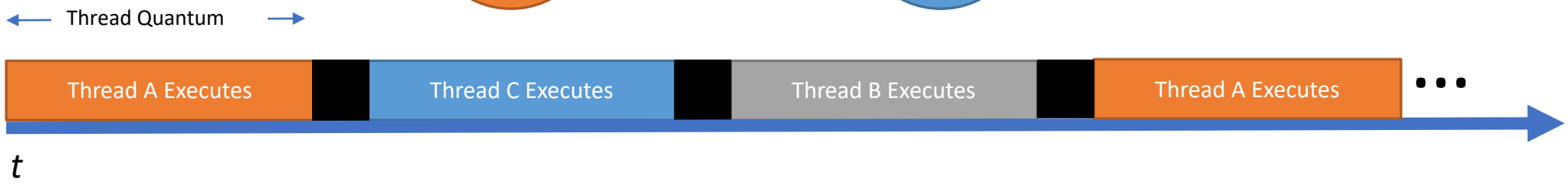
- Cooperative Thread Scheduling where the threads themselves yield the CPU to one another



RTOS: Scheduler

- We are looking for a **round-robin** scheduler in Lab2. Goes through a list and executes each thread for a given time and then moves to the next one.
- Guarantees fairness.
- No concept of priority





RTOS: Context Switching

- How to switch from one thread to another given that we have their two TCBs?
- Save the important stuff, switch to the new thread, later pick up the important stuff.
- Important stuff: The set of all registers a thread needs saved in order to correctly resume execution at a later time is known as it's **execution context.**
- For the ARM Cortex M4 (when not using floating point), a thread's context consists of the **Status Register, Program Counter, Link Register and Registers R0-R12**.
- In order to use multiple threads on a single CPU, we must be able to save the state if one thread and restore the state of another, this is known as a **context switch.**
- We need to interrupt the current thread's execution to perform a context switch either voluntarily (cooperative) or nonvoluntarily (preemptive).

RTOS: Context Switching

- In Lab2 we use the following two interrupts for scheduling and context switching

Systick Interrupt

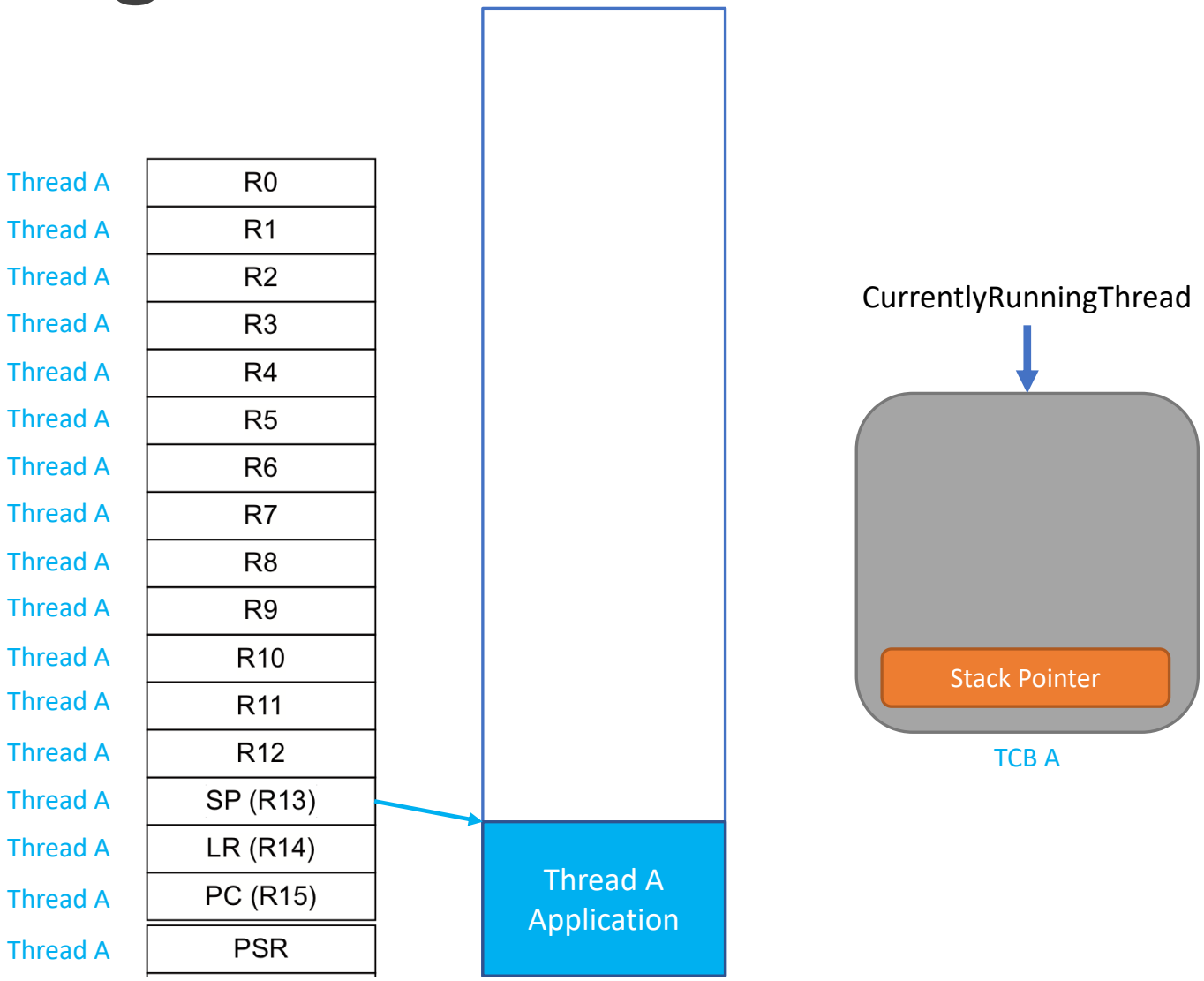
- Caused by Systick Timer reaching zero
- For our OS, it will interrupt every 1 millisecond.
- Increments a system time variable.
- Triggers the context switch interrupt.
- In the future, it will have more responsibilities such as modifying TCBs and running Periodic Events.

