EGEC 451 Lab 4: A Blocking RTOS Schedule

Dominic Jacobo

1. **OS.c:**

```
// os.c
// Runs on LM4F120/TM4C123
// A very simple real time operating system with minimal features.

#include <stdint.h>
#include "os.h"
#include "PLL.h"
#include "tm4c123gh6pm.h"

// 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);

#define NUMTHREADS  3        // maximum number of threads
#define STACKSIZE   100      // number of 32-bit words in stack
struct tcb{ // This is the thread control block
  int32_t *sp;       // pointer to stack (valid for threads not running
  uint32_t id; // thread identifier number
  struct tcb *next;  // linked-list pointer
  uint32_t *bi; // block identifier
};
typedef struct tcb tcbType;
tcbType tcbs[NUMTHREADS];
tcbType *RunPt; // Used to point to the running thread
int32_t Stacks[NUMTHREADS][STACKSIZE];

//uint32_t test = 7;


// ******** OS_Init ************
// initialize operating system, disable interrupts until OS_Launch
// initialize OS controlled I/O: systick, 16 MHz PLL
// Inputs: none
// Outputs: none
```

```
void OS_Init(void){
  OS_DisableInterrupts();
  PLL_Init(Bus8MHz);        // set processor clock to 8 MHz
  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)|0xE0000000; // priority 7
(Highest Prio for thread handler)
}

void SetInitialStack(int i){
  tcbs[i].sp = &Stacks[i][STACKSIZE-16]; // thread stack pointer
  Stacks[i][STACKSIZE-1] = 0x01000000;   // thumb bit
  Stacks[i][STACKSIZE-3] = 0x14141414;   // R14
  Stacks[i][STACKSIZE-4] = 0x12121212;   // R12
  Stacks[i][STACKSIZE-5] = 0x03030303;   // R3
  Stacks[i][STACKSIZE-6] = 0x02020202;   // R2
  Stacks[i][STACKSIZE-7] = 0x01010101;   // R1
  Stacks[i][STACKSIZE-8] = 0x00000000;   // R0
  Stacks[i][STACKSIZE-9] = 0x11111111;   // R11
  Stacks[i][STACKSIZE-10] = 0x10101010;  // R10
  Stacks[i][STACKSIZE-11] = 0x09090909;  // R9
  Stacks[i][STACKSIZE-12] = 0x08080808;  // R8
  Stacks[i][STACKSIZE-13] = 0x07070707;  // R7
  Stacks[i][STACKSIZE-14] = 0x06060606;  // R6
  Stacks[i][STACKSIZE-15] = 0x05050505;  // R5
  Stacks[i][STACKSIZE-16] = 0x04040404;  // R4
}

// ******** OS_AddThread ***************
// add three foreground threads to the scheduler
// Inputs: three pointers to a void/void foreground tasks
// Outputs: 1 if successful, 0 if this thread can not be added
int OS_AddThreads(void(*task0)(void),
                void(*task1)(void),
                void(*task2)(void)){ int32_t status;
  status = StartCritical();
  tcbs[0].next = &tcbs[1]; // 0 points to 1
  tcbs[1].next = &tcbs[2]; // 1 points to 2
  tcbs[2].next = &tcbs[0]; // 2 points to 0
  tcbs[0].id = 0; // id number for thread 0
  tcbs[1].id = 1; // id number for thread 1
  tcbs[2].id = 2; // id number for thread 2
  tcbs[0].bi = 0; // set block identifiers for threads to 0
```

```
  tcbs[1].bi = 0;
  tcbs[2].bi = 0;
  SetInitialStack(0); Stacks[0][STACKSIZE-2] = (int32_t)(task0); // PC
  SetInitialStack(1); Stacks[1][STACKSIZE-2] = (int32_t)(task1); // PC
  SetInitialStack(2); Stacks[2][STACKSIZE-2] = (int32_t)(task2); // PC
  RunPt = &tcbs[0];       // thread 0 will run first
  EndCritical(status);
  return 1;               // successful
}

/// ******** OS_Launch ***************
// start the scheduler, enable interrupts
// Inputs: number of bus clock cycles for each time slice
//         (maximum of 24 bits)
// Outputs: none (does not return)
void OS_Launch(uint32_t theTimeSlice){
  NVIC_ST_RELOAD_R = theTimeSlice - 1; // reload value
  NVIC_ST_CTRL_R = 0x00000007; // enable, core clock and interrupt arm
  StartOS();                   // start on the first task
}

// ******** OS_Wait ***************
void OS_Wait(uint32_t* sig){
    OS_DisableInterrupts();
    if((*sig) == 0){
        (*RunPt).bi = sig;
        OS_EnableInterrupts();
        NVIC_INT_CTRL_R |= NVIC_INT_CTRL_PENDSTSET;
    }
    else{
        (*sig) = 0;
        OS_EnableInterrupts();
    }
}

// ******** OS_Signal ***************
void OS_Signal(uint32_t* sig){
    uint8_t i;
    uint32_t status;
    tcbType* counter = (*RunPt).next;
    status = StartCritical();

    (*sig) = 1;
```

```
   for(i = 0; i < 2; i++){
      if((*counter).bi == sig){
         (*counter).bi = 0;
         break;
      }
      counter = (*counter).next;
   }
   EndCritical(status);
}
```

