EEL 4745C: Microprocessor Applications 2
Lab #4: G8RTOS IPC, Dynamic Threads, Priority Scheduling
Fall 2024

OBJECTIVES
In this lab, you will improve some existing features and implement new capabilities to G8RTOS that you implemented in lab-3. This includes communication between threads, thread yielding, thread priority, aperiodic and periodic threads, dynamic thread creation, as well as priority scheduling.

REQUIRED MATERIALS
Hardware
- Tiva TM4C123GH6PM Launchpad
- Laptop workstation with CCS setup
- Sensors Booster Pack
- Multimod Board

Software
- TivaWare SDK
- Code Composer Studio (CCS) >= 11.1.0
- Skeleton code provided for Lab-4

Documentations
- TM4C123GH6PM manual and datasheet
- Tiva Launchpad schematic
- Multimod Board schematic
- PCA9555 datasheet
- ST7789 datasheet
- Feel free to use books / browse the internet!

IN-LAB REQUIREMENTS

This spans two weeks. The Part-AB demo is due on your first week, Part CDE is due on the final week. We will be slightly flexible with grading in the first week, as long as you are making considerable progress.

Your Quiz-1 will be right after this lab, so these concepts are super important to grasp.

Part A: Implementing Blocking, Yielding, Sleeping, Priority. [25 points]

MOTIVATION
Previously, threads used simple software delays because thread *sleeping* had not been implemented. Without these software delays, there was the potential issue where the thread would immediately repossess the resource it released, causing that thread to run forever and *starve* other threads.

When a thread has completed its task, it should sleep for a certain period of time, so it will yield the CPU to another thread, so that minimal processing time is wasted.

When a thread becomes *blocked*, it means it is currently unable to run (usually due to a requested resource being in use), so it should yield so another thread can run instead of *spinlocking* as in lab-3.

Additionally, assigning priorities to threads can be useful. Audio threads are one such example where certain threads have higher priority, where it is important that an audio thread is able to complete its task as soon as possible.

To this end, you will be making modifications to many G8RTOS functions you have written in lab-3.

PREPARATION
- Review lecture material

Modify the TCB
File: `G8RTOS/G8RTOS_Structures.h`

Within the "`G8RTOS_Structures.h`" file, update the following:
- **struct** tcb_t

Add variables that hold the following information:
- Which semaphore the thread is blocked on
  - This will be 0 when the thread is not blocked.
- The priority of the thread
  - Priority is a range from [0, 255], with 0 being the highest priority.
- If the thread is alive
- If the thread is asleep
- What time the thread should wake up
- The thread name
  - This is a character array.
- The thread ID

Note that these do not have to be updated at this part, but the full structure will have these variables included. The sections below will specify what the TCB should be updated with.

Update your Semaphore Functions
File: `G8RTOS/src/G8RTOS_Semaphores.c`, `G8RTOS/src/G8RTOS_Scheduler.c`

Within the "`G8RTOS_Semaphores.c`" file, update the following functions:
- **G8RTOS_WaitSemaphore()**
- **G8RTOS_SignalSemaphore()**

Within the "`G8RTOS_Scheduler.c`" file, update the following functions:
- **G8RTOS_Scheduler()**

When a thread requests a resource and it is not available, it is now considered “blocked” on that resource. Since it can no longer complete its task, it should immediately yield so that another thread can run, and not continue until that resource is released.

When a thread releases a resource, it should search for the next thread blocked on the resource and unblock it.

To do this:

- Modify the TCB structure to include a pointer to the semaphore it is waiting on. This will equal the semaphore’s address if it is blocked, or **nullptr** if it is not blocked.
- Modify **G8RTOS_WaitSemaphore()**, so that you are no longer *polling* the semaphore. Instead, if the resource is not available, the semaphore pointer of the TCB should now point to the current semaphore and then **yield** control to allow another thread to run.
- Modify **G8RTOS_SignalSemaphore()** to now include a process that goes through the TCB linked list and unblock the first thread that is blocked on that semaphore.
- Modify **G8RTOS_Scheduler()** so that it does not select a thread that is blocked.

