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

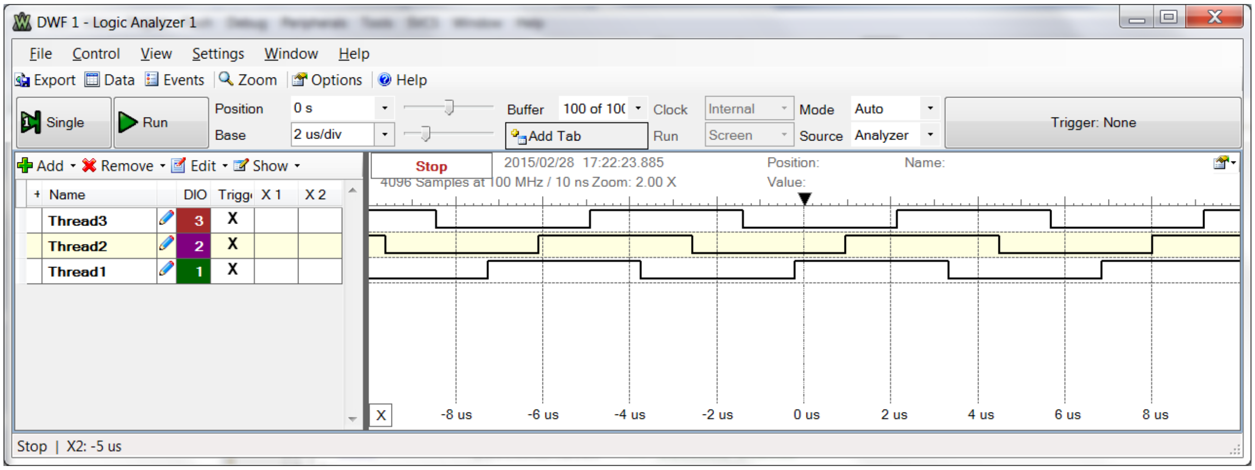


Figure 2.1 Cooperative Scheduling

