Experiment 6: Task Blocking and Priorities

Objectives:

- To gain experience writing your own RTOS on the TM4C123 Launchpad Development Board.
- To gain experience the Keil ARM development environment and debugging features.
- To learn more about concurrency, semaphores, and races conditions.

Reading:

- Tiva™ C Series Development and Evaluation Kits for Keil™ RealView® MDK (course website or http://www.ti.com/lit/ml/spmu355/spmu355.pdf
- Tiva C Series TM4C123G LaunchPad Evaluation Board User's Guide (course website)
- Tiva C Series TM4C123x ROM User's Guide (course website or http://www.ti.com/lit/pdf/spmu367)
- TivaWare Peripheral Driver Library User's Guide course website or http://www.ti.com/lit/spmu298)

Introduction:

In this experiment, you will improve the implementation of your spin-lock semaphores of the previous lab first by 1) having the blocked task surrender the processor instead of spinning and then 2) by having the Kernel mark the task as "blocked" and thus preventing it from ever being scheduled.

Experiment:

1. Modify your implementation of the spin-lock semaphore from the previous lab by having

```
void OS_Sem_Wait(unsigned int *s)
{
   while (*s == 0){
         OS_Suspend();      // surrender the processor
   }
   *s = *s – 1;
}
```

Again this code will not work as it is above because tasks can race on access to the variable *s. You must protect these accesses by disabling and enabling the interrupts as you did in the previous lab.

2. While the semaphore implementation above is better, it still incurs overhead each time the blocked task is scheduled, we will have to make a lap around the while (*s == 0) loop and surrender the processor. It would be much better if we could mark this task in some way that the scheduler could tell it was blocked and not schedule it

at all. To do this will require a new field in the TCB to indicate that a task is blocked on a semaphore. Let’s call this new filed “blocked”.

If we had such a capability, our semaphore Signal and Wait could become:

```
void OS_Sem_Wait(uint32_t *s){
  DisableInterrupts();
  (*s) = (*s) - 1;
  if((*s) < 0){
    Mark_this_task_blocked_on_s; // blocked on s
    EnableInterrupts();
    OS_Suspend();       // run scheduler
  }
  EnableInterrupts();
}
```

And

```
void OS_Sem_Signal(uint32_t *s){
  DisableInterrupts();
  (*s) = (*s) + 1;
  if((*s) <= 0){
    Find_a_task_blocked_on_s_and_clear_it;
                   //unblock some task to take the semaphore
  }
  EnableInterrupts();
}
```

Then our scheduler could have the following add code added:

```
unsigned char * Schedule(unsigned char * the_sp)
   {
    CURRENT_TASK->sp = the_sp;
    CURRENT_TASK->state = T_READY;
    CURRENT_TASK = CURRENT_TASK->next;

        while(CURRENT_TASK->blocked){  // skip task if blocked
             CURRENT_TASK = CURRENT_TASK->next;
        }

   // code to start new tasks omitted

        return(sp);
   }
```

To make this work, marking a task as blocked is fairly easy:

```
    Mark_this_task_blocked_on_s; // blocked on s
```

Could be:

```
    CURRENT_TASK->blocked = s; // blocked on s
```

However, the code block:

```
   Find_a_task_blocked_on_s_and_clear_it;
                   //unblock some task to take the semaphore
```

is a little more involved. We must search through the TCBs to find a task that is blocked on s and then clear it. This will not guarantee that the task we mark will get the semaphore (other tasks my callOS_Sem_Wait() before it gets to run), however, it will ensure that at least one task is unblocked and will try to acquire the semaphore.

```
{
   TaskControlBlock *task_ptr;
   task_ptr = CURRENT_TASK->next;
                              // search for a task blocked on s
   while(task_ptr->blocked != s){
     task_ptr = task_ptr->next;
   }
   pt->blocked = 0;         // unblock this task

}
```

3. Modify your cpu time code from the last labe to take into account that tasks may call OS_Suspend() and tasks may not get scheduled because of blocking and confirm that your total process time calculations are still correct and demonstrate for a signature.

4. Finally, we can add priority to our scheduler. This will require an additional field in the TCB to hold a priority number (perhaps "priority") and another change to the functions to start tasks to set priorities for each task and the scheduler to enforce the priorities. The new scheduler could be:

```
   void Scheduler(void){ // every time slice
     uint32_t max_priority = MAX_TASK_PRIORITY;
     TaskControlBlock *pt;
     TaskControlBlock *WinnerPt;

    CURRENT_TASK->sp = the_sp;
    CURRENT_TASK->state = T_READY;

     pt = CURRENT_TASK;
            // search for highest priority task (not blocked)
     do{
       pt = pt->next;
       if((pt->priority < max_priority)&&((pt->blocked)==0)){
         max_priority = pt->priority;
         WinnerPt = pt;
```

```
        }
      } while(CURRENT_TASK != pt); // look at all tasks

      CURRENT_TASK = WinnerPt;     // schedule the task found
    }
```

You do not have to implement priorities for the OS, we will switch to a commercial RTOS that already has this implemented for the next labs.

NAME:____________________

1-3: Instructor Signature:_________________________Date/Time:____________