COE4DS4 Lab #6
Introduction to Real-Time Operating Systems

Objective

To learn about the basic task management and resource sharing in a real-time operating system (OS).

Preparation

- Read uC/OS-II documents and learn about its basic features and services
- Read this document and get familiarized with the source code and the in-lab experiments

Experiment 1

Simply put, in a multitasking environment multiple tasks (also called programs or processes) run on a processor and, although the processor is time-shared between these tasks, each task considers that it owns the processor. The very purpose of multitasking is to switch between tasks and schedule their execution. Multitasking is a key functionality of operating systems. Examples of other functions performed by operating systems, which are not discussed in this document, are device drivers or file systems. The switching between tasks and the scheduler (also called dispatcher) are implemented as part of the operating system kernel. In a real-time environment, the kernel attempts to schedule tasks such that the real-time constraints are met. The real-time constraints can be either hard or soft, depending on whether failing to meet them will cause the system to crash or degrade its service or performance.

uC/OS-II is a portable multitasking kernel developed primarily for form-factor constrained embedded systems. Its source code is approximately six thousand lines of code written in ANSI C and it has been ported to over one hundred different architectures ranging from 8 to 64-bits. In the lab folder you are provided with the reference and configuration manuals for uC/OS-II, as well as with a one-page quick reference guide with its key functions for event/task management. At its core it has a deterministic real-time preemptive scheduler. Determinism implies that its services (used to create/delete tasks, pass data between tasks, ...) have a known execution time. A preemptive scheduler will stop the execution of a lower priority task whenever a higher-priority task is ready for execution. In a non-preemptive scheduling policy, a task will complete its execution before it releases the processor to service the other tasks. If the interrupts are enabled, the uC/OS-II scheduler will release the processor to service the interrupt service routines (ISRs).

The uC/OS-II kernel can handle 64 tasks. The 4 highest and the 4 lowest priority tasks are used by the operating system itself, which leaves 56 tasks that can be introduced by the user. The kernel provides functions for task creation and deletion. Each task has an infinite loop, in which it can be suspended or delayed. During the suspension/delay the processor is released, which enables other tasks to be scheduled to run on the processor. The suspension can be done through events such as semaphores, mailboxes and queues (these OS events will be discussed using code snippets in-class and in-lab). The above types of events can also be used to communicate between ISRs and the concurrent tasks managed by the scheduler. Delaying task execution is done through time management functions that rely on time periods determined by an interval timer. These intervals (called OS ticks, or just ticks) help keep track of time for task scheduling, as well as timeouts for event monitoring.

The uC/OS-II kernel manages a task ready list, a priority table, as well as the task control blocks for all the active tasks. A task control block contains the priority level and pointers to private data and the task stack. The priority level is used by the scheduler, which preemptively schedules the task with the highest priority when it is ready for execution. Whenever a task is suspended or delayed (and the processor is assigned to another task) the task-dependent info from the processor's registers and the system stack are saved on the task's private stack (called context saving). When a task resumes execution its context is restored. When a task is suspended/delayed and another task is started/resumed then one context will be saved and another context will restored; this is called context switching and it is managed by the OS.

Copyright © by Adam Kinsman, Henry Ko and Nicola Nicolici – Embedded Systems course at McMaster University, Canada

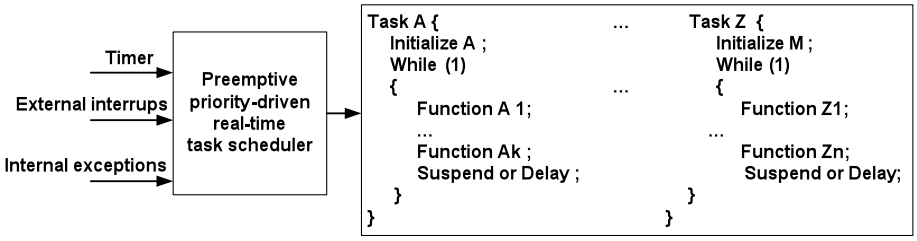

Figure 1: A simple overview of uC/OS-II.