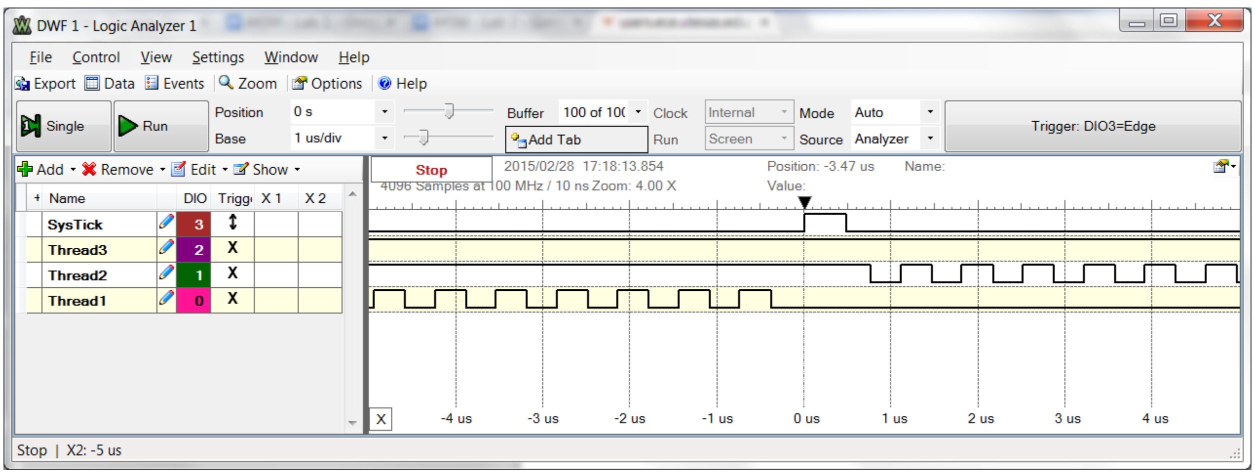


Figure 2.2 Preemptive Scheduling Using SysTick Round Robin

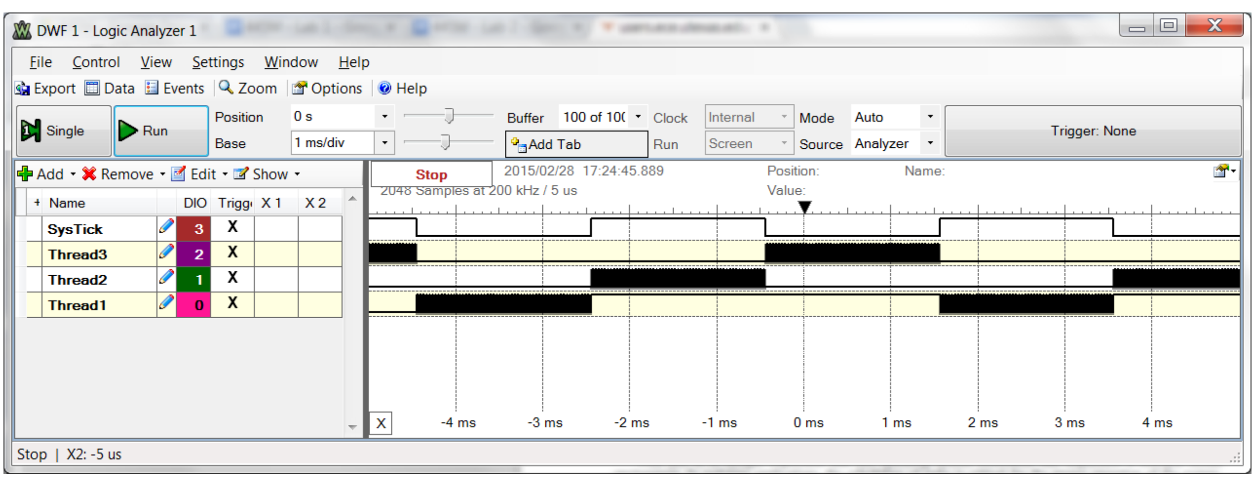