Ensure a deadlock does not occur, which is when thread A is waiting on a semaphore to be released by thread B, but thread B is waiting on a semaphore to be released by thread A.

Implement Sleeping
File: `G8RTOS/src/G8RTOS_Scheduler.c`

Before, when a repeatable task completes you implemented a simple software delay to wait to run the task again. You could also use timers for this purpose, however, there needs to be a way to synchronize this with CPU time so that CPU time can be better distributed.

For this problem, you will implement **sleeping**. This causes a thread to wait for a specified amount of time while allowing other threads to run. Thread sleeping is used with background threads when the accuracy of time is not as important as CPU usage.

Within “`G8RTOS_Scheduler.c`” file, implement or update the following two function:

- Implement: **G8RTOS_Sleep()**
- Update: **G8RTOS_Scheduler()**
- Update: **SysTick_Handler()**

You will also further update your TCB to keep track of **sleep duration** and **sleep status**.

Once a thread is asleep, it should immediately context switch to the next available thread. Furthermore, your scheduler should now check for if a thread is asleep.

Note: What would happen if all threads were asleep while the scheduler is running?

Implement Priority
In this section you will be converting your scheduler into a priority scheduler. For a thread to run, it must be active and the **highest priority** among other active threads.

Within the “`G8RTOS_Scheduler.c`” file, update the following functions:

- **G8RTOS_Scheduler()**
- **G8RTOS_AddThread()**

First, update the TCB with priority. This should be an **uint8_t** bit value, with lesser numbers having higher priority.

The scheduler will now traverse the linked list in order to find the thread with the **highest priority** that is **not blocked** and **not sleeping**.

G8RTOS_AddThread() will now include as a parameter an 8-bit priority value, to set the corresponding added thread’s priority.

Part B: Implementing Interprocess Communication [20 points]

Implement FIFO Initialization, Write, Read
File: `G8RTOS/src/G8RTOS_IPC.c`, `G8RTOS/G8RTOS_IPC.h`

A FIFO (First In, First Out) data structure can be used as a buffer for asynchronous communication between threads.

Within the “`G8RTOS_IPC.c`” file, implement the following functions:

- **G8RTOS_InitFIFO();**
- **G8RTOS_ReadFIFO();**
- **G8RTOS_WriteFIFO();**

Within the “`G8RTOS_IPC.h`” file, implement the following struct:

- **G8RTOS_FIFO_t**

The **G8RTOS_FIFO_t** structure contains:

- Buffer array of uint32_t with a predefined size
- pointer to the head of the buffer
- pointer to the tail of the buffer
- lost data counter
- semaphore: current size
- semaphore: mutex

G8RTOS_InitFIFO() will take in a FIFO_index as a variable, which selects the FIFO from an array of FIFOs defined by the G8RTOS. You should initialize the buffer pointers as well as the semaphores.

G8RTOS_ReadFIFO() will take as a parameter a FIFO_index. Before reading from the FIFO, it must check if the FIFO is already being read by another thread. Then it must check to see if there is any data. If it is allowed to read, then increment the head pointer and signal the semaphores. Be mindful of boundaries.

G8RTOS_WriteFIFO() will take as a parameter a FIFO_index and the data to be written. If it is full (current size > fifo_size - 1) or if the index is invalid, it should return an error. Otherwise, data should be written to the FIFO, the tail pointer should be updated, the current size semaphore should be signaled, and no error is returned. **Stale data should be overwritten if the FIFO is full.**

IN-LAB REQUIREMENTS
Complete the thread sleeping, yielding, and blocking. Implement the FIFOs. Demonstrate that these are working using simple test threads and the debugger. Show that if the thread gets blocked or is asleep, the next thread available immediately begins running. Show that once the semaphore is signaled that threads that are blocked on it are able to run again. Show that data can be published to a FIFO by one thread and read by another.

