EEL 4930 – Spring 2020

Lab 4: Thread Priority, Dynamic Thread Creation and Destruction, Aperiodic Events, and Interfacing with an LCD

OBJECTIVES

- Write a library with extensive functions that allow to interface with a touchscreen color LCD
- Incorporate aperiodic event threads in our RTOS
- Convert our round-robin scheduler into a priority scheduler

REQUIRED MATERIALS

Hardware

- MSP432 Launchpad
- Sensors Booster Pack

Software

- BSP
- Lab 3 G8RTOS

Part A: Interfacing with a touchscreen color LCD

Please look over the ILI9325C and XPT2046 datasheets before continuing reading this lab document.

On Canvas, you will find a header file and source file for the LCD driver library you will write. Most functions you will write, however some are provided to you for the sake of time. The header file contains defines for various registers used to configure the LCD. Additionally, the functions you are required to write have been commented on a high level for you.

Provided Functions:

LCD_Init(): Initializes the LCD

The provided LCD initialization function does most of the work for you, however you will need to initialize the SPI peripheral yourselves (use no pre-scaler for maximum performance) and enable the interrupt on P4.0 for the touchscreen if the user indicates to do so. Before configuring any registers, the LCD needs to be reset, so a simple low-true reset function is included in the source file and has been implemented for you.

PutChar(): Outputs a character to the display at some coordinate. This utilizes the ASCII library provided to you.

LCD_Text(): Outputs a string to the LCD

LCD_WriteIndex(): Sets the address for the register we want to write to

LCD_WriteData(): Writes 16-bit data to the register specified by LCD_WriteIndex()

LCD_ReadData(): Reads 16-bit data from the register specified by LCD_WriteIndex()

LCD_Write_Data_Start(): Sends out the starting condition for continuous data transmission. This will be useful for reasons mentioned in the LCD_DrawRectangle() function description.

Functions You Will Write:

LCD_DrawRectangle(): Draws rectangle with specified color

You'll notice that in the GUI_Text() function, the screen area is set to span the entire area again. This is because there is an optimization method we can exploit for drawing rectangles that will prove very useful to us in our applications. Page 34 in the datasheet shows the timing diagram for continuous data transmission, which allows us to write a constant stream of data without needing to raise and lower the CS after every transmission. Page 63 of the datasheet shows the automatic incrementing of the destination address when the entry mode register is set accordingly. Look over the LCD_Init function to see how this is configured for the orientation on our board.

So, whenever we want draw a rectangle, we can change the span of the screen size, and send out continuous data to the screen with the specified color, and the destination will increment automatically for us, saving a huge amount of time!

LCD_Clear(): Clears the screen with the specified color

LCD_SetPoint(): Draws one pixel with specified coordinate and color

LCD_Write_Data_Only(): Sends out only data (useful for continuous data transmission)

LCD_ReadReg(): Reads data from specified register

SPISendRecvByte(): Sends and receives a byte of data over SPI

LCD_WriteReg(): Writes data to specified register

LCD_SetCursor(): Places cursor at specified coordinate

TP_ReadXY(): Reads touched X and Y coordinates from LCD

Refer to pg. 23 for the timing diagram of the A-D conversion of the touched X and Y coordinates, as well as pg. 16 for finding out how to grab the X and Y coordinates in the XPT2046 datasheet. The defines for the two channels are in the LCD header file.

You must keep in mind that when touching the LCD, the screen will bounce! Therefore, you will need to implement a debounce technique once incorporated in the OS.

Tip 1: If you convert the A-D conversion to a value between 0 and 1, and multiply by the max screen size for either X or Y, you will get the position of the touched point.

Tip 2: Carefully look at the timing diagram to understand how to shift the bits to attain the correct A-D result.

Part B: Convert Round-Robin Scheduler to Priority Scheduler

Instead of using a round-robin scheduler, we will introduce priority to our RTOS, where every thread has its own priority. For example, this is one way to guarantee we are writing to the LCD at 30fps, for example. Or we can now make our idle thread the lowest priority, where it is only run when all threads are unable to run (either blocked or sleeping).

