Real-time Scheduling and Priority Inversion Understanding

COE718: Embedded Systems Design
Lab 4

1. Objectives

The purpose of this lab is to understand real-time scheduling using µVision for ARM Cortex-M3. Specifically, you will learn how to schedule and implement a Rate Monotonic Scheduling (RMS) technique, which is a popular technique for Fixed Priority Scheduling (FPS). Moreover, you will also have hands-on experience of priority inversion and its solution. To implement these approaches, you will be introduced to concepts of virtual timers, inter-thread communication methods (signals and waits for threads), along with static and dynamic priority inversions. You are expected to have a thorough understanding of the previous lab3 dealing with µVision and RTX for the completion of this lab. Refer to lecture notes on the background information pertaining to RMS and priority inversion.

2. Rate Monotonic Scheduling

2.1. Virtual Timers

Virtual timers are a type of countdown timer used in the CMSIS RTX API. Each timer possesses a callback function which is triggered once the timer has counted down. This callback indicates what action the timer is to perform once triggered. Therefore, the instantiation of multiple timers can countdown various periods of time, useful for the multiple tasks executed in a real-time system.

Virtual timers are defined after the #include and #define area in the .c code as:

```
osTimerDef(timer0_handle, callback);
```

Note its callback function must be declared before defining the timer(s). The virtual timer may then be instantiated within the main() as an RTX thread.

```
osTimerId timer_0= osTimerCreate(osTimer(timer0_handle), osTimerPeriodic, (void *)0);
```

The above statement creates a timer called timer_0 that specifies information for once its countdown has triggered. The timer0_handle will call the callback function with the argument (void *)0. In terms of the frequency of the countdown timer, osTimerPeriodic defines a periodic timer whereas osTimerOnce is used to declare a single-shot timer. The timer can then be started in the main() at any time using the following statement.

```
osTimerStart(timer_0, 3000);
```

It signifies that the timer timer_0 should start, with a countdown of 3000 milliseconds. The use of multiple virtual timers can trigger the callback function at various times and/or frequencies. An example application will be provided to you in the next section after explaining the importance of inter-thread communication in RTX or any other real-time operating system.

2.2. Inter-thread Communication - Signals and Waits

Up to now we have learned how to create threads, set their priorities, and use timers provided by RTX to create and schedule applications. In many applications however, there is a need to synchronize and communicate information among various threads. There are several means to communicate between threads in an RTOS. In the first part of this lab, we will focus on the use of signal and wait flags to synchronize execution between application threads. The concept is synonymous to signal and wait flags learned in general operating systems also.

A single thread in the RTX API may contain up to 8 signal flags stored in its thread control block. Signals are used to synchronize (signal or halt) the execution of threads. To synchronize threads, a thread usually "waits" for a "signal" to continue its execution. If a thread's signal flag number matches the wait flag number that was asserted by another thread, then the waiting thread can be released from the waiting state, and it will transition to the ready state for execution. Thus, this method is used to synchronize any number of threads - the signal thread must complete a certain task before the waiting thread can continue.

Like the previous lab, a thread is created and given an ID as given below.

```
void led_Thread1 (void const *argument);
osThreadDef(led_Thread1, osPriorityNormal, 1, 0);
 osThreadId T_led_ID1;

int main(void){
...
T_led_ID1 = osThreadCreate(osThread(led_Thread1), NULL);
...}
```

A wait flag may be set in the code as follows:

```
osSignalWait (0x03,osWaitForever);
```

The above statement signifies that the thread is waiting for the signal flag 0x03 to be asserted. The second parameter indicates the maximum duration (in milliseconds) that the thread should wait to be signalled. In this case the wait period is osWaitForever.

A signal may be sent to a thread or cleared using:

```
osSignalSet(T_led_ID2,0x01);    or    osSignalClear(T_led_ID2, 0x01);
```

2.3. Example Application

Launch the µVision application. Create a new project "virtual_demo" in your "lab4/example1" folder. Select the LPC1768 processor chip. Copy the files provided to you on D2L under lab4 folder, to your project directory. Configure your project workspace with the same settings as you did in lab3. Your project folder should resemble Figure 1.

