```c
/*****************************************************************************//**
*Lab 09: Semaphore Implementation
*****************************************************************************/
#include "sl_component_catalog.h"
#include "sl_system_init.h"
#include "app.h"
#if defined(SL_CATALOG_POWER_MANAGER_PRESENT)
#include "sl_power_manager.h"
#endif
#if defined(SL_CATALOG_KERNEL_PRESENT)
#include "sl_system_kernel.h"
#else // SL_CATALOG_KERNEL_PRESENT
#include "sl_system_process_action.h"
#endif // SL_CATALOG_KERNEL_PRESENT
#include "em_device.h"
#include "em_chip.h"

#include "segmentlcd.h"
#include "em_gpio.h"

/*****************************************************************************//**
 * Extern Includes for Lab04
 *****************************************************************************/
extern void task_A(), task_B(), task_C(), task_D();

extern void Yield();
extern void SysTick_Handler(void);


#define NUM_TASKS 5 // number of real-time tasks plus one

//create a struct for the semaphores
typedef struct
{
  int32_t id;
  int32_t count;
} semaphore;

typedef struct
{
  uint32_t *stack_pointer;
  uint32_t timer3_on;
  uint32_t task_mask;
  uint32_t ready_time;  // not used yet but will be later
  int32_t priority; // not used yet but may be later
  semaphore *smphr; //address of the semaphore
} TaskControlBlock;
TaskControlBlock TCB[NUM_TASKS];

//create the semaphores and initialize them to NULL like in lab08
semaphore semA;
semaphore semB;
semaphore semC;
semaphore semD;

semaphore semLCD; // semaphore for LCD display
semaphore semCD; // semaphore for CD
```



```c

//use the same tick_count and prog variables from lab08
int tick_count = 0;
int prog = 0;
//used to see what the timer3 is run at
int time_measured = 0;
int max_time = 0;

//use the same s and n variables from lab08
char s[8]; // string for LCD display
int n; // number for LCD display

volatile TaskControlBlock *CurrentTask = TCB;
const volatile uint32_t SystemTick = 0; //SystemTick is found in context.s

// stack space for each task
uint32_t stack1[100];
uint32_t stack2[100];
uint32_t stack3[100];
uint32_t stack4[100];

void timer3_start()
{
  TIMER3->CNT = 0; // reset timer to 0
  CurrentTask -> timer3_on = 1; // written to CMD when scheduled
  TIMER3->CMD = 1; // start the timer
}

int timer3_stop()
{
  TIMER3->CMD = 2; // stop timer
  CurrentTask -> timer3_on = 0;
  time_measured = TIMER3->CNT;
  return time_measured;
}

void update_max_time(int time_measured)
{
  if(time_measured > max_time)
  {
    max_time = time_measured;
  }
}

void SemaphoreCreate(semaphore *sem, int32_t id, int32_t count)
{
  sem->id = id;
  sem->count = count;
}

void give_semaphore(semaphore *sem)
{
  __disable_irq();
  if(sem->count > 0)
  {
    sem->count ++;
  }
```



```c
  else
  {
    sem->count = 1;
    int highest = 0;
    int index = 0;
    for(int i = 1; i < NUM_TASKS; i++)
    {
      if(TCB[i].smphr == sem && TCB[i].priority > highest)
        {
          highest = TCB[i].priority;
          index = i;
        }
    }
    if(index == 0)// There are three (3) different cases that could happen
    {
        __enable_irq();
        return;
    }
    else if(TCB[index].priority > CurrentTask->priority)
    {
        TCB[index].smphr = 0;
        __enable_irq(); //make sure this is before Yield();
        Yield();
        return;
    }
    else
    {
        TCB[index].smphr = 0;
        __enable_irq();
        return;
    }
  }
}

void take_semaphore(semaphore *sem)
{
  __disable_irq();
  if(sem->count > 0){sem->count--;}
  else
  {
    CurrentTask->smphr = sem;
    __enable_irq();
    Yield();
    //__disable_irq();
    sem->count--;
  }
  __enable_irq();
}

// create a new task, set up the stack frame and mark it ready-to-go
