Pre-emptive scheduling with RTX

COE718: Embedded Systems Design
Lab 3b

1. Objectives

This lab introduces the students about developing RTX-based multithreaded applications for ARM Cortex-M3/M4 processors. The students will learn how to schedule multithreaded applications by employing preemptive, and non-preemptive scheduling supported by µVision, RTX operating system and CMSIS libraries.

2. Implementing Different Scheduling Algorithms

Follow the steps from Lab3a manual to setup project for this Lab (i.e. Lab3b). Use the same starter code given for Lab 3a. To implement pre-emptive/non-pre-emptive scheduling techniques, you need to adjust the file '*RTX_Conf_CM*.c'. Specifically in the Configuration Wizard, the option System Configuration >> Round- Robin Thread switching must be disabled. However, make sure that the systick timers are enabled.

Exercise 1: Setting Priority

Access the Thread.c file and amend the following line as shown below:

```
osThreadDef(Thread1, osPriorityNormal, 1, 0); to
osThreadDef(Thread1, osPriorityAboveNormal, 1, 0);
```

Compile the program and return to Debug mode. Run the program and open the Event Viewer window.

Question 1: *What do you notice?*
By setting the priority of the Thread2 to a higher priority as compared to Thread1, a **pre-emptive** (interruptible) scheduling technique is created where the higher priority thread will execute to completion first. Since Thread1 was created first, it is also expected to run first. However, Thread1 will never be executed due to its "Normal" priority setting (in comparison to Thread2 having "AboveNormal" priority) and the fact that Thread2 executes infinitely. Conversely, if the code was programmed such that the Thread2 terminates after a finite time (when its workload completes), then Thread1 would thereafter be able to execute. It is recommended that the CMSIS-RTOS API Thread Management and *osPriority* enumerations be consulted to familiarize yourself with the available thread-based priority options during coding.

Note that it is also possible to create a **non-preemptive** scheduling algorithm by assigning appropriate priority levels to the tasks.

Exercise 2: Pre-emptive Scheduling

Access the 'Thread.c' file again. Change the Thread1 and Thread2 codes as follows.

```
void Thread2 (void const *argument) {
       for (;;){  // Infinite loop – runs while thread2 runs.
              countb++;   // Increment global variable countb indefinitely
              osThreadYield();
       }                        // suspend thread
}


void Thread1 (void const *argument) {
       for (;;){  // Infinite loop – runs while Thread1 runs.
              counta++;   // Increment global variable counta indefinitely
              osThreadYield();
       }     // suspend thread
}
```

Moreover, make sure to change:
`osThreadDef(Thread1, osPriorityAboveNormal, 1, 0);` back to
`osThreadDef(Thread1, osPriorityNormal, 1, 0);`

Recompile the files and Enter the Debug mode. Open a Watch window to track the *counta* and *countb* variables, along with the Event Viewer. Reset the program and click RUN.

Question 2: *How does the execution of the code using `osThreadYield()` differ from round-robin scheduling (studied in lab3a)?*
If you were successful, you will observe short execution time slices per thread in the Event Viewer, where it appears as if the threads are running as round-robin (after several msecs). With the changes made to the program, each thread should simply increment their counter by one and pass control to the next thread of equal or greater priority using `osThreadYield()`. Specifically, you should observe that on average a single thread runs for 2.52µs before passing control to the next thread (which is the equivalent time spent entering the thread, incrementing the counter, and passing control).

What is the utilization time of the processor?
Check the Idle Demon (refer Lab 3a manual for idle demon) variable and the task using performance-based tools.

