ECE3849
D-Term 2021

Real Time Embedded Systems

Module 3 Part 3

Module 3 Part 3 Overview

- Deadlock.
- Task Priority Inversion.
- Gates for mutual exclusion.
- Semaphores for signaling and synchronization.

Deadlock

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

Let sem_a and sem_b be two semaphores initialized to count = 1 before the tasks

sem_a guards the global variable a
sem_b guards the global variable b

```
int a;
int b;

void task1(UArg arg0, UArg arg1) { // high priority
  // init
  while (1) {
    ...
    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
  // init
  while (1) {
    ...
    Semaphore_pend(sem_b, BIOS_WAIT_FOREVER);
    Semaphore_pend(sem_a, BIOS_WAIT_FOREVER);
    b = a;
    Semaphore_post(sem_a);
    Semaphore_post(sem_b);
    ...
  }
}
```

0 task 1 is blocked waiting for ISR.

3 4 1 2 5

ISR unblocks task 1

Both tasks are blocked, waiting for the other to post.

0 System initialization.
- sem_a & sem_b, count = 1.
- task1 is blocked waiting on ISR.

1 task2 starts and pends on sem_b, countB = 1 so it can run.
- countB decremented, countB = 0.
- Task2 continues to run.

2 ISR unblocks task1 (high priority).
- task2 is placed in ready state.
- task1 starts running.

3 task1 pends on sem_a, countA = 1 so it can run.
- countA decremented, countA = 0.
- task1 continues.

4 task1 pends on sem_b, countB = 0.
- task1 is blocked.
- task2 starts running.

5 task2 pends on sem_a, countA = 0.
- task2 is blocked.

Ece3849_rtos_latency: Deadlock

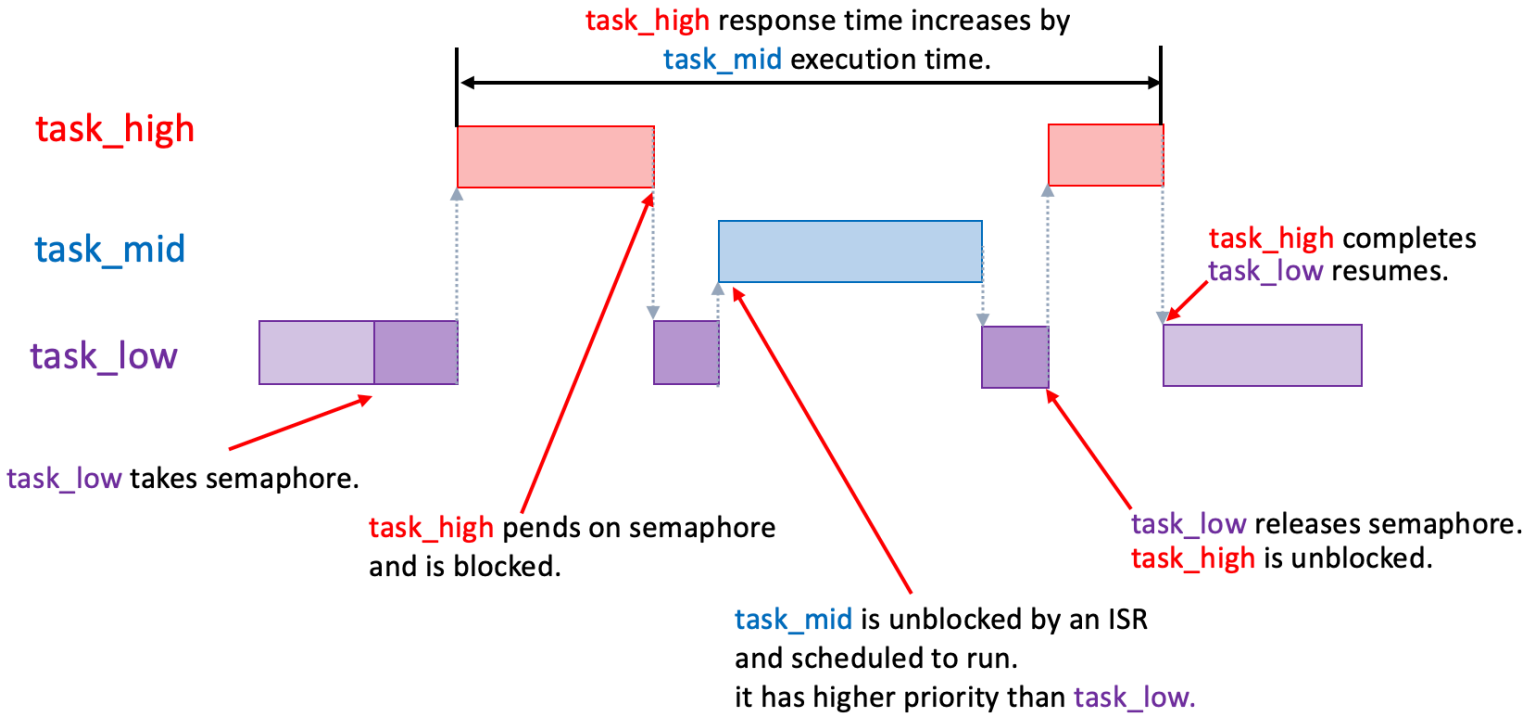
- Ece3849_rtos_latency – deadlock simulation mode.
  - #define ENABLE_DEADLOCK 1
  - Adds the code from the deadlock on previous page, to event0_func and event2_func.
  - Event1_func is left unchanged
  - We observe the event counters to verify which events are still running.
    - Event1 continues to run for the whole test.
    - Event0 and 2 stop incrementing almost immediate, showing they are deadlocked.
- If you are using two semaphores they should always pend in the same order in both tasks.
  - We change Event 2 so that both events pend in the same order

```
Semaphore_pend(sem_a, BIOS_WAIT_FOREVER);
Semaphore_pend(sem_b, BIOS_WAIT_FOREVER);
```

