**Lab 3 RTOS with Blocking and Priority Scheduling**

**Goals**
- Extend the RTOS to include blocking and priority,
- Extend the RTOS to include two real-time periodic tasks,
- Develop minimally invasive tools to determine performance measures,
- Record debugging/performance data and download this data to the PC.

**Starter files**
- **`Switch_4C123`** project (SW2 input)
- **`RTOS_4C123`** project (basic OS)
- **`OS.h`** and **`Lab2.c`**

**Background**

In this lab you will convert your spinlock semaphores to blocking semaphores, and convert your round robin scheduler to a priority scheduler. A priority field will be added to the TCB, which is set at the time the user calls **`OS_AddThread`**. In Lab 2, you implemented a single periodic task using a timer interrupt and a single aperiodic task using the PF4 interrupt. In this lab, you will add the possibility of two periodic tasks and two aperiodic tasks. An additional parameter will allow the user to set the priority of the four background tasks. In the example user program, **Lab2.c**, one periodic task will sample the ADC and a second periodic task will run a digital PID motor controller. The two aperiodic tasks will be triggered by the two buttons on the board (PF4=SW1 and PF0=SW2). The background tasks should have priority (preempt) any foreground task, regardless of the user-defined priorities. When modifying your Lab 2, you will need to uncomment this line in the main() program

```
OS_AddSW2Task(&SW2Push,2);  // add this line in Lab 3
```

A thread is in the **blocked state** when it is waiting for some external event like input/output (serial input data available, LCD free, I/O device available). If a thread communicates with other threads, it can be blocked waiting to receive data or for room to be available in the transmit buffer. Both types of blocking will be implemented in this lab. If a thread wishes to output to the LCD display, but another thread is currently outputting, it will block. We will use a blocking semaphore to implement the sharing of the display output among multiple threads.

All of these features can be implemented by changing **`OS_Wait`**, **`OS_Signal`**, **`OS_bWait`**, **`OS_bSignal`**, and your scheduler. One possible way to implement blocking semaphores uses linked-list data structures to hold the ready and blocked threads. You will need to create multiple blocked linked lists. In general, you would have one blocked list with each blocking semaphore. You could extend the semaphore structure to include both the semaphore value and a pointer to a TCB list containing threads that are blocked on the semaphore. The semaphore initialization should be extended to clear the linked-list of blocked threads on that semaphore. Except for the semaphore structure, everything else in the user program should remain exactly the same.

**New `OS_Wait`**

1) Save the I bit, then disable interrupts
2) Decrement the semaphore counter, **`(*pt->Value) = (*pt->Value)-1`**
3) If the semaphore counter is less than zero then this thread will be blocked
   set the **`status`** of this thread to blocked,
   specify this thread is to be blocked onto the linked list of this semaphore (semaphore pointer)
   suspend thread causing the thread switch operation to occur
4) Restore I bit to its previous value

**Add this step somewhere in your thread switch process**

1) If this thread is to be blocked
   move the **TCB** of this thread from the active list to the end of the blocked list of the specified semaphore

**New `OS_Signal`**

1) Save the I bit, then disable interrupts
2) Increment the semaphore counter, **`(*pt->Value) = (*pt->Value)+1`**
3) If the semaphore counter is less than or equal to zero then
   wake up one thread from the **TCB** linked list
   with Round Robin, choose the one waiting the longest (bounded waiting)
   with Priority, wake up the highest priority thread

move the **TCB** of the "wakeup" thread from the blocked list to the active list
what to do with the thread that called **`OS_Signal`**
with Round Robin, do not suspend execution
with Priority, do suspend execution if you wake up a higher priority thread
be careful not to suspend a background thread

4) Restore the I bit to its previous value

There is a second simpler implementation. In this implementation, a **`BlockPt`** field is added to the TCB. If the **`BlockPt`** is null, the thread is not blocked. If the **`BlockPt`** contains a semaphore pointer, it is blocked on that semaphore. This simple implementation will not allow you to implement bounded waiting. There are other ways to implement blocking, and you are free to implement other blocking schemes. Although bounded waiting is important, it is not necessary for you to implement it.

In uCOS-III, if a low priority thread signals a semaphore that wakes up a higher priority thread, the thread switch occurs immediately, and the higher priority thread is run, without waiting for the time slice of the lower priority thread to finish. This is a good feature, but you do not have to implement it in this lab.

You will implement a blocking scheduler. If multiple threads are blocked, when it is time to wakeup a thread, one option is to have the OS wakeup the one that has been blocked the longest (bounded waiting). If a thread requests a resource that is unavailable, your system should move the thread to the appropriate blocked linked-list. Careful thought should go into when to remove a thread from the blocked list. There are lots of ways in which to implement blocking semaphores. You are allowed to choose whichever way you wish as along as

1) Blocked threads are not allowed to run;
2) The Lab3.c user program runs for 30 minutes without crashing;
3) The Lab3.c user program runs for 30 minutes without hitting a deadlock; and
4) You understand how to implement blocking as described in class and as implemented in uCOS-II.