Exercise 3: Stop the previous program and exit Debug mode to gain access to the 'Thread.c' file. Remove the *`osThreadYield()`* functions in the last exercise you implemented. Change the Thread1 and Thread2 functions code as given below.

```
void Thread2 (void const *argument) {
  for(;;) {
       countb++;
       osDelay(1);
  }
}

void Thread1 (void const *argument) {
 for(;;) {
       counta++;
       osDelay(2);
    }
}
```

Recompile the files and enter Debug mode. Setup the Watch 1 window with the variables counta, countb, and countIDLE. RUN the program

Question 3: *Assess the Watch window and note the difference between the execution of this code and the previous code. Use the Performance Analyzer and Event Viewer to verify your findings. What is the utilization time of the CPU?*

2. Lab Assignment

This lab is due in **Week 7** at the beginning of your lab session. The following outlines the specifications for two different scheduling applications (Questions 1, and 2). You must create TWO versions for each application: i) an analysis version and ii) a demo version. The analysis will be used for debug mode to analyze the performance of your applications for your report and **must not** include any LED or LCD code. The demo version will include LCD and LED functions for your demo (Take care of stack size to accommodate the LED/LCD code in *RTX_Conf_CM.h*). You will be marked on both versions of the code; however, you are only required to submit the analysis version for grading.

1. Table I provides a list of pre-emptive tasks, with their function and priority listed. Note: The lower the number in the Priority column, the higher the priority. Write the **pre-emptive** code for a scheduling algorithm which invokes the tasks and functionalities in Table I based on their priority level (i.e., Task C should finish computing first etc.). Each task should print their final result to stdout (using printf or the watch window). For the demo version only, use the LEDs and the LCD to indicate the threads that are currently executing in your program.

TABLE I: LIST OF PRE-EMPTIVE TASKS

Task	***Functionality***	***Thread Priority***
A	$A = \sum_{x=0}^{256}[x + (x + 2)]$	2
B	$B = \sum_{n=1}^{16} \frac{2^n}{n!}$	3
C	$C = \sum_{n=1}^{16} \left(\frac{n+1}{n}\right)$	1
D	$D = 1 + \frac{5}{1!} + \frac{5^2}{2!} + \frac{5^3}{3!} + \frac{5^4}{4!} + \frac{5^5}{5!}$	2
E	$E = \pi r^2 + 2\pi r^2 + 3\pi r^2 + \ldots + 12\pi r^2$	3

2. Create a pre-emptive scheduling algorithm for the following Operating System (OS) based problem where each task has been assigned a priority :

Priority	**OS Task Function**	**Functionality**
1	Memory Management	-Increments its memory access counter. -Implements a bit band computation to a volatile memory location. -Passes control to CPU management. -After obtaining a signal back from CPU management, delays the OS for 1 tick and deletes itself.
1	CPU Management	-Increments its CPU management access counter. -Creates a conditional execution and barrel shifting scenario (to check its internal hardware). -Signals back to the memory management task and deletes itself.

2	Application Interface	-Ensure that Application executes before Device Management. -Accesses the global variable logger using a mutex. -Writes a partial message to logger. Waits for Device Management to finish writing to logger. -Increases its global counter. Delays the OS for 1 tick and deletes itself.
2	Device Management	-Writes the ending to logger variable started by the Application Interface. -Signals back to the App Interface. -Increases its global counter. Delays for 1 tick to ensure the file is closed and deletes itself.
4	User Interface	Increments the number of users (its variable), delays the OS for 1 tick then deletes itself.

Commence the task execution in main() with the memory management task. Implement each task's functionality as specified in the above table, noting that each function has its own global variable and may be dependent on another task. The tasks presented here are of a finite workload, with the highest priority task has priority one. For the demo version only, use the LEDs and the LCD to indicate the threads that are currently executing in your program. (Note you may need to add delays in your demo version).

Comment on pros and cons of pre-emptive scheme for the operating system problem.

Compare round-robin and pre-emptive schemes for the operating system problem.

For all the solutions:

Hand in the printout of the **analysis version** of your .c code, *RTX_Conf_CM.c* Configuration Wizard file, and snapshots of your Event Viewer and Performance Analyzer windows for **each** application. Question 2 may require several snapshots of your Event Viewer, and a well thought out choice regarding where you should halt your program for the Performance Analyzer window. Your TA will ask you to demonstrate the **demo version** for each application during your lab session. You may also be required to answer questions regarding the implementation and simulations of your applications.