This exam is open book, open notes. Calculators are allowed. Internet access is not permitted. Show all work for full credit. Partial credit is also assigned.

Semaphores

1) 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 after 50 ms, enough time for both tasks to block. 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 is in the first row of the table. The semaphores are binary, and unblock tasks in FIFO order. Semaphore actions, scheduler actions and each execution of `<task1 code>` and `<task2 code>` should be separate steps. You do not need to label the steps in the code.

```
void Hwi0(UArg arg) {  // Hwi0 interrupt service routine
    <clear interrupt flag>;
    Semaphore_post(sem1);
    Semaphore_post(sem2);
}

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

void task2(UArg arg0, UArg arg1) {  // low priority
    // <- start
    while (1) {
        Semaphore_pend(sem2, BIOS_WAIT_FOREVER);
        <task2 code>;
        Semaphore_post(sem1);
    }
}
```

Step	Action	task1	task2	sem1		sem2	
				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 pends on sem2**		**Blocked**				**task2**
5	**scheduler selects the Idle thread to run**						
6	**Hwi0 runs after 50 ms, posts to sem1**	**Ready**			**-**		
7	**Hwi0 posts to sem2**		**Ready**				**-**
8	**scheduler runs when Hwi0 exits**	**Running**					
9	**<task1 code> runs**						
10	**task1 pends on sem2**	**Blocked**					**task1**
11	**scheduler runs**		**Running**				
12	**<task2 code> runs**						
13	**task2 posts to sem1**			**1**			
14	**task2 pends on sem2**		**Blocked**				**task1, task2**

Semaphores and mailboxes

2) Implement a single-slot mailbox using only global variables and TI-RTOS Semaphores. The mailbox should have a capacity of **one item** of the type `int32_t`. You are not allowed to use the TI-RTOS Mailbox object. The function Mbox_send() should block if the mailbox is full. The function Mbox_receive() should block if the mailbox is empty.

   a) List all semaphores and their initial count.

Semaphore	Initial count
Mbox_full	0
Mbox_empty	1

   b) Declare global variables.

```
int32_t Mbox_data;
```

   c) Implement Mbox_send(). This function must block if the mailbox is full.

```
void Mbox_send(int32_t data) { // place data into the mailbox
    Semaphore_pend(Mbox_empty, BIOS_WAIT_FOREVER);
    Mbox_data = data;
    Semaphore_post(Mbox_full);
}
```

(problem continues on the next page)

d) Implement Mbox_receive() that returns the data received from the mailbox. This function must block if the mailbox is empty.

```
int32_t Mbox_receive(void) { // retrieve data from the mailbox

    int32_t temp;

    Semaphore_pend(Mbox_full, BIOS_WAIT_FOREVER);

    temp = Mbox_data;

    Semaphore_post(Mbox_empty);

    return temp;

}
```

Critical sections, TI-RTOS gates

3) You have two TI-RTOS Tasks (`task1` and `task3`) sharing a global array `a[]`, and another unrelated task of intermediate priority. Assume ISRs periodically post to `semTask1`, `semTask2` and `semTask3`. The initial semaphore settings are in the following table.

Semaphore	Initial count	Type	Unblock order
sem1	1	binary	FIFO
semTask1	0	binary	FIFO
semTask2	0	binary	FIFO
semTask3	0	binary	FIFO

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

void task1(UArg arg0, UArg arg1) {  // highest priority
  int32_t b[1000];
  while (1) {
    Semaphore_pend(semTask1, BIOS_WAIT_FOREVER);
    Semaphore_pend(sem1, BIOS_WAIT_FOREVER); // <- replace with enter
    memcpy(&b[0], &a[0], 1000*sizeof(int32_t)); // critical section 1
    Semaphore_post(sem1);                       // <- replace with leave
    <other code, not using a[]>;
  }
}

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
  int32_t i;
  while (1) {
    Semaphore_pend(semTask3, BIOS_WAIT_FOREVER);
    <other code, not using a[]>;
    Semaphore_pend(sem1, BIOS_WAIT_FOREVER); // <- replace with enter
    for (i = 0; i < 1000; i++) {    // critical section 2
      a[i]++;
    }
    Semaphore_post(sem1);                       // <- replace with leave
  }
}
```

(problem continues on the next page)

a) Which of the following issues occurs in this code? Indicate **which task**(s) is/are affected and **in what way**.

☐ Forgetting to take a semaphore
☐ Forgetting to release a semaphore
☑ Priority inversion
☐ Deadlock

`task1` is affected by priority inversion, caused by the semaphore `sem1`. If critical section 2 is preempted by both `task1` and `task2`, `task1` will block on `sem1`, and `task2` will be able to run to completion. Effectively, `task2` is higher priority than `task1` during that critical section, contrary to the assigned priorities.

`task1` (max) response time is increased by the execution time of `task2` (and also by the duration of Critical Section 2). `task1` latency is unaffected, however.

b) Which TI-RTOS Gate object fixes the problem found in part (a) without introducing new problems? Explain. Assume other tasks of even higher priority may exist in this system, but only `task1` and `task3` share `a[]`. On the previous page, **indicate** the locations of the gate `enter` and `leave` calls.

☐ GateMutex
☑ GateMutexPri
☐ GateTask
☐ GateSwi
☐ GateHwi
☐ GateAll

GateMutexPri is a semaphore-based object that implements priority inheritance. If `sem1` is replaced by this gate, the priority inversion problem found in part (a) is eliminated.

Other objects that also eliminate priority inversion, such as GateTask, disable the task scheduler. Since the critical section is relatively long (accessing a 1000-element array), the latency introduced to other high-priority tasks by disabling the scheduler is a significant issue. GateMutexPri does not have his problem, and is very similar to a Semaphore in performance.

The locations of GateMutexPri `enter` and `leave` calls is highlighted on the previous page.

Real-time debugging

4) Short answers.

a) You are trying to **measure** the **response time** of an ISR. This ISR is triggered by a rising edge of an external signal. Outline your approach.

1. **Toggle a GPIO output at the end of the ISR.**
2. **Exercise all code paths in the system that may affect the latency of this ISR.**
3. **Monitor both the external trigger signal and the GPIO output using an oscilloscope in infinite persist mode.**
    a. **Trigger the oscilloscope of the rising edge that triggers the ISR.**
    b. **Measure the max time delay from the trigger to the GPIO edge.**

b) Suppose your highest priority TI-RTOS Task is not meeting deadlines because its latency is too high. What other parts of the system could be optimized to improve this Task's latency? List all the possibilities you can think of without any further information about this system.

1. **All Hwi.**
2. **All Swi.**
3. **Critical sections that disable Task, Swi, or Hwi scheduling. The longest such critical section must be optimized first.**

Note: Critical sections that use semaphore and derivative objects do not affect the latency of the highest priority task. They can still affect the response time if the task can block on one of these objects.