PendSV Interrupt

- Request for system-level service.
- Specifically created by ARM for performing context switches in an OS setting.
- Will be used for our context switch.
- Since we will be modifying CPU register, the ISR will need to be written primarily in Assembly.

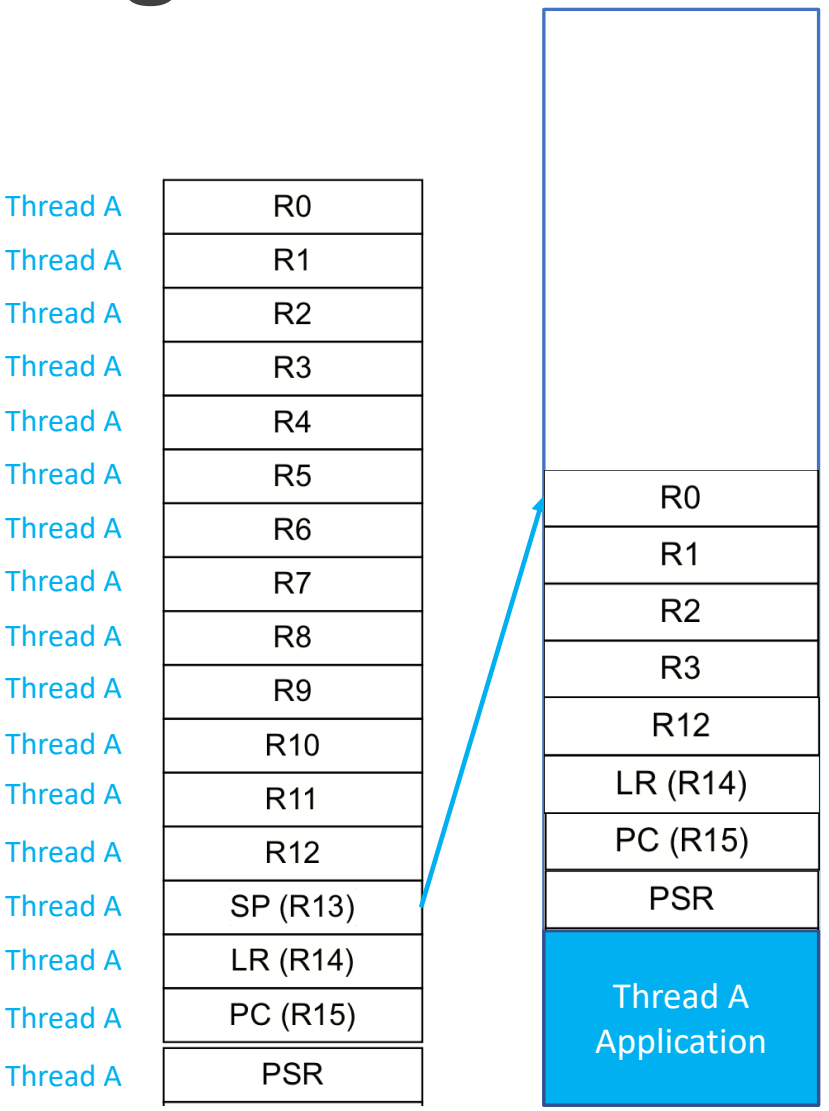
RTOS: Context Switching

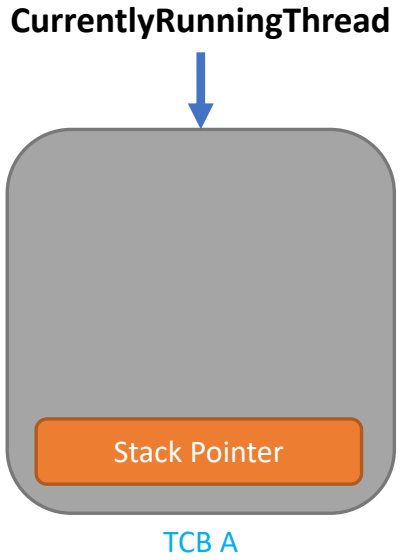
- Context-switch done by



RTOS: Context Switching

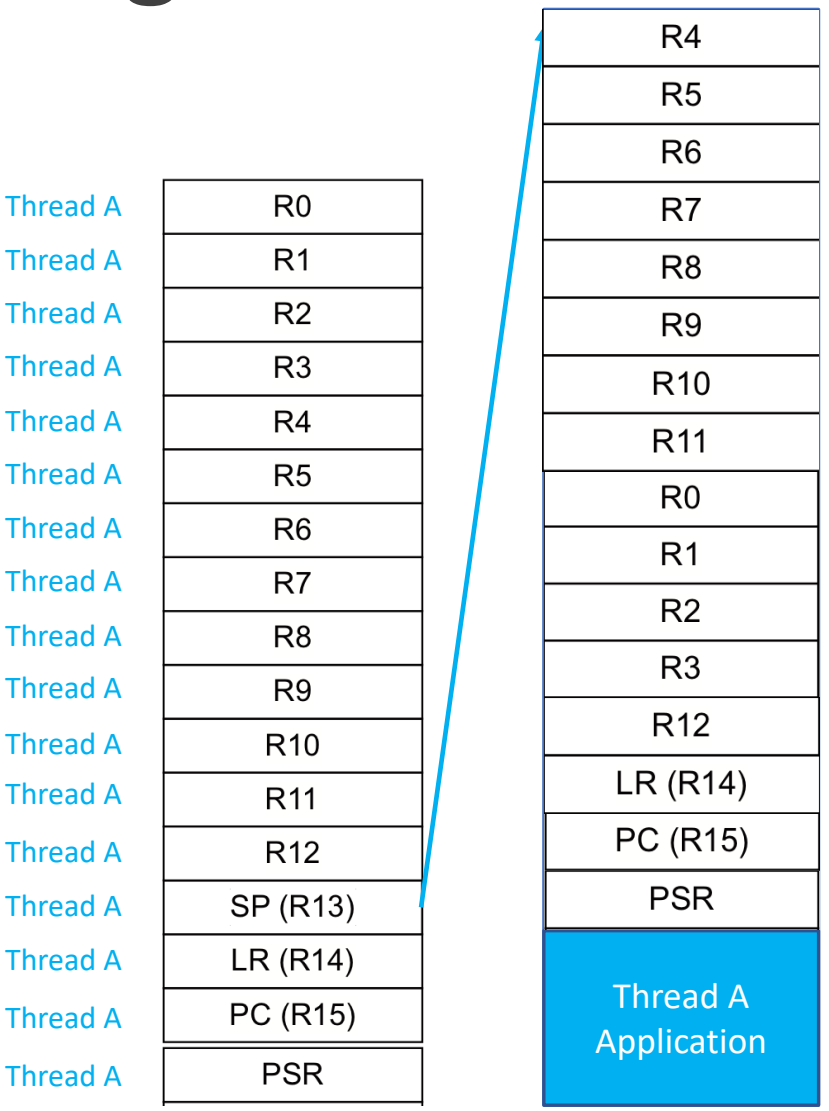
- Context-switch done by:
  - Calling PendSV through software-triggered interrupt (ICSR) from the Systick handler. (pushes R0-R3, R12-R15, PSR to the stack automatically)