Part C: Dynamic threading, Aperiodic and Periodic Events [25 points]

Implementing Dynamic Threading
File: `G8RTOS/src/G8RTOS_Scheduler.c`

Currently, once the RTOS is running there is no way to add or delete threads. To rectify this, we will be modifying **G8RTOS_AddThread** again, and adding two new ones.

Within the "`G8RTOS_Scheduler.c`" file, modify the following function:

- **G8RTOS_AddThread()**

And implement the following functions:

- **G8RTOS_KillThread()**
- **G8RTOS_KillSelf()**

To begin, modify the TCB struct to include:

- threadID
- Thread Name
- Alive status

G8RTOS_AddThread() should be updated to now take in a character string as a name, which will be used to set the TCB name. The function will now also be a critical section. It will be necessary to enter a critical section when the function is called and exit it while returning. **Keep in mind that you will need to account for cases where there are dead threads between alive threads!**

G8RTOS_KillThread() will take in a threadID, indicating the thread to kill. Consider boundary conditions, such as if no thread with that ID exists, if there is only one thread running, if the thread removal should be in a critical section. **You will need to update the next and previous thread's TCB pointers.**

G8RTOS_KillSelf(), similar to KillThread, causes the thread to cease running and frees up its space in the TCB linked list.

Implementing Aperiodic Events
File: `G8RTOS/src/G8RTOS_Scheduler.c`

PREPARATION

- Review the interrupt vector table in the tm4c123gh6pm datasheet.

Within "`G8RTOS_Scheduler.c`" file, implement the following function:

- **G8RTOS_Add_APeriodicEvent()**

This is performed similarly to the timer interrupts you did in a previous lab. Each interrupt is associated with a specific interrupt request number, associated with a specific interrupt condition according to the interrupt vector table. Therefore, when utilizing an aperiodic event, be sure to also enable the corresponding interrupt in the relevant registers.

We must also verify that the *IRQn* is less than the last exception (155). It must also be greater than the last acceptable user interrupt (0) and verify that the priority is not greater than 6.

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, you want to relocate the interrupts vector to SRAM yourselves. This is done for you in G8RTOS_Init().

Implementing Periodic Events
File: `G8RTOS/src/G8RTOS_Scheduler.c`

Within "`G8RTOS_Scheduler.c`" file, implement:

- **G8RTOS_Add_PeriodicEvent()**

And update the following function:

- **SysTick_Handler()**

Periodic threads will be created with three parameters: the pointer to the function to be run, the execution time, and the period. Background threads with sleeping are useful when CPU usage is more important. When timing accuracy is more important, periodic events should be used.

Periodic threads also have their own doubly-linked container; see `G8RTOS_Structures.h`.

Periodic threads are similar to functions synced to a hardware timer, in that they occur at specific time intervals. However, periodic threads are synced to system time. The **execution time** is **when the function will first run** after the G8RTOS is launched and the **period** is the **amount of time between each subsequent execution**.

Each time **SysTick_Handler** triggers, it will also evaluate and run eligible periodic events, before updating the periodic event's next execution time.

Note: that if two periodic events have the same period, it is possible to stagger them so there are not multiple events occurring in the same SysTick.

IN-LAB REQUIREMENTS
Complete the dynamic thread functions, as well as the periodic and aperiodic events above. As an example, have a thread spawn more threads and have these threads terminate themselves. See if the TCB is overwritten on subsequent add threads. Show that the periodic events occur on specific intervals of SysTick. Show that the aperiodic events trigger.

Part D: Interfacing TFT Display, Joystick [15 points]

File:
`MultimodDrivers/src/multimod_ST7789.c`,
`MultimodDrivers/src/multimod_buttons.c`,
`MultimodDrivers/src/multimod_joystick.c`

PREPARATION
- Review ADC module contents in the tm4c123gh6pm datasheet.
- Review Multimod Board schematics
- Review ST7789 Datasheet.