  - Operation returns to normal and all events counter increment for entire test.

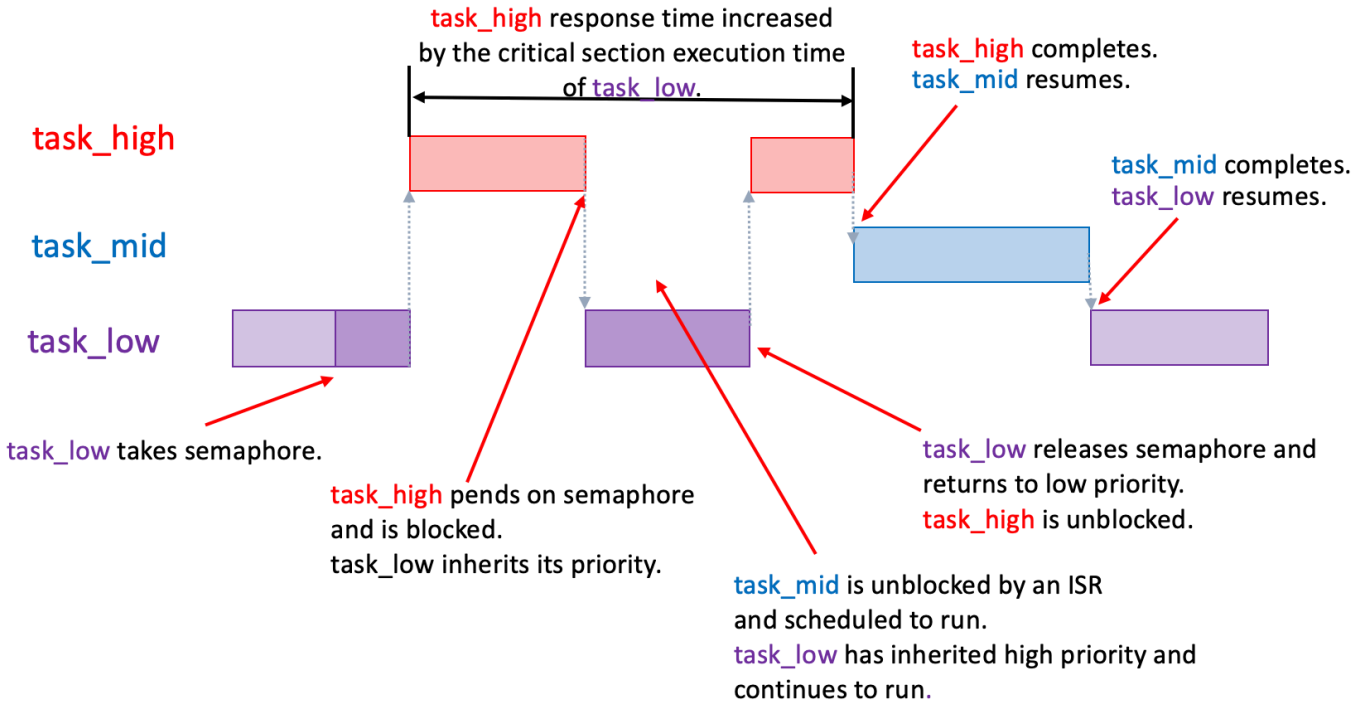
Task Priority Inversion

- Task priority inversion.
  - Low priority task, task_low, takes the semaphore0, protecting the shared data.
  - High priority task, task_high, preempts task_low, runs and pends on the semaphore0 and is blocked.
  - Medium priority task, task_mid, preempts task_low, runs and completes.
  - Low priority task continues, posting to semaphore0 to finally unblock task_high.
- Response time of task_high increases by execution time of task_mid.
  - Scheduling priorities are violated and relative deadlines maybe missed.



Priority Inheritance

- Priority inheritance.
  - Temporarily raises the priority of the task holding the semaphore.
  - The priority is raised to the highest priority task blocked on the semaphore.
  - Once the high priority task is unblocked, priorities return to normal.
- Restrictions.
  - Priority inheritance is restricted to use with binary semaphores with the role of protecting critical sections.
- Priority Inheritance prevents task_mid from running while task_high is blocked.



Priority Inheritance: GateMutexPri()

- Ti-RTOS implements priority inheritance using a specialized object GateMutexPri().
  - GateMutexPri_enter() is placed at the start of the critical section.
  - GateMutexPri_leave() is placed at the end of the critical section.
  - These are used in place of Semaphore_pend() and Semaphore_post() calls.
- GateMutexPri_enter() returns a key
  - The key is later used by GateMutexPri_leave() to restore the thread preemption to the state just prior to entering.

```
IArg keyMutexPri0;
...
keyMutexPri0 = GateMutexPri_enter(mutexPri0);
// critical section
GateMutexPri_leave(mutexPri0, keyMutexPri0);
```

Ece3849_rtos_latency: GateMutexPri()

- ece3849_rtos_latency: adds the following statements to event0 and event2 to simulate protecting critical section.