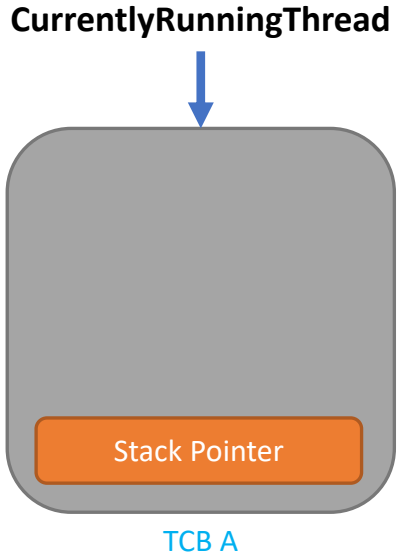




RTOS: Context Switching

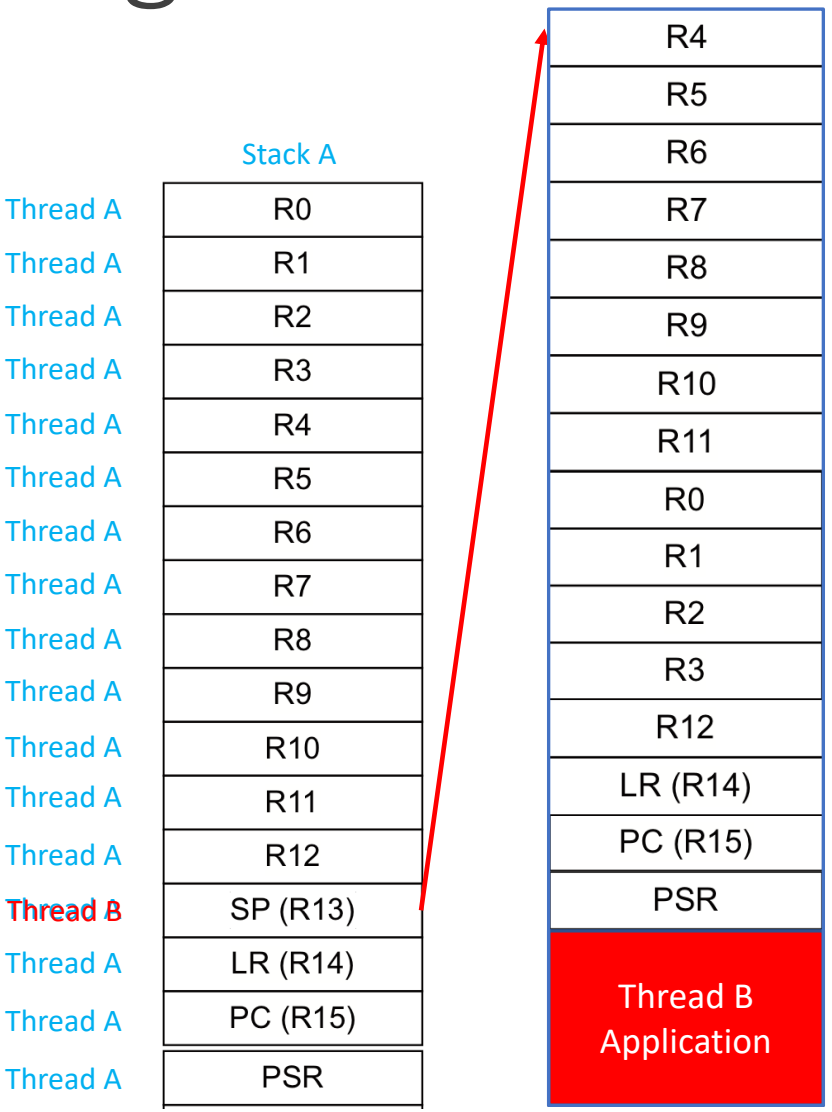
- Context-switch done by:
  - Calling PendSV through software-triggered interrupt (ICSR) from the Systick handler. (pushes R0-R3, R12-R15, PSR to the stack automatically)
  - PendSV handler should push the rest of the registers (R4-R11) to stack.

