Jonathan W. Valvano First Name: ________________ Last Name:____________________
March 3, 2017, 10:00 to 10:50am

Open book and open notes. No calculators or any electronic devices (turn cell phones off). Please be sure that your answers to all questions (and all supporting work that is required) are contained in the space (boxes) provided. Anything outside the boxes will be ignored in grading. For all questions, unless otherwise stated, find the most efficient (time, resources) solution.

(15) Question 1. Consider these three implementations that set bit two of an output port, as performed on different architectures. The port does have multiple bits that are shared with other threads. In each case, however, the code does indeed set bit 2 of the output port **without** having a critical section. The normal address of Port F is 0x400253FC, which we use to access all bits.

```
;Freescale 9S12
;Set Port T bit 2
  BSET PTT,#4
```

```
;Cortex M version 1
;Set PF2 on TM4C123
   LDR     R0,=0x40025010
   LDR     R1, [R0]
   ORR     R1, #4
   STR     R1, [R0]
```

```
;Cortex M version 2
   LDR     R0,=0x400253FC
L: LDREX   R1, [R0]
   ORR     R1, #4
   STREX   R2,R1,[R0]
   CMP     R2, #0
   BNE     L
```

(5) Part a) Why does the 9S12 code not have a critical section?

(5) Part b) Why does the Cortex M version 1 code not have a critical section?

(5) Part c) Why does the Cortex M version 2 code not have a critical section?

(10) Question 2. The thread scheduler described in book section 3.3 (RTOS starter project) is used to schedule multiple tasks including this one. The semaphore **`ready`** is signaled when it is time to take a sample. The previous **`SIZE`** samples are recorded. The system operates properly at **`SIZE`**=10, but you get a hard fault when **`SIZE`** is increased to 1000.

```
#define SIZE 10
void Task1(void){ uint16_t x[SIZE];uint16_t y;
  while(1){
    for(int i=SIZE-1; i>0; i--)x[i] = x[i-1];
    OS_Wait(&ready);
    PD1 ^= 0x02; x[0] = ADC_In();
    y = DigitalFilter(x);
  }
}
```

(6) Part a) What caused the hard fault?

(4) Part b) Give two ways to remove the hard fault.

Hint: fix the user code

Hint: fix the OS

(10) Question 3. There are four hardware-triggered ISRs, with priorities 1, 3, 3, and 5. Each of the ISRs executes a real-time task. There are five main threads running with blocking semaphores. Consider the one real-time task running as a priority 3 ISR; give an equation for the worst-case latency for this real-time task. Let

A_{min}, A_{ave}, A_{max} be the minimum, average, and maximum time interrupts are disabled
B_{min}, B_{ave}, B_{max} be the minimum, average, and maximum time interrupts are enabled
C_{min}, C_{ave}, C_{max} be the minimum, average, and maximum time to run the priority 1 ISR
D_{min}, D_{ave}, D_{max} be the minimum, average, and maximum time to run the other priority 3 ISR
E_{min}, E_{ave}, E_{max} be the minimum, average, and maximum time to run the priority 5 ISR
F_{min}, F_{ave}, F_{max} be the minimum, average, and maximum time to execute one instruction

You may assume the ISRs do not interrupt themselves, and the actual times between interrupts of the same task are long compared to the time it takes to execute an ISR.

(20) Question 4. Consider this spinlock semaphore implementation with cooperation. In this system, if a thread were to wait for more than 1 second, a deadlock has occurred. Modify this implementation so it calls an OS function, **`OS_Deadlock()`**, if a thread waits for more than 1 second. You may call **`OS_Time()`**, which returns the time in 12.5 ns units. You can specify if **`OS_Time`** counts up or down. You do not have to write **`OS_Time`** or **`OS_Deadlock()`**. Multiple threads can call **`OS_Wait`** on the same or different semaphores. Do not add any fields to TCB.