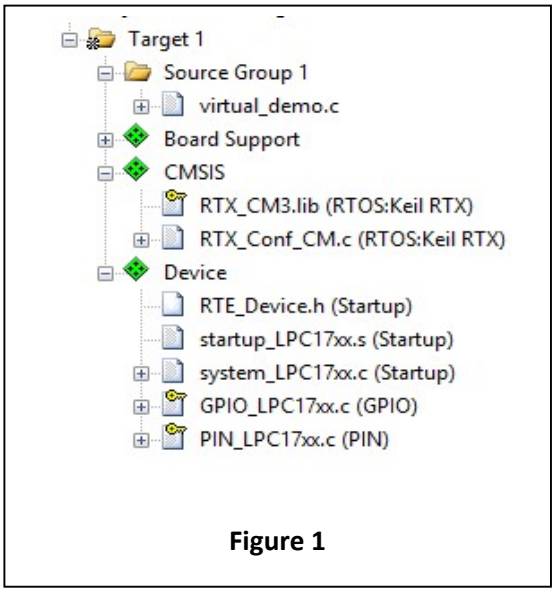

Figure 1

Check the "Options for Target"->C/C++ -> C99 Mode. Refer to lab-3 for reference on setting up an RTOS application. Make sure that the timers are enabled in RTX_Conf_CM.c.

Open the virtual_demo.c file and examine the code. Note the following: two virtual timers created and started with the countdown period of 3000 and 1000 respectively. The callback function is passed with the specified timer parameters once the timer has triggered. Three threads are created for the signal and wait demonstration (T_led_ID1, T_led_ID2, T_led_ID3). LEDs will flash according to the signal/wait pair that have been matched, or virtual timer currently being called back and executed.

Build the project and analyze the application execution (LEDs), its timing characteristics, and how it correlates with the sample code is given. You will use the debugger and analysis tools to trace variables to get an in-depth understanding. Make sure to comment out the osDelay() while debugging.

3. Priority Inversion

3.1. Introduction

Priority inversion has been a common design RTOS error in the past. An example of priority inversion occurs when a lower-priority (P3) thread blocks a higher-priority (P1) thread because P1 must perform a critical function before P3 can continue to execute. Within this time, it is possible that a medium priority (P2) thread pre-empts P3. Therefore, the medium priority thread is free to execute until it is blocked by a higher priority thread, where the high-priority thread is still waiting on the lower-priority thread to finish

executing. Therefore, the high-priority thread continues to be blocked while a medium priority thread executes, causing priority inversion.

A solution to this problem is dynamic priority elevation. To avoid priority inversion in the previous example, before P1 makes a call to the P3 thread, it must raise the priority level of P3. Once P3 finished executing the critical function, its priority level is reverted to low so that normal execution may continue. To change the priority level, the following code is used.

```
osThreadSetPriority(P3, osPriorityHigh);
```

where the priority of thread P3 is set to high and can be reverted later to osPriorityLow using the same function.

3.2. Example Application

Launch the µVision application. Create a new project "virtual_demo" in your "lab4/example1" folder. Select the LPC1768 chip. Copy the files provided to you on D2L under lab4 folder ,to your project directory. Configure your project workspace with same settings as in lab-3. Your project folder should resemble that of Figure 2.

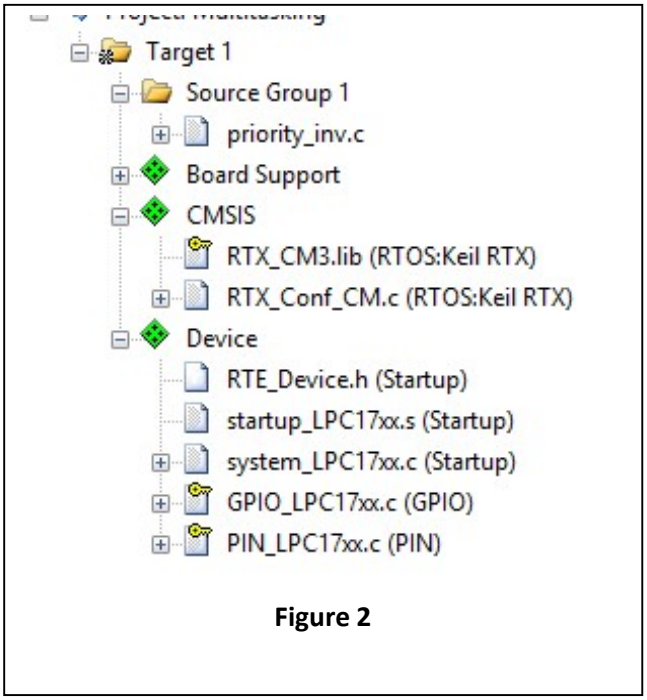

Figure 2

Check the "Options for Target"->C/C++ -> C99 Mode. Refer to Lab-3 for reference on setting up an RTOS (Real-Time Operating System) application. Ensure that the timers are enabled in RTX_Conf_CM.c.

