Lab 3 RTOS with Blocking and Priority Scheduling

Michael Coulter and Jake Klovenski

A) Objectives (1/2 page maximum)

The objectives of this lab are to 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, and to record debugging/performance data and download this data to the PC.

B) Hardware Design (none for this lab)

C) Software Design

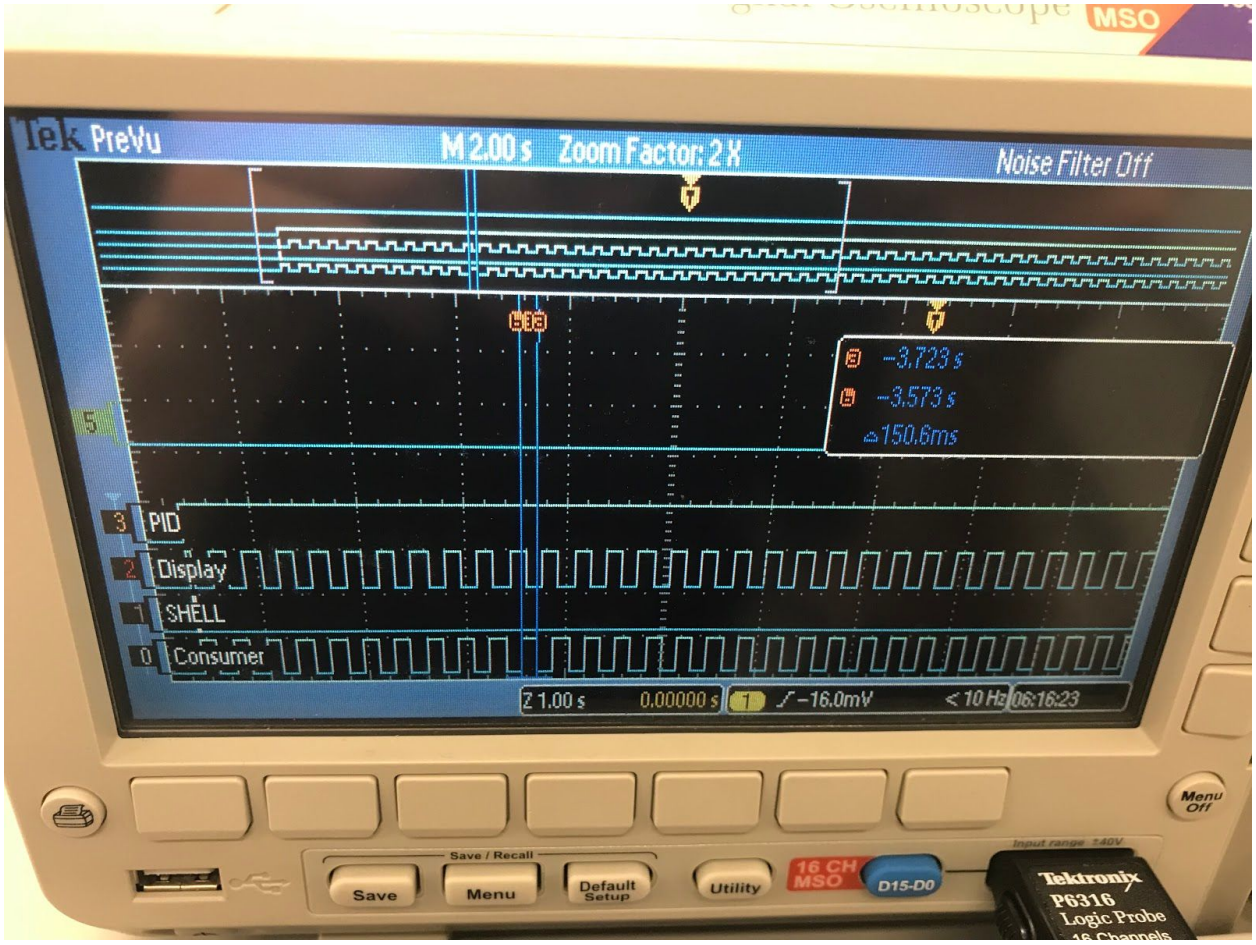
1) Documentation and code of main program used to measure time jitter in Procedure 2
The jitter function runs during timer interrupts and returns the absolute value of the time difference from when the jitter function is called and from when the next time the jitter function is called. It then takes this value and compares it to the ideal time of when the interrupt should have occurred, then returns the difference.