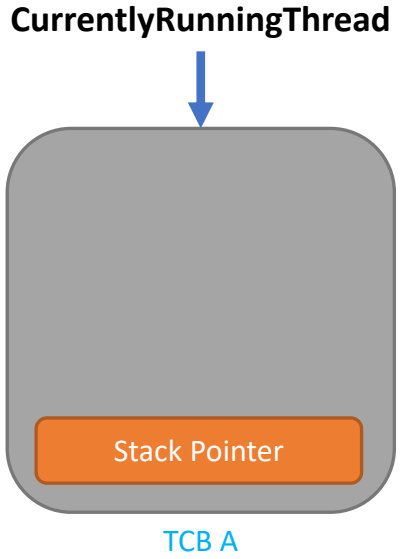




RTOS: Context Switching

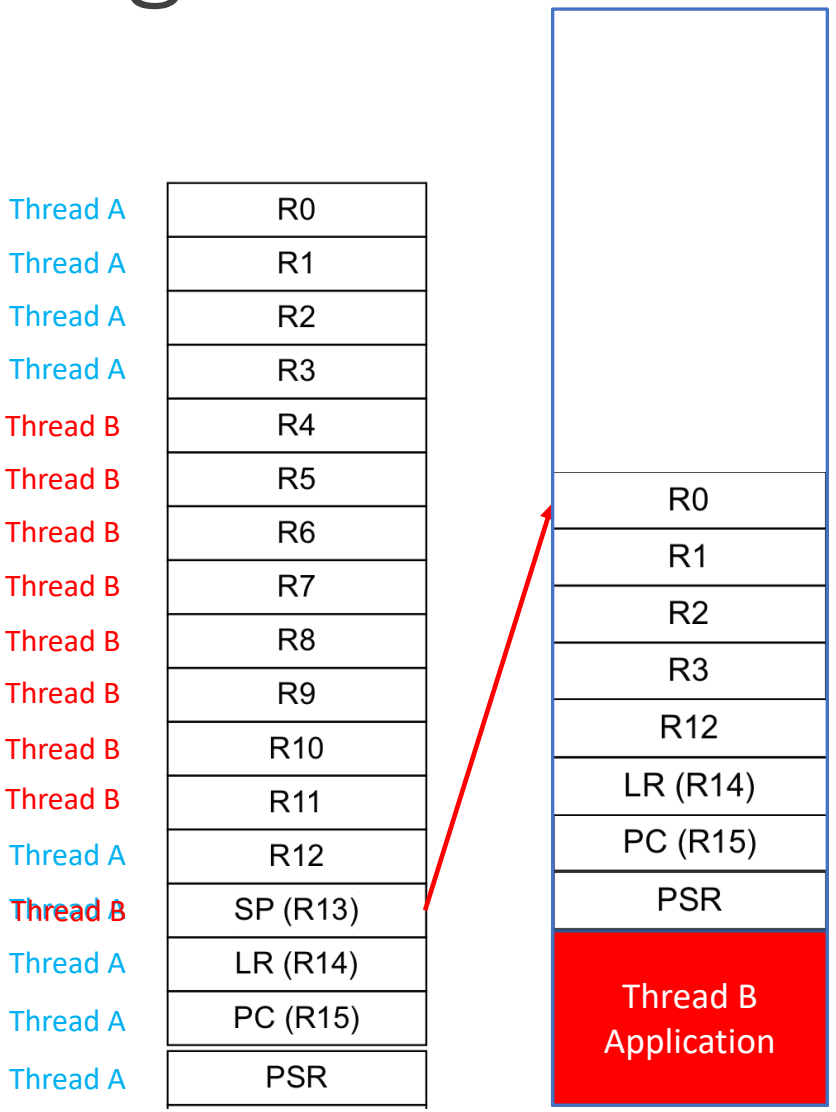
- Context-switch done by:
  - Calling PendSV through software-triggered interrupt (ICSR) from the Systick handler. (pushes R0-R3, R12-R15, PSR to the stack automatically)
  - PendSV handler should push the rest of the registers (R4-R11) to stack. Now we are done with saving Thread-A's context.
  - Call the scheduler to update CurrentlyRunningThread.
  - Load the new-thread's (Thread-B's) stack pointer. This should point to the top of Thread-B's context.

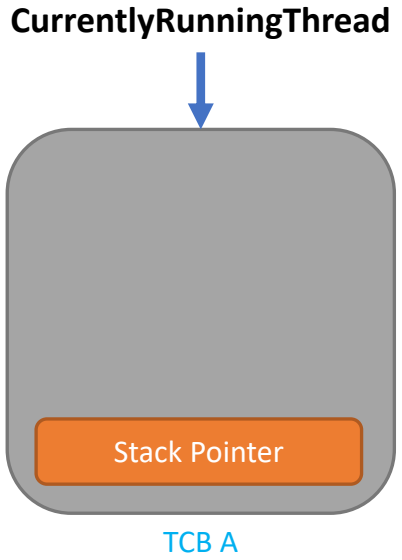




RTOS: Context Switching

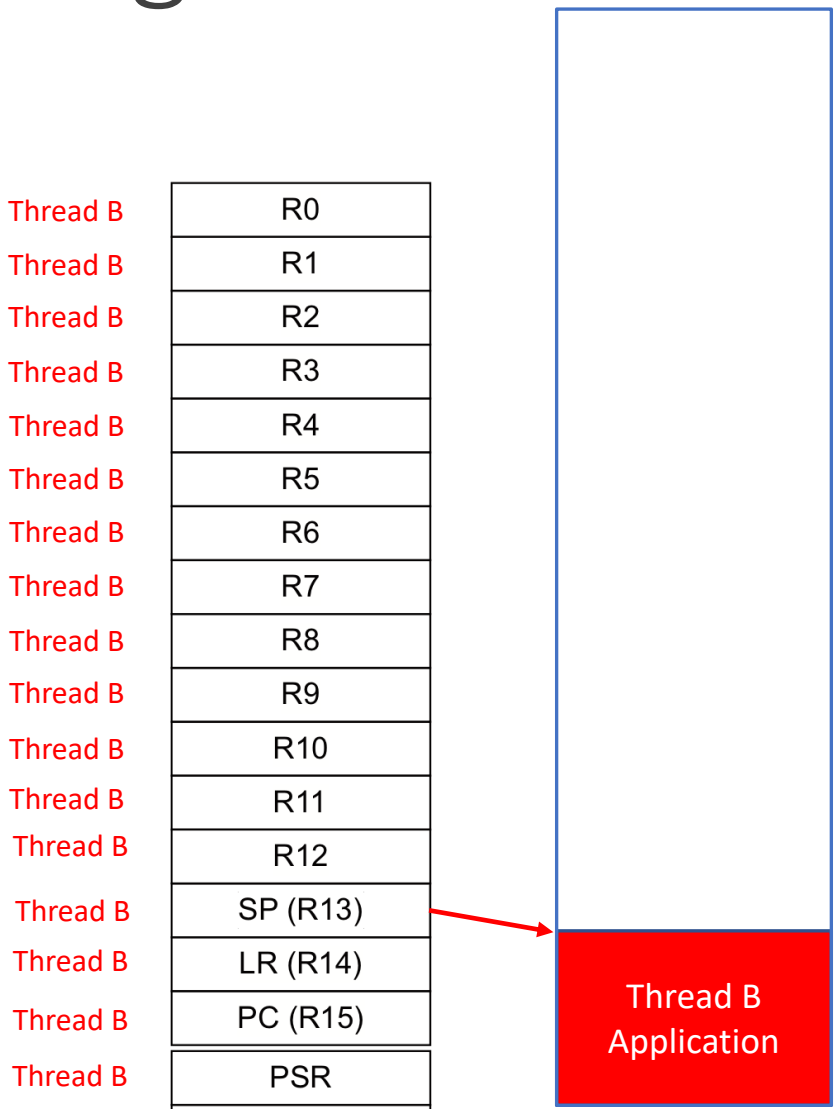
- Context-switch done by:
  - Calling PendSV through software-triggered interrupt (ICSR) from the Systick handler. (pushes R0-R3, R12-R15, PSR to the stack automatically)
  - PendSV handler should push the rest of the registers (R4-R11) to stack. Now we are done with saving Thread-A's context.
  - Call the scheduler to update CurrentlyRunningThread.
  - Load the new-thread's (Thread-B's) stack pointer. This should point to the top of Thread-B's context.
  - Pop Thread-B's R4-R11 (they were pushed during the context-save of Thread-B).

