Module 3: Lecture Questions

Module 3 - Lecture 10 (Friday – 4/9/21)

1) What are the best practices when writing a reentrant function? What are common indicators that your function may not be reentrant?
2) If a function requires non-atomic access to shared data or calls a non-reentrant function, what must be done to make it reentrant?
3) Why is it discouraged to use pointers in a reentrant function?
4) In order for the CPU to receive and Interrupt Request what must be enabled?
5) How does the interrupt controller decide when to issue an interrupt request and which request to issue?
6) Interrupts can have three states that is stored in the IRQ memory in the interrupt controller. What are they and what causes them to transition from one state to the next?
7) Who is responsible for clearing the peripheral interrupt flag? Why is it important to do this as soon as possible?
8) What is the core functionality that every Real-Time Operating system provides?
9) What are the characteristics of a task? How does the RTOS synchronize activities between the ISRs and the tasks?
10) If I have a high priority task running and a lower priority interrupt occurs what happens?

More questions on next page.

11) (This exercise is from D. E. Simon, “An Embedded Software Primer,” Addison-Wesley Professional, 1999.) Which of the numbered lines (lines 1-5) in the following function would lead you to suspect that this function is probably not reentrant? **Explain**.

```
static int iCount;

void vNotReentrant (int x, int *p)
{
    int y;

/* Line 1 */  y = x * 2;
/* Line 2 */  ++p;
/* Line 3 */  *p = 123;
/* Line 4 */  iCount += 234;
/* Line 5 */  printf ("\nNew count: %d", x);
}
```

Line 1 is safe: y and x are local variables.

Line 2 is safe: p is a function argument, stored locally. The pointer p is not to be confused with *p, the `int` to which p is pointing.

Line 3 should be treated with caution: It is writing to whatever p is pointing to, which could be a shared variable. If this were the only shared data access in this function, and the write is atomic, this line should be safe. A write to an `int` (at least 16 bits per C standard) is atomic on 32-bit and 16-bit architectures. If the programmer is disciplined enough to never have p point to a shared variable, then this line is also safe.

Line 4 makes this function non-reentrant (on many, but not all architectures): It is performing a read-modify-write, a non-atomic operation, on a global variable.

Line 5 potentially makes this function non-reentrant: printf() accesses output hardware that is a shared resource. It is possible that printf() is reentrant in this particular C library.

> **Aside: printf() should never be called from real-time code, because its execution time is long and potentially non-deterministic. RTOSs usually supply a fast, feature-limited version of printf() that can be called from real-time code for debugging purposes.**

12) (This exercise is slightly modified from D. E. Simon, “An Embedded Software Primer,” Addison-Wesley Professional, 1999.) Where in the following code do you need to disable and re-enable interrupts to make the function reentrant? Rewrite this function to make it reentrant.

```
static int iValue;

int iFixValue (int iParm)
{
    int iTemp;

    iTemp = iValue;
    iTemp += iParm * 17;

    if (iTemp > 4922)
        iTemp = iParm;
    iValue = iTemp;

    iParm = iTemp + 179;
    if (iParm < 2000)
        return 1;
    else
        return 0;
}
```

```
static int iValue;           // global variable

int iFixValue (int iParm)
{
    int iTemp;

    IntMasterDisable();
    iTemp = iValue;          // read global
    iTemp += iParm * 17;

    if (iTemp > 4922)
        iTemp = iParm;
    iValue = iTemp;          // write to global
    IntMasterEnable();

    iParm = iTemp + 179;
    if (iParm < 2000)
        return 1;
    else
        return 0;
}
```