OSasm.asm:

```
;/*******************************************************************
*******/
; OSasm.asm: low-level OS commands, written in assembly                    */
; Runs on LM4F120/TM4C123
; A very simple real time operating system with minimal features.

      .thumb
      .text
      .align 2


      .global  RunPt          ; currently running thread
      .global  OS_DisableInterrupts
      .global  OS_EnableInterrupts
      .global  StartOS
      .global  SysTick_Handler
      .global Time_Slice_Counter ; global time slice varible here
      .global Global_Thread_Id ; global thread identifier number


OS_DisableInterrupts:  .asmfunc
      CPSID   I
      BX      LR
     .endasmfunc

OS_EnableInterrupts:  .asmfunc
      CPSIE   I
      BX      LR
     .endasmfunc

SysTick_Handler:  .asmfunc     ; 1) Saves R0-R3,R12,LR,PC,PSR
```

```
   CPSID   I                 ; 2) Prevent interrupt during switch
Loop:
   PUSH    {R4-R11}          ; 3) Save remaining regs r4-11
   LDR     R0, RunPtAddr      ; 4) R0=pointer to RunPt, old thread
   LDR     R1, [R0]          ;    R1 = RunPt
   STR     SP, [R1]          ; 5) Save SP into TCB
   LDR     R1, [R1,#8]       ; 6) R1 = RunPt->next
   STR     R1, [R0]          ;    RunPt = R1 (The thread switches here)
   LDR     SP, [R1]          ; 7) new thread SP; SP = RunPt->sp;
   POP     {R4-R11}          ; 8) restore regs r4-11

        LDR             R1, [R0]                ; R1 = new RunPt
        LDR             R2, [R1,#12]      ; R2 = RunPt->bi
        CMP             R2, #0
        BNE             Loop                    ; If R2 is not equal to zero that means its
blocked

   LDR                  R2, [R0]                ; R2 = new RunPt
   LDR          R4, [R2,#4]       ; R4 = value of new RunPt.id
   LDR     R3, Global_Thread_IdAddr ; R3 points to Global_Thread_IdAddr
   STR          R4, [R3]                ; R3 (Global_Thread_Id) = R2 (Value of new
RunPt.id)

   LDR                  R0, Time_Slice_CounterAddr ; update the Time_Slice_Counter
        LDR R1, [R0]
        ADD R1, #1
        STR R1, [R0]

   CPSIE   I                 ; 9) tasks run with interrupts enabled
   BX      LR                ; 10) restore R0-R3,R12,LR,PC,PSR
  .endasmfunc
RunPtAddr .field RunPt,32
Time_Slice_CounterAddr .field Time_Slice_Counter,32
Global_Thread_IdAddr .field Global_Thread_Id,32

StartOS:  .asmfunc
        LDR    R0, Time_Slice_CounterAddr
        LDR             R1, [R0]                ; Load in value of Time_Slice_Counter to
R1
        MOV   R1, #0                    ; May not be needed because Time_Slice_Counter
is 0 by default
        ADD             R1, #1
```

```
        STR              R1, [R0]

   LDR     R0, RunPtAddr      ; currently running thread
   LDR     R2, [R0]           ; R2 = value of RunPt

   LDR           R1, [R2,#4]       ; R1 = value at RunPt.id
   LDR           R3, Global_Thread_IdAddr ; R3 points to Global_Thread_IdAddr
   STR     R1, [R3]                  ; R3 (Global_Thread_Id) = R1 (Value of RunPt.id)

   LDR     SP, [R2]           ; new thread SP; SP = RunPt->stackPointer;
   POP     {R4-R11}           ; restore regs r4-11
   POP     {R0-R3}            ; restore regs r0-3
   POP     {R12}
   POP     {LR}               ; discard LR from initial stack
   POP     {LR}               ; start location
   POP     {R1}               ; discard PSR
   CPSIE   I                  ; Enable interrupts at processor level
   BX      LR                 ; start first thread
  .endasmfunc
  .end
```