Figure 2.3 Preemptive Scheduling Using SysTick Round Robin

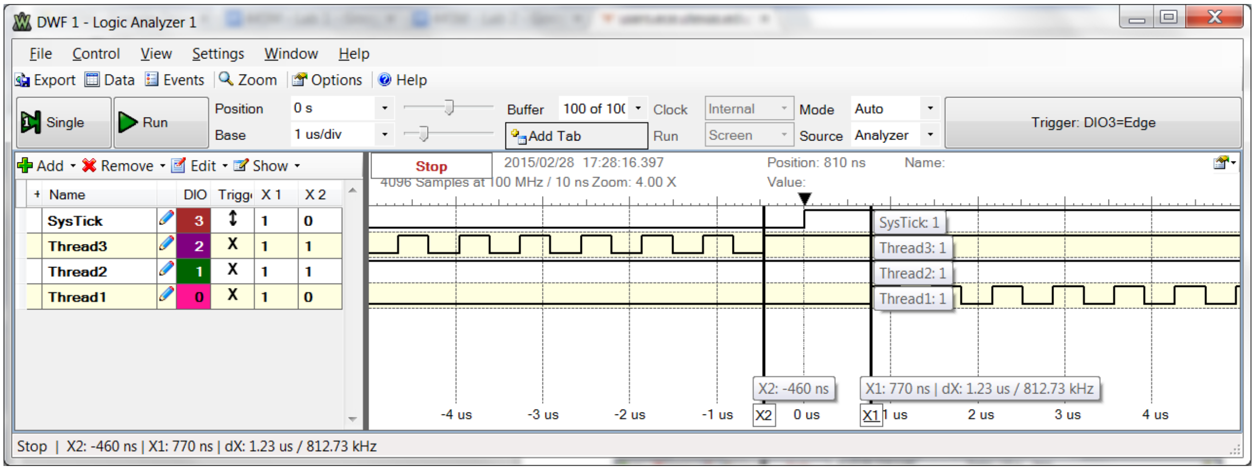


Figure 2.8 Context Switch Time

2) measurement of the thread-switch time
