Jonathan W. Valvano First Name: _____ Last Name:_____
February 28, 2014, 10:00 to 10:50am

Quiz 1 is a closed book exam. You may have two-sided 8.5 by 11 inch sheet of hand-written crib notes, but no books or electronic devices. You may put answers on the backs of the pages.

(10) Question 0. Please staple your crib sheet to your exam. Your crib sheet will be graded on content and correctness.

(5) Question 1. Consider the reader-writer problem. If a thread wished to read it calls **ROpen()**, reads, and then calls **RClose()**. If a thread wished to write it calls **WOpen()**, writes, and then calls **WClose()**. The two lines in *italics* have been switched from the classical implementation.

ROpen **wait**(&**mutex**); **ReadCount**++; ***signal(&mutex);*** ***if(ReadCount==1) wait(&wrt)***	**WOpen** **wait**(&**wrt**);
RClose **wait(&mutex);** **ReadCount--;** **if(ReadCount==0) signal(&wrt)** **signal(&mutex);**	**WClose** **signal**(&**wrt**);

Circle the best answer that describes what will happen with this implementation of reader-writer.

A) Rotating the two italicized lines has no effect; it works either way.

B) Rotating the two italicized lines makes it better; because signal occurs earlier, the overall bandwidth is increased.

C) Because of the read-modify-write the solution now has a critical section.

D) It is now broken, and may allow multiple readers to read at the same time a writer is writing.

E) It is now broken, and may allow multiple writers to write at the same time a reader is reading.

(5) Question 2.

Part a) Give a general definition of stabilize with respect to debugging.

Part b) Give an example from Labs 2 and 3 were stabilization is used.

(20) Question 3. Consider a problem of running two foreground threads (**client** and **server**) using a preemptive scheduler with semaphore synchronization. The **Init()** function, which you do not need to write, will initialize Ports B and D as inputs and initialize Port E as an output. There are three shared 32-bit global variables:

```
unsigned long In1,In2,Out;
```

First, the **client** thread should initialize the ports. Each time through the loop, the **client** thread should create two pieces of data and store them in **In1** and **In2**. Once new data are available in **In1** and **In2**, the **server** thread should calculate the average of these two numbers and place the result in **Out**. Once a new calculation is complete and the result is available in **Out**, the **client** thread should output the result. The basic shell of this operation is given. Define one or more semaphores, and then add calls to the following semaphore functions in order to properly synchronize the interactions between **client** and **server**.

```
void OS_Wait(Sema4Type *semaPt);
void OS_Signal(Sema4Type *semaPt);
```

You will define one or more semaphores and place calls to the wait and signal functions, otherwise no other changes are allowed. For each semaphore you add, explain what it means to be 0, 1 etc. You may not assume **client** runs first. Specify the initial values for each semaphore you add.

```
void client(void){
  Init(); // set up B,D,E ports
  while(1){
    In1=GPIO_PORTB_DATA_R; // read
    In2=GPIO_PORTD_DATA_R; // read
    GPIO_PORTE_DATA_R=Out; // write
  }
}
```

```
void server(void){
  while(1){
    Out = (In1+In2)/2; // average
  }
}
```

(10) Question 4. Consider this example of two background threads. Timer0A runs with a priority of 2 and SysTick runs with a priority of 3. The assembly code generated by the compiler follows.

```
void SysTick_Handler(void){
  static unsigned long cnt=0;
  cnt = cnt + 1;
}
```

```
void Timer0A_Handler(void){
  static unsigned long cnt=0;
  cnt = cnt + 1;
  TIMER0_ICR_R = TIMER_ICR_CAECINT;
}
```

```
                SysTick_Handler
0x00000414 4808     LDR           r0,[pc,#32]   ; @0x00000438 (address of cnt)
0x00000416 6800     LDR           r0,[r0,#0x00] ; value of cnt
0x00000418 1C40     ADDS          r0,r0,#1
0x0000041A 4907     LDR           r1,[pc,#28]   ; @0x00000438 (address of cnt)
0x0000041C 6008     STR           r0,[r1,#0x00] ; update cnt
0x0000041E 4770     BX            lr
                Timer0A_Handler
0x00000420 4806     LDR           r0,[pc,#24]   ; @0x0000043C (address of cnt)
0x00000422 6800     LDR           r0,[r0,#0x00] ; value of cnt
0x00000424 1C40     ADDS          r0,r0,#1
0x00000426 4905     LDR           r1,[pc,#20]   ; @0x0000043C (address of cnt)
0x00000428 6008     STR           r0,[r1,#0x00] ; update cnt
0x0000042A 2004     MOVS          r0,#0x04      ; TIMER_ICR_CAECINT
0x0000042C 4904     LDR           r1,[pc,#16]   ; @0x00000440 (address of ICR)
0x0000042E 6248     STR           r0,[r1,#0x24]
0x00000430 4770     BX            lr
0x00000438 20000000 DCD           0x20000000
0x0000043C 20000004 DCD           0x20000004
0x00000440 40030000 DCD           0x40030000
```

Part a) Does this system have a critical section?