User.c:

```
//#include <stdio.h>
#include <stdint.h>
#include "os.h"
#include "tm4c123gh6pm.h"
#include "SSEG.h"
#include "Timer0A.h"
#include "Timer1A.h"
#include "Timer2A.h"
#include "Timer3A.h"
#include "LCD.h"
#include "Switch.h"

#define TIMESLICE 8000000  // The thread switch time in number of SysTick counts
(bus clock cycles at 8 MHz)

#define FREQUENCY 8000000.0f
#define SEC_PER_MIN 60
#define GEARBOX_RATIO 120
#define PULSE_PER_ROTATION 8

uint32_t Count1;   // number of times thread1 loops
```

```
uint32_t Count2;   // number of times thread2 loops
uint32_t Count3;   // number of times thread3 loops

uint32_t sig = 1;

int pw = 0;
uint32_t RPM = 0;

void Task1(void){
  Count1 = 0;

  // set direction
  GPIO_PORTB_DATA_R &= ~0x04;
  GPIO_PORTB_DATA_R |= 0x08;

  //OS_Wait(&sig);

  for(;;){
    Count1++;
    //GPIO_PORTF_DATA_R = (GPIO_PORTF_DATA_R & ~0x0E) | (0x01<<1);  //
Show red (Save this line for task 3)
    //Below is my attempt at the PWM DC Motor Code

    // set direction
    //GPIO_PORTB_DATA_R &= ~0x04;
    //GPIO_PORTB_DATA_R |= 0x08;



    // reverse direction
   // GPIO_PORTB_DATA_R &= ~0x08;
    //GPIO_PORTB_DATA_R |= 0x04;



    if(Mailbox_Desired_Flag == 1){ // Check for new desired speed
        OS_Wait(&sig);
        pw = (28 * Desired_RPM) + 120; // equation for setting the pw based on desired
speed
        PWM1_3_CMPB_R = pw;
        OS_Signal(&sig);
        Timer3A_Wait1ms(250); // Do I even need this?
        RPM =
(SEC_PER_MIN*FREQUENCY)/(PULSE_PER_ROTATION*GEARBOX_RATIO*CC
_Difference)*10;
```

```
        Mailbox_Desired_Flag = 0;
        Mailbox_Actual_Flag = 1;
    }
  }
}

void Task2(void){
  Count2 = 0;
  //OS_Signal(&sig);
  for(;;){
    Count2++;
   //GPIO_PORTF_DATA_R = (GPIO_PORTF_DATA_R & ~0x0E) | (0x02<<1);  //
Show blue (Save this line for task 3)
    //RPM =
(SEC_PER_MIN*FREQUENCY)/(PULSE_PER_ROTATION*GEARBOX_RATIO*CC
_Difference)*10;
    if((Mailbox_Actual_Flag == 1) || (Mailbox_ActualLCD_Flag == 1)){
        Timer0A_Wait1ms(40);
        LCD_Clear();
        LCD_OutUDec(Desired_RPM); // Put on first line
        LCD_command(0xC0); // Cursor moved to row 2 col 1
        LCD_OutUFix(RPM); // Put on second line
        Mailbox_Actual_Flag = 0;
        Mailbox_ActualLCD_Flag = 0;
    }
  }
}

void Task3(void){
  Count3 = 0;
  uint32_t Actual_RPM = 0;
  float Expected_RPM = 0.0f;
  int8_t Diff = 0;
  int32_t Percent_Error = 0;
  uint8_t temp_desired_RPM = 0;
  for(;;){
    Count3++;
    //GPIO_PORTF_DATA_R = (GPIO_PORTF_DATA_R & ~0x0E) | (0x03<<1);  //
Show red + blue = purple/magenta
    Actual_RPM =
(SEC_PER_MIN*FREQUENCY)/(PULSE_PER_ROTATION*GEARBOX_RATIO*CC
_Difference);
    Expected_RPM = (pw-120)/28;
```

```
    Diff = Expected_RPM - Actual_RPM;
