**CG2271 Real Time Operating Systems**

**Lab 5**

**Answer Book**

|  |  |
| --- | --- |
| Name: | Matric No: |
| Name: | Matric No: |

Question 1 (6 marks)

As in the previous lab, when the OSCreateTask**(**int prio**,** void **(\***rptr**)(**void **\*),** void **\***rarg**)** function is called, the kernel automatically allocates a dedicated TCB for that task (in the *\_tasks* array of TCBs). Each time this is done, the kernel allocates a new stack (with its pointer) which is also stored in that task’s TCB, together with other things like the pointer to the function representing the task, the arguments for the function, as well as the task’s priority. Even though both task 1 and 2 are using the same code, separate TCBs are created for each of them. This allows them to maintain their own context (hardware registers and stack) separately, so they behave as though they are two independent tasks despite being created from the same function.

Question 2 (3 marks)

By inspection, we would expect to see something similar to this in the console:

Hello there! I am task #1

Hello there! I am task #2

Hello there! I am task #1

Hello there! I am task #2

…

The numbers 1 and 2 are the arguments passed to the functions which were specified when creating the tasks.

Question 3 (5 marks)

From experimentation, it is observed that the output would be fine if either one of the tasks was disabled. The problem occurs when the second task attempts to write into the serial buffers immediately after the first task goes to sleep. The data has not had the time to transmit fully before it is being written to again, causing data corruption.

Question 4 (5 marks)

void OSCreateQueue**(**int **\***buffer**,** unsigned char length**,** TMsgQ **\***queue**)** accepts an integer buffer (an array allocated by the progammer), the length of the space, and a *TMsgQ* structure to hold information about the queue. The function simply fills in basic information in the provided into the *TMsgQ* struct.

void OSEnqueue**(**int data**,** TMsgQ **\***queue**)** accepts data written to the queue, as well as a TMsgQ representing the queue. If the queue is full, the data is ignored and the function returns. If it is not full, then the data is appended to the back of the queue. If a task had been waiting for the queue (and is currently blocked), it is also unblocked and placed back into the ready queue. (This is mechanism is further elaborated below).

int OSDequeue**(**TMsgQ **\***queue**)** takes in a TMsgQ struct and dequeues the item at its head if it exists. If there is nothing in the queue, the process is immediately blocked and the scheduler is called to swap to another task. The *blockproc* integer value of the TMsgQ struct is also updated to reflect the number of the task being blocked. This is so that when new data is available, the enqueue function knows which task to unblock. If data is indeed available, the item at the head is polled and returned.

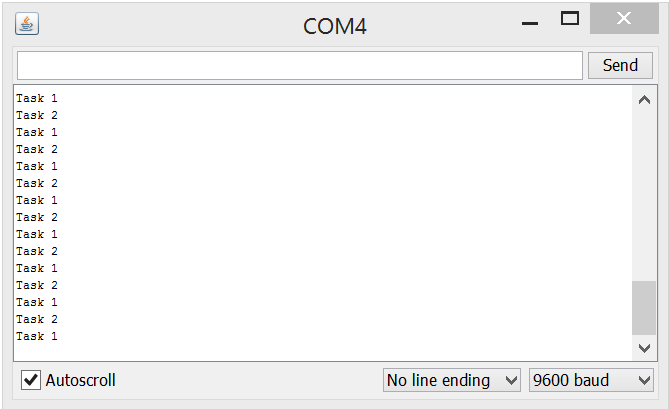
The difference between the queues learnt in data structures modules and the queue used here is that this queue blocks when there is no data available, and enqueuing can wake up a task that is waiting for that queue. There is some sort of inter-task communication taking place here.

Another difference is that the typical queue we learnt in data structure uses a higher-level abstraction (linked lists of ListNodes, and so on) than the one used here. The queue here is implemented simply with an array and two pointers.

Note: A potential issue is observed in OSDequeue. Since *procblock* is a simple integer value, it is only safe for one task to dequeue from the queue. If multiple tasks dequeue from this queue when no data is available, all other tasks except the last one will be permanently blocked.

Question 5 (3 marks)

The output of the program is as shown:



“Task 1” and “Task 2” are now printed in alternating fashion with no text corruption.

Question 6 (5 marks)

Data output through the serial port is not instantaneous, and requires time for data to be sent out. Since the time taken to transmit the data serially exceeds the rate at which the two tasks are writing data into the queue (5ms per task, which results in an average of 2.5ms per message), if we do not limit the rate at which the serialPrint task is run, the output buffers will overflow before its contents can be fully transmitted, resulting in text corruption. Therefore, there is a need force serialPrint to sleep in order to limit the rate at which Serial.print(…) and Serial.println(…) is called.

Question 7 (5 marks)

It is observed that the rate at which the two tasks write into the queue exceeds the rate at which the queue is being emptied. Therefore, the queue is eventually maxed out, and new data being inserted will be lost. When this happens, it is useful to observe how the tasks are being run.