Open the priority_inv.c file and examine the code. Note the two lines of code which are commented out. Once you have gained a firm understanding of what the code should do, build the project, and enter Debug mode. Reset and run the code. Use the Event viewer to see the priority inversion occurring

between the threads. This is a visual of the priority inversion example given in the introductory Section 3.1. The middle priority (taskB) thread will continue executing, blocking the higher and lower priority threads.

Uncomment the two lines of code to dynamically adjust the priority level of taskC. Save and build the project. Enter debug mode, reset, and run the code. Notice that priority elevation has eliminated the inversion that was occurring previously. Note: osThreadTerminate(t_main) is used to eliminate the main thread after the three threads have been created (since it is no longer needed).

4. Lab Assignment

The following two parts outline your assignment for the Lab-4.

4.1. Part 1

Implement an RMS algorithm using the following process set given in Table 1.

Table 1: 3-Process Set

Process	Period (T)	Computation Time (C)	Priority (P)
A	40000	20000	3
B	40000	10000	2
C	20000	5000	1

Schedule the above process set using inter-thread communication mechanisms and virtual timers. Since RMS is a Fixed Priority Scheduling method, ensure that the priorities are followed accordingly (i.e., the lower the number, the higher the priority). Note that the periods are much longer than the lecture examples due to debugging and demo purposes. Use a custom delay() function and the LEDs to represent the execution of a thread or process.

Hand in the .c code for the scheduling of this process set. Moreover, include an execution timeline for the process set (may be drawn, or by using Visio, or any other Drawing software). Submit this timeline with your code. Make sure that your program execution matches your calculated timeline for verifying the correctness. You may also be quizzed on this implementation and asked to demo the RMS algorithm to your TA or Lab supervisor.

4.2. Part 2

The Mars Pathfinder (MP) is a well-known priority inversion problem. Figure 3, given below represents a simplistic timeline we will use to represent the MP problem.

As seen in the example of Figure 3, the lower priority task C is executing its thread's workload on resource R1 and is pre-empted by the medium priority task B at 50ms. Therefore, when task A (the highest priority) is executed at 60ms which needs to execute on resource R1, it will continuously be blocked by task C. In fact, task C will not execute again since it is pre-empted by both Tasks B and A. Task B will also be blocked by Task C since it has a medium priority.

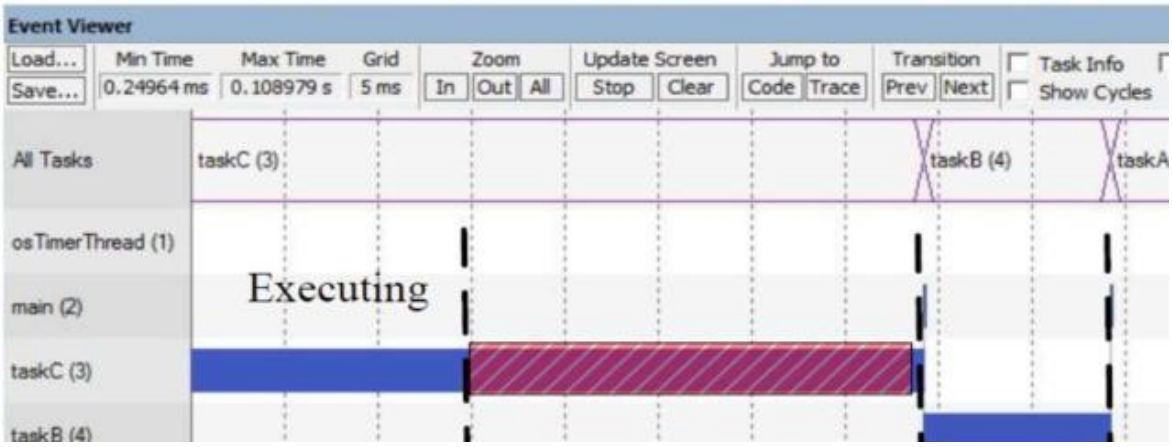

Figure 3

Schedule and implement both the problem and solution within a single .c code for the above example. No demo (LEDs) is necessary for this exercise. You will only need to use the debugger (Event Viewer, RTX Task Viewer etc.) to visualize the problem. There are many ways to eliminate the priority inversion. It is to be noted that RTX is a modern RTOS that will never allow a priority inversion. If you decide to use a mutex to represent the necessary resource, do not use the RTX defined os_mut as this automatically eliminates priority inversion. Use a Boolean variable instead. Place comments and or preprocessor flags where necessary to clearly outline which code represents the solution. Provide two printouts of the Event Viewer displaying both the MP problem and its respective solution. Also provide a printout of the RTX_Conf_CM.C wizard editor (screenshot). You may also be quizzed by your TA on your understanding of the priority inversion problem and your solution during the demo and submission.