A uint8_t priority field will be added to the TCB struct, which is set when the user calls G8RTOS_AddThread(). The lower the priority number, the higher the thread priority (we are using the same convention as the ARM Cortex M priorities). Since the priority will be a uint8_t, the lowest priority possible is 255, and the highest will be 0.

Now we need modify our scheduler to no longer be strictly round-robin. The implementation will be as follows:

- Set tempNextThread to be the next thread in the linked list (allows for round-robin scheduling of equal priorities) and iterate through all the other threads
- If tempNextThread is neither sleeping or blocked, we check if its priority value is less than a currentMaxPriority value (initial currentMaxPriority value will be 256)
- If it is, we set the CurrentlyRunningThread equal to the thread with the higher priority, and reinitialize the currentMaxPriority
- Set the next thread to check equal to the next thread in the linked list to keep checking for the rest of the threads (potentially of higher priority).

Lastly, modify G8RTOS_Launch to choose the thread with the highest priority to run first.

Part C: Dynamic Thread Creation and Destruction

Currently, we cannot add or kill threads once our OS has been launched. In order to accomplish this task, we will need to modify the TCB struct and the G8RTOS_AddThread function, add a function to attain a thread's ID, and add two new functions to the scheduler: **G8RTOS_KillSelf()** and **G8RTOS_KillThread()**.

Begin by adding the following to the TCB struct:

- bool isAlive
- threadId_t threadID
  - typedef threadId_t as a uint32_t inside of scheduler header
- char threadName[MAX_NAME_LENGTH]
  - define max to 16

The thread name will be convenient for us to keep track of threads inside of the variable explorer (not relevant to this part, but we'll add it to the TCB while we're already modifying it).

The isAlive bit will indicate whether that thread is alive, or if it's been killed and no longer in the linked list of active threads.

The thread ID will allow every thread to have its unique ID so the user can request the ID of the thread to kill and kill it while in another thread.

Modifications to G8RTOS_AddThread():

The AddThread function will now take in not only a thread's priority, but also its name to initialize. Since we want to be able to add a thread while our OS is running, we will need to enter a critical section and exit it prior to returning.

Lab 4: Thread Priority, Dynamic Thread Creation and Destruction, Aperiodic Events, and Interfacing with an LCD

To create a unique ID, we while use a static uint16_t IDCounter, which will increment every time a TCB is initialized. The TCB's threadID will be:

((IDCounter++) << 16) | tcbToInitialize

where the tcbToInitialize is the first TCB not alive. If there are none that are dead, you should return THREADS_INCORRECTLY_ALIVE.

The last things to do is to initialize the char array for the thread's name and set the alive bit to true.

threadId_t G8RTOS_GetThreadId()

Create this function that returns the CurrentlyRunningThread's thread ID.

Thread Destruction Functions:

sched_ErrCode_t G8RTOS_KillThread(threadId_t threadId):

The KillThread function will take in a threadId, indicating the thread to kill. The implementation will be as follows:

- Enter a critical section
- Return appropriate error code if there's only one thread running
- Search for thread with the same threadId
- Return error code if the thread does not exist
- Set the threads isAlive bit to false
- Update thread pointers
- If thread being killed is the currently running thread, we need to context switch once critical section is ended
- Decrement number of threads
- End critical section

sched_ErrCode_t G8RTOS_KillSelf():

The KillSelf function will simply kill the currently running thread. The implementation will be as follows:

- Enter a critical section
- If only 1 thread running, return appropriate error code
- Change isAlive bit to false
- Update thread pointers
- Start context switch
- Decrement number of threads
- End critical section

Part D: Aperiodic Event Threads

Since an aperiodic event thread will essentially be an interrupt routine, we will need to initialize the appropriate NVIC registers accordingly. Too add an aperiodic event, we provide it with a function pointer that will serve as the ISR, a priority, and the IRQ interrupt number.

sched_ErrCode_t G8RTOS_AddAPeriodicEvent(void (*AthreadToAdd)(void), uint8_t priority, IRQn_Type IRQn):

The implementation will be as follows:

- Verify the IRQn is less than the last exception (PSS_IRQn) and greater than last acceptable user IRQn (PORT6_IRQn), or else return appropriate error
- Verify priority is not greater than 6, the greatest user priority number, or else return appropriate error
- Use the following core_cm4 library functions to initialize the NVIC registers:
    - __NVIC_SetVector
    - __NVIC_SetPriority
    - NVIC_EnableIRQ