Part b) If you think it has a critical section, specify the exact beginning and end of the critical section by adding two arrows pointing into the assembly code. If you do not think it has a critical section, justify your choice with a reason why no critical section exists.

(20) Question 5. There are three FIFOs like the following, implemented with blocking semaphores. Each of the three FIFOs has its own globals and semaphores (replace ? with 1,2,3)

```
int FIFO?_Put(long data){
  OS_Wait(&RoomLeft?);
  OS_Wait(&mutex?);
  *(PutPt?++) = data;    // Put
  if(PutPt? == &Fifo?[FIFOSIZE]){
    PutPt? = &Fifo?[0];  // wrap
  }
  OS_Signal(&mutex?);
  OS_Signal(&CurrentSize?);
}
```

```
int FIFO?_Get(void){long data;
  OS_Wait(&CurrentSize?);
  OS_Wait(&mutex?);
  data = *(GetPt?++); // Get
  if(GetPt? == &Fifo?[FIFOSIZE]){
    GetPt? = &Fifo?[0];   // wrap
  }
  OS_Signal(&mutex?);
  OS_Signal(&RoomLeft?); return data;
}
```

The system as three foreground threads that communicate using the FIFOs in a circular fashion.

```
void T1(void){long x,y;
  while(1){
    x = FIFO3_Get();
    y = x+1;
    FIFO1_Put(y);
  }
}
```

```
void T2(void){long x,y;
  while(1){
    x = FIFO1_Get();
    y = 2*x;
    FIFO2_Put(y);
  }
}
```

```
void T3(void){long x,y;
  while(1){
    x = FIFO2_Get();
    y = x-1;
    FIFO3_Put(y);
  }
}
```

The system is initialized with one element, value=0, in FIFO1. The other two FIFOs are initially empty. The three foreground threads are run with a priority scheduler. T1 has a high priority. T2 and T3 have an equal but lower priority.

Part a) Can you choose a FIFO large enough so a deadlock never occurs and the FIFOs never fill? Justify your answer.

Part b) If it can run without deadlock/full error, how large does the FIFO need to be to prevent a deadlock/full error? If a deadlock/full will occur, give an execution sequence that causes the deadlock/full.

(10) Question 6. The following system samples PE3 at 1000 Hz. You may assume this **OS_FIFO** never fills. Assume there are no bugs in this code, this is timer-triggered ADC sampling.

```
void ADC0_InitTimer0ATriggerSeq3(void){ // sample PE3 (channel 0)
  volatile unsigned long delay;
  SYSCTL_RCGCGPIO_R |= SYSCTL_RCGCGPIO_R4; break;
  delay = SYSCTL_RCGCGPIO_R;  // 2) allow time for clock to stabilize
  delay = SYSCTL_RCGCGPIO_R;
  GPIO_PORTE_DIR_R &= ~0x08;  // 3.0) make PE3 input
  SYSCTL_RCGC0_R |= SYSCTL_RCGC0_ADC0;      // activate ADC0 (legacy code)
  SYSCTL_RCGC1_R |= SYSCTL_RCGC1_TIMER0;    // activate timer0 (legacy code)
  delay = SYSCTL_RCGC1_R;                   // allow time to finish activating
  TIMER0_CTL_R &= ~TIMER_CTL_TAEN;          // disable timer0A during setup
  TIMER0_CTL_R |= TIMER_CTL_TAOTE;          // enable timer0A trigger to ADC
  TIMER0_CFG_R = TIMER_CFG_16_BIT;          // configure for 16-bit timer mode
  TIMER0_TAMR_R = TIMER_TAMR_TAMR_PERIOD;   // configure for periodic mode
  TIMER0_TAPR_R = 79;                       // prescale so 1us
  TIMER0_TAILR_R = 999;                     // triggers at 1kHz, every 1ms
  TIMER0_IMR_R &= ~TIMER_IMR_TATOIM;        // disable timeout interrupt
  TIMER0_CTL_R |= TIMER_CTL_TAEN;           // enable timer0A 16-b, periodic
  ADC0_PC_R &= ~ADC_PC_SR_M;                // clear max sample rate field
  ADC0_PC_R |= ADC_PC_SR_125K;              // configure for 125K samples/sec
  ADC0_SSPRI_R = 0x0123;
  ADC0_ACTSS_R &= ~ADC_ACTSS_ASEN3;         // disable sample sequencer 3
  ADC0_EMUX_R &= ~ADC_EMUX_EM3_M;           // clear SS3 trigger select field
  ADC0_EMUX_R += ADC_EMUX_EM3_TIMER;        // configure for timer trigger event
  ADC0_SSMUX3_R = 0;                        // channel 0
  ADC0_SSCTL3_R = 0x06;                     // END and IE
  ADC0_IM_R |= ADC_IM_MASK3;                // enable SS3 interrupts
  ADC0_ACTSS_R |= ADC_ACTSS_ASEN3;          // enable sample sequencer 3
  NVIC_PRI4_R = (NVIC_PRI4_R&0xFFFF00FF)|0x00004000; // bits 13-15
  NVIC_EN0_R = NVIC_EN0_INT17;              // enable interrupt 17 in NVIC
  EnableInterrupts();
}
void ADC0Seq3_Handler(void){ unsigned long ADCvalue;
  ADC0_ISC_R = ADC_ISC_IN3;                 // acknowledge ADC sequencer 3
  OS_FIFO_Put(ADC0_SSFIFO3_R);
}
```

