# ECE3849-D21

| Question # | Grader | Total Points | Points Received | Comments |
|---|---|---|---|---|
| #1: Reentrancy | | 23 | | |
| #2 RTOS Latency | | 11 | | |
| #2: Thread Scheduling | | 30 | | |
| #3 Semaphores and TI-RTOS objects. | | 36 | | |
| Total | | | | |

**Name: ________________________________**

# Instructions & Restrictions

## Do's:

- Read the instruction on canvas before starting the test.
- Enter your name on page 1.
- This is an open book, open notes exam.
- If asked to write C code, you may use CCS as your editor to check your syntax.
- **<u>Submit you test in pdf format to canvas by 10:15</u>** or agreed to time if accommodations have been made.
    - Points may be deducted for lateness.
    - If you experience technical difficulties, contact the Professor immediately via zoom or by cell phone 508-523-0606.
- Make sure your answers are readable on your submitted version. Points cannot be given if the answers cannot be read.
- Review your pdf submission to verify all pages are present.
- All work must represent your knowledge and all code must be written by you.
- Read each question carefully, make sure you follow the specification when implementation details are given.
- You may ask questions of the instructor via zoom or by texting or calling 508-523-0606.

## Don'ts

- You many not collaborate with or communicate with others during the exam. Not verbally or in writing. Not in-person or electronically.
- Do not lose track of time. It is more important to answer every question then to get one question 100% correct.

## Question 1: Reentrancy

A. Below is a non-reentrant function running on a 32-bit processor, for each line numbered line in the function below indicate if it is safe or not safe to use in a reentrant function. Explain for your reasoning. [12]

```
void enableTimer(uint32_t x, uint32_t *y) {
   static uint32_t timerPriority[5] = {0, 32, 64, 96, 128};

  /*Line 1 */    uint32_t iPriority = *y;

 /* Line 2 */  while (x > 4) x--;

 /* Line 3 */    if (iPriority  > timerPriority[x]) {
IntMasterDisable():
                                  TimerDisable(TIMER0_BASE, TIMER_BOTH);
                                  IntPrioritySet(INT_TIMER0A, timerPriority[x]);
 /* Line 4 */             TimerIntEnable(TIMER0_BASE, TIMER_TIMA_TIMEOUT);
                                  TimerEnable(TIMER0_BASE, TIMER_A); // comment for CPU load testing
IntMasterEnable();
   }
}
```

1. Line 1 – Safe / Not Safe, Explain.

Safe.
It performs an atomic read into a local variable. It does not matter that it maybe pointing to a global variable as it is a onetime atomic operation.

2. Line 2 – Safe / Not Safe, Explain.

Safe.
x is a local variable, it can be modified, read from, written without limit.

3. Line 3 – Safe / Not Safe, Explain.

Safe.
timerPriority is a static variable, however it is a constant. Its values are never changed and does not pose a shared data problem.  IPriority and x are also local variables.

4. Line 4 – Safe / Not Safe, Explain.

Not safe.
TimerIntEnable is not a reentrant function. Calling non-reentrant functions makes the function non-reentrant.

B. Modify the enableTimer coed in Part A to make it reentrant. [6]

See highlighted and red updates in-line with code.

C. In general, why should using pointers be avoided when writing a reentrant function? [5]

Pointers should be avoided when writing a reentrant function as they could be pointing to a global shared data variable. There is no control within the function to guarantee that the pointer is not pointing to a global shared data variable. If the function performs non-atomic operations and the data is global data then the function is non-reentrant.

## Question 2: RTOS Latency

A. Why does a “Zero Latency Interrupt” have a shorter latency than a standard Hwi? What functions can be performed in a non-zero latency Hwi that cannot be done in a “Zero Latency Interrupt”? [6]

A “Zero Latency Interrupt” bypasses the RTOS, it does not have to wait for the RTOS to receive the interrupt, prioritize and schedule it. It has the highest priority preempting everything controlled by the RTOS.

Because it bypasses the RTOS it is not allowed to engage with the RTOS and so cannot post semaphores or issue commands that would change the RTOS scheduling.