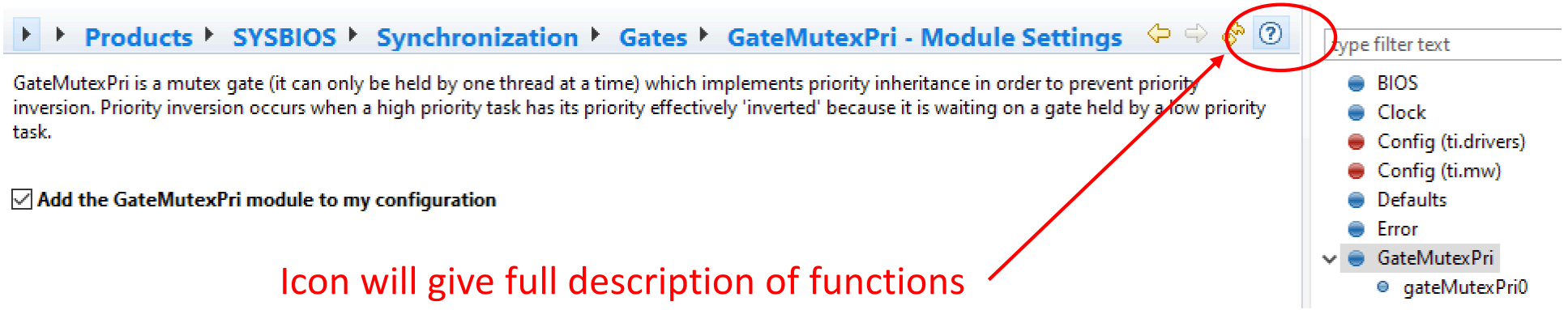
```
130 #ifdef PRIORITY_INVERSION
131       Semaphore_pend(sem_a, BIOS_WAIT_FOREVER);
132       delay_us(100); // "critical section"
133       Semaphore_post(sem_a);
134 #endif
```

Critical section protected by Semaphore

OR

```
135 #ifdef ENABLE_GATEMUTEXPRI
136         key = GateMutexPri_enter(gateMutexPri0);
137         delay_us(100); // "critical section"
138         GateMutexPri_leave(gateMutexPri0, key);
139 #endif
```

Critical section protected by GateMutexPri



Icon will give full description of functions

Ece3849_rtos_latency: Highest Priority Task Response Time

Expected task0 execution time results:

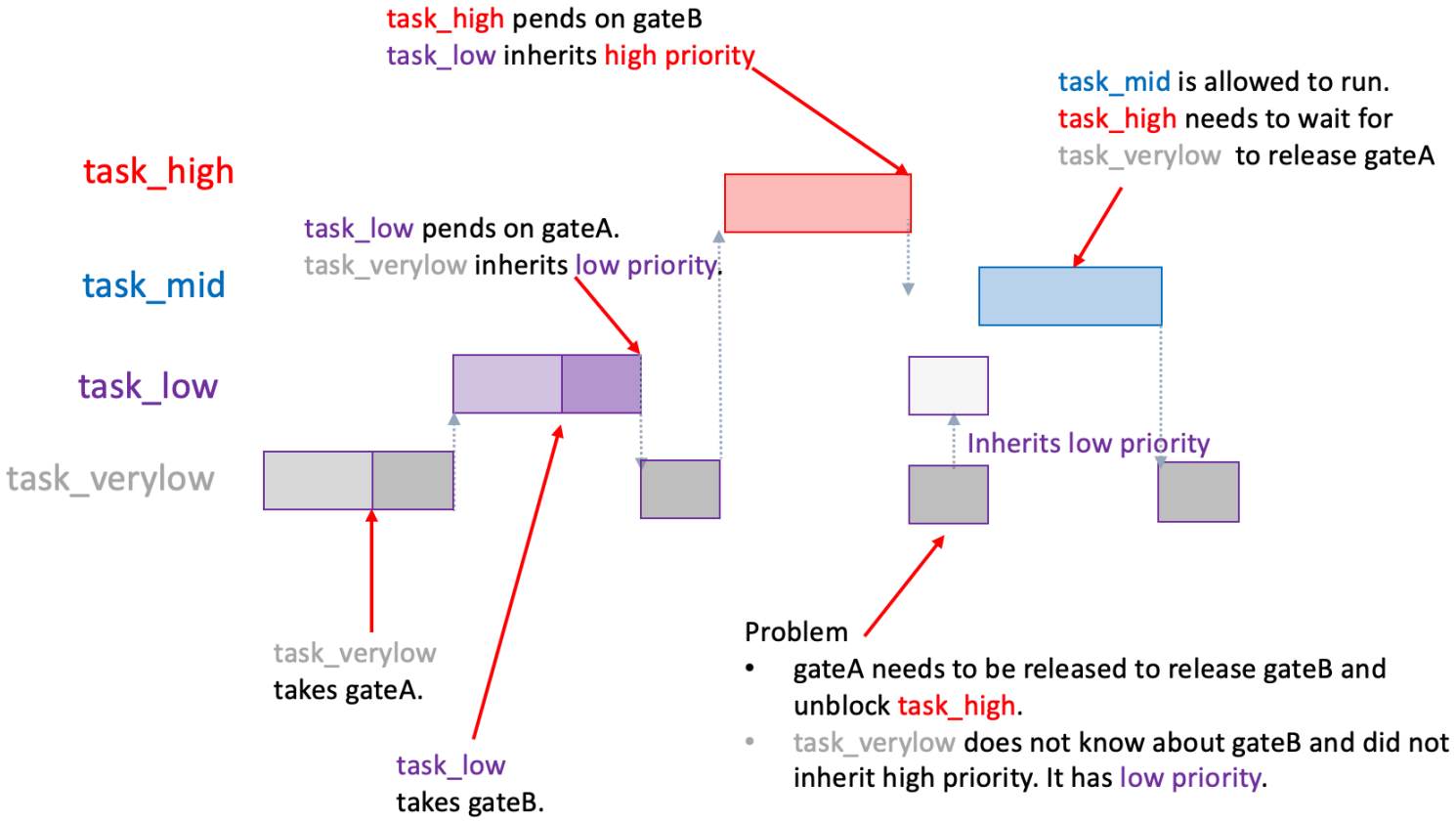
- No critical section (original conditions):
  - Response time = 18 usec latency + 2 msec execution time = ~2 msec.
- Critical section protected with Semaphore sem_a => causes priority inversion.
  - Expected response time = ~3.2 msec.
    - + 18 usec original latency +
    - + 2 msec task0 original execution time
    - + 1 msec task 1 execution time
    - + 100 usec task2 critical data section protection
    - + 100 usec task0 critical data section execution.
- Critical section protected with GateMutexPri() => removes priority inversion.
  - Expected response time = ~2.1 msec
    - Semaphore response time of ~3.2 – 1.1 msec task1 execution time because priority inversion is removed.

