Real Time Operating Systems Design and Programming

LAB 3

OS Mutex Lab

Implementing Mutex and Dodging Deadlock

Contents

1 Overview..2
2 The environment..2
3 Lab Procedure...2

1 Overview

In this exercise, a simulated environment based on the RTX RTOS is created. From here you will explore some of the concepts and algorithms introduced during the lectures.

2 The environment

The Discovery board is a typical uniprocessor system. Therefore you can encounter preemption problems but not multiprocessor-related problems. With the help of RTX, however, it is possible to at least create an “illusion” that tasks are running in parallel.

Look back at the already completed lab code from the provided Keil project. You can see that the code has two tasks, which share an exclusive resource. For demonstration purposes, the exclusive resource here is an on-board red LED (non-mutex access does no harm). The blue and green LEDs indicate whether task0 or task1 is running respectively.

Verify that the configuration for RTX is correct, in particular make sure the scheduling algorithm is RR (as it provides the multiprocessing “environment”). You are asked to insert delay functions in some questions to make sure preemption occurs during the delay period, so that both tasks can be synchronised and “run in parallel”. Feel free to insert the function wherever you want to see the result of exact synchrony between two tasks to simulate the effect of multiprocessing, but make sure the delay period is long enough for preemption to take place. You can also try other algorithms (non-preemptive) during the lab and you should see that the sequential program does not usually involve difficult concurrent issues.

You may also use an oscilloscope, in which case you should attach the probe to PD12 for the green LED, PD13 for the orange LED, PD14 for the red LED and PD15 for the blue LED.

3 Lab Procedure

1. Open the template project. Scan through the code.
2. Compile, download and run the code, you can see task0 finishes its task (red and blue LEDs on and off) without competition from task1.
3. What would happen if you uncomment the line //taskID1 = os_tsk_create(Task1, 0); try and run the code. What do you see? Is that what you expected?

   Green, blue and red LEDs goes on and off almost at the same time. There is a moment when all three LEDs are on. This is clearly not mutex access.

You will now try to implement a mutex for these two tasks. You can create new global variables and add new code in task0 and task1, but do not change the sequence of the code in the critical sections. You can assume all the writings are atomic.

4. The first mutex scheme suggests that you can check if the resource is accessed before entering the critical section. If the resource is available, enter the critical section, otherwise busy-waiting. Use the Check function. Check (RED), for example, returns 1 when red LED is on and 0 when off. Implement this by adding just one line just before the critical section for both tasks (while loop should be enough), run and describe what you see. Does this enforce mutex?

   See the solution project for the code - should add while(Check(RED)){} before each critical section.

   Red and blue go on at first, then blue goes off and green goes on. Green and red go off in the end. This result implies that this scheme enforces mutex.

   However, this does not enforce mutex! See next question for the explanation.

5. Now, assume you have the while loop like while(Check(RED)){} for question 4. Try to add this line: Delay(rand() % RDIV*RAMT + RMIN); right between the while loop and the critical sections of both tasks. It keeps the task doing nothing for a while before entering the critical section. Rerun your code. How will this affect the result? Is there any moment that all three LEDs are on? What does the result imply?

   There is a moment when three LEDs are on. This means it does not enforce mutex.

   You can think of the problem without adding the Delay function. What if both tasks examine the availability of the resource at the same time? They will both find it available and both enter the critical section. The Delay functions have been added to simulate this precise scenario as neither task is able to get to the critical section of the code and get hold of the RED LED resource in one single tick. Both tasks will clear the Check(RED) test before either task locks the other.

6. The second mutex scheme suggested is as follows. First, create a global integer variable called token and initialise it as 0. This variable indicates which task's turn it is to use the critical resource. Second, add this code to protect each critical section. Let $i \in \{0,1\}$ and i' be $1-i$.

   ```
   while ( token != i ) { }

   /* critical section */

   token = i';
   ```

   Implement this for both tasks, run and describe what you see. Does this enforce mutex? If it does, what are the main problems for such mutex?

   See the solution project for the code.

   Red and blue go on at first, then blue goes off and green goes on. Green and red go off in the end. This enforces mutex.

However, the disadvantage is that tasks take turn to enter the critical section instead of a need basis (i.e. a task wishing to execute the task). This will lead to starvation if the task that has the turn does not enter the critical section.

7. The third mutex scheme suggests the use of flags as follows (Note that *i'* is equal to *1-i*):

   ```
   flag[i] = true;

   while ( flag[i'] ) { }

   /* critical section */

   flag[i] = false;
   ```

   Here, a set flag indicate a wish to execute. Implement the above code for both tasks, run and describe what you see. Does this enforce mutex? If it does, compare to the second scheme, how does this improve the mutex?

   See the solution project for the codes.

   Red and blue go on at first, then blue goes off and green goes on. Green and red go off in the end. This enforces mutex, and tasks can access critical section based on their needs.

8. As in question 5, add this line of code: Delay(rand() % RDIV*RAMT + RMIN); between where you set the flag and the while loop. How will this affect the result? Run the code and describe what you see. What does the result imply?

   No LEDs will be on.

   Deadlock occurs. Each task grabs its flag and will not release it until the other does so.

   The Delay function plays the same role here as it does in question 5.

9. The fourth mutex scheme modifies the third to solve the deadlock problem by being "polite" and turning the flag from non-preemptible resource into preemptible resource:

   ```
   flag[i] = true;

   while ( flag[i'] ) {

   flag[i] = false;

   Delay(Random_Time);

   flag[i] = true;

   }

   /* critical section */

   flag[i] = false;
   ```

Implement this for both tasks, repeat question 8 for this scheme. Run and describe what you see. Does this scheme solve the problem that the third scheme has?

Red and blue go on at first, then blue goes off and green goes on. Green and red go off in the end. This enforces mutex, and solves the deadlock problem.

10. The fourth scheme is believed to be problematic. It will potentially cause **"livelock"** where two tasks are too "polite" so that no task actually consumes any resource while waiting for the other. Try to use the delay function again to show how that could happen. Hint: the delay function is effectively a synchronisation mechanism between the tasks; use it whenever you want both tasks to run in "real" parallel.

    The delay function can be inserted into the first line within the while loop. Both tasks can check each other's flag at the same time and then give up their own flags at the same time and so on and so forth.

    You will see no LEDs flashing in this case.

11. Implement the Dekker's algorithm for both tasks. A pseudocode version can be found from the lecture slide. You may start from reusing the token and flags implemented for the previous questions. Understand how it works. Hint: the fourth scheme is actually pretty close to the Dekker's algorithm; try to combine the second scheme and the fourth scheme.

    See the solution project for the code.

12. (Optional) Another successful algorithm for mutex is the Peterson's algorithm. Look it up in an OS textbook or online[1]. Show how it works. Why is it better?

    flag[i] = true;

    turn = i';

    while (flag[i] && turn == i') { }

    /* critical section */

    flag[i] = false;

    This algorithm is more "polite" than the fourth scheme. However the most important thing is it does not depend on the atomic operations.

As you may have noticed from the lab, all the mutexes apply the busy-waiting mechanism. Although in some cases this is desirable, this is generally a waste of CPU resources. You may use the os_tsk_pass() instead of task busy-waiting, but OS services like semaphores are better alternatives in most cases. We will come back to these topics later.

[1] e.g. https://en.wikipedia.org/wiki/Peterson%27s_algorithm