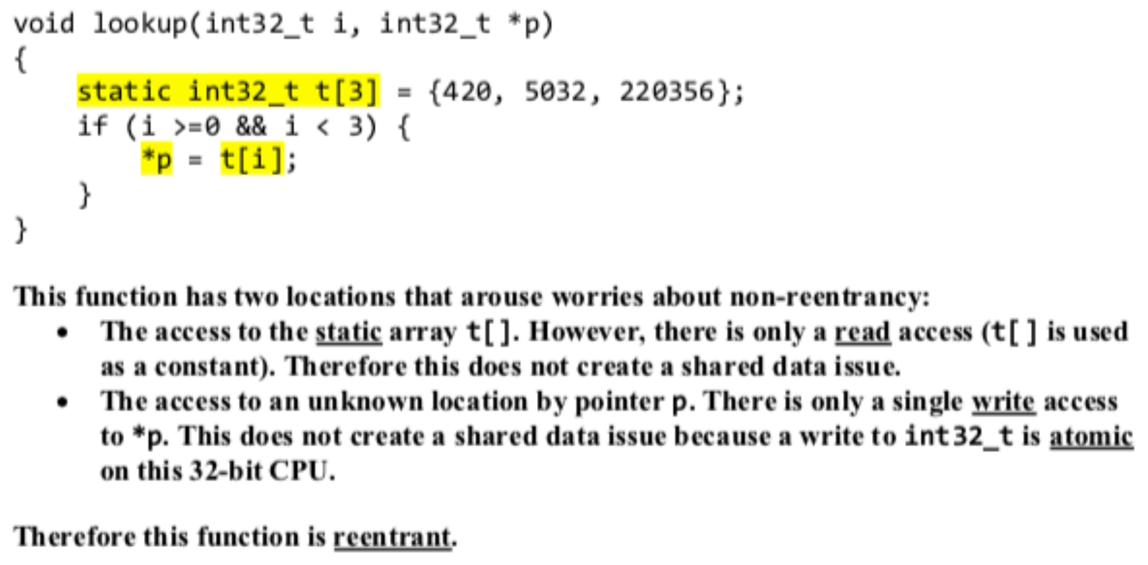
The code that updates `iValue` is a read-modify-write operation that makes one continuous critical section. The rest of the function only operates on local variables (including function arguments), so does not need to be protected.

13) The function `lookup()` is to be used by multiple threads in a preemptive multitasking system. This code runs on a **32-bit** CPU: 32-bit reads and writes are atomic. Is this function reentrant, non-reentrant, or reentrant under certain conditions? **Label** and **explain** all the locations in `lookup()` that cause you to suspect non-reentrancy.

```
void lookup(int32_t i, int32_t *p)
{
    static int32_t t[3] = {420, 5032, 220356};
    if (i >=0 && i < 3) {

*p = t[i]; }

}
```



```
void lookup(int32_t i, int32_t *p)
{
    static int32_t t[3] = {420, 5032, 220356};
    if (i >=0 && i < 3) {
        *p = t[i];
    }
}
```

This function has two locations that arouse worries about non-reentrancy:

- **The access to the static array `t[]`. However, there is only a read access (`t[]` is used as a constant). Therefore this does not create a shared data issue.**
- **The access to an unknown location by pointer `p`. There is only a single write access to `*p`. This does not create a shared data issue because a write to `int32_t` is atomic on this 32-bit CPU.**

Therefore this function is reentrant.

Module 3 - Lecture 11 (Tuesday – 4/13/21)

14) What is RTOS features are universally supported?
15) What are the characteristics of a task?
16) Sketch the task state transition diagram and explain how each task state is entered and exited. How does the RTOS scheduler decide which task is run?
17) What task is run if no threads are running? What is commonly done in this task?
18) What is a "zero-latency interrupt"? How is it different from a RTOS controlled Hwi Interrupt?
19) How much does the TI-RTOS affect the latency of ISRs / Hwi interrupts?
20) What is the maximum latency of the highest priority task? Why is it greater than the latency of the highest priority interrupt?
21) What is a Swi? How is the same as an Hwi? How is it different from a Hwi?
22) Order the following threads from highest priority to lowest priority.