GateMutex Object

- The GateMutex object is GateMutexPri without the priority inheritance function.
  - It is a binary semaphore for mutual exclusion.
  - It enables or disables access to shared resources.
  - They can not be used by Hwi or Swi threads.
  - It use GateMutex_enter() and GateMutex_leave() functions to implement this functionality.
- It behaves similar to a semaphore except, it can be entered multiple times in the same function with out blocking.
  - If a semaphore pends multiple times, a post must occur in between each pend to unblock it.
  - If GateMutex_enter() is called multiple times in the same task and it already holds the key, it continues without being blocked.
  - It is only used for mutual exclusion and can not be used for task synchronization.
  - Mutual exclusion guarantees only one thread at a time can access the shared resource.

Priority Inheritance Limitations

- Priority Inheritance is on a per Gate basis.
- When multiple gates are used in a single task they are not aware of each others inherited traits.
  - This can lead to further priority inversion.
- task_high is preempted by task_mid below.



GateTask: Disabling Task Switching

- Task switching can be disabled.
  - Disabling switching will increase task latency.
    - Latency will increase by the longest time tasks switching is disabled.
    - This is a similar to the affect that disabling interrupts has on interrupt latency.
  - Uses fewer CPU cycles.
  - No priority inversion or deadlock.
- The GateTask object disables task switching.
  - GateTask_enter() function disables the task switching
    - Returns a key to be used in restoring the thread preemption state when leaving the GateTask.
  - GateTask_leave() function enables task switching and restores task state to the thread preemption state just prior to entering.
- These functions can not be called in Hwi or Swi interrupts.

```
IArg keyGateTask0;
...
keyGateTask0 = GateTask_enter(gateTask0);
// critical section
GateTask_leave(gateTask0, keyGateTask0);
...
```

Ece3849_rtos_latency: GateTask

- event0_func and event2_func now call the GateTask functions instead of the Semaphore or GateMutexPri functions.

```
#ifdef ENABLE_GATETASK
        key = GateTask_enter(gateTask0);
        delay_us(100); // "critical section"
        GateTask_leave(gateTask0, key);
#endif
```

Latency now increased from 18 usec to 108 usec.