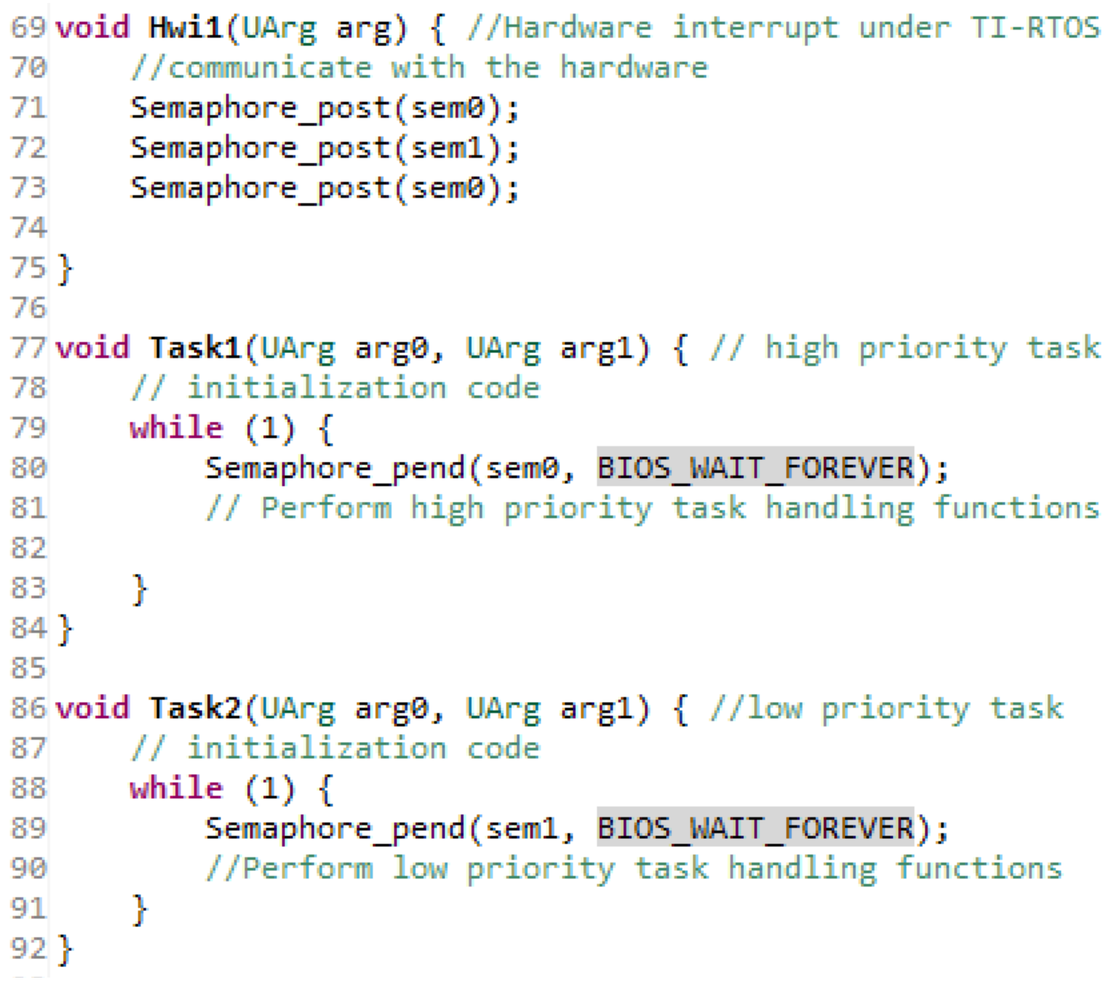
B. Which has longer latency, the lowest priority software interrupt or the highest priority task? Explain. [5]

The lowest priority software interrupt has shorter latency than the highest priority task.

The maximum task latency is the execution time of all hardware and software interrupts plus the time to schedule the task and context switch. The software interrupt only needs to wait for the execution time of all the hardware interrupts and the software interrupts with higher priority.