2) Documentation and code of main program used test the blocking semaphores Preparation 4
In order to do OS blocking in our RTOS, each of our semaphore objects contain a queue. When a semaphore needs to block a thread, the thread is added to the end of the semaphore's block queue. If there are no other threads currently being blocked by that semaphore, it will be the only thread in the blocked queue. When a thread is blocked, it will be taken out of the doubly linked list of actively running threads so that no time is spent context switching and executing the blocked thread. To unblock a thread, the thread is removed from the semaphore's blocked queue and is placed back in the linked list of running threads behind NextRunPt, and will run at next appropriate time.

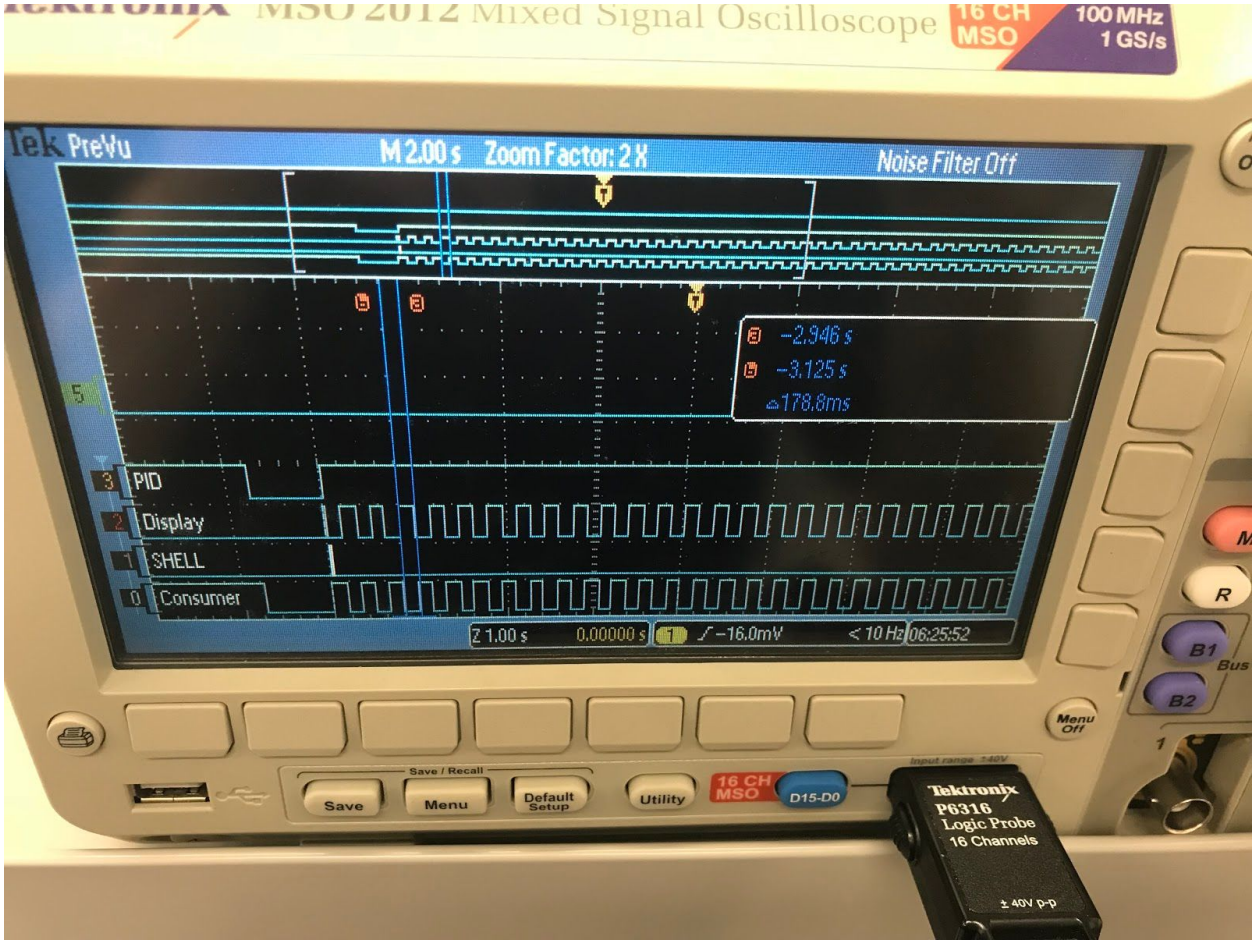
3) Documentation and code of your blocking/priority RTOS, OS.c and any associated assembly files
The blocking functions the same as described in the C.1. The priority scheduler functions by finding the thread with the "highest" priority that is not sleeping in the linked list of running threads and setting NextRunPt to it. If the scheduler is being called from OS_Block, it will disregard the thread corresponding to RunPt to prevent the blocked thread from being run, causing the semaphores queue from becoming the execution list.