void CreateTask(int task, void (*funct)(), void *stack, uint32_t stack_words, uint32_t
  priority, uint32_t ready_time)
{
  uint32_t *ptr = (uint32_t *)stack + (stack_words-1); // last byte of stack
  *ptr-- = 0x01000000; // xPSR, Thumb state only
  *ptr-- = (uint32_t)funct;
  for (int i=0; i<6; ++i) *ptr-- = 0; // lr, r12, r3, r2, r1, r0
```



```c
  *ptr = -7; // exception link register
  for (int i=0; i<8; ++i) *--ptr = 0; // r11, r10, r9, r8, r7, r6, r5, r4
  TCB[task].stack_pointer = ptr;
  TCB[task].ready_time = ready_time;
  TCB[task].priority = priority;
  TCB[task].task_mask = 1<<(task-1); //shift
}

void vWaitUntil(int i)
{
  CurrentTask->ready_time = i;
  Yield();
}

void TaskA(void *params)
{
  int release_time = 10; // release all tasks at t = 10
  (void) params; // suppress warning
  for(;;)
  {
    vWaitUntil(release_time);
    task_A();
    release_time += 1; //period of task A
  }
}

void TaskB(void *params)
{
  int release_time = 10; // release all tasks at t = 10
  (void) params; // suppress warning
  for(;;)
  {
    vWaitUntil(release_time);
    task_B();
    release_time += 25; //period of task B
    take_semaphore(&semLCD); // wait for LCD semaphore
    SegmentLCD_ARing(prog,0); // turn off previous segment
    prog = (prog +1) & 7;
    SegmentLCD_ARing(prog,1); // turn on next segment
    give_semaphore(&semLCD); // give LCD semaphore
  }
}

void TaskC(void *params)
{
  int release_time = 10; // release all tasks at t = 10
  (void) params; // suppress warning
  for(;;)
  {
    vWaitUntil(release_time);
    take_semaphore(&semCD);
    task_C(s);
    release_time += 7; //period of task C
    give_semaphore(&semCD);
    take_semaphore(&semLCD);
    SegmentLCD_Write(s);
    give_semaphore(&semLCD);
```

```
  }
}

void TaskD(void *params)
{
  int release_time = 10; // release all tasks at t = 10
  (void) params; // suppress warning
  for(;;)
  {
    vWaitUntil(release_time);
    timer3_start();
    take_semaphore(&semCD);
    task_D(&n);
    release_time += 10; //period of task D
    give_semaphore(&semCD);
    take_semaphore(&semLCD);
    SegmentLCD_Number(n);
    give_semaphore(&semLCD);
  }
}

TaskControlBlock* scheduler()
{
  int highest_priority = 0;
  int highest_priority_task = 0;
  for(int i=1; i<NUM_TASKS; i++)
  {
    if(TCB[i].priority < 1){continue;}
    if(SystemTick >= TCB[i].ready_time && TCB[i].smphr == 0)
    {
      if(TCB[i].priority > highest_priority)
      {
        highest_priority = TCB[i].priority;
        highest_priority_task = i;
      }
    }
  }
  return TCB+highest_priority_task;
}

int idle_count = 0; // used to count idle time

int main(void)
{
  SystemCoreClock = 14000000; // 14 MHz for this device
  // Vendor function to work around bugs in some versions of the hardware
  CHIP_Init();
  // enable clock to GPIO & TIMER3
  CMU->HFPERCLKEN0 |= 0x2100;
  // Divide by 8, 1.75 MHz count rate
  TIMER3->CTRL = 0x03000000;

  // set pins 0-5 to mode 4 (output)
  GPIO->P[gpioPortD].MODEL = (GPIO->P[gpioPortD].MODEL & ~0x0FFFFFF) | 0x00444444;

  // Initialize the LCD
  SegmentLCD_Init(false);
```



```
  // Write to the display
  SegmentLCD_Number(1234);
  SegmentLCD_Write("HELLO");

  // create the semaphores

  SemaphoreCreate(&semLCD, 1, 1); //semaphore &semLCD, id 1, count 1
  SemaphoreCreate(&semCD, 2, 1); //semaphore &semCD, id 2, count 1

  // configure 5ms timer tick
  if (SysTick_Config(5*SystemCoreClock / 1000)) while (1) ;

  // create the real-time tasks
  CreateTask(1,TaskA,stack1,100,4,0);
  CreateTask(2,TaskB,stack2,100,3,0);
  CreateTask(3,TaskC,stack3,100,2,0);
  CreateTask(4,TaskD,stack4,100,1,0);

  /* Infinite loop for aperiodic and sporadic tasks */
  while (1)
  {
  idle_count++;
  }
}
```