    Percent_Error = (Diff/Expected_RPM)*100;

    if ((Diff > 0) && (Expected_RPM > 0)){
        if((Percent_Error > 13) && (Percent_Error < 35)){
            OS_Wait(&sig);
            Mailbox_Desired_Flag = 1;
            pw = 0;
            PWM1_3_CMPB_R = pw;
            temp_desired_RPM = Desired_RPM;
            while(Desired_RPM <= temp_desired_RPM){}
            OS_Signal(&sig);

        }
    }
  }
}

int main(void){
  OS_Init();           // initialize, disable interrupts, set PLL to 8 MHz (equivalent to
PLL_Init()) checked(this means should be ready)
  SYSCTL_RCGCGPIO_R |= 0x20;            // activate clock for Port F
  while((SYSCTL_PRGPIO_R&0x20) == 0){}; // allow time for clock to stabilize
  GPIO_PORTF_DIR_R |= 0x0E;             // make PF3-1 out
  GPIO_PORTF_AFSEL_R &= ~0x0E;          // disable alt funct on PF3-1
  GPIO_PORTF_DEN_R |= 0x0E;             // enable digital I/O on PF3-1
  GPIO_PORTF_PCTL_R &= ~0x0000FFF0;     // configure PF3-1 as GPIO
  GPIO_PORTF_AMSEL_R &= ~0x0E;          // disable analog functionality on PF3-1
  OS_AddThreads(&Task1, &Task2, &Task3);

  Timer0A_Init(8000000);
  Timer1A_Init(8000000);
  Timer2A_Init();
  Timer3A_Init(8000000);
  sevenseg_init(); // This is equivalent to SSI2_init()
  LCD_init();

  //Code for PWM init is below

   SYSCTL_RCGCPWM_R |= 0x02;        // enable clock to PWM1
   SYSCTL_RCGCGPIO_R |= 0x20;       // enable clock to GPIOF
   SYSCTL_RCGCGPIO_R |= 0x02;       // enable clock to GPIOB
```

```
  //delayMs(1);                  // PWM1 seems to take a while to start
  Timer0A_Wait1ms(1);

  SYSCTL_RCC_R &= ~0x00100000;     // use system clock for PWM
  PWM1_INVERT_R  |= 0x80;          // positive pulse
  PWM1_3_CTL_R = 0;                // disable PWM1_3 during configuration
  PWM1_3_GENB_R = 0x0000080C;      // output high when load and low when match
  PWM1_3_LOAD_R = 3999;            // 4 kHz
  PWM1_3_CTL_R = 1;                // enable PWM1_3
  PWM1_ENABLE_R |= 0x80;           // enable PWM1

  GPIO_PORTF_DIR_R |= 0x08;            // set PORTF 3 pins as output (LED) pin
  GPIO_PORTF_DEN_R |= 0x08;            // set PORTF 3 pins as digital pins
  GPIO_PORTF_AFSEL_R |= 0x08;          // enable alternate function
  GPIO_PORTF_PCTL_R &= ~0x0000F000;    // clear PORTF 3 alternate function
  GPIO_PORTF_PCTL_R |= 0x00005000;     // set PORTF 3 alternate funtion to PWM

  GPIO_PORTB_DEN_R |= 0x0C;            // PORTB 3 as digital pins
  GPIO_PORTB_DIR_R |= 0x0C;            // set PORTB 3 as output
  GPIO_PORTB_DATA_R |= 0x08;           // enable PORTB 3

 //Code for PWM init ends on the line above

  Switch_Init();


 OS_Launch(TIMESLICE); // doesn't return, interrupts enabled in here
 return 0;            // this never executes
}
```

