Embedded Systems

Postlab 10 Threads and Semaphores

By Rajendra Singh (111601017), final year, CSE, IIT PALAKKAD

25 Oct 2019

Objective

To learn how to control the multiple LED in freedom board using threads and semaphores.

Abstract

Thread:

What is a Thread?

- A ***thread*** is a basic unit of CPU utilization, consisting of a program counter, a stack, and a set of registers, (and a thread ID.
- A thread is a path of execution within a process. A process can contain multiple threads.
- Traditional (heavyweight) processes have a single thread of control - There is one program counter, and one sequence of instructions that can be carried out at any given time.

Why Multithreading?

A thread is also known as lightweight process. The idea is to achieve parallelism by dividing a process into multiple threads. For example, in a browser, multiple tabs can be different threads. MS

Word uses multiple threads: one thread to format the text, another thread to process inputs, etc. More advantages of multithreading are discussed below

Process vs Thread?

The primary difference is that threads within the same process run in a shared memory space, while processes run in separate memory spaces.

Threads are not independent of one another like processes are, and as a result threads share with other threads their code section, data section, and OS resources (like open files and signals). But, like process, a thread has its own program counter (PC), register set, and stack space.

Advantages of Thread over Process

1. Responsiveness: If the process is divided into multiple threads, if one thread completes its execution, then its output can be immediately returned.

2. Faster context switch: Context switch time between threads is lower compared to process context switch. Process context switching requires more overhead from the CPU.

3. Effective utilization of multiprocessor system: If we have multiple threads in a single process, then we can schedule multiple threads on multiple processor. This will make process execution faster.

4. Resource sharing: Resources like code, data, and files can be shared among all threads within a process.

Note: stack and registers can't be shared among the threads. Each thread has its own stack and registers.

5. Communication: Communication between multiple threads is easier, as the threads shares common address space. while in process we have to follow some specific communication technique for communication between two process.

6. Enhanced throughput of the system: If a process is divided into multiple threads, and each thread function is considered as one job, then the number of jobs completed per unit of time is increased, thus increasing the throughput of the system.

Types of Threads

There are two types of threads.

User Level Thread

Kernel Level Thread

Semaphore

Semaphore was proposed by Dijkstra in 1965 which is a very significant technique to manage concurrent processes by using a simple integer value, which is known as semaphore. Semaphore is simply a variable which is non-negative and shared between threads. This variable is used to solve the critical section problem and to achieve process synchronization in the multiprocessing environment.

The two most common kinds of semaphores are counting semaphores and binary semaphores. Counting semaphore can take non-negative integer values and Binary semaphore can take the value 0 & 1. only.

Kindly look the code for more information regarding the port and pin selection.

EXERCISE

1: Control the red and blue led based on the thread.

=> In below is detailed code with comments for each step for controlling the red and blue led based on the thread is presented.

CODE:

Below code can also be found at github.

```
#include<MKL25Z4.h>

#include "cmsis_os.h" // Include header file for RTX CMSIS-RTOS

#include "led.h"



// System runs at 48MHz // LED #0, #1 are port B, LED #2 is port D

extern void LED_Config(void);

extern void LED_Set(void);

extern void LED_Clear(void);

extern __INLINE void LED_On(uint32_t led);

extern __INLINE void LED_Off(uint32_t led);



osThreadId t_blinky; /* Thread IDs */ // Declare a thread ID for blinky

// ========================================================================

void blinky(void const *argument); // Thread /* Function declaration */

void blinky(void const *argument) { // Blinky

    while(1) {
```

```c
        LED_On(2); // Green LED on

        osDelay(500); // delay 500 msec

        LED_Off(2); // Green LED off

        osDelay(500); // delay 500 msec

    } // end while

} // end of blinky

// ========================================================================

osThreadDef(blinky, osPriorityNormal, 1, 0); // define blinky as thread function




int main(void)

{

    SystemCoreClockUpdate();

    LED_Config();// Configure LED outputs

    t_blinky = osThreadCreate(osThread(blinky), NULL);// Create a task "blinky" and
assign thread ID to t_blinky

    while(1){

        LED_On(0); // Red LED on

        osDelay(500); // delay 500 msec
```

```
        LED_Off(0); // Red LED off

        osDelay(500); // delay 200 msec

    };

}
```

2: Control the red and blue led based on the thread with osSignalWait.

=> In below is detailed code with comments for each step for controlling the red and blue led based on the thread with osSignalWait is presented.

