EE445M – Lab 2: RTOS Kernel

Kristian Doggett and Akshar Patel

3/2/15

A) Objectives

Goals:

Develop OS facilities for real-time applications
Coordinate multiple foreground and background threads
Design a round robin multi-thread scheduler
Implement spinlock semaphores and use them for thread synchronization
Implement inter-thread communication

Summary:

Lab 2 introduces all the core components needed to run an RTOS such as thread scheduling, semaphores and inter-thread communication. In part one of the lab, we design a simple cooperative and preemptive versions of thread scheduling. Moving in part two, we use the preemptive thread scheduler to manage five various tasks. In task one, we use software triggered ADC sampling to measure time-jitter; when a periodic task is supposed to run and when it actually runs. In task two, we learn to use aperiodic threads such as a push of a button. We use a debounce task to handle mechanical jitter in the button to properly record a user press. In task three, we use a hardware trigger ADC and a FIFO to produce then send data to a consumer that outputs to the LCD. In task four, a single foreground thread, PID, is used as a default thread that runs if all other thread are blocked, dead, or sleeping. Finally, in task five, we add our interpreter as a thread.

B) Hardware Design

None for this lab.

C) Software Design (printout of spinlock/round-robin operating system)

See end of lab report.

D) Measurement Data

1) plots of the logic analyzer like Figures 2.1, 2.2, 2.3, 2.4, and 2.8

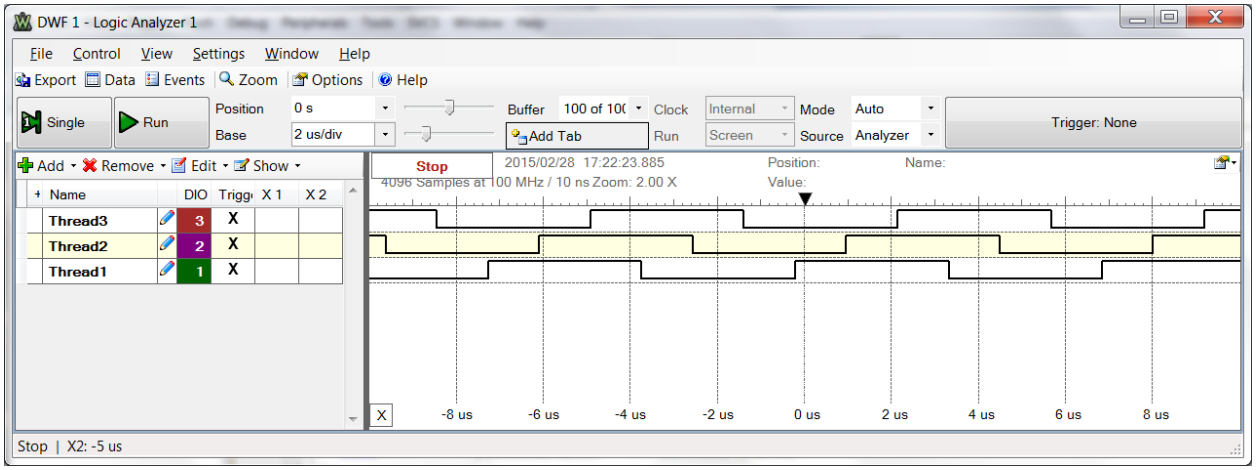


Figure 2.1 Cooperative Scheduling

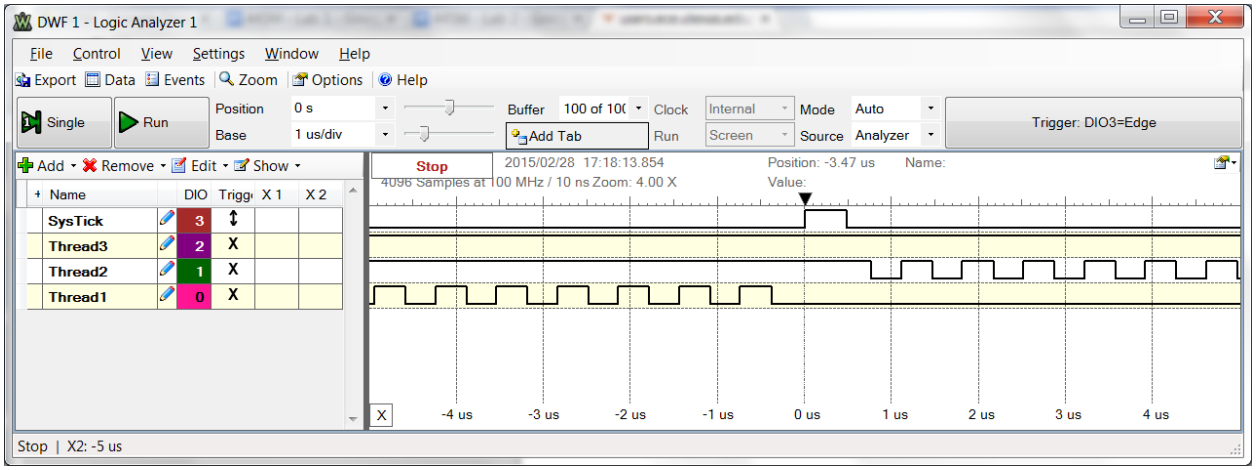


Figure 2.2 Preemptive Scheduling Using SysTick Round Robin

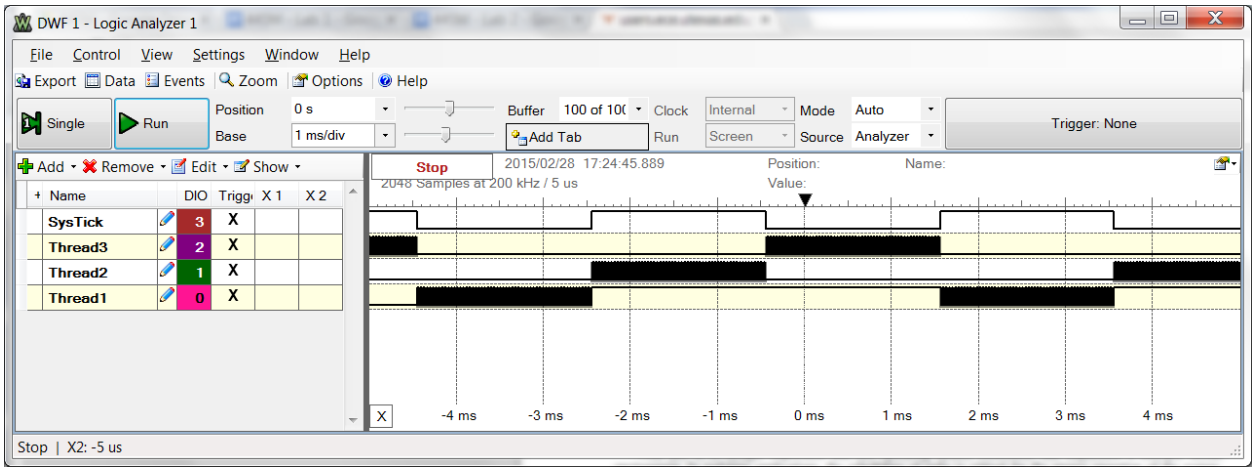


Figure 2.3 Preemptive Scheduling Using SysTick Round Robin

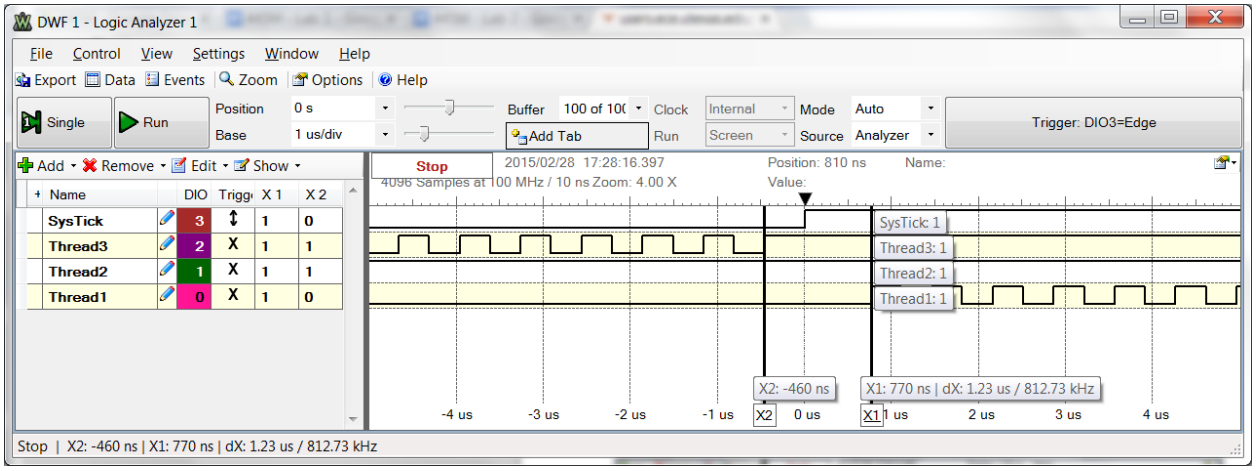
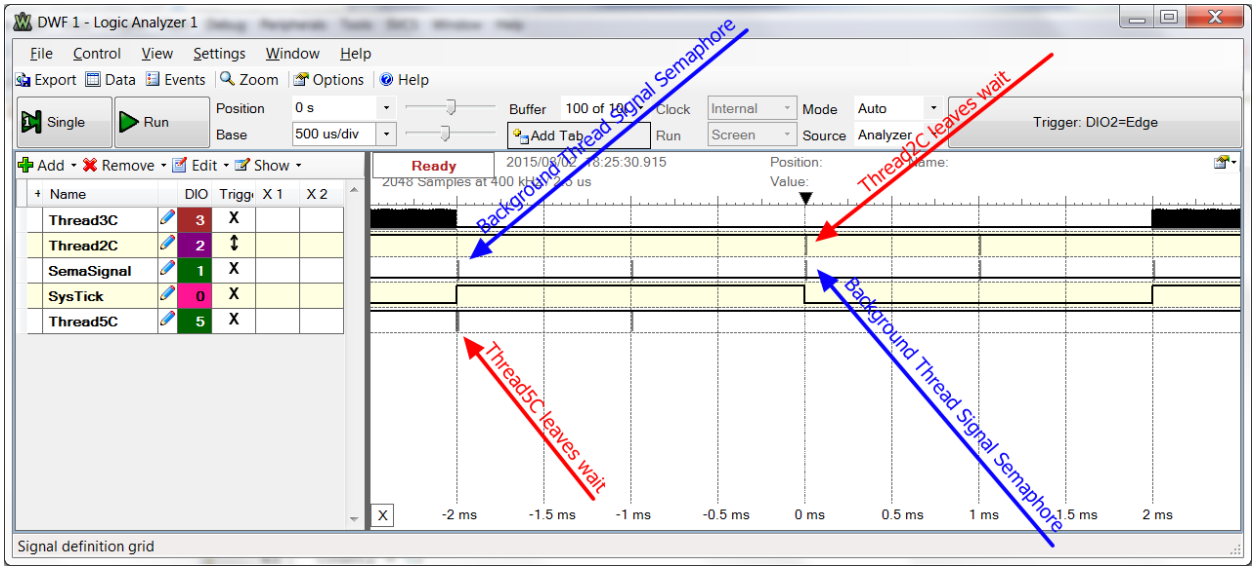