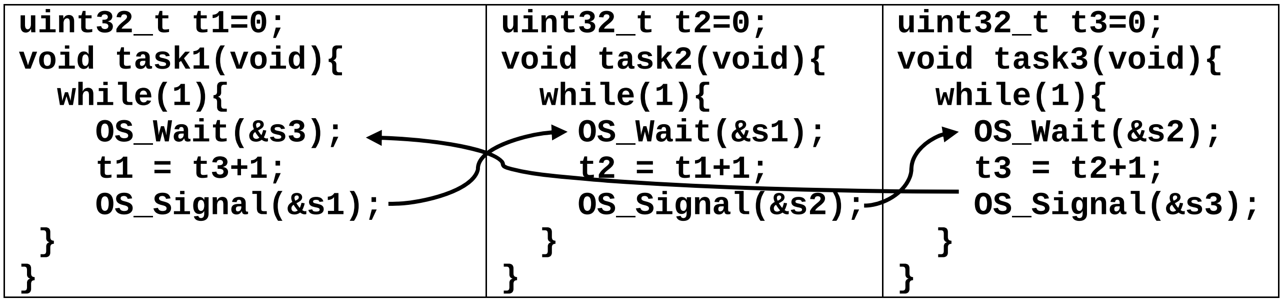
```
void OS_Wait(int32_t *s){

  DisableInterrupts();

  while((*s) == 0){

    EnableInterrupts();

    OS_Suspend();

    DisableInterrupts();

  }

  (*s) = (*s) - 1;

  EnableInterrupts();

}
```

(30) Question 5. In this question you will implement a simple preemptive round-robin OS scheduler using SysTick interrupts. In this OS, the TCBs are a simple linear array containing only the saved stack pointer. There are exactly four threads with no sleeping, no priority, no blocking, and no killing. Your OS should run the threads in the 0,1,2,3,0,1,2,3,… order using SysTick.

```
#define STACKSIZE 100  // 400 bytes of stack
int32_t *tcbs[4];      // saved stack pointer for each thread
uint32_t RunI;         // index of currently running thread (0,1,2,3)
int32_t Stacks[4][STACKSIZE];
void OS_AddThreads(void(*task0)(void), void(*task1)(void),
                   void(*task2)(void), void(*task3)(void)){
  tcbs[0] = &Stacks[0][STACKSIZE-16];        // thread stack pointer
  Stacks[0][STACKSIZE-1] = 0x01000000;       // thumb bit
  Stacks[0][STACKSIZE-2] = (int32_t)(task0); // PC
  tcbs[1] = &Stacks[1][STACKSIZE-16];        // thread stack pointer
  Stacks[1][STACKSIZE-1] = 0x01000000;       // thumb bit
  Stacks[1][STACKSIZE-2] = (int32_t)(task1); // PC
  tcbs[2] = &Stacks[2][STACKSIZE-16];        // thread stack pointer
  Stacks[2][STACKSIZE-1] = 0x01000000;       // thumb bit
  Stacks[2][STACKSIZE-2] = (int32_t)(task2); // PC
  tcbs[3] = &Stacks[3][STACKSIZE-16];        // thread stack pointer
  Stacks[3][STACKSIZE-1] = 0x01000000;       // thumb bit
  Stacks[3][STACKSIZE-2] = (int32_t)(task3); // PC
  RunI = 0;                                  // thread 0 will run first
}
```

Assume SysTick is configured to interrupt every 1 ms at priority 7. There are other ISRs running other background tasks running at higher priority than 7. **Show the assembly code** for the SysTick ISR. You may not add variables or make any changes to above code. Use comments to explain what you are doing.

(15) Question 6. Consider these foreground threads that will run with your Lab 2 OS. Together they form an assembly line, taking data from the previous thread, incrementing the value, and then passing the data to the next thread. These three threads are exactly as shown; no other code other than these threads exists. You may assume the three blocking semaphores are all initialized to zero before the threads are launched. When running properly, the variables should increase by 3 each time through the loop.



```
uint32_t t1=0;
void task1(void){
  while(1){
    OS_Wait(&s3);
    t1 = t3+1;
    OS_Signal(&s1);
 }
}
```

```
uint32_t t2=0;
void task2(void){
  while(1){
    OS_Wait(&s1);
    t2 = t1+1;
    OS_Signal(&s2);
  }
}
```

```
uint32_t t3=0;
void task3(void){
  while(1){
    OS_Wait(&s2);
    t3 = t2+1;
    OS_Signal(&s3);
  }
}
```

(7) Part a) Explain why this system gets stuck. It has a bug, tell me the bug

(8) Part b) Give one way to remove the bug. When working properly, **task1** runs through its while loop, then **task2** runs through its while loop, and then the **task3** runs through its while loop. This cycles runs over and over. One of the variables should go 1,4,7,10... Another variable should go 2,5,8,11... Another variable should go 3,6,9,12,...