    Hwi with a priority of 0
    Task with a priority of 0
    Swi with a priority of 1
    Hwi with a priority of 64
    Task with a priority of 15
    Swi with a priority of 10,

Module 3 - Lecture 12 (Thursday– 4/15/21)

23) When you configuring a semaphore in CCS, you can select the type of semaphore you want implemented. What are the options? What are the functional differences between the options?
24) In lecture there were four common semaphore problems listed, for the cases below which error was most likely made by the embedded designer?
    a) The system is reporting frequently that it is missing its relative deadlines, even though the RMS scheduling shows there is plenty of timing margin.
    b) The system has been running for two weeks and just hangs, requiring a power cycle to get running again.
    c) The customer is reporting data errors as well as intermittent hangs requiring power cycling.
    d) The system is responsible for sampling an analog waveform and displaying it on an LCD screen. The waveform is supposed to be a sine wave but every once in a while, it is discontinuous like it is sampling the wrong data.
25) Name one advantage and one disadvantage that semaphores have over using Hwi / ISR / interrupts.
26) What is deadlock? What affect does it have on the operation of your tasks? How can it be avoided in your code?
27) When does priority inversion occur? How does it affect system performance? How does using the GateMutexPri Object instead of a Semaphore object resolve priority inversion?

28) Trace the execution of the following multithreaded code on a single-CPU system. The tasks Task1 and Task2 start in the Ready state at the beginning of their task functions. A **single Hwi0 interrupt** occurs at the point specified in the comments. Stop when all tasks block, requiring action beyond this setup to unblock them. Complete the table below, numbering all steps in the code. Log the actions that modify the Task and/or Semaphore state and the state after each action. Include each scheduler execution as a separate step.

```
void Hwi0(UArg arg) {  // Hwi0 interrupt service routine
    <clear interrupt flag>;
    Semaphore_post(semTask1);
}

void Task1(UArg arg0, UArg arg1) {  // high priority task
①   while (1) {
②       Semaphore_pend(semTask1, BIOS_WAIT_FOREVER);

        Semaphore_pend(sem0, BIOS_WAIT_FOREVER);
        // critical section
        Semaphore_post(sem0);
    }
}

void Task2(UArg arg0, UArg arg1) {  // low priority task
③   while (1) {
        Semaphore_pend(semTask2, BIOS_WAIT_FOREVER);

        Semaphore_pend(sem0, BIOS_WAIT_FOREVER);
        // critical section <- Hwi0 interrupt occurs once
        Semaphore_post(sem0);
    }
}
```

Step	Action	Task1	Task2	semTask1		semTask2		sem0	
				count	list	count	list	count	list
	Start	Ready	Ready	0	-	1	-	1	-
①	Scheduler runs	Running	Ready	0	-	1	-	1	-
②	Task1 pends on semTask1	Blocked	Ready	0	Task1	1	-	1	-
③	Scheduler runs	Blocked	Running	0	Task1	1	-	1	-

(add more rows as needed)

Answers on next page.

```
void Hwi0(UArg arg) {  // Hwi0 interrupt service routine
      <clear interrupt flag>;
⑥     Semaphore_post(semTask1);
} // <- ⑦ scheduler runs after Hwi0 returns

void Task1(UArg arg0, UArg arg1) {  // high priority task
①  while (1) {
②⑦⑬    Semaphore_pend(semTask1, BIOS_WAIT_FOREVER);

⑧⑪      Semaphore_pend(sem0, BIOS_WAIT_FOREVER);
        // critical section
⑫       Semaphore_post(sem0);
    }
}

void Task2(UArg arg0, UArg arg1) {  // low priority task
③  while (1) {
④⑮      Semaphore_pend(semTask2, BIOS_WAIT_FOREVER);

⑤       Semaphore_pend(sem0, BIOS_WAIT_FOREVER);
⑥⑨      // critical section <- Hwi0 interrupt occurs once
⑩⑭      Semaphore_post(sem0);
    }
}
```