## Question 3: Thread Synchronization

The multitasking code below is run on a single-CPU system using the TI-RTOS. The semaphores used are counting semaphores.



```
void Hwi1(UArg arg) { //Hardware interrupt under TI-RTOS
    //communicate with the hardware
    Semaphore_post(sem0);
    Semaphore_post(sem1);
    Semaphore_post(sem0);

}

void Task1(UArg arg0, UArg arg1) { // high priority task
    // initialization code
    while (1) {
        Semaphore_pend(sem0, BIOS_WAIT_FOREVER);
        // Perform high priority task handling functions

    }
}

void Task2(UArg arg0, UArg arg1) { //low priority task
    // initialization code
    while (1) {
        Semaphore_pend(sem1, BIOS_WAIT_FOREVER);
        //Perform low priority task handling functions
    }
}
```

A. Assume that the Hwi occurs infrequently and all tasks can complete prior to the next interrupt occurring. Both Task1 and Task2 are pending on their respective semaphores when Hwi1 interrupt occurs. In the table provided on the next page, trace the execution of the above code until all tasks are blocked, requiring another interrupt to continue. <u>Semaphore and Scheduler actions should be separate steps.</u> [24]

B. If the Hwi interrupt occurred at a high rate of speed relative to the task execution time how might the execution of the tasks change. Explain. [6]

Task1 is higher priority and will always be scheduled ahead of Task2 if its semaphore is available. If the Hwi interrupt occurs faster than Task1 and/or Task2 can complete, then Task1 will take priority over Task2 and run constantly or constantly preempt Task2. Task2 could potentially be starved and may never finish.

Add lines to the table as needed.

| Step | Action | Task1 | Task2 | Sem0 | | Sem1 | |
|---|---|---|---|---|---|---|---|
| | | | | count | wait list | count | wait list |
| 0 | Start | Blocked | Blocked | 0 | Task1 | 0 | Task2 |
| 1 | Hwi posts to sem0 | Ready | Blocked | 0 | -- | 0 | Task2 |
| 2 | Hwi posts to sem1 | Ready | Ready | 0 | -- | 0 | -- |
| 3 | Hwi posts to sem0 | Ready | Ready | 1 | -- | 0 | -- |
| 4 | Scheduler runs task1 higher priority and runs. | Running | Ready | 1 | -- | 0 | -- |
| 5 | Task1 pends on sem0 and continues to run. | Running | Ready | 0 | -- | 0 | -- |
| 6 | Task1 pends on sem0. | Blocked | Ready | 0 | Task1 | 0 | -- |
| 7 | Scheduler runs task2 is ready | Blocked | Running | 0 | Task1 | 0 | -- |
| 8 | Task2 pends on sem1 | Blocked | Blocked | 0 | Task1 | 0 | Task2 |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |
| | | | | | | | |

## Question 4: Semaphores and TI-RTOS Objects

The multitasking code below is run on a single-CPU system using the TI-RTOS. The semaphores used are counting semaphores. Assume that the Hwi's occur infrequently and all tasks have sufficient time to complete before the next interrupt if they are able. Hwi0 and Hwi1 can occur anytime relative to each other.

```
void Hwi0(UArg arg) { //Hardware interrupt under TI-RTOS
  //communicate with the hardware
  Semaphore_post(sem0);
}
void Hwi1(UArg arg) { //Hardware interrupt under TI-RTOS
  //communicate with the hardware
  Semaphore_post(sem1);
  Semaphore_post(sem1);
}

void Task0(UArg arg0, UArg arg1) { // high priority task
  // initialization code
  while (1) {
        Semaphore_pend(sem0, BIOS_WAIT_FOREVER);
        // Perform high priority task handling functions
  }
}
void Task1(UArg arg0, UArg arg1) { // medium priority task
  // initialization code
  while (1) {
        Semaphore_pend(sem1, BIOS_WAIT_FOREVER);
        // Perform medium priority task handling functions
        Semaphore_pend(dataMutex, BIOS_WAIT_FOREVER);
        // Execute critical data operations
        Semaphore_post(dataMutex);
  }
}
void Task2(UArg arg0, UArg arg1) { //low priority task
  // initialization code
  while (1) {
       Semaphore_pend(sem1, BIOS_WAIT_FOREVER);
       //Perform low priority task handling functions
       Semaphore_pend(dataMutex, BIOS_WAIT_FOREVER);
       Semaphore_post(sem0);
       // Execute critical data operations
       Semaphore_post(dataMutex);
  }
}
```

**QUESTIONS ON NEXT PAGE**

A. What is deadlock? Will deadlock occur in the code in this question? Explain. [8]

Deadlock occurs when tasks are pending on each other and thus neither is able to post. They are forever blocking each other.