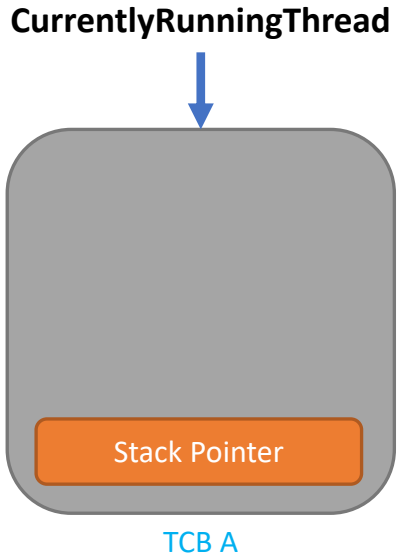




RTOS: Context Switching

- Context-switch done by:
  - Calling PendSV through software-triggered interrupt (ICSR) from the Systick handler. (pushes R0-R3, R12-R15, PSR to the stack automatically)
  - PendSV handler should push the rest of the registers (R4-R11) to stack. Now we are done with saving Thread-A's context.
  - Call the scheduler to update CurrentlyRunningThread.
  - Load the new-thread's (Thread-B's) stack pointer. This should point to the top of Thread-B's context.
  - Pop Thread-B's R4-R11 (the ones we pushed during context save).
  - Returning from PendSV will pop the remaining registers for Thread-B. Thread-B should have all of its context now.





RTOS: Context Switching

- ASM skeleton will be provided to you:

```
; Functions Defined
.def G8RTOS_Start, PendSV_Handler

; Dependencies
.ref CurrentlyRunningThread, G8RTOS_Scheduler

.thumb; Set to thumb mode
.align 2; Align by 2 bytes (thumb mode uses
allignment by 2 or 4)
.text; Text section

; Need to have the address defined in file
; (label needs to be close enough to asm code to
be reached with PC relative addressing)
RunningPtr: .field CurrentlyRunningThread, 32
```

```
; PendSV_Handler
; - Performs a context switch in G8RTOS
; - Saves remaining registers into thread stack
;- Saves current stack pointer to tcb
;- Calls G8RTOS_Scheduler to get new tcb
;- Set stack pointer to new stack pointer from new tcb
;- Pops registers from thread stack
PendSV_Handler:

.asmfunc
;Implement this
.endasmfunc

; end of the asm file
.align
.end
```

RTOS: Critical Section

- In multi-threading, multiple threads need access to the same device or shared memory
- This causes threads to fight over the resource resulting in errors
  - Ex. 1. Two threads writing to an LCD, the second thread interrupting the first
  - Ex. 2. Two threads initializing a port differently. The order the threads are run in will determine the final state of the port
- Critical section: the part of the program where shared memory is accessed
- Consequences of not handling critical sections are race conditions and flawed results
- The above context switching code especially the saving and loading of registers should be in a critical section.

RTOS: Critical Section

- The Lab2 skeleton includes some assembly code and its relative C declarations for entering and exiting critical sections.

```
; Functions Defined
.def StartCriticalSection, EndCriticalSection

.thumb; Set to thumb mode
.align 2; Align by 2 bytes (thumb mode uses
allignment by 2 or 4)
.text; Text section
```

```
; Starts a critical section
; - Saves the state of the current PRIMASK (I-bit)
; - Disables interrupts
; Returns: The current PRIMASK State
StartCriticalSection:
.asmfunc

MRS R0, PRIMASK; Save PRIMASK to R0 (Return Register)
CPSID I; Disable Interrupts
BX LR; Return

.endasmfunc
```

```
; Ends a critical Section
; - Restores the state of the PRIMASK given an input
; Param R0: PRIMASK State to update
EndCriticalSection:
.asmfunc

MSR PRIMASK, R0; Save R0 (Param) to PRIMASK
BX LR; Return

.endasmfunc
```

RTOS: Add-Thread Function

- Before launching our operating system, we must initialize our threads and put them in the scheduler.
- In Lab2, you will be creating a function `Add_Thread` that will initialize a thread and place it in the scheduler given a Void-Void Function pointer.
- The first step to adding a thread is to check if there are any available empty threads in the operating system.
- In order to keep our thread addition fast and deterministic, our TCBs and thread stacks will be statically allocated. Therefore these is a limited number of TCBs and accompanying stacks available.
- The operating system keeps track of the number of threads currently in the scheduler with a `NumThreads` Variable. To check if we are at our limit, we will check if `NumThreads` is equal to our limit. If it is, the function will return an error code instead of adding a thread.
- If there is available space in the thread list the function will add a new TCB to the list. Set the next and prev pointers.