State changes are highlighted:

				semTask1		semTask2		sem0	
Step	Action	Task1	Task2	count	list	count	list	count	list
	Start	Ready	Ready	0	-	1	-	1	-
①	Scheduler runs	Running	Ready	0	-	1	-	1	-
②	Task1 pends on semTask1	Blocked	Ready	0	Task1	1	-	1	-
③	Scheduler runs	Blocked	Running	0	Task1	1	-	1	-
④	**Task2 pends on semTask2**	**Blocked**	**Running**	**0**	**Task1**	**0**	**-**	**1**	**-**
⑤	**Task2 pends on sem0**	**Blocked**	**Running**	**0**	**Task1**	**0**	**-**	**0**	**-**
⑥	**Hwi0 interrupts, posts to semTask1**	**Ready**	**Running (Hwi)**	**0**	**-**	**0**	**-**	**0**	**-**
⑦	**Scheduler runs after Hwi0 returns**	**Running**	**Ready**	**0**	**-**	**0**	**-**	**0**	**-**
⑧	**Task1 pends on sem0**	**Blocked**	**Ready**	**0**	**-**	**0**	**-**	**0**	**Task1**
⑨	**Scheduler runs**	**Blocked**	**Running**	**0**	**-**	**0**	**-**	**0**	**Task1**
⑩	**Task2 posts to sem0**	**Ready**	**Running**	**0**	**-**	**0**	**-**	**0**	**-**
⑪	**Scheduler runs**	**Running**	**Ready**	**0**	**-**	**0**	**-**	**0**	**-**
⑫	**Task1 posts to sem0**	**Running**	**Ready**	**0**	**-**	**0**	**-**	**1**	**-**
⑬	**Task1 pends on semTask1**	**Blocked**	**Ready**	**0**	**Task1**	**0**	**-**	**1**	**-**
⑭	**Scheduler runs**	**Blocked**	**Running**	**0**	**Task1**	**0**	**-**	**1**	**-**
⑮	**Task2 pends on semTask2**	**Blocked**	**Blocked**	**0**	**Task1**	**0**	**Task2**	**1**	**-**

29) Trace the execution of the following multitasking code on a single-CPU system until all tasks block, requiring action beyond this setup to unblock. The tasks start execution at the points labeled in the comments. A **single Hwi0 interrupt** occurs during the first execution of `<task2 code>`, as indicated in the comments. Fill in the table on the next page (you may leave table cells blank if unchanged from the previous row). The initial task and semaphore state are in the first row of the table. The semaphores are counting, and unblock tasks in FIFO order. Semaphore actions, scheduler actions and each execution of `<task1 code>` and `<task2 code>` should be separate steps.

```
void Hwi0(UArg arg) {  // Hwi0 interrupt service routine
    <clear interrupt flag>;
    Semaphore_post(sem1);
}

void task1(UArg arg0, UArg arg1) {  // high priority
    // <- start
    while (1) {
        Semaphore_pend(sem1, BIOS_WAIT_FOREVER);
        Semaphore_post(sem2);
        <task1 code>;
    }
}

void task2(UArg arg0, UArg arg1) {  // low priority
    // <- start
    while (1) {
        <task2 code>; // <-Hwi0 interrupts during first loop iteration
        Semaphore_pend(sem2, BIOS_WAIT_FOREVER);
    }
}
```

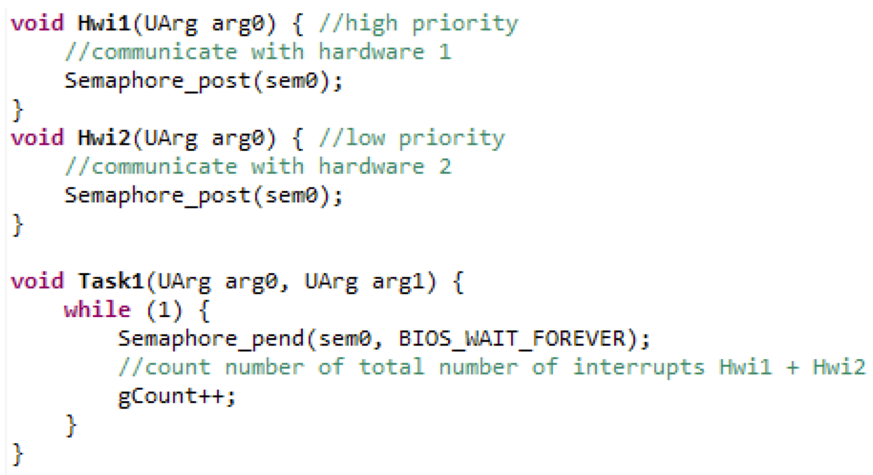
Add lines to the table as needed.