Latency = Original latency + maximum time switching is disabled.

(x)= event0_latency/120.0f	float	108.300003
(x)= event1_latency/120.0f	float	2230.6001
(x)= event2_latency/120.0f	float	3152.80835
(x)= event0_response_time/120.0f	float	2227.3584
(x)= event1_response_time/120.0f	float	3240.16675
(x)= event2_response_time/120.0f	float	14573.333

Response time task0 similar to GateMutexPri functionality.

RTOS: Gating Interrupts

- Ti-RTOS can also disable interrupts that it controls.
- GateSwi object.
  - Uses GateSwi_enter() and GateSwi_leave() functions and a key to restore the state on leaving.
  - They can be called from Swi or tasks.
  - GateSWi object disables Swi and task scheduling.
  - Affects latency of all Swi and tasks.
- GateHwi object.
  - Uses GateHwi_enter() and GateHwi_leave() functions and a key to restore the state on leaving.
  - They can be called from Task, Swi or Hwi threads.
  - Guarantees exclusive access to the CPU.
  - Does not affect “zero-latency interrupts”.
  - Affects latency of all Hwi, Swi and tasks.

Summary of Objects used for Mutual Exclusion

Mutual Exclusion: only one thread can access a shared resource at a time.

Object	Pros and Cons
1. Semaphore (**pend** & **post**) • not optimal for mutual exclusion • has other applications	+ targeted − slow (higher CPU load) − does not work with ISRs − priority inversion − deadlock
2. GateMutex (**enter** & **leave**) • binary semaphore for mutual exclusion • cannot be used for signaling/synchronization	+ can be "entered" multiple times by the same task without blocking otherwise same as Semaphore
3. GateMutexPri • binary semaphore with **priority inheritance** • tasks are unblocked in **priority** order • preferred for longer critical sections	+ fixes most priority inversion cases otherwise same as GateMutex
4. GateTask • disables Task scheduling • preferred for short, frequently occurring critical sections	+ faster (lower CPU load) + no priority inversion + no deadlock − affects latency of **all Tasks** (but not Swi or Hwi)
5. GateSwi • disables Swi & Task scheduling	+ works with Swi and Tasks − affects latency of all Tasks and Swi
6. GateHwi • disables Hwi, Swi & Task scheduling • does not disable "zero-latency interrupts"	+ works with Hwi, Swi and Taks − affects latency of all Hwi, Swi and Tasks

Semaphores for Signaling

- Semaphores allow Hwi, Swi and other tasks to signal a task.
  - A task is unblocked when an event occurs.
- Below Hwi issues a Semaphore_post() to unblock Task1 which is pending.
  - If the Hwi is periodic this allows the task to run at a regular interval.
  - This keeps the CPU load under 100% because pending on semaphores do not take CPU cycles.
- Choice of semaphore type.
  - If this is a binary semaphore, then Hwi interrupts will be missed if the task gets unexpectedly delayed and the interrupt runs multiple time between Task1 pends.
  - A counting semaphore ensures that task1 is called once for every interrupt occurrence even if intermittently it misses its relative deadline.

Initialize semaphore `sem0` to count = **0** before it is first used.

```
void Hwi1(UArg arg0) { // hardware interrupt under TI-RTOS
  // communicate with the hardware
  Semaphore_post(sem0);
}

void Task1(UArg arg0, UArg arg1) {
  // init
  while (1) {
    Semaphore_pend(sem0, BIOS_WAIT_FOREVER);
    // handle event
  }
}
```

Semaphore Signaling: Multiple Tasks

- During Initialization,
  - semaphore count = 0.
  - Task1 runs first because it is higher priority and pends. It is blocked and put on the wait list.
  - Task2 runs second it pends and is placed on the wait list.

```
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
  }
}
```

Some time later Hwi1 interrupt occurs. It posts twice.