Since the printing task (Tp = 50ms) has the highest priority, followed by task 1 (Tp = 5ms) and 2 (Tp = 5ms), the tasks will be run in the following order (suppose that the queue is already maxed out):

|  |  |  |
| --- | --- | --- |
| Time | Task | Comment |
| 0 | serialPrint  task1  task2 | Clears a slot in the queue  Task 1 gets the slot  Ignored |
| 5 | task1  task2 | Ignored  Ignored |
| 10 | task1  task2 | Ignored  Ignored |
| 15 | task1  task2 | Ignored  Ignored |
| 20 | task1  task2 | Ignored  Ignored |
| 25 | task1  task2 | Ignored  Ignored |
| 30 | task1  task2 | Ignored  Ignored |
| 35 | task1  task2 | Ignored  Ignored |
| 40 | task1  task2 | Ignored  Ignored |
| 45 | task1  task2 | Ignored  Ignored |
| 50 | serialPrint  task1  task2 | Clears a slot in the queue  Task 1 gets the slot  Ignored |
| 55 | task1  task2 | Ignored  Ignored |
| 60 | task1  task2 | Ignored  Ignored |

At every 50ms, all tasks would be ready, so serialPrint will be run, followed by task1 and then task2. There is a greater chance for task1 to be enqueued ahead of task2, and therefore allowing it to enqueue its data in the empty slot first.

Sidenotes: From experimentation, this problem has the most likelihood to occur whenever the sleep time of the serialPrint task is in a multiple of the task 1 and 2’s sleep time. There are some combinations where the problem never occur (for example 44ms for serialPrint). Even though setting the serialPrint’s sleep time to 50ms resulted in task 1 being able to get all the slots, setting it to 90ms actually resulted in strange patterns where we have one *task 2* for every 10 *task 1*’s printed. Setting the serialPrint’s sleep time to 80ms resulted in one *task 2* for every 4 *task 1*’s printed. The actual reason behind this may be too technical to discover, but the general idea is that **since task 1 has a higher priority, it has the highest chance to enqueue once a free slot is available**.

Question 8 (8 marks)

void OSCreateBarrier**(**unsigned int count**,** struct OSBarrier **\***barrier**)** **{**

barrier**->**count **=** count**;**

OSCreateSema**(&**barrier**->**sema**,**0**,**1**);**

**}**

This function simply initializes the barrier. The *count* of the barrier is set to the *count* passed in the parameter, and the OSCreateSema(…) function is used to initialize the semaphore to a binary semaphore with a starting value of 0. All tasks will be blocked and enqueued in the semaphore’s queue first, and then these tasks are then released until there are no more tasks in the list. A binary semaphore is used because the count of the semaphore will never be more than 0. A non-binary semaphore also works over here because it does not affect anything.

Question 9 (10 marks)

void OSReachBarrier**(**struct OSBarrier **\***barrier**)** **{**

**if** **(--**barrier**->**count**)** **{**

OSTakeSema**(&**barrier**->**sema**);**

**}**

**if** **(**procPeek**(&**barrier**->**sema**.**taskQ**)** **!=** 255**)** **{**

OSGiveSema**(&**barrier**->**sema**);**

**}**

**}**

The first condition (--barrier->count) checks if there are any more tasks that needs to reach the barrier other than the current task itself. If there are more tasks then the task is blocked using OSTakeSema**(&**barrier**->**sema**).** The condition evaluates to false only if this current task is the last task to reach the barrier.

The second condition uses procPeek to see if there are any more tasks waiting for the semaphore. If there is at least 1 task in the semaphore’s queue, OSGiveSema**(&**barrier**->**sema**)** will release it. This released task will in turn do the same and release another task waiting for the queue. This process cascades until there are no more tasks waiting for the semaphore. In actual practice, it is safe to simply omit the second condition and call OSGiveSema**(&**barrier**->**sema**)**. This will cause the final value of the semaphore to be 1 instead of 0. To save a few clock cycles, we can reduce the code to:

void OSReachBarrier**(**struct OSBarrier **\***barrier**)** **{**

**if** **(--**barrier**->**count**)** **{**

OSTakeSema**(&**barrier**->**sema**);**

**}**

OSGiveSema**(&**barrier**->**sema**);**

**}**

Question 10 (5 marks)

The tasks reach the barrier in the order 2, 1, 3 because they are forced to wake up in that order using the OSSleep(…) command.

However, whenever a task goes into blocking due to the semaphore, it is actually enqueued in the semaphore’s queue using PrioEnq(…) which enforces priority order. The tasks are also dequeued from the semaphore’s queue and woken according to their priority (highest first, i.e. lowest priority number), so the crossing message prints in that order.

A more detailed analysis is as follows: When task 3 releases a semaphore, task 2 is actually dequeued from the semaphore’s queue and placed in the ready queue. During this time, there are actually 2 tasks in the queue. OSPrioSwap(…) causes task 2 to preempt task 3 because its priority is higher. When task 2 runs, it releases task 1. At this point, task 2 will be allowed to continue (because releasing uses OSPrioSwap(…), allowing the highest priority task to continue if it is running). Finally, when task 2 goes to sleep again, task 3 will run followed by task 1 according to their priorities.

**Total: \_\_\_\_\_\_\_ / 55**