Step	Action	task1	task2	sem1		sem2	
				count	wait list	count	wait list
0	Start	Ready	Ready	0	-	0	-

				sem1		sem2	
Step	Action	task1	task2	count	wait list	count	wait list
0	Start	Ready	Ready	0	-	0	-
1	**scheduler runs**	**Running**	**Ready**				
2	**task1 pends on sem1**	**Blocked**	🟨		**task1**		
3	**scheduler runs**	🟥	**Running**		🟥		
4	**<task2 code> starts running**	🟥	🟩		🟥		
5	**Hwi0 interrupts, posts to sem1**	**Ready**	🟩		-		
6	**scheduler runs when Hwi0 exits**	**Running**	**Ready**				
7	**task1 posts to sem2**	🟩	🟨			**1**	
8	**<task1 code> runs**	🟩	🟨			🟩	
9	**task1 pends on sem1**	**Blocked**	🟨		**task1**	🟩	
10	**scheduler runs**	🟥	**Running**		🟥	🟩	
11	**<task2 code> finishes**	🟥	🟩		🟥	🟩	
12	**task2 pends on sem2**	🟥	🟩		🟥	**0**	
13	**<task2 code> runs**	🟥	🟩		🟥		
14	**task2 pends on sem2**	🟥	**Blocked**		🟥		**task2**

Module 3 - Lecture 13 (Friday– 4/16/21)

30) Under what conditions is the GateMutexPri object unable to resolve priority inversion? For these cases what is the alternative solution? Given the alternative solution will always work, why not just use it all the time?
31) How is GateMutex different from a Semaphore? Why is the GateMutex object not suitable for synchronization of tasks?
32) Which RTOS objects are susceptible to deadlock? Which are susceptible to priority inversion?
33) What kind of Semaphore do you want to use in the following application to guarantee all hardware interrupts are counted? Explain.



```
void Hwi1(UArg arg0) { //high priority
    //communicate with hardware 1
    Semaphore_post(sem0);
}
void Hwi2(UArg arg0) { //low priority
    //communicate with hardware 2
    Semaphore_post(sem0);
}

void Task1(UArg arg0, UArg arg1) {
    while (1) {
        Semaphore_pend(sem0, BIOS_WAIT_FOREVER);
        //count number of total number of interrupts Hwi1 + Hwi2
        gCount++;
    }
}
```

34) In lecture, we looked at the execution of the below code assuming that Hwi happened at a slow enough rate that both task1 and task2 completed before Hwi posted again.

a) What would happen if Hwi1 occurred more frequently and posted to semaphore sem0 while task2 was still executing its other code?

b) If you wanted to guarantee that task1 and task2 each ran once every time the Hwi1 occurred. What would the minimum Hwi period need to be?

```
void Hwi1(UArg arg) {  // hardware interrupt under TI-RTOS
  // communicate with the hardware
  Semaphore_post(sem0);
  Semaphore_post(sem0);
}
void Task1(UArg arg0, UArg arg1) {  // high priority
  // init
  while (1) {
    Semaphore_pend(sem0, BIOS_WAIT_FOREVER);
    // other code
  }
}
void Task2(UArg arg0, UArg arg1) {  // low priority
  // init
  while (1) {
    Semaphore_pend(sem0, BIOS_WAIT_FOREVER);
    // other code
  }
}
```