In the source code provided in the *experiment1* folder, we investigate the difference between preemptive and non-preemptive schedules.There are five tasks that are used in this experiment. A task launcher with low priority creates tasks 0 and 3 of higher priority. Task 0 has a higher priority than task 1, which has a higher priority than task 2, which has a higher priority than task 3. Note, *in uC/OS-II the lower the integer code for the priority level of a task, the higher the priority that will be given to it by the scheduler* (e.g., the task with the integer code 10 has higher priority than the task with the integer code 11). The tasks can be suspended by not proceeding further with the execution until an event, such as setting up a semaphore (as used in this experiment), has occurred.

Task 0 monitors a semaphore that is set by an interrupt service routine after push button 0 is pressed; the status of each task in the system will then by displayed; then the task suspends itself by waiting for the semaphore which is set by push button 0. Task 3 creates task 2 and then it will monitor if push button 3 has been pressed; after push button 3 was pressed, task 3 will delete itself. Task 2 will first create task 1 and then it will monitor if push button 2 has been pressed; after push button 2 was pressed, task 2 will delete itself. Task 1 will monitor if push button 1 has been pressed and then it will delete itself. The reference code implements the implicit scheduling policy, which is preemptive. Under this policy, because task 1 will have a higher priority than tasks 2 and 3, it will preempt tasks 2 and 3 (i.e., they will be suspended until task 1 completes); for example even if push button 2 is pressed, task 2 will not finish because it is suspended until task 1 completes its execution. First push button 1 must be pressed, which will lead to the completion of task 1 and only then the semaphore set by push button 2 is read by task 2.

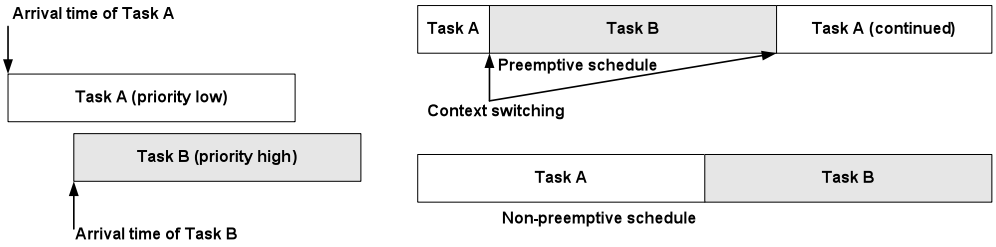

Figure 2: The basic concept of preemptive and non-preemptive schedules.

In the lab you will have to do the following:

- Run the reference code and understand how preemptive scheduling works
- Modify the source to implement non-preemptive scheduling; to achieve this you will need to disable the scheduler each time you enter a task and enable it before you leave the task (this is achieved using service functions from the OS kernel); understand the non-preemptive schedule

Experiment 2

In this experiment, periodic tasks are created and deleted dynamically when push buttons are pressed. Each task is characterized by its execution time and its period, which are both assigned random values when the task is created. The period is the sum of the execution time (when useful computations are done) and the “non-busy” (or idle) time when the task is suspended and the processor is assigned to other tasks.

Copyright © by Adam Kinsman, Henry Ko and Nicola Nicolici – Embedded Systems course at McMaster University, Canada

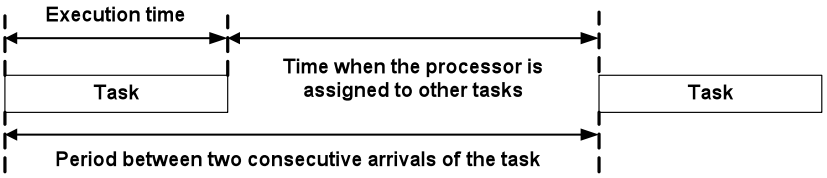

Figure 3: The period and execution time of a task.

To emulate longer execution times than a few instructions, we use a custom delay function. It uses a dummy for loop, which requires computation from the OS perspective. The timer in Qsys is set up to generate an interrupt with a frequency of 1 KHz, which implies that each OS tick is 1 ms. The `custom_delay()` function, which will be used in our labs is given the number of OS ticks as a parameter and a dummy `for` loop will keep the processor busy for the specified time. For the "non-busy" time we use one of the delay functions provided by uC/OS-II: `OSTimeDly()`or `OSTimeDlyHMSM()`, which can specify the suspension time either in terms of OS ticks or in terms of hours, minutes, seconds and milliseconds. When using these OS delay functions, the scheduler knows the task does not need the processor, which can be re-assigned to other tasks that need it.