470 ns

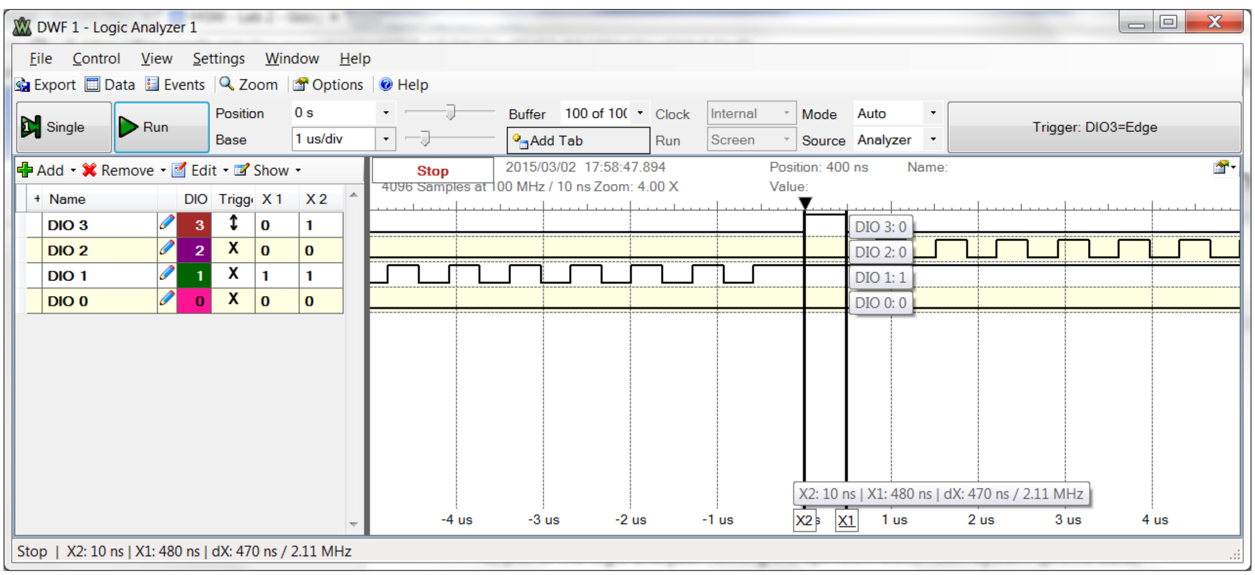
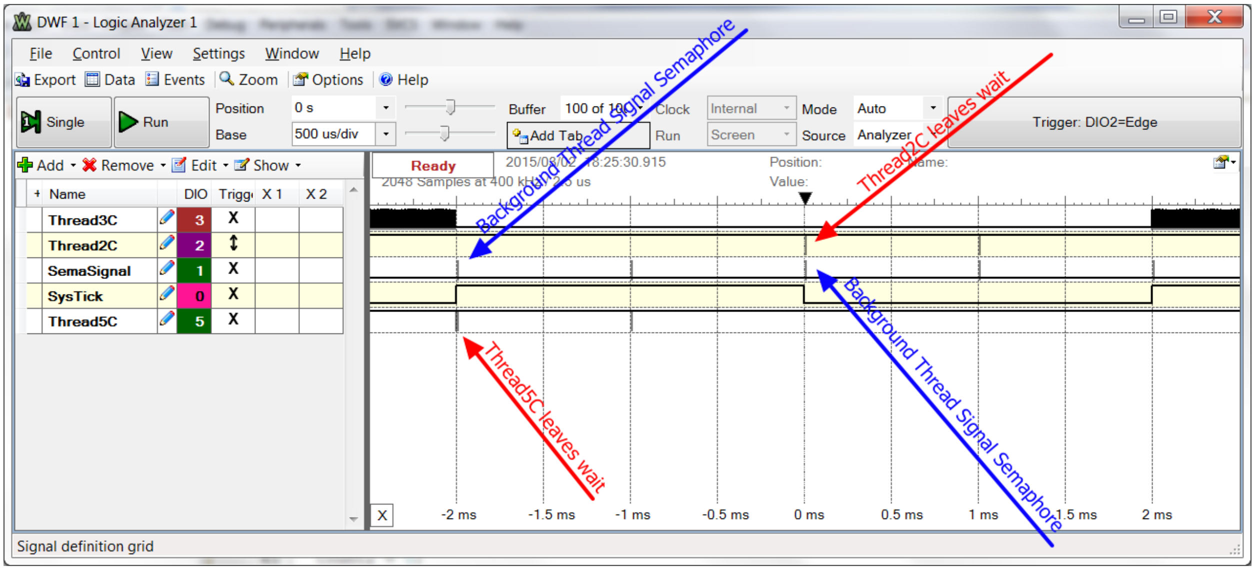
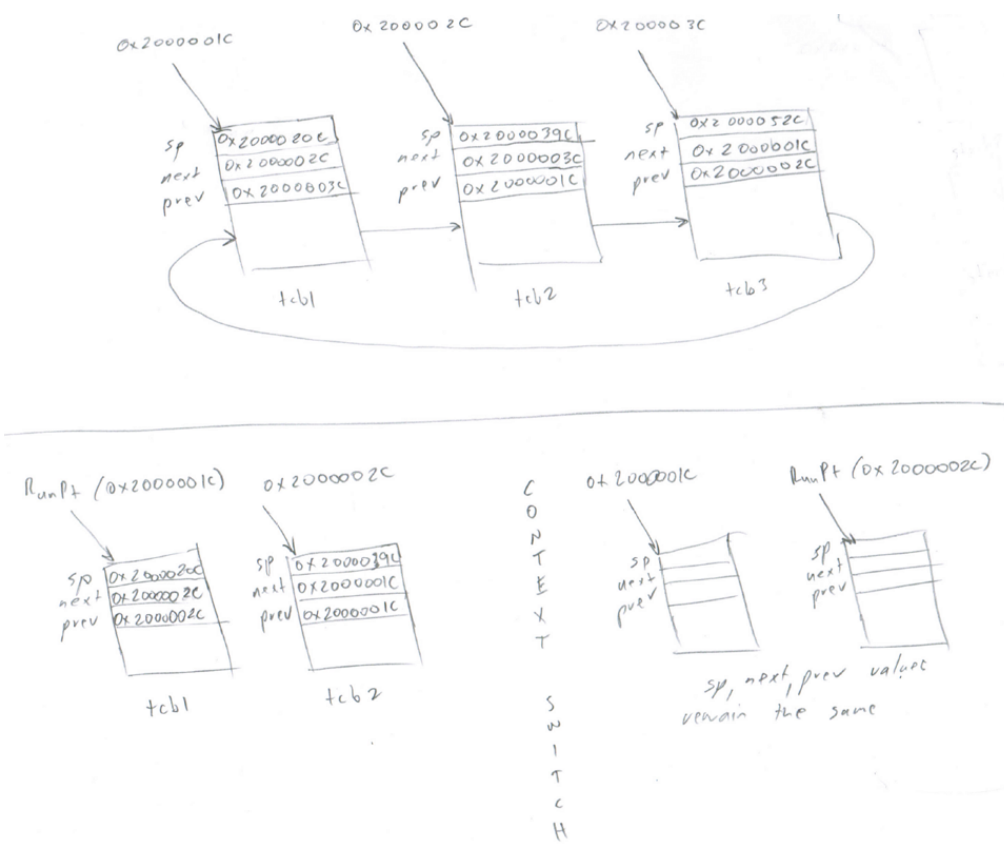