**Preparation**

1) Design at least two ways to implement the second periodic background thread. A **`priority`** parameter in the **`OS_AddPeriodicThread`** function allows the user to specify the relative priority of the four background threads. This priority does not affect the fact that all background threads (two periodic, two aperiodic) will preempt any foreground thread. In this lab, **`OS_AddPeriodicThread`** will be called 0, 1, or 2 times. Think about how would your implementation be different if there were 10 background threads? Write (pseudo) code to implement this new expanded periodic thread feature.

2) Add features to the OS to measure and record time jitters for the two periodic threads. Record two maximum jitters and two histograms of jitter values. In particular, move the jitter measurements from Lab2 into the OS, and make two copies of it. Feel free to change any of the user code in **`Lab2.c`** so they are compatible with your RTOS.

3) Write (pseudo) code to implement the second aperiodic background thread, triggered by a falling edge on PF0. In hardware this is implemented with the **SW2** button. On the robot, you could add a bumper switch and use this feature to detect collisions. Your OS should allow the user to activate and arm this feature by passing a user function to execute, and using the external event mechanism to trigger the background task. You should also provide an OS function to disarm this feature. The **`priority`** parameters for **`OS_AddPeriodicThread`** **`OS_AddSW1Task`** and **`OS_AddSW2Task`** allow the user to specify the relative priority of the four background threads.

4) Design at least two ways to implement blocking semaphores. In this class, we have 1 to 10 foreground threads. How would your implementation be different if there were 100 foreground threads? Write (pseudo) code to implement the blocking semaphores. Take one of the test programs from **`Lab2.c`** and modify it to test the blocking semaphores. In particular, there should be multiple threads signaling the same semaphore, and multiple threads waiting on that same semaphore. You should signal both in the background and in the foreground, but you can only wait in the foreground. Use counters to make verify the total number of times signal is called matches the total number of times wait allows a thread to pass. The key is to run the system in an exhaustive manner increasing the likelihood of finding bugs. For example, if there is a 1 in $10^4$ chance of **Condition A** occurring (e.g., PC is executing at a particular spot) and a 1 in $10^4$ chance of **Condition B** occurring, there will be a 1 in $10^8$ chance of **Conditions A and B** occurring at the same time. You also have to be careful not to introduce critical sections in the test program itself. Edit **`Lab2.c`** so it is compatible with your RTOS. Modify the numbers in **`Signal2`**() so that the period of this thread is variable.

5) Design at least two ways to implement the priority scheduler. Again, think about how would your implementation be different if there were 100 foreground threads? Write (pseudo) code to implement the priority scheduler.

**Procedure**

1) Implement, test and debug the second periodic task feature (Preparation 1). See **`Lab2.c`**.

2) Write the function **`jitter`** called by **`Thread7`**. Using main program **`Testmain5`** evaluate the time jitters over a finite amount of time (e.g., 10 seconds) for these six conditions (Prep 2). Make Task A higher priority than Task B. Feel free to adjust the specific times. The 2.99 ms means any value slightly different than 3ms, not related to 2ms.

- Three periodic times $p_A$ ={1, 2.99 ms} for Task A, i.e., Task A is invoked every $p_A$,
- Three execution times $t_A$ = {5, 50, 500 µs} for Task A, i.e., Task A takes $t_A$ to complete,
- One periodic time $p_B$= 2 ms for Task B, i.e., Task B is invoked every $p_B$,
- One execution time $t_B$ = 250 µs for Task B, i.e., each time it runs, Task B takes $t_B$ to complete, with $t_B$ = ¼ $p_B$

You are allowed to change any of these numbers to be compatible with your OS.

3) Implement, test and debug the second aperiodic task feature (Preparation 3). See **`Lab2.c`**.

4) Implement, test and debug the blocking semaphore implementation with the round-robin scheduler using test program written in Preparation 4. Using **`Lab2.c`**, test the blocking semaphore solution on the same five test cases you ran in Lab 2 procedure 5. An example performance data is shown in the middle of Table 3.1.

5) Implement, test and debug the blocking priority implementation. Using **`Lab2.c`**, test the solution on the same test cases you ran in Lab 2, Procedure 5. Create a side by side table of the performance data (e.g., Table 3.1) of

- spinlock semaphores, round robin scheduler (Lab 2)
- blocking semaphores, round robin scheduler (Lab 3 Procedure 4)
- blocking semaphores, priority scheduler (Lab 3 Procedure 5)

| | | Spinlock/Round Robin | | | Spin/Round Robin/Cooperative | | | Block/Priority | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **FIFO Size** | **Tslice (ms)** | **Data Lost** | **Jitter (µs)** | **PID Work** | **Data Lost** | Jitter (µs) | **PID Work** | **Data Lost** | **Jitter (µs)** | **PID Work** |
| 4 | 2 | | | | | | | | | |
| 32 | 2 | 0 | 0.4 | 3230 | 0 | 0.4 | 11949 | 0 | 11.6 | 11559 |
| 32 | 1 | | | | | | | | | |
| 32 | 10 | | | | | | | | | |

*Table 3.1. Example performance data for Labs 2 and 3 (collected with no interpreter or switch input). Data from Valvano's Spring 2014 lab solutions running 20 seconds.*

6) Add features to the OS to record the following information.