RTOS: Add-Thread Function

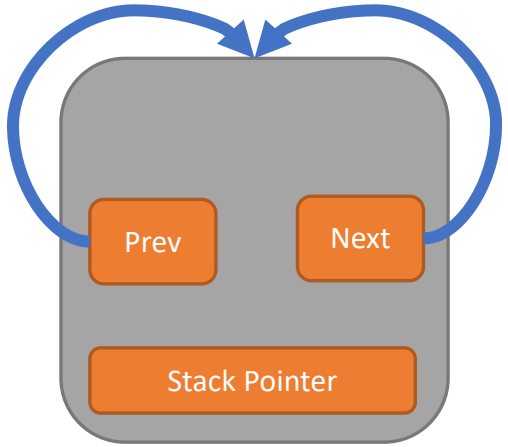
- The `threadStacks` is a static array that holds all the stack areas in memory.

```
/* Thread Stacks */
static int32_t threadStacks[MAX_THREADS][STACKSIZE];
```

- `STACKSIZE` is set to 1024.
- When initializing a thread *we have to note that when the first context-switch arrives and picks up our thread it expects a full execution context (R0-R15, PC, PSR). Simply focus on how you implemented the context-switch and what it expects from the stack.* For the context switch described earlier the thread initialization looks like this:
  - R0-R15 to dummy values.
  - PC to the thread's function pointer (the void-void function pointer that the Add_Thread function receives)
  - PSR to some default value that has the thumb-bit set.
  - The stack-pointer in the TCB struct should point to the top of all of these (`&threadStacks[threadnum][STACKSIZE-16]` because the stack grows from high addresses to low ones).

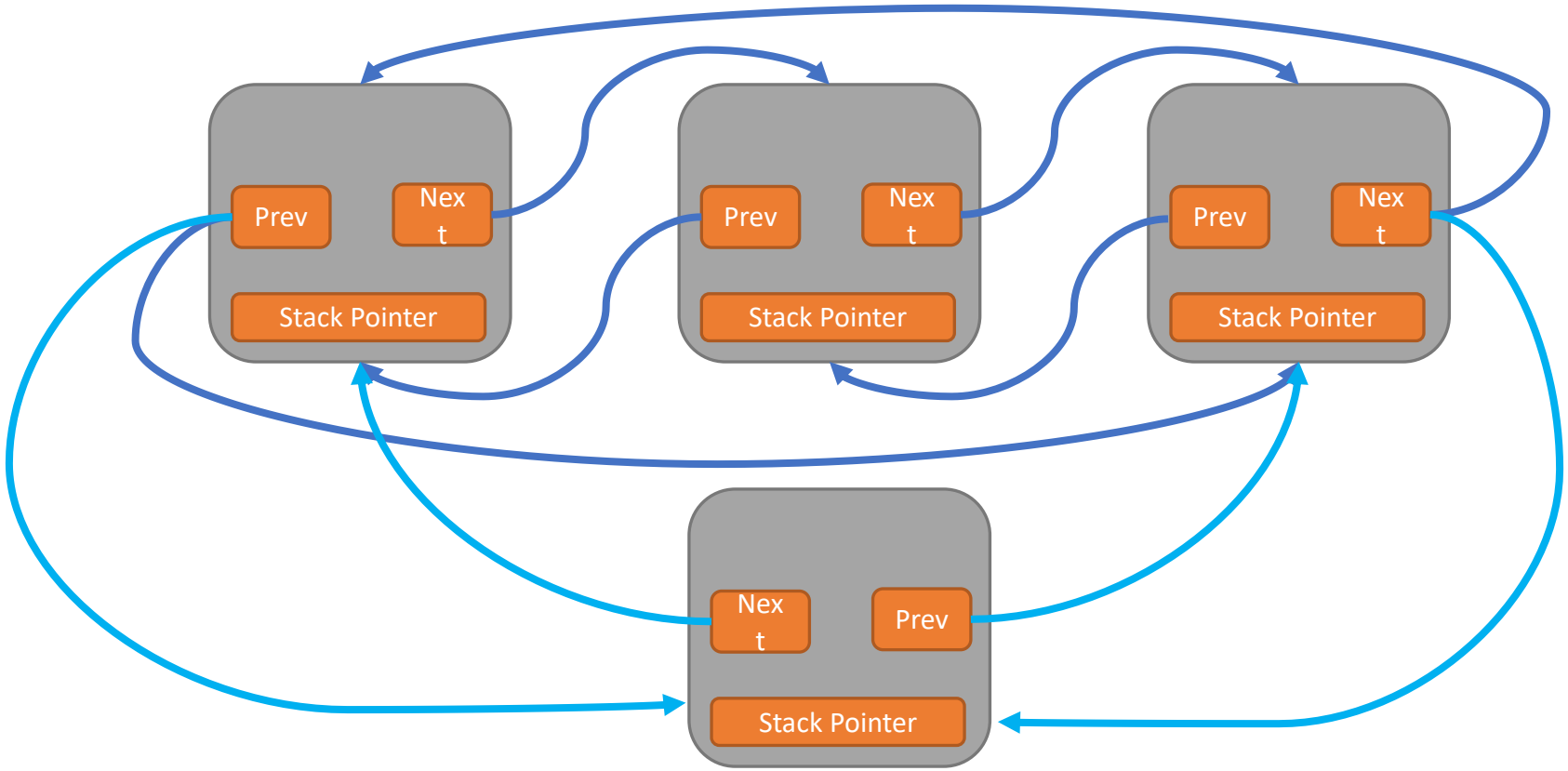
RTOS: Add-Thread

First thread to be added



TCB

Multiple threads in a round-robin scheduler



TCB

RTOS final steps

- The assembly context-switch function calls the C scheduler function which picks the next thread to run. In Lab2 this function simply picks the next thread using the next-pointer in the TCB.
- When you have finished configuring G8RTOS, your OS is now 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 said thread into the CPU, and enable interrupts. This task will be split into two functions:
  - `G8RTOS_Launch:`

  This C function will be called from main and will accomplish the following:
    - Set CurrentlyRunningThread
    - Initialize SysTick
    - Set the priorities of PendSV and SysTick to the lowest priority
    - Call G8RTOS_Start
  - `G8RTOS_Start:`