Figure 1.1 Thread Switch Time: 470 ns

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.

```c
#include "OS.h"
#include "PLL.h"
#include "tm4c123gh6pm.h"
#include <stdint.h>
#include "pins.h"
#include "Timer2A.h"

/*--------- TCB Stucture ---------*/

#define NUMTHREADS  10        // maximum number of threads
#define STACKSIZE   128      // number of 32-bit words in stack

struct tcb{
  int32_t           *sp;                // pointer to stack (valid for threads not running
  struct tcb        *next;                  // linked-list pointer to next
    struct tcb      *prev;                  // linked-list pointer to previous
    char                    ID;                         // identifies thread
    uint32_t            sleep;                  // sleep status
    char                    priority;               // priority of thread
    char                    blockedState;       // blocked status
    char                    empty;                  // Tell whether or not a thread is empty.
};
typedef struct tcb tcb;         // typedef tcb as tcbType
tcb tcbs[NUMTHREADS];                // allocate memory for NUMTHREADS threads
tcb *RunPt;                                 // pointer to running thread
int32_t Stacks[NUMTHREADS][STACKSIZE];      // allocate memory on stack outside TCB

/*--------- Set Up Initial Stack for Thread ---------*/

// function definitions in osasm.s
void OS_DisableInterrupts(void); // Disable interrupts
void OS_EnableInterrupts(void);  // Enable interrupts
int32_t StartCritical(void);
void EndCritical(int32_t primask);
void StartOS(void);
int threadMaxed = 0;
tcb *firstThread = &tcbs[0];
int OS_AddThread(void(*task)(void), unsigned long stackSize, unsigned long priority){
    int32_t status;
    status = StartCritical();
    int threadNum = 0;
    while(tcbs[threadNum].empty == 1){
        threadNum++;
    }
// Successfully add thread to linked list
    tcbs[threadNum].sp = &Stacks[threadNum][stackSize-16]; // thread stack pointer

    /* Check next thread condition */
    if(threadNum == 0){ tcbs[0].next = &tcbs[0];}                       // If there is only one
    thread, then the thread loops back to itself
    else{
        tcbs[threadNum].next = &tcbs[0];                        // Make sure second to last thread
        points to last thread now, and last thread loops back
```

```
        tcbs[0].prev->next = &tcbs[threadNum];
    }
    /******************************/
    /* Check previous thread condition */
    if(threadNum == 0) { tcbs[0].prev = &tcbs[0];}                  // If there is one one
    thread in system, then previous loops back to intself
    else {
        tcbs[threadNum].prev = tcbs[0].prev;                        // Make sure Thread1->prev
        points to last thread, and LastThread->prev points to second to last thread
        tcbs[0].prev = &tcbs[threadNum];
    }
    /**********************************/
    tcbs[threadNum].sleep = 0;
    tcbs[threadNum].priority = priority;
    Stacks[threadNum][stackSize-1] = 0x01000000;   // thumb bit
    Stacks[threadNum][stackSize-2] = (int32_t)(task); // PC
  Stacks[threadNum][stackSize-3] = 0x14141414;   // R14
  Stacks[threadNum][stackSize-4] = 0x12121212;   // R12
  Stacks[threadNum][stackSize-5] = 0x03030303;   // R3
  Stacks[threadNum][stackSize-6] = 0x02020202;   // R2
  Stacks[threadNum][stackSize-7] = 0x01010101;   // R1
  Stacks[threadNum][stackSize-8] = 0x00000000;   // R0
  Stacks[threadNum][stackSize-9] = 0x11111111;   // R11
  Stacks[threadNum][stackSize-10] = 0x10101010;  // R10
  Stacks[threadNum][stackSize-11] = 0x09090909;  // R9
  Stacks[threadNum][stackSize-12] = 0x08080808;  // R8
  Stacks[threadNum][stackSize-13] = 0x07070707;  // R7
  Stacks[threadNum][stackSize-14] = 0x06060606;  // R6
  Stacks[threadNum][stackSize-15] = 0x05050505;  // R5
  Stacks[threadNum][stackSize-16] = 0x04040404;  // R4
// Sort Linked List based off of priorites
    if(threadNum > 0) {
        tcb *lastThread = &tcbs[threadNum];
        while(firstThread->next != &tcbs[0]) {
            if(lastThread->priority < firstThread->priority) {
                firstThread->prev->next = firstThread->next;
                firstThread->next->prev = firstThread->prev;
                firstThread->prev = firstThread->next;
                firstThread->next = firstThread->next->next;
                firstThread->prev->next = firstThread;
                firstThread->next->prev = firstThread;
            }
            firstThread = firstThread->next;
        }
        firstThread = &tcbs[0];
        for(int threadIndex = 0; threadIndex <= threadNum; threadIndex++) {