Switch.c:

```
// Switch.c
// Runs on Tiva-C

#include <stdint.h>
#include "tm4c123gh6pm.h"
#include "Switch.h"
#include "Timer0A.h"

void Switch_Init(void){
   SYSCTL_RCGCGPIO_R |= 0x08; // activate clock for Port D
   //GPIO_PORTD_LOCK_R = 0x4C4F434B; // unlock GPIO Port D
   GPIO_PORTD_AMSEL_R &= ~0x0D; // disable analong function on PD3-0 (NEW)
```

```
  GPIO_PORTD_PCTL_R &= ~0x0000FF0F; // configure PD3-0 as GPIO (NEW)
  //GPIO_PORTD_CR_R = 0x0D; // allow changes to PD3-0
  GPIO_PORTD_DIR_R &= ~0x0D; // make PD3-0 in
  GPIO_PORTD_AFSEL_R &= ~0x0D; // disable alt funct on PD3-0
  GPIO_PORTD_DEN_R |= 0x0D; // enable digistal I/O on PD3-0
  //GPIO_PORTD_PUR_R |= 0x0F; // pullup on PD3-0
  GPIO_PORTD_IS_R &= ~0x0D; // PD3-0 are edge-sensitive
  GPIO_PORTD_IBE_R &= ~0x0D; // PD3-0 are single edge
  GPIO_PORTD_IEV_R |= 0x0D; // PD3-0 rising edge triggered
  GPIO_PORTD_ICR_R = 0x0D; // clear flags
  GPIO_PORTD_IM_R |= 0x0D; // arm interrupts on PD3-0
  NVIC_PRI0_R = (NVIC_PRI0_R&0x00FFFFFF) | 0x20000000; // priority 1
  NVIC_EN0_R = 0x08; // enable interrupt 3 in NVIC
}

void GPIOPortD_Handler(void){
  Timer0A_Wait1ms(25); // Wait for switch to stabilize (aka wait for debouncing)

  if(GPIO_PORTD_RIS_R&0x08){ // poll PD3 (aka SW2 check)
    GPIO_PORTD_ICR_R = 0x08; // acknowledge flag3

    //Speed up by 10 RPM (Use global variable for desired speed and update it here)
    if((Desired_RPM <= 140) && (Desired_RPM >= 0)){
      Desired_RPM += 10;
    }
    //Add a mailbox here to set the flag, which will tell the task a new speed has been requested
    Mailbox_Desired_Flag = 1;
    Mailbox_ActualLCD_Flag = 1;
  }
  if(GPIO_PORTD_RIS_R&0x04){ // poll PD2 (aka SW3 check)
    GPIO_PORTD_ICR_R = 0x04; // acknowledge flag2

    //Slow down by 10 RPM (Use global variable for desired speed and update it here)
    if(Desired_RPM >= 10){
      Desired_RPM -= 10;
    }
    //Add a mailbox here to set the flag, which will tell the task a new speed has been requested
    Mailbox_Desired_Flag = 1;
    Mailbox_ActualLCD_Flag = 1;
  }
  if(GPIO_PORTD_RIS_R&0x01){ // poll PD0 (aka SW5 check)
```

```
        GPIO_PORTD_ICR_R = 0x01; // acknowledge flag0

        //Set speed to 0 RPM (Use global variable for desired speed and update it here)
        Desired_RPM = 0;
        //Add a mailbox here to set the flag, which will tell the task a new speed has been
requested
        Mailbox_Desired_Flag = 1;
        Mailbox_ActualLCD_Flag = 1;
    }
}
```

