Comp3402 Critical Sections, MUTEX

C.B.Price March 2023

Purpose	(i) To investigate shared memory problem (ii) To investigate shared resources problem. (iii) To solve these using either Critical Sections and Semaphores.
Files Required	Arduino **portable** sketchbook
ILO Contribution	6
Send to Me	
Assignment Info.	
Homework	Read Chapter 8 (10)

Activities

1 Critical Sections – Sharing Resources

Here we shall investigate the situation where two tasks share the same system resource, the Serial Monitor. We shall uncover a problem and find out how to fix it.

(a) Open the sketch **RTOS_Rel_6** where you will see two tasks that count from 0 to 100 and send the result to the serial monitor. Run the sketch and you will see that some lines are 'mashed' where the Serial.print commands in the two tasks interfere with each other.

(b) The first way to solve this problem is to use a critical region wrapped around all calls to the serial monitor. This is a crude solution since it involves suspending all interrupts on the MCU including the interrupt which ticks the scheduler. Here's the syntax

```
taskENTER_CRITICAL();
   calls to serial monitor go here
taskEXIT_CRITICAL();
```

You should find this solves the problem, and you have a nice output where you can see the Tasks outputting in an interleaved manner as they are swapped in and out.

(b) The second way to solve this is using a MUTEX. Remember this is rather like a baton which only one task can possess. The task asks for it at the beginning of a critical region, and if it's free the task will get it. The other task will then be unable to get it, so will wait at the start of its critical region. Only when the first task has finished its critical processing, and releases the baton, can the second task get it and run.

First declare the mutex up top (I named mine 'xMutex'): **SemaphoreHandle_t xMutex;**

Then create it in setup() **xMutex = xSemaphoreCreateMutex();**

Here's the syntax for taking and releasing the mutex

```
xSemaphoreTake(xMutex,portMAX_DELAY);
    calls to serial monitor go here
xSemaphoreGive(xMutex);
```

2 Critical Sections – Sharing variables in Memory

Here we shall look at two tasks which change the value of a variable **sum** shared between tasks. Task 1 increments sum 100000 time and Task 2 decrements sum 100000 times. So, when both tasks are complete, sum should be zero. The value of sum is printed out by Task3 when both Tasks1 and 2 inform Task3 that they have finished their work.

(a) Have a look at the sketch **RTOS_Rel_7**. All three tasks have the same priority. Here's what happens (but we do not know which task (1 or 2) will be swapped in first since they have the same priority. Let's assume it's Task 1

- Task1 does its summing
- It sets flag **done1** to true
- It suspends itself
- Task2 does its summing
- It sets flag **done2** to true
- It suspends itself

Of course, Task3 may be swapped in at any point. The logic in Task3 so it does its job is

- If both **done1** and **done2** are true
    - print out the value of **sum**
    - set done1 and done2 to false
    - resume both tasks.

This way you will see a repetition of calculations and they will all be wrong.

Maybe you do not fully understand how this problem actually happens, since it does not matter the *order* of the expressions sum = sum + 1; and sum = sum – 1; The answer (see chapter 8) is that it scheduler swaps occur at the *machine code level* and the above expressions may not translate into a single line of machine code.

(b) Use you experience with mutexes to add code and solve the problem. The mutex is already declared and defined. Your task is to decide where to place mutexes to protect the necessary code.