- Maximum time the system runs with interrupts disabled
- Percentage of time the system runs with interrupts disabled

Add interpreter commands to clear and restart the measurements and to dump results to the PC. The 10-second time jitter measurements performed in Lab 2 and Procedure 2 are not long enough to capture rare events. For example, what if the timer interrupt is triggered during the time a thread is being killed? Recording jitter itself (Procedure 2) is a direct measure of the latency. Conversely, recording the times when interrupts are disabled is an indirect measurement of latency. The maximum time running with interrupts disabled provides an upper bound on the latency of your real-time system. You can run **`Testmain7`** to measure the time is takes to switch from one foreground task to another foreground task (assuming no I/O interrupts); but you do not need to add interpreter commands to print out the task switch time.

7) Create a thread profile recorder. Using **`Lab2.c`**, collect timestamp data for the first 100 events. In this recording, you can ignore aperiodic events (i.e., no Select, Down, or UART I/O). For each event, record the time and the thread effected. The events include

- A foreground thread is started (whenever PendSV is triggered)
- A periodic thread is started
- A periodic thread finishes (observing if one periodic event suspends another periodic event)

Add interpreter commands similar to 6) to clear buffers, restart the measurements and dump results to the PC. Organize these profile data in a graphical manner.

**Checkout (show this to the TA)**
1) Demonstrate the final periodic thread system running to the TA.
2) Show the TA where in your final system the worst case time jitter arises.
3) Demonstrate your method to visualize the real time execution pattern.
4) Discuss your results of the three tables (e.g., Table 3.1 with data from Lab 2, Procedures 4 and 5).

**Deliverables (exact components of the lab report and lab submission)**
A) Objectives (1/2 page maximum)
B) Hardware Design (none for this lab)
C) Software Design
   1) Documentation and code of main program used to measure time jitter in Procedure 2
   2) Documentation and code of main program used test the blocking semaphores Preparation 4
   3) Documentation and code of your blocking/priority RTOS, **OS.c** and any associated assembly files
D) Measurement Data
   1) Plot of the logic analyzer running the blocking/sleeping/killing/round-robin system (**Lab2.c**)
   2) Plot of the scope window running the blocking/sleeping/killing/priority system (**Lab2.c**)
   3) Table like Table 3.1 each showing performance measurements versus sizes of the Fifo and timeslices
E) Analysis and Discussion (2 page maximum). In particular, answer these questions
   1) How would your implementation of **`OS_AddPeriodicThread`** be different if there were 10 background threads? (Preparation 1)
   2) How would your implementation of blocking semaphores be different if there were 100 foreground threads? (Preparation 4)
   3) How would your implementation of the priority scheduler be different if there were 100 foreground threads? (Preparation 5)
   4) What happens to your OS if all the threads are blocked? If your OS would crash, describe exactly what the OS does? What happens to your OS if all the threads are sleeping? If your OS would crash, describe exactly what the OS does? If you answered crash to either or both, explain how you might change your OS to prevent the crash.
   5) What happens to your OS if one of the foreground threads returns? E.g., what if you added this foreground

```
void BadThread(void){ int i;
  for(i=0; i<100; i++){};
  return;
}
```

   What should your OS have done in this case? Do not implement it, rather, with one sentence, say what the OS should have done? Hint: I asked this question on an exam.
   6) What are the advantages of spinlock semaphores over blocking semaphores? What are the advantages of blocking semaphores over spinlock?
   7) Consider the case where thread T1 interrupts thread T2, and we are experimentally verifying the system operates without critical sections. Let **n** be the number of times T1 interrupts T2. Let **m** be the total number of interruptible locations within T2. Assume the time during which T1 triggers is random with respect to the place (between which two instructions of T2) it gets interrupted. In other words, there are **m** equally-likely places within T2 for the T1 interrupt to occur. What is the probability after **n** interrupts that a particular place in T2 was never selected? Furthermore, what is the probability that all locations were interrupted at least once?

**Hints**
0) Run the test mains and debug each component separately before combining into one system. If something doesn't work create a set of test mains and debug each component separately. Remember to use the logic analyzer to see what is running when.

1) Feel free to change how the user programs are organized (**Lab2.c**).

2) Since each of the labs is built on top of the previous lab, time spent debugging this lab will greatly simplify subsequent labs. In other words, some students report that significant time is wasted during Labs 4, 5 and 6 because their Lab 3 OS has bugs.

3) Port F, Pin 0 (PF0 for SW2) is protected by the GPIO Lock/Commit Control registers, i.e. needs to be unlocked and uncommitted before it can be reprogrammed (see **`Switch_4C123`** starter code).