main.c

```c

#include <stdint.h>
#include <stdbool.h>
#include "main.h"
#include "drivers/pinout.h"
#include "utils/uartstdio.h"


// TivaWare includes
#include "driverlib/sysctl.h"
#include "driverlib/debug.h"
#include "driverlib/rom_map.h"
#include "driverlib/rom.h"
#include "driverlib/timer.h"
#include "driverlib/inc/hw_memmap.h"
#include "driverlib/inc/hw_ints.h"

// FreeRTOS includes
#include "FreeRTOSConfig.h"
#include "FreeRTOS.h"
#include <timers.h>
#include <semphr.h>
#include "task.h"
#include "queue.h"
#include "limits.h"


#define FIB_LIMIT_FOR_32_BIT 47
#define TIME_TO_RUN 240 //ms

SemaphoreHandle_t task1SyncSemaphore, task2SyncSemaphore;
TickType_t startTimeTick;
TaskHandle_t Task1_handle, Task2_handle;
uint32_t counter = 0;
double Hz = 100;
uint32_t ulPeriod;


void fiboncacci(int ms){
    TickType_t xStartTick = xTaskGetTickCount();
    TickType_t xCurrentTick = xTaskGetTickCount();
    uint32_t fib = 1, fib_a = 1, fib_b = 1;
    uint32_t i;
    while((xCurrentTick - xStartTick) < (pdMS_TO_TICKS(ms) -1)){
        for (i = 0; i < FIB_LIMIT_FOR_32_BIT; i++){
            if(((xCurrentTick - xStartTick) >= pdMS_TO_TICKS(ms)-1)) break;
            fib_a = fib_b;
            fib_b = fib;
            fib = fib_a + fib_b;
        }
        xCurrentTick = xTaskGetTickCount();
    }
}
```

```c




void Timer0Isr(void)
{
    TickType_t xCurrentTick = xTaskGetTickCount();
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;
    ROM_TimerIntClear(TIMER0_BASE, TIMER_TIMA_TIMEOUT);  // Clear the timer interrupt
    counter ++;
    if (counter % 3 == 0){
        xTaskNotifyFromISR(Task1_handle, xCurrentTick, eSetValueWithOverwrite, &
xHigherPriorityTaskWoken);
        portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
    }
    else if(counter % 8 == 0){
        xTaskNotifyFromISR(Task2_handle, xCurrentTick, eSetValueWithOverwrite, &
xHigherPriorityTaskWoken);
        portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
        counter = 0;
    }
}



// Process 1
void xTask1(void * pvParameters)
{
    TickType_t xLastWakeTime;
    xLastWakeTime = xTaskGetTickCount();
    const TickType_t xMaxBlockTime = pdMS_TO_TICKS( 5000 );
    BaseType_t xResult;
    uint32_t ulNotifiedValue;

    while((xLastWakeTime - startTimeTick) < TIME_TO_RUN){

        xResult = xTaskNotifyWait( pdFALSE,
                                /* Don't clear bits on entry. */
                                ULONG_MAX,
                                /* Clear all bits on exit. */
                                &ulNotifiedValue, /* Stores the notified value. */
                                xMaxBlockTime );

        if( xResult == pdPASS )
        {
//              xSemaphoreTake(task1SyncSemaphore, xMaxBlockTime);
            TickType_t xCurrentTick = xTaskGetTickCount();
            UARTprintf("Task 1 started at %d ms and Timer interrupt data: %d\n",
xCurrentTick, ulNotifiedValue);
            fiboncacci(10);
            TickType_t xFibTime = xTaskGetTickCount();
            UARTprintf("Task 1 completed at %d ms (Execution: %d ms)\n",
                        xFibTime, (xFibTime - xCurrentTick));
            xLastWakeTime = xCurrentTick;
//              xSemaphoreGive(task1SyncSemaphore);
        }
    }
```

```c
}



// Process 2
void xTask2(void *pvParameters)
{
    TickType_t xLastWakeTime;
    xLastWakeTime = xTaskGetTickCount();
    const TickType_t xMaxBlockTime = pdMS_TO_TICKS( 5000 );
    BaseType_t xResult;
    uint32_t ulNotifiedValue;

    while ((xLastWakeTime - startTimeTick) < TIME_TO_RUN)
    {

        xResult = xTaskNotifyWait( pdFALSE,
                        /* Don't clear bits on entry. */
                        ULONG_MAX,
                        /* Clear all bits on exit. */
                        &ulNotifiedValue, /* Stores the notified value. */
                        xMaxBlockTime );

        if( xResult == pdPASS )
        {
//            xSemaphoreTake(task1SyncSemaphore, xMaxBlockTime);
            TickType_t xCurrentTick = xTaskGetTickCount();
            UARTprintf("Task 2 started at %d ms (Preempted Task 1) and  Timer
interrupt data %d\n", xCurrentTick, ulNotifiedValue);
            fiboncacci(40);
            TickType_t xFibTime = xTaskGetTickCount();
            UARTprintf("Task 2 completed at %d ms (Execution: %d ms)\n",
                       xFibTime, (xFibTime - xCurrentTick));
            xLastWakeTime = xCurrentTick;
//            xSemaphoreGive(task1SyncSemaphore);
        }
    }
}



// Main function
int main(void)
{
    // Initialize system clock to 120 MHz
    uint32_t output_clock_rate_hz;
    output_clock_rate_hz = ROM_SysCtlClockFreqSet(
                               (SYSCTL_XTAL_25MHZ | SYSCTL_OSC_MAIN |
                                SYSCTL_USE_PLL | SYSCTL_CFG_VCO_480),
                               SYSTEM_CLOCK);
    ASSERT(output_clock_rate_hz == SYSTEM_CLOCK);


    // Initialize the GPIO pins for the Launchpad
    PinoutSet(false, false);
    UARTStdioConfig(0, 230400, SYSTEM_CLOCK);

    ROM_SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER0);
    ROM_TimerConfigure(TIMER0_BASE, TIMER_CFG_PERIODIC);   // 32 bits Timer
    TimerIntRegister(TIMER0_BASE, TIMER_A, Timer0Isr);    // Registering  isr


    ulPeriod = (SYSTEM_CLOCK / Hz);
    ROM_TimerLoadSet(TIMER0_BASE, TIMER_A, ulPeriod -1);

    ROM_TimerEnable(TIMER0_BASE, TIMER_A);
    ROM_IntEnable(INT_TIMER0A);
    ROM_TimerIntEnable(TIMER0_BASE, TIMER_TIMA_TIMEOUT);

    task1SyncSemaphore = xSemaphoreCreateBinary();
    task2SyncSemaphore = xSemaphoreCreateBinary();


    xTaskCreate(xTask1, "Task1", configMINIMAL_STACK_SIZE, NULL, 2, &Task1_handle);
    xTaskCreate(xTask2, "Task2", configMINIMAL_STACK_SIZE, NULL, 1, &Task2_handle);
    startTimeTick = xTaskGetTickCount();
    xTaskNotifyFromISR(Task1_handle, startTimeTick, eSetValueWithOverwrite, NULL);
    xTaskNotifyFromISR(Task2_handle, startTimeTick, eSetValueWithOverwrite, NULL);

    vTaskStartScheduler();

    return (0);
}


/*  ASSERT() Error function
 *
 *  failed ASSERTS() from driverlib/debug.h are executed in this function
 */
void __error__(char *pcFilename, uint32_t ui32Line)
{
    // Place a breakpoint here to capture errors until logging routine is finished
    while (1)
    {
    }
}
```