The priority of this ADC interrupt is level 2. Assume there are three other ISRs in addition to this ADC ISR. One ISR is priority level 1 with a maximum execution time of 10μs, one ISR is priority level 2 with a maximum execution time of 20μs, and the last ISR is priority level 3 with a maximum execution time of 30μs. The maximum slew rate of the ADC input voltage is 1V/ms. First, estimate the maximum jitter in the ADC sampling (causing the ADC data to be imperfect), and then use the ADC sampling jitter to calculate the maximum voltage error in the data caused by ADC sampling jitter.

(20) Question 7. Consider this round robin scheduler OS. A sleep parameter exists in the TCB, and the preemptive thread switch occurs every *Δt*. Non-cooperative spinlock semaphores are used. The italicized lines are added from the scheduler in the book, which implement sleeping. SysTick runs at interrupt priority level 7. Assume there are no other interrupt service routines (just SysTick and the timer that decrements the sleep parameters).

```
struct tcb{
  long *sp;          // pointer to stack, valid for threads not running
  struct tcb *next;  // linked-list pointer
  unsigned long sleep;
};
SysTick_Handler                 ; 1) Saves R0-R3,R12,LR,PC,PSR
     CPSID   I                  ; 2) Prevent interrupt during switch
     PUSH    {R4-R11}           ; 3) Save remaining regs r4-11
     LDR     R0, =RunPt         ; 4) R0=pointer to RunPt, old thread
     LDR     R1, [R0]           ;    R1 = RunPt
     STR     SP, [R1]           ; 5) Save SP into TCB
loop LDR     R1, [R1,#4]        ; 6) R1 = RunPt->next
     LDR     R2, [R1,#8]        ; sleep parameter
     CMP     R2, #0
     BNE     loop               ; skip this thread if sleeping
     STR     R1, [R0]           ;    RunPt = R1
     LDR     SP, [R1]           ; 7) new thread SP; SP = RunPt->sp;
     POP     {R4-R11}           ; 8) restore regs r4-11
     CPSIE   I                  ; 9) tasks run with interrupts enabled
     BX      LR                 ; 10) restore R0-R3,R12,LR,PC,PSR
```

When a thread wishes to sleep it calls this **OS_Sleep** function

```
void OS_Sleep(unsigned long time){
  RunPt->sleep = time;
  NVIC_ST_Current = 0; // next thread gets a full Δt time slice
  NVIC_INT_CNTR_R = 0x04000000; // writing 1 to bit 26 triggers a SysTick ISR
}
```

Part a) To implement sleeping, a periodic interrupt runs every 10ms (priority level 6), decrementing the sleep parameter for every sleeping thread. There are at most *n* threads in the circular linked-list. If a thread calls **OS_Sleep** with a parameter of *m* (*m*=2 means sleep 20ms), what is the minimum and maximum time the thread will actually sleep? Show your work and explain your equations.

Part b) What happens to this OS if all threads are sleeping? Will it crash? Or, will some threads eventually stop sleeping and the scheduler will continue to run again?