  This assembly function will accomplish the following:
    - Loads the currently running thread's context into the CPU
    - Enable interrupts

RTOS final steps

- The assembly context-switch function calls the C scheduler function which picks the next thread to run. In Lab2 this function simply picks the next thread using the next-pointer in the TCB.
- When you have finished configuring G8RTOS, your OS is now 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 said thread into the CPU, and enable interrupts. This task will be split into two functions:
  - `G8RTOS_Launch:`

  This C function will be called from main and will accomplish the following:
    - Set CurrentlyRunningThread
    - Initialize SysTick
    - Set the priorities of PendSV and SysTick to the lowest priority
    - Call G8RTOS_Start
  - `G8RTOS_Start:`

  This assembly function will accomplish the following:
    - Loads the currently running thread's context into the CPU
    - Enable interrupts
    - Jump to first thread somehow

You can test all of this by using a couple of threads to increment counters or play with LEDs.

RTOS: Semaphores

- Historically semaphores were first turn signals
- Generally speaking are indicators of intent
- In computer context, a semaphore is a type of flag
- Two uses of semaphores
  - Mutual exclusion – Only one thread should be accessing a shared resource at a time
  - Synchronization – Threads A should start doing something only when Thread B signals it to do so
- Make flag more informative: counter
- Mutex vs. Semaphore
  - A mutex is a lock with only one state – locked/unlocked. A mutex can only be released by the thread that acquired it
    - Similar to a binary semaphore, but cannot be signaled by another thread i.e. LCD
  - A semaphore can have a value greater than 1 and can be signaled by any other thread
    - i.e. 4K buffer split into 4 1K buffers

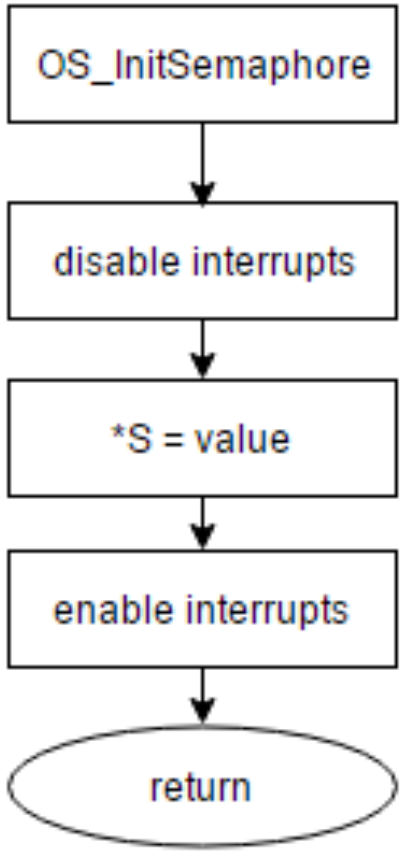
RTOS: Semaphores

- Semaphore is a counter with three functions:
  - `G8RTOS_InitSemaphore(*SEMAPHORE_NAME, SEMAPHORE_VALUE)`
  - `G8RTOS_WaitSemaphore(*SEMAPHORE_NAME)`
  - `G8RTOS_SignalSemaphore(*SEMAPHORE_NAME)`

- In G8RTOS_Semaphores.h, you should create a new type of:
  - `typedef int32_t semaphore_t`

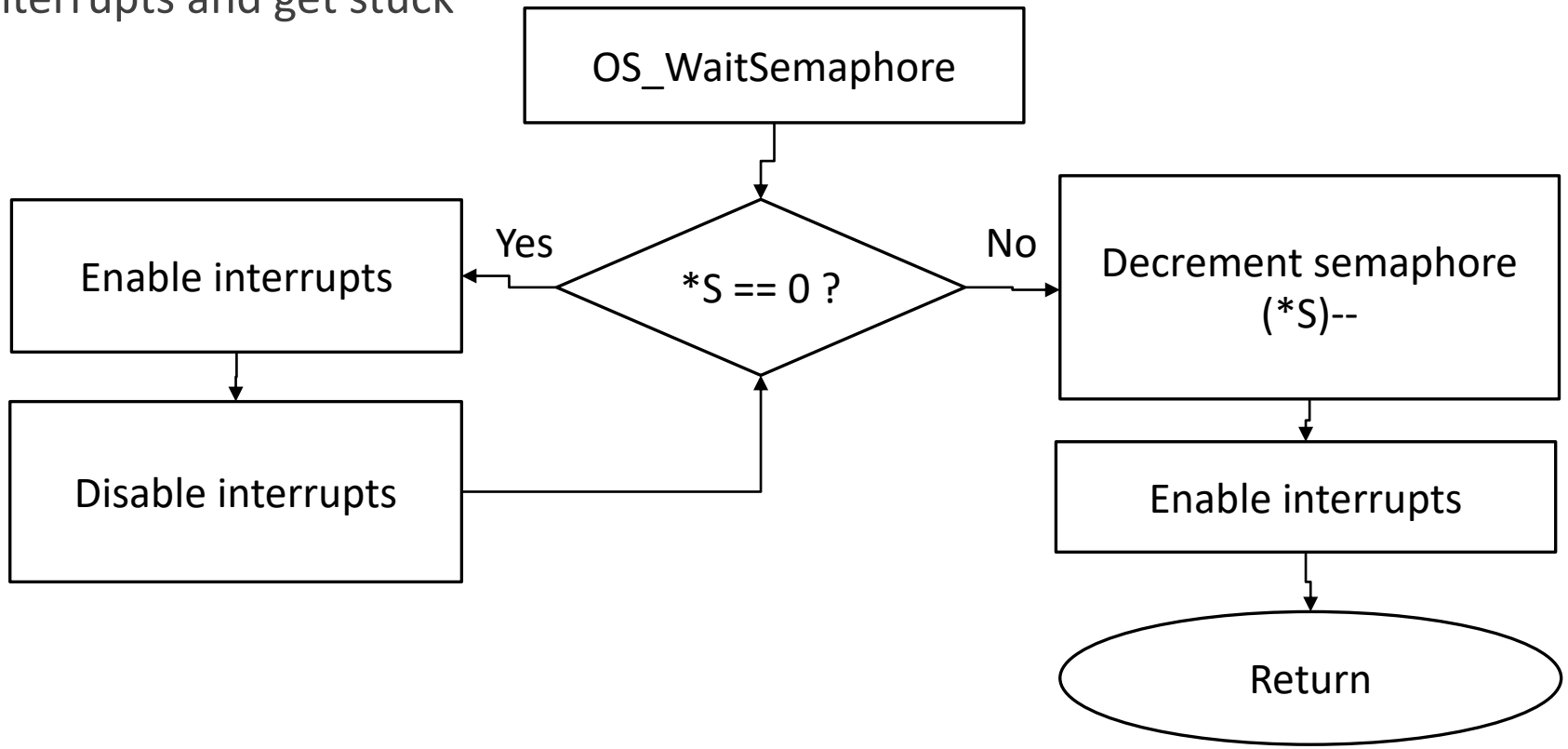
`G8RTOS_InitSemaphore(semaphore_t *S, int32_t value)`