Deadlock does not occur in this code.

Hwi1 is called with a hardware interrupt occurs. It unblocks Task 1 and Task2 to run by posting to sem1. Hwi1 does not pend on sem1 and cannot be blocked.

Hwi0 and Task0 behave in a similar way, Hwi0 cannot be blocked as it is called in response to a hardware interrupt.

dataMutex is used to protect the critical shared data sections either Task1 or Task2 can pend on this semaphore but never both.

B. What is priority inversion? Will priority inversion occur in the code in this question? Explain. [8]

Priority inversion occurs when a low priority task is blocking a high priority task and is then preempted by a medium priority task. The medium priority task is allowed to run ahead of the high priority task. This causes the response time of the high priority task to be increased by the execution time of the medium priority task.

Priority inversion does not occur in this code.

Task 1 and Task 2 are sharing the dataMutex semaphore and can block and unblock each other as needed to share the data. Task0 is a high priority task and will preempt Task1 and Task2 when needed because they are lower priority. The high priority task is never blocked by lower priority tasks.

C. Modify the code in this question to use a GateTask object to protect the critical shared data sections in Task1 and Task2 instead of the dataMutex semaphore. [8]

```
void Task1(UArg arg0, UArg arg1) { // medium priority task
   IArg keyGateTask0;
// initialization code
   while (1) {
            Semaphore_pend(sem1, BIOS_WAIT_FOREVER);
            // Perform medium priority task handling functions
             keyGateTask0 = GateTask_enter(gateTask0);
            // Execute critical data operations
```

```
              GateTask_leave(gateTask0, keyGateTask0);
    }
}
void Task2(UArg arg0, UArg arg1) { //low priority task
    IArg keyGateTask0;
  // initialization code
    while (1) {
            Semaphore_pend(sem1, BIOS_WAIT_FOREVER);
            //Perform low priority task handling functions
           keyGateTask0 = GateTask_enter(gateTask0);
           Semaphore_post(sem0);
            // Execute critical data operations
          GateTask_leave(gateTask0, keyGateTask0);

    }
}
```

D. The code in this question uses the dataMutex semaphore to protect the critical shared data sections in Task1 and Task2. In general, name one advantage of using a semaphore for this purpose versus using the GateTask object. Name one disadvantage or possible problem that semaphores have when used to protect critical shared data sections versus the GateTask object. [6]

Semaphore Advantage:

Any one of the following or other reasonable answers are fine.

- When using a semaphore only the performance of the tasks that are using the semaphore are affected. The semaphore does not block other unrelated tasks or affect higher priority tasks.
- The latency and response time of unrelated tasks in the system are unaffected.

Semaphore disadvantage / possible problem:

Any one of the following or other reasonable answers are fine.

- Using semaphores have a higher CPU load than Gate objects because it must update state of the tasks when they are issued and then the schedule runs to determine which task should run.
- If used improperly deadlock can occur.
- If used improperly priority inversion can occur.
- Does not work with ISRs (Semaphore pends cannot be used in ISRs.)

More questions on next page.

E. In general, name one advantage of using the GateTask instead of a semaphore for protecting critical shared data sections. Name one disadvantage or possible problem using GateTask versus semaphores. [6]

GateTask Advantage:

Any one of the following or other reasonable answers are fine.

- It has a lower CPU load than semaphores.
- It does not experience deadlock.
- It does not experience priority inversion.

GateTask disadvantage / possible problem

Other reasonable answers are fine.

- Affects the latency of all tasks in the system. If the critical data section is long this can lead to significant increases in latency and potentially missed deadlines.