Within the "`multimod_ST7789.c` file, implement the following functions:
- **ST7789_SetWindow()**
- **ST7789_DrawPixel()**

Within the "`multimod_buttons.c`" file, implement the following functions:
- **MultimodButtons_Init()**
- **MultimodButtons_Get()**

Within "`multimod_joystick.c`" file, implement:
- **JOYSTICK_Init()**
- **JOYSTICK_IntEnable()**
- **JOYSTICK_GetXY()**

The ST7789 uses SPI to communicate. The SPI functions have already been written for you, as it should have been covered in a previous course. Be sure to review the tm4c123gh6pm manual.

Part E: Putting it all together! [15 points]

Completing the program
The final product will be using what you have implemented before. The program will use a button press to determine when to spawn a cube. This cube will be spawned at random locations. A different button will cause a random ball to terminate. The joystick will be used to move the camera around, allowing perspective shifts and the joystick can also be used to toggle between moving up and down or moving further or closer away.

Cube_Thread, which is detailed below, has already been done for you, but you will have to implement your RTOS functions in some areas.

Background Threads
- **Idle_Thread**
  - Does nothing. Lowest priority!
- **CamMove_Thread**
  - Reads JOYSTICK_FIFO
  - Normalize values received to a range of [-1, 1], then use this result to move the camera position. The Joystick Y axis will control the Y or Z coords depending on the state of a toggle.
- **Cube_Thread**
  - Reads SPAWNCOOR_FIFO, spawns cube at that position with L, W, H = 50.
  - Continuously draws cube to & clears cube from display.
- **Read_Buttons**
  - Wait on the PCA9555 semaphore. Sleep for some time (to debounce) then read from the PCA9555 to determine which buttons were pressed.
  - If SW1 on the Multimod board is pressed, spawn a cube thread in random x, y, z positions using the rand() function.
    - x should be [-100,100]
    - y should be [-100,100]
    - z should be [-120,-20]
    - Send to SPAWNCOOR_FIFO.
  - If SW2 on the Multimod board is pressed, terminate a random cube.
- **Read_JoystickPress**
  - Toggles the joystick_flag to determine if the Y-axis moves the Y coordinates or Z coordinates.

Periodic Threads
- **Print_WorldCoords() (Period: 100 ms)**
  - Use UART to print out the x, y, z values of the current camera position in the world space.

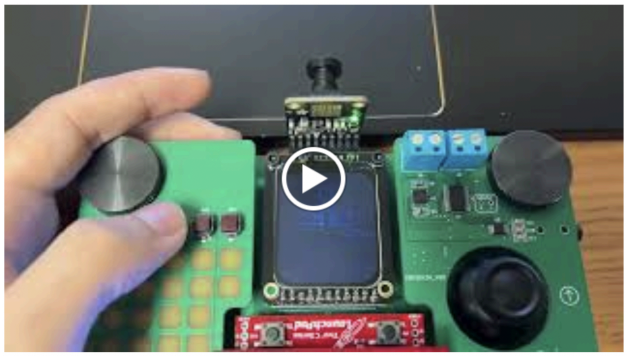
- **Get_Joystick() (Period: 100 ms)**
  - Gets XY values from Joystick, sends it to JOYSTICK_FIFO.

APeriodic Threads

- **GPIOE_Handler**
  - When the relevant pin detects a falling edge from the PCA9555, increment a semaphore to show that there is something to read from it.
- **GPIOD_Handler**
  - When the relevant pin detects a falling edge from the Joystick, increment a semaphore to show that there is something to read.

IN-LAB REQUIREMENTS

Show the program working end-to-end. See the following video demonstration for reference.



Link: ***https://youtu.be/yDRkCWjt62c***

Bonus points:

There is an LED matrix on your Multimod Board, interfaceable by the **PCA9956b**. Use your accelerometer to transmit data to a FIFO so that another thread can read it and then command the LED matrix to light up accordingly to the x-y acceleration measured by the IMU. [up to +5 points].