Figure 2.8 Context Switch Time

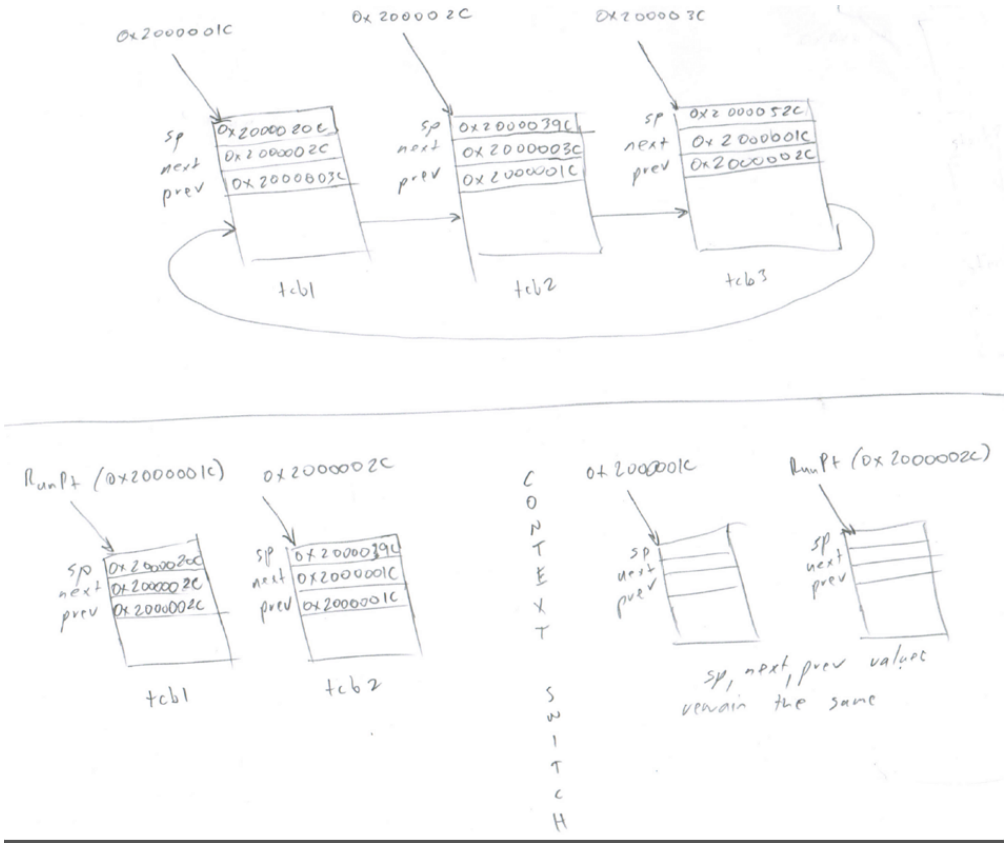
2) measurement of the thread-switch time
470 ns

Figure 1.1 Thread Switch Time: 470 ns

3) plot of the logic analyzer running the spinlock/round-robin system (profile data)



4) the four sketches (from first preparation parts 3 and 5), with measured data collected during testing



5) a table like Table 2.1 each showing performance measurements versus sizes of OS_Fifo and timeslices

Did not implement.

6) a table showing performance measurements with and without debugging instruments

Did not implement.

E) Analysis and Discussion (2 page maximum)

1) Why did the time jitter in my solution jump from 4 to 6 µs when interpreter I/O occurred?

Did not implement interpreter.

2) Justify why Task 3 has no time jitter on its ADC sampling.

Not sure.

3) There are four (or more) interrupts in this system DAS, ADC, Select, and SysTick (thread switch). Justify your choice of hardware priorities in the NVIC?

The ADC used for the DAS task is software triggered so it does not have an NVIC priority.

Task	NVIC Priority
Hardware Triggered ADC	2
Select	5
SysTick	6
PendSV	7

We choose PendSV to be 7 and SysTick to be 6 so that PendSV never preempts SysTick. Select and hardware triggered ADC were chosen to be higher priority than PendSV and SysTick so they could preempt the RTOS context switch timers.

4) Explain what happens if your stack size is too small. How could you detect stack overflow? How could you prevent stack overflow from crashing the OS?

If your stack size is too small, a stacks could accidently overwrite each other and cause erratic behavior or system crash. You could detect stack overflow by checking whether you are in the bounds of a stack before writing using some type of pointer. You prevent a crash by having some default functioning stack stored in a safe location and run when a stack overflow is detected.

5) Both Consumer and Display have an OS_Kill() at the end. Do these OS_Kills always execute, sometime execute, or never execute? Explain.

Not sure.

6) The interaction between the producer and consumer is deterministic. What does deterministic mean? Assume for this question that if the OS_Fifo has 5 elements data is lost, but if it has 6 elements no data is lost. What does this tell you about the timing of the consumer plus display?

Deterministic means there is no randomness involved. I am not sure what effect that has.

7) Without going back and actually measuring it, do you think the Consumer ever waits when it calls OS_MailBox_Send? Explain.

The chances that the Consumer thread ever waits on OS_MailBox_Send are slim to none. The reason behind that is because Consumer's only purpose is to get the data from the producer via the FIFO and output it to the Display. Seeing that the Display thread is only created in the Consumer thread and killed immediately after being called consumer would not have to wait on Display thread. On the other hand, MailBox also waits on if there is room left in the FIFO but if we're taking data out faster than we're collecting, then the Consumer doesn't have to wait on the FIFO being full or not.