Each time push button "i" is pressed, task "i" will be deleted if it is active or created if it is inactive. The priorities of each task are updated dynamically (at runtime) each time a task is deleted/added, according to the following simple policy. Whenever a new task is created it will be assigned the lowest priority of the four tasks controlled by the four push-buttons. When a task is deleted, all the tasks with lower priority than it (i.e., higher integer code) will by updated by decrementing their integer code (i.e., their priority will be increased). Each time a push button is pressed a custom function is used to re-assign the priorities of the tasks and pass these new priority levels to the OS scheduler. The scheduler will consequently resume the task execution with the updated priority levels.

In the lab, all you need to do is understand how the priorities of the tasks are updated at runtime.

Experiment 3

When multiple tasks access shared resources, a synchronization mechanism must be provided by the operating system. For example, if multiple tasks require access to the character liquid crystal display (LCD) peripheral on the DE2 board, if each of them would send characters without some form of synchronization between tasks, then characters (or groups of characters) from different tasks will be "mixed" on the character LCD screen. In the reference code provided to you, messages from two different tasks will be displayed on the screen if the lower priority task is preempted by the higher priority one after it wrote only the first line on the LCD. To avoid this problem, the mutex event (provided by the OS) can be employed as shown in the figure below. The mutex needs to be created before it is used. Subsequently, when each task needs to access the LCD it will first check if the LCD is used by another task. If yes, the task will wait until the resource has been released, at which time it can proceed further with its execution. Note, mutexes can lead to the priority inversion problem and examples will be discussed in-class.

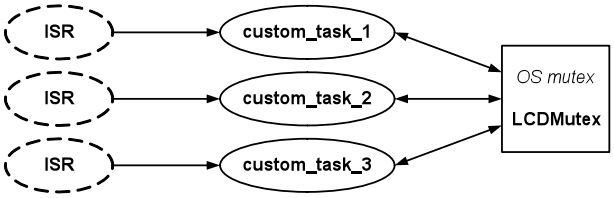

Figure 4: Multiple tasks synchronizing accesses to a common resource through a mutex.

In the lab you will have to do the following:

- Understand the behavior of the reference code with and without the mutex
- Modify the mutex pending call from the infinite timeout (as it is currently implemented) to a pre-defined value, such as 1 second. If the mutex is not released before timeout, print a message in the terminal and proceed further with the code execution for the respective task

Copyright © by Adam Kinsman, Henry Ko and Nicola Nicolici – Embedded Systems course at McMaster University, Canada

Write-up Template

COE4DS4 – Lab #6 Report
Group Number
Student names and IDs
McMaster email addresses
Date

There are 3 take-home exercises that you have to complete within one week. Label the projects as exercise1, exercise2 and exercise3.

Exercise 1 (1 mark) – Register two periodic tasks with the OS. The execution time ("busy" time or the *execution_time* field in the *task_info_struct* from *experiment2*) and the idle time ("non-busy" time or the *os_delay* field in the *task_info_struct*) for task 1 should be assigned as follows: the *execution_time* in terms of OS ticks will be equal to the number of letters in the first name of the first group member multiplied by 250; the *os_delay* in terms of OS ticks will be equal to the number of letters in the first name of the second group member multiplied by 850. To emulate *execution_time*, use the `custom_delay()` function, whereas for *os_delay* use the `OSTimeDly()` function. For task 2, which has a lower priority than task 1, *execution_time* and *os_delay* are assigned based on the same principle, however the numbers of letters from the surnames of each group member (instead of first names) are used as the scale factors. The two tasks should run indefinitely and print the OS tick, obtained through `OSTimeGet()`, each time they are started. You must run both the preemptive and the non-preemptive schedules (as in *experiment1*) and discuss the differences. Note, because the tasks are periodic, for the non-preemptive case, re-enable the scheduler before you call the `OSTimeDly()` function. Submit your sources and in your report write a third-of-a-page paragraph describing your reasoning.