- Enter a critical section and initialize the semaphore in it.
- Pass the pointer to the semaphore around so that its value can be modified



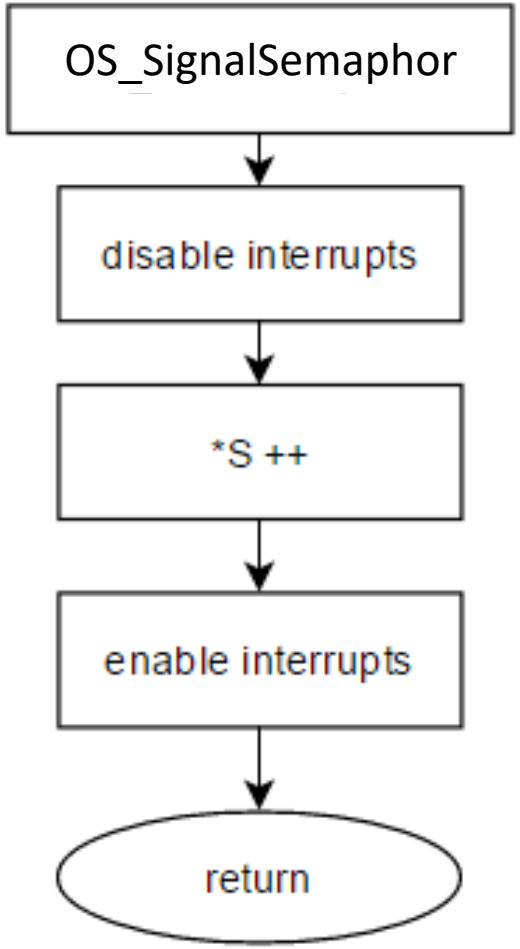
G8RTOS_WaitSemaphore(semaphore_t *S)

- Wait for semaphore to become non-zero.
- While waiting enter and exit the critical section so that we won't miss all the interrupts and get stuck in the loop forever.



G8RTOS_SignalSemaphore(semaphore_t *S)

- Enter critical section.
- Increment the semaphore.
- Make sure to increment the semaphore itself not its pointer (*S)++;



Mutual Exclusion

- An example use of semaphores for mutually exclusive tasks.

```
G8RTOS_InitSemaphore(&LCDReady,1);

void Task0{

    G8RTOS_WaitSemaphore(&LCDReady);
    LCD_Write("Coming to you live from task 0");
    G8RTOS_SignalSemaphore(&LCDReady);
}

void Task1{

    G8RTOS_WaitSemaphore(&LCDReady);
    LCD_Write("Coming to you live from task 1");
    G8RTOS_SignalSemaphore(&LCDReady);
}
```

Synchronization

- Enter critical section.
- Increment the semaphore.
- Make sure to increment the semaphore it self not its pointer (*S)++;

```
uint32_t ADCData;
G8RTOS_InitSemaphore(&ADCReady,0);

void TaskADC{

    ADCData = ADC_Read();
    G8RTOS_SignalSemaphore(&ADCReady);
}

void TaskLCD{

    G8RTOS_WaitSemaphore(&ADCReady);
    LCD_Write(ADCData);
}
```

RTOS: Threads

- Implement the above semaphore functions in Lab2. Now we can add threads to the RTOS and synchronize them using semaphore.
- We will use two semaphore to make sure the access to the LED I2C interface and the sensors I2C interface is exclusive.
- You developed functions for writing to LEDs in Lab1. They should be included in the BSP.
- Functions for reading sensor data are available in the BSP.

```
/* reading X location from accelerometer. Initialization is done
in the BSP initialization function*/
#include <BSP.h>
int16_t accelerometerX;
while(bmi160_read_accel_x(&accelerometerX));
```

```
/* reading Light sensor data. */
uint16_t lightData;
while(!sensorOpt3001Read(&lightData));
```

```
/* reading Light sensor data. */
int16_t gyroZ;
while(bmi160_read_gyro_z(&gyroZ));
```

RTOS: Threads

- You need the following three threads:
  - Thread 0:
    - waits for the sensor I2C semaphore.
    - Reads accelerometer's x-axis and saves it.
    - Releases the sensor I2C semaphore.
    - Waits for the LED I2C semaphore. Once available output data to red LEDs as shown in the Lab manual. And then releases it.
  - Thread 1:
    - waits for the I2C semaphore.
    - Reads the light sensor and save to local variable.
    - Releases the I2C semaphore.
    - Wait for the I2C semaphore. Once available output data to green LEDs as shown in the Lab manual.
  - Thread 2:
    - waits for the I2C semaphore.
    - Reads from the z-axis of the gyro and save into local variable
    - Releases the I2C semaphore.
    - Wait for the I2C semaphore. Once available output data to blue LEDs as shown in the Lab manual. Don't forget to release the semaphore by signaling it (signal function).

RTOS: Threads

- Write all the threads in a `Threads.c` file with a header file `Threads.h` as well.
- Initialize the RTOS. Add the threads to the RTOS.
- Start the RTOS using the launch and start functions.