Switch.h:

```
// Timer3A.h
// Runs on Tiva-C

#ifndef __SWITCH_H__
#define __SWITCH_H__

//Global Variables
volatile uint8_t Desired_RPM;
volatile uint8_t Mailbox_Desired_Flag;
volatile uint8_t Mailbox_ActualLCD_Flag;

void Switch_Init(void);

#endif
```

2. **Explain your interface to the buttons which allow the user to request the DC motor's speed. Which button performs which operations, and how did you check for this in you code?**

- After initializing the GPIO buttons, configuring edge sensitivity for rising edge behavior, arming the buttons for interrupts, and setting their priority, I utilized a handler function to implement the button functionality. Because the buttons were mapped to portD, I implemented button press functionality using the GPIOPortD_Handler. When a button is pressed, the handler is invoked. When the handler is invoked, it starts by delaying for 22 milliseconds. This is essential to steady the button press due to oscillation caused by button debouncing. After the delay, the button's true value should be determined. Next the if statements examine each initialized button to verify if it was the one that produced the interrupt. Each If statement will acknowledge the flag that triggered the interruption. Depending on which button is pressed, different functions will occur. SW2 (PD3) will increase the desired speed by 10 and configure the mailbox flags. SW3 (PD2) reduces the desired speed by ten and sets the mailbox flags. SW5 will change the desired speed to 0 and set the mailbox flags. These mailbox flags are used in user tasks 1 and 2 for knowing when to recalculate the PWM to send to the DC motor and when to output the new desired and calculated RPM to the LCD.

3. **Explain what specific changes you made to the kernal/RTOS; i.e, the os.c and OSasm.asm code. How exactly did you add support for blocking?**

-There were three main changes that were made to the os.c file. First was the implementation of the block identifier for each of the TCBs. I initialized this by creating a uint32_t pointer and set this to zero (NULL) when the threads are added to the OS. The reasons it's a uint32_t pointer is so it can point to the variable that is blocking the thread, which in this case would be a semaphore. The second and third main change to the os.c file was the declaration and definition of the OS_Wait function and the OS_Signal function. The OS_Wait function is a binary blocking semaphore. In this function the semaphore resource is checked if it has already been taken. If it has not been taken the semaphore resource is set to zero and we leave the function. If it has been taken (sig is equal to zero) we set the threads block identifier equal to the semaphore resource (its address) and end the threads time slice period early. Lastly, the OS_Signal function is also a binary semaphore but is used for releasing the semaphore resource so another thread can take it if needed. In this function first the semaphore resource (sig) is set to 1. Next a loop is used to unblock the first thread being blocked by sig. This loop will loop through all other threads checking to see if its block identifier is equal to sig if it is that means it was blocked by this resource so we free it by setting its identifier to zero and break from the loop. If no thread's block identifier is found to equal sig the loop will finish on its own. This outcome is fine, it just means that no thread was blocked while the resource was being used. The changes made to the OSasm.asm works together with what has already been discussed. The changes made to the assembly code involve the creation of a loop toward the end of SysTick_Handler. This loop checks to see if the currently thread that has been switch to has been blocked. It does this by dereferencing the RunPt to get its bi (block indentifier) value. If this value is not zero that means it is blocked, so it should skip this thread by switching to the next thread before this thread will run. So if its blocked it will jump to a label located at the beginning of the SysTick_Handler, which will restart the process of switching the thread.

4. **Explain your use of "signal" and "wait" in the DC motor user task and the safety stop user task. How do these tasks properly share the control of the DC motor?**

-For the DC motor user task I used signal and wait as prompted in our lab report. So right before the pw is calculated the OS_Wait function is called to get access to the semaphore before adjusting the PWM. After the motor speed is changed due to the change in PWM the OS_Signal function is called to release access of the semaphore. As for the safety task when a high percentage of error is detected between Expected_RPM and Actual_RPM the OS_Wait function is called to take the resource. This resource is not given back until the desired RPM is increased from what it was before it was stopped. When this increase happens OS_Signal function is called to give the resource back.

5. **List any sources/websires used or students with whom you discussed this assignment.**
   -N/A