More questions on next page

35) The following Tasks share two global variables. **Explain your answers.**

`sem_a` and `sem_b` are semaphores initialized to count = 1.
`semTask1` and `semTask2` are semaphores initialized count = 0, and posted by ISRs.

```
int a, b;  // shared globals

void Task1(UArg arg0, UArg arg1) {  // high priority
    while (1) {
        Semaphore_pend(semTask1, BIOS_WAIT_FOREVER);
        ...
        Semaphore_pend(sem_a, BIOS_WAIT_FOREVER);
        Semaphore_pend(sem_b, BIOS_WAIT_FOREVER);
        a = b;
        Semaphore_post(sem_b);
        Semaphore_post(sem_a);
        ...
    }
}

void Task2(UArg arg0, UArg arg1) {  // low priority
    while (1) {
        Semaphore_pend(semTask2, BIOS_WAIT_FOREVER);
        ...
        Semaphore_pend(sem_a, BIOS_WAIT_FOREVER);
        Semaphore_pend(sem_b, BIOS_WAIT_FOREVER);
        b = a;
        Semaphore_post(sem_a);
        Semaphore_post(sem_b);
        ...
    }
}
```

a) [10 pts] Can deadlock occur in this case? Assume ISRs can post to `semTask1` and `semTask2` at any time.

No. The `sem_a` and `sem_b` semaphore pends are in the same order in both Tasks. The order of the posts does not matter.

b) [10 pts] Can priority inversion occur in this case? Assume other Tasks of various priorities exist in the system.

Yes. If Task1 blocks on any of the two semaphores held by Task2, a medium priority Task can preempt Task2 and run to completion. Regular semaphores do not protect from priority inversion.

c) [10 pts] Which TI-RTOS Gate objects would protect this code from priority inversion? Assume no other Tasks access the globals a and b.

- GateMutexPri
- GateTask
- GateSwi
- GateHwi

All gate objects except GateMutex (which is based on a regular semaphore) protect from preemption by medium-priority Tasks. GateMutexPri does this through priority inheritance. The rest disable Task scheduling, so preemption by another Task is not possible.

d) [10 pts] Which TI-RTOS Gate object would be the most appropriate for protecting these critical sections? Your goal is to minimize negative impact on the schedule of the real-time system.

Either GateMutexPri or GateTask can be used to protect critical sections are short and occur only in Tasks.

GateTask can be used to protect critical sections are short and occur only in Tasks. Turning off Task scheduling is a simple operation, so the CPU load is not increased as much as with semaphore-based gates. However, Task latency is increased for all tasks.

GateMutexPri is also a good choice as it does not affect the latency of the tasks, but would result in a slightly higher CPU load as it is semaphore based.

Other gates are not optimal because they either affect the latency of Swi/Hwi unnecessary (GateSwi and GateHwi), or do not protect against priority inversion (GateMutex).

e) [10 pts] If the critical section were longer, e.g. accessing a shared array with 1000 entries, would your selection of the appropriate TI-RTOS Gate object change? If so, to which Gate object?

GateMutexPri becomes the most appropriate because the critical sections are long (accessing 1000 entries is more than CPU 1000 cycles) and occur only in Tasks. The execution time of the critical sections will not affect the response time of other threads (including Tasks) that do not access this shared resource. The CPU load will only incrementally increase.

If we keep GateTask, the increase in Task latency due to the long critical sections would likely be too high.