            if(firstThread->priority > tcbs[threadIndex].priority){
                firstThread = &tcbs[threadIndex];
            }
        }
    }
    tcbs[threadNum].empty = 1;
    EndCritical(status);
```

```
    return threadMaxed;
}

void OS_Init(void){
  DisableInterrupts();
  PLL_Init();                 // set processor clock to 80 MHz
    Debug_Port_Init();
    tcbs_Init();
    Timer2A_Init();
  NVIC_ST_CTRL_R = 0;         // disable SysTick during setup
  NVIC_ST_CURRENT_R = 0;      // any write to current clears it
  NVIC_SYS_PRI3_R =(NVIC_SYS_PRI3_R&0x00FFFFFF)|0x60000000; // priority 6
    NVIC_SYS_PRI3_R =(NVIC_SYS_PRI3_R&0xFF00FFFF)|0x00E00000; // priority 7
}

void OS_Launch(unsigned long theTimeSlice){
    NVIC_ST_RELOAD_R = theTimeSlice - 1;
    NVIC_ST_CTRL_R = 0x07; // enable, core clock and interrupt arm
    RunPt = firstThread;        // firstThread is the highest priority thread
    StartOS();
}

int count = 0;
void SysTick_Handler(){
    DIO0 ^= BIT0;
    while(RunPt->next->sleep > 0) {
        RunPt->next->sleep = RunPt->next->sleep - 1;
        RunPt = RunPt->next;
    }
        NVIC_INT_CTRL_R = NVIC_INT_CTRL_PEND_SV;
}



/* Initialize all tcbs to empty */
void tcbs_Init(void){
    int threadIndex;
    for(threadIndex = 0; threadIndex < NUMTHREADS; threadIndex++) {
        tcbs[threadIndex].empty = 0;
    }
}
//Semaphore Functions
void OS_InitSemaphore(Sema4Type *semaPt, long value){
    semaPt->Value = value;
}
void OS_Wait(Sema4Type *semaPt){
    DisableInterrupts();
    while(semaPt->Value <= 0){
        EnableInterrupts();
        DisableInterrupts();
    }
    semaPt->Value = semaPt->Value - 1;
    EnableInterrupts();
}
```

```
void OS_Signal(Sema4Type *semaPt){
    long status;
    status = StartCritical();
    semaPt->Value = semaPt->Value + 1;
    EndCritical(status);
}

void OS_bWait(Sema4Type *semaPt){
    DisableInterrupts();
    while(semaPt->Value == 0){
        EnableInterrupts();
        DisableInterrupts();
    }
    semaPt->Value = semaPt->Value - 1;
    EnableInterrupts();
}

void OS_bSignal(Sema4Type *semaPt){
    long status;
    status = StartCritical();
    semaPt->Value = 1;
    EndCritical(status);
}

void (*SW1Task)(void);

int OS_AddSW1Task(void(*task)(void), unsigned long priority){
    volatile unsigned long delay;
    SW1Task = task;
  SYSCTL_RCGCGPIO_R |= 0x00000020; // (a) activate clock for port F
    delay = SYSCTL_RCGC2_R;                        // settle
  GPIO_PORTF_DIR_R &= ~0x10;    // (c) make PF4 in (built-in button)
  GPIO_PORTF_AFSEL_R &= ~0x10;  //     disable alt funct on PF4
  GPIO_PORTF_DEN_R |= 0x10;     //     enable digital I/O on PF4
  GPIO_PORTF_PCTL_R &= ~0x000F0000; // configure PF4 as GPIO
  GPIO_PORTF_AMSEL_R = 0;       //     disable analog functionality on PF
  GPIO_PORTF_PUR_R |= 0x10;     //     enable weak pull-up on PF4
  GPIO_PORTF_IS_R &= ~0x10;     // (d) PF4 is edge-sensitive
  GPIO_PORTF_IBE_R &= ~0x10;    //     PF4 is not both edges
  GPIO_PORTF_IEV_R &= ~0x10;    //     PF4 falling edge event
  GPIO_PORTF_ICR_R = 0x10;      // (e) clear flag4
  GPIO_PORTF_IM_R |= 0x10;      // (f) arm interrupt on PF4 *** No IME bit as mentioned in Book
  ***
  NVIC_PRI7_R = (NVIC_PRI7_R&0xFF00FFFF)|0x00A00000; // (g) priority 5
  NVIC_EN0_R = 0x40000000;      // (h) enable interrupt 30 in NVIC
    return 1;
}

int handler_count = 0;

void GPIOPortF_Handler(void){
    DisableInterrupts();
```

```
    DIO3 ^= BIT3;
    handler_count++;
    GPIO_PORTF_ICR_R = 0x10;
    EnableInterrupts();
    (*SW1Task)();
}

int OS_AddPeriodicThread(void(*task)(void), unsigned long period, unsigned long priority){
    Timer2A_Launch(task, period);
    return 1;
}

void OS_Sleep(unsigned long sleepTime){
    DisableInterrupts();
    RunPt->sleep = sleepTime;
    EnableInterrupts();
    SysTick_Handler();
}
void OS_Kill(void){
    DisableInterrupts();
    RunPt->prev->next = RunPt->next;
    RunPt->next->prev = RunPt->prev;
    RunPt->empty = 0;
    EnableInterrupts();
    SysTick_Handler();
}

void OS_MailBox_Init(void){}
void OS_Fifo_Init(unsigned long size){}
int OS_AddSW2Task(void(*task)(void), unsigned long priority){}
unsigned long OS_Time(void){ unsigned long time;
    DisableInterrupts();
    time = NVIC_ST_CURRENT_R;
    EnableInterrupts();
    return time;
}
unsigned long OS_TimeDifference(unsigned long start, unsigned long stop){}

/*---------- Future OS Functions -----------*/

unsigned long OS_Id(void){}
int OS_Fifo_Put(unsigned long data){}
unsigned long OS_Fifo_Get(void){}
long OS_Fifo_Size(void){}
void OS_MailBox_Send(unsigned long data){}
unsigned long OS_MailBox_Recv(void){}
void OS_ClearMsTime(void){}
unsigned long OS_MsTime(void){}
```