- The first post unblocks Task1.
  - It is higher priority AND first in the list.
  - Task1 is placed in Ready state.
- The second post unblocks Task2 and it is placed in Ready state.
- When Hwi1 exits, the scheduler runs.
  - Task1 is higher priority and enters Running state.
- Task1 completes its loop and pends again and is blocked.
- Task2 enters running state, completes its loop and pends.

Semaphore Signaling: Multiple Tasks

- What happens if Hwi1 only posts once?

```
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
  }
}
```

First Hwi occurs…

- ?

Second Hwi occurs…

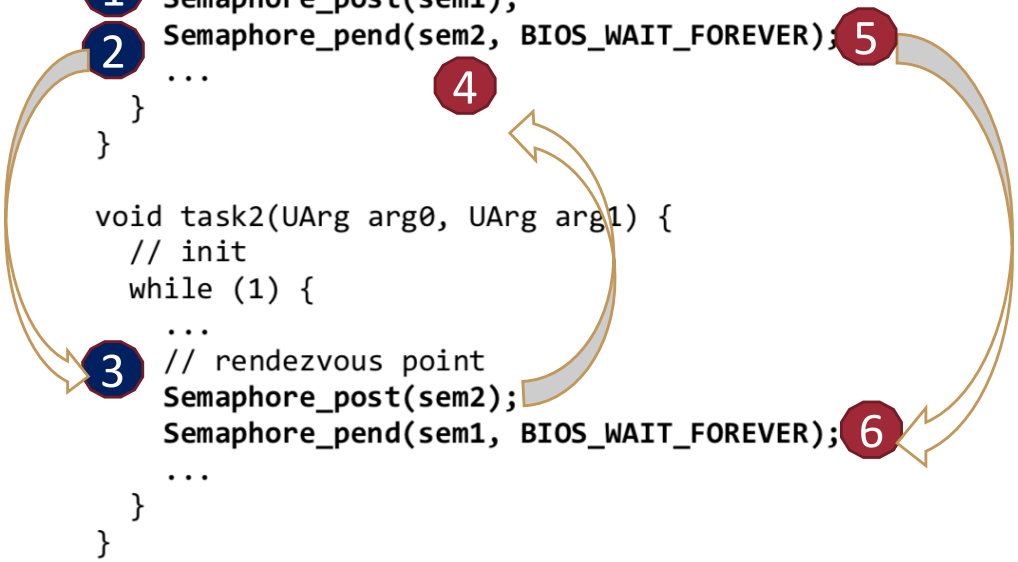
- ?

Semaphore for synchronization

- Rendezvous: We would like two tasks to arrive at a specific point in their code before either can continue.

```
void task1(UArg arg0, UArg arg1) {
  // init
  while (1) {
    ...
    // rendezvous point
    Semaphore_post(sem1);
    Semaphore_pend(sem2, BIOS_WAIT_FOREVER);
    ...
  }
}

void task2(UArg arg0, UArg arg1) {
  // init
  while (1) {
    ...
    // rendezvous point
    Semaphore_post(sem2);
    Semaphore_pend(sem1, BIOS_WAIT_FOREVER);
    ...
  }
}
```



Steps 0 -3: Just once on initialization

0. System initialization
   - Sem1 & sem2, count = 0
   - task1 is higher priority
1. task1 starts and posts to sem1.
   - task2 has not run yet, nothing on waitlist
   - sem1 count is incremented.
2. task1 pends on sem2.
   - sem2 count = 0, task1 is put on wait list.
   - task1 is blocked.
3. task2 runs and posts to sem2.
   - task1 is in waitlist and moved to Ready state.
4. task1 is higher priority and runs its while loop and posts to sem1.
5. task1 pends on on sem2 and is blocked.
6. Task2 starts running,
   - takes sem1 and decrements sem1 count.
   - Continues its while loop and then posts to sem2.

Repeats this sequence forever

- It is like a relay race.
- Each task runs and hands off to the next task a the desired point.