More questions on next page

36) You have two TI-RTOS Tasks (`task1` and `task3`) sharing a global array `g[]`, and another unrelated task of intermediate priority. Assume ISRs periodically post to `semTask1`, `semTask2` and `semTask3`. All Semaphores in this example are binary, initialized to 0.

```
int32_t g[1000];  // shared global

void task1(UArg arg0, UArg arg1) {  // high priority
  int32_t b[1000];


  while (1) {
    Semaphore_pend(semTask1, BIOS_WAIT_FOREVER);


    memcpy(&b[0], &g[0], 1000*sizeof(int32_t)); // critical section 1


    <other code, not using g[]>;
  }
}

void task2(UArg arg0, UArg arg1) {  // mid priority
  while (1) {
    Semaphore_pend(semTask2, BIOS_WAIT_FOREVER);
    <other code unrelated to task1 and task3>;
  }
}

void task3(UArg arg0, UArg arg1) {  // low priority
  IArg key;
  int32_t i;
  while (1) {
    Semaphore_pend(semTask3, BIOS_WAIT_FOREVER);
    <other code, not using g[]>;
    key = GateTask_enter(gate1);


    for (i = 0; i < 1000; i++) {    // critical section 2
      g[i]++;
    }
    GateTask_leave(gate1, key);


  }
}
```

Solution part c

```
int32_t g[1000];  // shared global

void task1(UArg arg0, UArg arg1) {  // high priority
  int32_t b[1000];
  IArg key;

  while (1) {
    Semaphore_pend(semTask1, BIOS_WAIT_FOREVER);

    key = GateMutexPri_enter(gate2);
    memcpy(&b[0], &g[0], 1000*sizeof(int32_t)); // critical section 1
    GateMutexPri_leave(gate2, key);

    <other code, not using g[]>;
  }
}

void task2(UArg arg0, UArg arg1) {  // mid priority
  while (1) {
    Semaphore_pend(semTask2, BIOS_WAIT_FOREVER);
    <other code unrelated to task1 and task3>;
  }
}

void task3(UArg arg0, UArg arg1) {  // low priority
  IArg key;
  int32_t i;
  while (1) {
    Semaphore_pend(semTask3, BIOS_WAIT_FOREVER);
    <other code, not using g[]>;
    ~~key = GateTask_enter(gate1);~~

    key = GateMutexPri_enter(gate2);
    for (i = 0; i < 1000; i++) {    // critical section 2
      g[i]++;
    }
    ~~GateTask_leave(gate1, key);~~

    GateMutexPri_leave(gate2, key);
  }
}
```

a) Check the box for each issue that occurs in the given code. Briefly **explain** your answer, **why** or **why not**, in each case.

☐ Shared data bug

No, there are no shared data bugs here. Critical section 2 in task3 cannot be preempted by task1 because GateTask disables task scheduling. Critical Section 1 in task1 cannot be preempted by task3 because task1 is higher priority.

☐ Priority inversion

No, priority inversion does not occur because GateTask disables task scheduling in the low-priority task3. This prevents task2 from preempting Critical Section 2 while task1 is waiting for Critical Section 2 to finish.

☐ Deadlock

No, deadlock does not occur here. There are no situations where a Task needs more than one semaphore to enter a critical section.

b) Assume other Tasks of even higher priority exist in this system, but only `task1` and `task3` share `g[]`. What additional concern does this raise?

Because Critical Section 2 is relatively long (accessing a 1000-element array), and disables task scheduling, this raises concern over the latency of Tasks that are higher priority than task1. This latency is increased by the duration of Critical Section 2.

c) Assuming the setup of part (b), which of the following TI-RTOS objects do you think is the most appropriate for protecting the given critical sections? **Insert** any additional code needed into the blank spaces between the lines on the previous page, crossing out anything unnecessary.

☐ Semaphore
☐ GateMutex
☐ GateMutexPri
☐ GateTask
☐ GateSwi
☐ GateHwi

☐ Semaphore
☐ GateMutex
☑ GateMutexPri
☐ GateTask
☐ GateSwi
☐ GateHwi

GateMutexPri is similar to a Semaphore in the sense that it is targeted: the duration of the critical section does not affect the latency or response time of higher priority threads that do not need this gate. It is protected from priority inversion using priority inheritance. However, it uses slightly more CPU time. This is not a large issue because the critical sections are relatively long. (This explanation is not required for credit.)