EGEC 451 Lab Report 5

Dominic Jacobo

1. **Os.c:**

```
#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
  uint32_t prio; // current priority
  uint32_t aprio; // assigned priority
};
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;
  tcbs[0].prio = 1; // setting thread priority
```

```
  tcbs[1].prio = 1;
  tcbs[2].prio = 2;
  tcbs[0].aprio = 1;
  tcbs[1].aprio = 1;
  tcbs[2].aprio = 2;
  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;
/* Free all threads because its binary and also just because this function was called doesn't mean another thread got blocked */
    for(i = 0; i < 2; i++){
        if((*counter).bi == sig){
            (*counter).bi = 0;
            break;
        }
        counter = (*counter).next;
    }
    EndCritical(status);
}
```

Osasm.asm:

```
        .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:

```
    ;Switching to next thread code (not a problem)
   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 (I think I need to add this step each time I change threads)
   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


    ; Check if new thread is blocked (not a problem)
    LDR             R1, [R0]              ; R1 = new RunPt
    LDR             R2, [R1,#12]     ; R2 = RunPt->bi
    CMP             R2, #0
    BNE             Blocked                         ;If R2 is not equal to zero that means its blocked


    ;If resource is not blocked reset thread to its original prio (not a problem)
    LDR             R0, RunPtAddr
    LDR             R1, [R0]
    LDR             R2, [R1,#20]     ; R2 = RunPt.aprio
    STR             R2, [R1,#16]     ; RunPt.prio = R2



    ; Prio checking code
    LDR    R1, [R0]                  ; R1 = RunPt (The one just switched to)
```

```
    MOV     R2, #0          ; This will be a counter to use to know when other two threads have been compared with this ones

    LDR     R3, [R1,#16]    ; R3 = RunPt.prio

Label:

    LDR   R0, RunPtAddr

    LDR   R1, [R0]          ; R1 = RunPt (Because we have a new RunPt we need to put this in R1 again)

    LDR     R4, [R1,#8]     ; R4 = RunPt->next

    LDR     R5, [R4,#16]    ; R5 = R4.prio

    CMP     R3, R5

    BLT     Loop            ; If branch is taken that means higher prio is the next thread


    ;STR    SP, [R1]        ; Save SP into TCB (Newly added)

    ;STR    R4, [R0]        ; RunPt = RunPt->next (aka RunPt is switched to the next thread)

    ;LDR    SP, [R4]        ; 7) new thread SP; SP = RunPt->sp; (New added)


    ; Switch to the next thread

    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 (I think I need to add this step each time I change threads)

   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


    ADD     R2, #1          ; R2 = R2 + 1
```

```
CMP         R2, #2
BNE         Label                  ; Branch means we continue looping to compare to the next thread


;LDR        R1, [R0]               ; R1 = RunPt(Because we have a new RunPt we need to put this in R1 again) We have exited the loop
;STR        SP, [R1]               ; Save SP inot TCB (New)
;LDR        R4, [R1,#8]            ; R4 = RunPt->next
;STR   R4, [R0]                    ; RunPt = RunPT->next (We have come full circle)
;LDR   SP, [R4]


; Switch to the next thread
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 (I think I need to add this step each time I change threads)
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


; Assigning global thread id code
LDR       R0, RunPtAddr             ; R0 won't have the right address so you have to do reload it I think
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)
```

```
    ; Incrementing time slice counter code
   LDR                R0, Time_Slice_CounterAddr ; update the Time_Slice_Counter
    LDR    R1, [R0]
    ADD    R1, #1
    STR    R1, [R0]
    B                 END

Blocked: ; Sets threads prio to 0 because it is blocked
    LDR    R0, RunPtAddr
    LDR               R1, [R0]
    MOV               R2, #0                    ; R2 = 0
    STR               R2, [R1,#16]      ; RunPt.prio = R2
    B                 Loop

END:
   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"
```

```c
#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) + 100; // 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_Differ
ence)*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_Differ
ence)*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_Differ
ence);
    Expected_RPM = (pw-100)/28;
    Diff = Expected_RPM - Actual_RPM;
    Percent_Error = (Diff/Expected_RPM)*100;


    OS_Wait(&mutex);
    if ((Diff > 0) && (Expected_RPM > 0)){
        if((Percent_Error > 10) && (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();

 mutex = 1;

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

}

2. **Explain what specific changes you made to the kernel/RTOS; i.e., the os.c and OSasm.am code to support task priorities.**

-In order to implement support for task priorities in our RTOS I start by adding a priority value field to the TCB in os.c. This was pretty simple and just involved adding an uint32_t variable. Next, I assigned each thread its priority, so I gave tasks 1 and 2 a priority of 1 and task 3 a priority of 2. The priority assigning was done in the OS_AddThreads function. After this I needed to change the SysTick_Handler in the assembly code. My thought process was that after a thread switched the thread would compare its priority with the other two threads after it. If its own priority was found to be less than another's it would quickly switch to that thread with the high priority. If the thread's priority was found to be greater than or equal to the two other threads it would continue to run itself.

I implemented this priority checking after the thread had switched to the new thread and was checked for blocking. The priority checking code works by having a large loop that will loop at most two times. The loop begins by storing the RunPt->next value into a register which is than used to get the value of the next threads prio. The prio of the current running thread is compared with the prio of the next thread. If the prio of current thread is less than the prio of next thread the program should jump to the top of the thread switcher because the next thread has higher priority, so switch to that thread. If this does not happen that means the prio of the next thread is either the same or less than the current thread. Next, increment the running thread and loop counter to preform the loop again. If we leave this loop we will increment the running thread one more time, doing this will start us back with the original thread, so priority checking is done and this thread may continue.

After implementing this to the program it experienced deadlock because task 3 always had the highest priority so it would always run leading to the other threads being starved out. To fix this I first implemented a new mutex that would be use in the Timer2A_Handler function and in Task 3. In the Time2A_Handler function it will call the OS_Signal() function and right before task 3 checks the speed of the motor it will call the OS_Wait() function. Doing this will cause task 3 to be blocked when there is no change in the motor, so task 3 being blocked will let the other two tasks run which will fix our deadlock problem.

After implementing this the code would get stuck in the thread switcher because the thread switcher would keep trying to switch to the highest priority thread (task 3) but would find it blocked and skip it. This lead to an infinite loop inside the thread switcher. I fixed this by adding an assigned and current priority value field in the TCB. So when a thread is blocked its priority is set to zero. When the thread is unblocked its priority is reset by to its original assigned priority.

3. **Explain your plan for demonstrating priority inversion. Provide detail when describing the sequence of events that must happen in order for priority inversion to occur and how you will demonstrate this with your system.**
   -I would implement priority inversion by changing my OS_Wait() and OS_Signal() functions. In OS_Wait() I would preform a check to see if the mutex has been taken and if it was by a lower priority task. If it was I would use the global RunPt to see which thread it was and temporarily raise that lower priority thread to the same priority as the higher prio thread that wants this resource. After this lower prio thread gives up the resource in OS_Signal() its priority will be reset to its original priority. This could be demonstrated in my system by putting in break points after the lower priority thread takes the resource and watching the priority changes of the lower priority thread when the higher priority thread tries to take the resource.