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

		Round Robin/Spinlock			Round Robin/Blocking			Priority/Blocking		
F/IFO Size	Tslice (ms)	Data Lost	Jitter (us)	PID Work	Data Lost	Jitter (us)	PID Work	Data Lost	Jitter (us)	PID Work
4	2	0	4	6286	127	18	4199	191	36	2
32	2	0	4	6286	0	19	4194	0	34	2
32	1	0	4	6265	0	20	6281	0	37	1
32	10	0	4	6304	0	19	4208	0	27	13

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)
For our OS, we are using a separate timer to implement each individual background thread. So in the case of 10 background threads, it would be possible for our OS to support that many background threads, in that the TM4C has at least 10 timers. However, our OS can only support as many background threads as the TM4C has timers.

2) How would your implementation of blocking semaphores be different if there were 100 foreground threads?(Preparation 4)
Our blocking semaphores would work the same, in that if a shared resource is being used by a thread and is wanted to be used by another thread, the second thread is blocked until the resource becomes available.

3) How would your implementation of the priority scheduler be different if there were 100 foreground threads?(Preparation 5)
Our priority scheduler would work the same way in that it will always run he highest priority thread that is not sleeping. However, our scheduler would run a lot slower with 100 foreground threads so the system jitter would most likely be much higher. Therefore we would most likely have to come up with a more efficient scheduler that would run faster.

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.
In our OS, if all of the threads are blocked, then the system will hardfault. This is because that, at the end of OS_Block as the last thread is blocked, PendSV would be called which would have put NextRunPt into RunPt, however, since NextRunPt will contain an invalid thread TCB, the system will not be able to run an active thread, and will hardfault. Theoretically this should not happen in our OS because threads are only blocked if that thread wants to use a resource that a currently active thread is already using. If all threads are sleeping, our priority scheduler would go into an infinite loop in that it would never pick a thread to run next because of the fact that it skips over sleeping threads. To fix this infinite loop, we could implement code inside of our scheduler that would call sleeper (which helps to wake up threads) on certain threads if it detected that each thread was sleeping so that it could break out of the scheduler and run a thread.
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;
}
```

In our OS, this thread would return to address 0x14141414 because we initialize every threads LR to address 0x14141414 when we add new threads.

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.
Threads should either run infinite loops so that they can be interrupted and switched, or the thread should call OS_Kill instead of returning.

6) What are the advantages of spinlock semaphores over blocking semaphores? What are the advantages of blocking semaphores over spinlock?
A spinlock semaphore is simpler to implement and is less memory intensive. Blocking semaphores have an advantage in that they save a lot of time compared to spinlock in that if a thread is blocked, there is no chance of it even running and all of the time and work spent by the thread switcher is saved.

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?
Probability that a particular place in T2 was never selected after n interrupts = 1 - (n/m)
Probability that all locations were interrupted at least once = (n/m)^n