Semaphores

1) [35 pts] 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.

```
void Hwi0(UArg arg) {  // Hwi0 interrupt service routine
     <clear interrupt flag>;
⑦    Semaphore_post(semTask1);
} // <- ⑧ scheduler runs after Hwi0 returns

void Task1(UArg arg0, UArg arg1) {  // high priority task
①  while (1) {
②⑧⑭  Semaphore_pend(semTask1, BIOS_WAIT_FOREVER);

⑨⑫    Semaphore_pend(sem0, BIOS_WAIT_FOREVER);
      // critical section
⑬     Semaphore_post(sem0);
    }
}

void Task2(UArg arg0, UArg arg1) {  // low priority task
③  while (1) {
④⑯    Semaphore_pend(semTask2, BIOS_WAIT_FOREVER);

⑤     Semaphore_pend(sem0, BIOS_WAIT_FOREVER);
⑥⑩    // critical section <- Hwi0 interrupt occurs once
⑪⑮    Semaphore_post(sem0);
    }
}
```

State changes are highlighted:

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	-
④	**Task2 pends on semTask2**	**Blocked**	**Running**	**0**	**Task1**	**0**	**-**	**1**	**-**
⑤	**Task2 pends on sem0**	**Blocked**	**Running**	**0**	**Task1**	**0**	**-**	**0**	**-**
⑥	**Hwi0 preempts Task2**	**Blocked**	**Running (Hwi)**	**0**	**Task1**	**0**	**-**	**0**	**-**
⑦	**Hwi0 posts to semTask1**	**Ready**	**Running (Hwi)**	**0**	**-**	**0**	**-**	**0**	**-**
⑧	**Scheduler runs after Hwi0 returns**	**Running**	**Ready**	**0**	**-**	**0**	**-**	**0**	**-**
⑨	**Task1 pends on sem0**	**Blocked**	**Ready**	**0**	**-**	**0**	**-**	**0**	**Task1**
⑩	**Scheduler runs**	**Blocked**	**Running**	**0**	**-**	**0**	**-**	**0**	**Task1**
⑪	**Task2 posts to sem0**	**Ready**	**Running**	**0**	**-**	**0**	**-**	**0**	**-**
⑫	**Scheduler runs**	**Running**	**Ready**	**0**	**-**	**0**	**-**	**0**	**-**
⑬	**Task1 posts to sem0**	**Running**	**Ready**	**0**	**-**	**0**	**-**	**1**	**-**
⑭	**Task1 pends on semTask1**	**Blocked**	**Ready**	**0**	**Task1**	**0**	**-**	**1**	**-**
⑮	**Scheduler runs**	**Blocked**	**Running**	**0**	**Task1**	**0**	**-**	**1**	**-**
⑯	**Task2 pends on semTask2**	**Blocked**	**Blocked**	**0**	**Task1**	**0**	**Task2**	**1**	**-**

Grading note: The number of steps does not need to match exactly.

Deadlock, priority inversion, TI-RTOS Gates

2) [35 pts] 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) [7 pts] Can deadlock occur in this case? Assume ISRs can post to `semTask1` and `semTask2` at any time.

<u>No</u>. 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) [7 pts] Can priority inversion occur in this case? Assume other Tasks of various priorities exist in the system.

<u>Yes</u>. 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) [7 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**
- **GateAll**

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) [7 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.

GateTask should be the most appropriate because the critical sections are short and occur only in Tasks. Turning off Task scheduling is a simple operation, so the CPU utilization is not increased as much as with semaphore-based gates. The impact on Task latency is minimal due to short critical sections. Finally, preemption by Hwi and Swi threads is still allowed.

GateMutexPri is also a reasonable choice, but would result in higher CPU utilization. (GateMutexPri is acceptable for full credit.)

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

e) [7 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 utilization will only incrementally increase.

Mailboxes and Events

3) [30 pts] The following Task uses a TI-RTOS Mailbox object to handle multiple event types.

```
void Task1(UArg arg0, UArg arg1) {
    int msg;
    while (1) {
        Mailbox_pend(mailbox1, &msg, BIOS_WAIT_FOREVER);
        switch(msg) {
            case 0:
                // handle event 0
                break;
            case 1:
                // handle event 1
                break;
        }
    }
}
```

a) [6 pts] Write a code fragment that signals Task1 that "event 1" (referring to the comments) has occurred.

```
int msg;
msg = 1;  // specify event 1
Mailbox_post(mailbox1, &msg, BIOS_NO_WAIT);
```

It is best to use the BIOS_NO_WAIT timeout to make this call non-blocking, just like posting to a Semaphore or Event. (This is not required to receive full credit.)

b) [24 pts] Rewrite Task1, replacing the Mailbox with a TI-RTOS Event object to achieve approximately the same functionality. Also write replacement code for part (a). Refer to the SYS/BIOS (TI-RTOS Kernel) User's Guide (link available on Canvas under Pages/Datasheets) and CCS online documentation for detailed information about Event objects.

Handling multiple events through Event objects is different in that more than one event may need to be handled per loop iteration. The return value of Event_pend() needs to be examined to determine which events to handle. We cannot use a single switch() statement here.

Create an Event object called `event1`.

```
void Task1(UArg arg0, UArg arg1) {
    UInt posted;
    while (1) {
        posted = Event_pend(event1, Event_Id_NONE /* 0 also OK */,
                        Event_Id_00 | Event_Id_01, BIOS_WAIT_FOREVER);

        if (posted & Event_Id_00) {
            // handle event 0
        }

        if (posted & Event_Id_01) {
            // handle event 1
        }
    }
}
```

Replacement for part (a):

```
Event_post(event1, Event_Id_01);
```