Exercise 2 (4 marks) – Modify the *experiment2* as follows. In this take-home exercise you must change the priority assignment algorithm as follows. After the NIOS system has been started, for the first time when the push button "i" is pressed the task "i" will be created and it will run with the highest priority of the 4 periodic tasks that have been created up to that point in time. Every subsequent time when push button "i" is pressed again, it will be assigned the lowest priority of the 4 periodic tasks that have been created up to that point in time. Note, tasks are created only once and they are never deleted (they will run until the program execution on NIOS is stopped from the NIOS SDK). Note also, the created tasks can be re-assigned priorities before all 4 tasks have been created. When the tasks are first created use the same reasoning as in-lab or **exercise1** with functions such as `custom_delay()` or `OSTimeDly()`. However, for this exercise, the period for each task should be a random value between 10 and 12 seconds and the *execution_time* for each task, emulated through `custom_delay()`, should be a random value between 1,000 and 1,750 ms.

As in the in-lab experiment, the custom scheduler should run with a higher priority than any of the 4 periodic tasks controlled by the push buttons. Unlike the in-lab experiment, where the custom scheduler checks the status of push buttons every 500 ms, for this take-home exercise the custom scheduler should run every 3 seconds. Therefore, because one or more push buttons can be pressed multiple times in between two subsequent runs of the custom scheduler, it is your responsibility to keep track of the order in which push buttons have been pressed (it is assumed that within 3 seconds, push buttons cannot be pressed more than 10 times). For example, if in between two runs of the custom scheduler, we press push button 1, then push button 2, and then push button 1 again, the custom scheduler will assign the lowest priority to task 1 and task 2 will be assigned the second lowest priority of the 4 tasks controlled by push buttons. For the sake of simplicity, it is assumed that no push buttons are pressed while the custom scheduler is running and when the custom scheduler is done with the priority re-assignment, all the push button semaphores will be cleared. Submit your sources and in your report write a third-of-a-page paragraph describing your reasoning.

Copyright © by Adam Kinsman, Henry Ko and Nicola Nicolici – Embedded Systems course at McMaster University, Canada

Exercise 3 (3 marks) – Modify the *experiment3* as follows. There are four tasks (0, 1, 2 and 3) that are competing for four shared resources: input switch group A, input switch group B, red light-emitting diodes (LEDs) and green LEDs. Due to sharing (as explained below), each of these four resources is assigned a mutex used for arbitration.

- Each time when push button 0 is pressed, task 0 reads the value from the input switch group A and it prints it to the corresponding least significant red LEDs for 3.5 seconds
- Each time when push button 1 is pressed, task 1 reads the value from the input switch group A and it prints it to the corresponding least significant green LEDs for 3.5 seconds
- Each time when push button 2 is pressed, task 2 reads the value from the input switch group B and it prints it to the corresponding least significant red LEDs for 2.5 seconds
- Each time when push button 3 is pressed, task 3 reads the value from the input switch group B and it prints it to the corresponding least significant green LEDs for 2.5 seconds

Tasks 0 to 3 should have priorities in the increasing order of the task index (that is task 0 has the highest priority). Any mutex should have a higher priority than any task, as for the order between them, the mutex for the input switch group A should have the highest priority; then the mutex for the input switch group B; then the mutex for the red LEDs and then the mutex for the green LEDs. The mutex for each shared resource should be implemented with an infinite timeout. Note, in order to execute task 0 must obtain both the mutex for the input switch group A and the mutex for the red LEDs (the same principle applies to any task). Note also, because SWITCH_I[17] is used as a reset active low, switch group B has only 8 signals.

Implement the above spec and in your report provide a concise explanation for the observed behaviors for the following two experiments: (i) push buttons are pressed in the increasing order 1 second after each other; (ii) push buttons are pressed in the decreasing order 1 second after each other.

Submit your sources and in your report write a third-of-a-page paragraph describing your reasoning.

VERY IMPORTANT NOTE:

This lab has a weight of 8% of your final grade. The report has no value without the source files, where requested. Your report must be in "pdf" format and together with the requested source files it should be included in a directory called "coe4ds4_group_xx_takehome6" (where xx is your group number). Archive this directory (in "zip" format) and upload it through Avenue to Learn before noon one week after the day you have done the in-lab experiments for lab 6. Late submissions will be penalized.

Copyright © by Adam Kinsman, Henry Ko and Nicola Nicolici – Embedded Systems course at McMaster University, Canada