Note: To relocate an ISRs interrupt vector, the interrupt vector table must be relocated to SRAM. Depending on the compiler, this may or may not be done automatically. Therefore, to be compliant with all compilers, we want to relocate the interrupts vector to SRAM ourselves. We will do this in G8RTOS_Init(). The following code with relocate the vector table to 0x20000000:

```
uint32_t newVTORTable = 0x20000000;
memcpy((uint32_t *)newVTORTable, (uint32_t *)SCB->VTOR, 57*4);  // 57 interrupt vectors to copy
SCB->VTOR = newVTORTable;
```

Also note: Some of you might have an outdated version of core_cm4.h, where the vector functions should be. We have included the most up-to-date header file in the repo inside of libraries. Simply copy and paste the code in the existing file in CCS.

Updated Error Codes for Scheduler:

```
/*
 * Error Codes for Scheduler
 */
typedef enum
```

Lab 4: Thread Priority, Dynamic Thread Creation and Destruction, Aperiodic Events, and Interfacing with an LCD

```
{
        NO_ERROR                    =  0,
        THREAD_LIMIT_REACHED        = -1,
        NO_THREADS_SCHEDULED        = -2,
        THREADS_INCORRECTLY_ALIVE   = -3,
        THREAD_DOES_NOT_EXIST       = -4,
        CANNOT_KILL_LAST_THREAD     = -5,
        IRQn_INVALID                = -6,
        HWI_PRIORITY_INVALID        = -7
} sched_ErrCode_t;
```

Periodic, Aperiodic Events, and Init Sempahores for dynamic creation and destruction:

In the functions used to add periodic, aperiodic threads, and initSempahore, you will also want to make these critical sections now so that they may be called once the OS is running. You are not required to make a kill periodic and kill aperiodic function, although you can if you want to!

Application:

The goal of this part of the lab is to demonstrate all the new features in G8RTOS, as well as the LCD driver you wrote in an exciting way, as well as get you started on writing some helpful functions that you'll use in lab 5.

General description of expected final product:

Program will launch with nothing on the screen, waiting for a touch on the screen. Once touched, a ball (4x4 rectangle in our case) should be drawn on the screen with a random color (you may use the time.h library for randomness). Depending on the accelerometer x and y values, the ball will move accordingly. To make it more interesting, every new ball created should have a random speed (just a scaling factor for its velocity). If one of the balls is touched, you should delete the ball. There will be a max number of 20 balls allowed at one time. If a ball hits an edge, it should wrap around to the other side.

Initially, you will have the following threads active:

Read Accelerometer: (Background)
- Reads accelerometer x and y values
- Sleep for some amount of time

LCD tap: (Aperiodic)
- ISR for tap on the screen
- Should just set a flag saying a touch occurred

Wait for Tap: (Background)
- Waits for flag from ISR
- Reads touch coordinates
- Determines whether to delete or add ball
- If a ball is to be created:
    - Write the coordinates to a FIFO
    - Create a Ball thread
- If a ball is to be deleted, wait for any semaphores the ball thread might be using and call G8RTOS_KillThread with the ball's thread ID
- Delay for ~500ms to account for screen bouncing before checking touch flag again

Idle (Low priority idle thread)
- Now only runs when all other threads are either blocked or sleeping

Ball Thread: (Background)
- Finds a dead ball and makes it alive
- Read FIFO and initialize coordinates accordingly
- Get thread ID and store it (I recommend using a struct)
- Within while(1):
    - Move position depending on velocity and acceleration
    - Update ball on screen and sleep for ~30ms

Alternative method for LCD bouncing:

In the aperiodic event, you can add a new thread, then disable interrupts on the port's pin. This new thread will determine whether a ball is to be created or destroyed, take the appropriate action like mentioned earlier, and then sleep for some amount of time (enough until the bouncing has stopped), then re-enable the interrupt on the port's pin, clear the flag, and then kill itself.

Note: For the ball struct, a good thing to do would be to store the following attributes:
- x position
- y position
- x velocity
- y velocity
- alive
- thread ID
- color