CODE:

Below code can also be found at github.

```
#include<MKL25Z4.h>

#include "cmsis_os.h" // Include header file for RTX CMSIS-RTOS

#include "led.h"



// System runs at 48MHz // LED #0, #1 are port B, LED #2 is port D

extern void LED_Config(void);

extern void LED_Set(void);

extern void LED_Clear(void);

extern __INLINE void LED_On(uint32_t led);
```

```
extern __INLINE void LED_Off(uint32_t led);


osThreadId t_blinky; /* Thread IDs */ // Declare a thread ID for blinky

// =====================================================================

void blinky(void const *argument); // Thread /* Function declaration */

void blinky(void const *argument) { // Blinky

    while(1) {

        osSignalWait(0x0001, osWaitForever);

        LED_On(2); // Green LED on

        osDelay(500); // delay 500 msec

        LED_Off(2); // Green LED off

        osDelay(500); // delay 500 msec

    } // end while

} // end of blinky

// =====================================================================

osThreadDef(blinky, osPriorityNormal, 1, 0); // define blinky as thread function
```

```
int main(void)

{

    SystemCoreClockUpdate();

    LED_Config();// Configure LED outputs

     t_blinky = osThreadCreate(osThread(blinky), NULL);// Create a task "blinky" and
assign thread ID to t_blinky

    while(1){

        LED_On(0); // Red LED on

        osDelay(500); // delay 500 msec

        LED_Off(0); // Red LED off

        osSignalSet(t_blinky, 0x0001); // Set Signal

        osDelay(500); // delay 200 msec

    };

}
```

3: Control the red and blue led based on the Semaphores.

=> In below is detailed code with comments for each step for controlling the red and blue led based on the Semaphores is presented.

CODE:

Below code can also be found at github.

```
#include<MKL25Z4.h>

#include "cmsis_os.h" // Include header file for RTX CMSIS-RTOS

#include "led.h"



// System runs at 48MHz // LED #0, #1 are port B, LED #2 is port D

extern void LED_Config(void);

extern void LED_Set(void);

extern void LED_Clear(void);

extern __INLINE void LED_On(uint32_t led);

extern __INLINE void LED_Off(uint32_t led);



osThreadId t_blinky_red; /* Thread IDs */ // Declare a thread ID for blinky

osThreadId t_blinky_green; /* Thread IDs */ // Declare a thread ID for blinky

osThreadId t_blinky_blue; /* Thread IDs */ // Declare a thread ID for blinky



void blinky_red(void const *argument); // Thread /* Function declaration */

void blinky_green(void const *argument); // Thread /* Function declaration */

void blinky_blue(void const *argument); // Thread /* Function declaration */
```

```
osSemaphoreDef(two_LEDs);// Declare a Semaphore for LED control

osSemaphoreId two_LEDs_id; // Declare a Semaphore ID for LED control


// ======================================================================

void blinky_red(void const *argument) { // Blinky

    while(1) {

        osSemaphoreWait(two_LEDs_id, osWaitForever);

        LED_On(0); // Green LED on

        osDelay(400); // delay 400 msec

        LED_Off(0); // Green LED off

        osSemaphoreRelease(two_LEDs_id);

        osDelay(600); // delay 600 msec

    } // end while

} // end of blinky


void blinky_green(void const *argument) { // Blinky

    while(1) {

        osSemaphoreWait(two_LEDs_id, osWaitForever);
```

```
        LED_On(1); // Green LED on

        osDelay(400); // delay 400 msec

        LED_Off(1); // Green LED off

        osSemaphoreRelease(two_LEDs_id);

        osDelay(600); // delay 600 msec

    } // end while

} // end of blinky




void blinky_blue(void const *argument) { // Blinky

    while(1) {

        osSemaphoreWait(two_LEDs_id, osWaitForever);

        LED_On(2); // Green LED on

        osDelay(400); // delay 400 msec

        LED_Off(2); // Green LED off

        osSemaphoreRelease(two_LEDs_id);

        osDelay(600); // delay 600 msec

    } // end while

} // end of blinky
```

```
// ========================================================================

osThreadDef(blinky_red, osPriorityNormal, 1, 0); // define blinky as thread
function

osThreadDef(blinky_green, osPriorityNormal, 1, 0); // define blinky as thread
function

osThreadDef(blinky_blue, osPriorityNormal, 1, 0); // define blinky as thread
function



int main(void)

{

    SystemCoreClockUpdate();

    LED_Config();// Configure LED outputs


    two_LEDs_id = osSemaphoreCreate(osSemaphore(two_LEDs), 2);// Create
Semaphore with 2 tokens


    //Here order matter, change the order to view changes

    t_blinky_red = osThreadCreate(osThread(blinky_red), NULL);// Create a task
"blinky" and assign thread ID to t_blinky
```

```
    t_blinky_green = osThreadCreate(osThread(blinky_green), NULL);// Create a
task "blinky" and assign thread ID to t_blinky

    t_blinky_blue = osThreadCreate(osThread(blinky_blue), NULL);// Create a
task "blinky" and assign thread ID to t_blinky



    osThreadTerminate(osThreadGetId());

    while(1){

        osDelay(1000);

    };

}
```

Conclusion

- In this exercise, we learnt more about Threads and Semaphores.
- Learned to control multiple LED at once using Threads and Semaphores.
- Overall it was a nice exercise.

Reference

1) Google
2) [Cortex M0+ Generic User's Guide]
3) [Cortex M0+ Technical Reference Manual]