**CG2271 Real Time Operating Systems**

**Lab 6 Answer Book**

|  |  |
| --- | --- |
| Name | Student ID (AxxxxxY) |
| John Woo Yi Kai | A0272561E |
| Rishi Moorthy | A0273618X |
| Wang Yanjie | A0276202M |
| Kenneth Wong Cun Wi | A0303203A |
| Kuek Yeau Hao Jonathan | A0258485M |

**Question 1.** (3 MARKS)

The manageTasks() function is a simple, round-robin, time-sliced scheduler implementation. It uses a fixed busy-wait delay of delayMS(300) at the end of each loop iteration, establishing a 300ms time slice or base period. The function uses a counter variable, count, to determine which task to execute in the current time slice. The count variable cycles from 1 to 4 using a switch statement:

* Case 1 (count=1): Prints "Hello!"
* Case 2 (count=2): Toggles the Red LED.
* Case 3 (count=3): Toggles the Green LED.
* Case 4 (count=4): Toggles the Blue LED.

After each task execution, the loop delays for 300ms and increments the counter, wrapping back to 1 after 4. Since each task is executed only once every 4 iterations of each taking 300ms, all tasks have a period of 1200 ms (4 x 300ms = 1200ms), with a stagger of 300ms:

* “Hello!” starts at 0ms
* Red LED at 300ms
* Green LED at 600ms
* Blue LED at 900ms

There will be an action every 300ms (every loop switch to the next task), but each task themselves cycles every 1200ms.

**Question 2.** (2 MARKS)

It will not be possible to achieve those timings simply using a busy-wait delay and the constraint of a minimum of 5ms delay. If we look at the numbers 210, 322, 333, and 425, they have no significant common divisors. In fact, the 4 numbers are mutually coprime, which means their GCD is 1ms, which is lower than the minimum of 5ms. There is no other possible delay value that can be used to cycle from one task to the next sequentially in the required task periods.

**Question 3.** (1 MARK)

Change RED\_DELAY\_MS, GREEN\_DELAY\_MS, and BLUE\_DELAY\_MS to respective values of 322, 333, and 425 for the values in vTaskDelay(), and similarly for the hello\_task() to 210ms.  
  
It was easier because we could simply change the delay for each LED through a simple invocation of vTaskDelay() instead of modifying a busy-wait loop. Tasks now run concurrently (independently of each other) and has its own period and delay that does not depend on a common divisor, unlike the previous method which would changing the single-loop structure and the common delay since all tasks shared a single timeline.

**Question 4a.** (1 MARK)

Expected result of sum: 500 000

Actual value of sum: 500 000

**Question 4b.** (2 MARKS)

The actual value of the sum now is 422851. We get the wrong results (sum less than 500000 and varies every run) when the time slicing is ON because time slicing forces the CPU to interrupt the running task frequently at every tick, and a context switch can happen between these steps, causing one task to overwrite the other task’s update. This means multiple tasks may perform sum++ concurrently, causing some increments to overlap and are lost. However, when time slicing is OFF, one task runs to completion before the other, basically executing sequentially with no concurrent access, hence resulting in the correct value.

**Question 5.** (2 MARKS)

The primary advantage of using time-slicing is that it allows multiple tasks (or parts of an application) to share the CPU fairly, giving each task a chance to run for a short time before switching to the next. By rapidly switching the CPU's attention among tasks in fixed time intervals (time slices), the system ensures that no single task can occupy the CPU for an indefinite period. This prevents starvation for lower-priority tasks and guarantees a measure of responsiveness.

**Question 6.** (2 MARKS)

Yes, the answer is correct now (500000) because the use of a mutex prevents multiple tasks from updating the shared variable sum at the same time. This mutex (which is mutual exclusion) allows only one task at a time to access the section where sum is updated, without overwriting each other. Basically, the first task that executes xSemaphoreTake on the mutex will decrement it to 0, “locking” the mutex and blocks all other tasks that try to access it. A second task executing xSemaphoreTake on the mutex that is now 0 will block, until the initial task releases the mutex using xSemaphoreGive(mutex), unblocking the second task. When the second task executes xSemaphoreGive, the mutex will increment back to 1 and runs. This prevents race conditions, giving us the correct sum value.

**Question 7.** (1 MARK)

Yes. blinkLEDTask and helloTask are of equal priority. It will ensure that both tasks get to run.

**Question 8.** (2 MARKS)

Yes it runs now. FreeRTOS executes the highest priority ready task, hence helloTask, a higher-priority task of (2) that is ready to run will preempt blinkLEDTask of lower-priority (1). As soon as helloTask is created and scheduled, it will preempt blinkLEDTask and execute. blinkLEDTask will only be able to run when helloTask is in a blocked state.

**Question 9.** (2 MARKS)

The delay causes blinkLEDTask to block for the specific time of 250ms, giving up CPU control for other tasks. This allows helloTask to run even if both tasks have the same priority. We can say that vTaskDelay causes blinkLEDTask to voluntarily yield, allowing helloTask to run.

**Question 10.** (2 MARKS)

With the blink variable, the task responsible for checking it must constantly consume CPU cycles in a loop, even when there is no work to do, or simply known as polling. In contrast, a semaphore allows event-driven synchronisation, where a task enters the Blocked state while waiting for an event. When the event occurs, the task is immediately transitioned to the Ready state, consuming CPU time only when work is genuinely available. This results in more efficient CPU utilization by saving CPU cycles and ensures that the task executes immediately when signaled.

**Total: \_\_\_\_\_\_\_\_\_\_\_\_\_\_/ 20**