CS466 Lab 3 – Simple Queues, Producer/Consumer, Serial IO and gdb debugging
Due by Midnight Friday 2-8-2019. (Probably a 2-week lab)

!!! Must use provided lab format!!!

You may hand in a team lab per, Individual or Two members are the maximum number that I want working together.

Overview:

- This lab is similar in thread structure to Lab 2, I suggest that you start with a working lab2 solution.

Lab Preparation:

- Review your Lab 2 thread structure.
- Read over the FreeRTOS API documentation for queues (http://www.freertos.org/a00018.html). Specifically look at the documentation for xQueueCreate(), xQueueSend(), and xQueueReceive() in detail, similar to semaphores. You have to create the queue structure before you can call the queue API operations.

- Carefully follow the steps in the lab3 file "CS466_lab03_Build_and_Debug_Basics.pdf" Understanding what is happening during build and debug will help you follow the steps.

Objective:

- Using the classical paradigm of a producer/consumer example you are to re-implement this instance in the multiple task RTOS environment. You will start using serial IO to monitor output and start debugging your code. You will also connect gdb and use and simple yet very powerful debugger to analyze assert failures in your code.

Lab Work

1. Copy your Lab02 code to this lab03. Name the main program producerConsumer.c. You will need to modify the Makefile to adjust the main code file name.

```
FreeRTOSConfig.h
Makefile
assert.c
assert.h
makedefs
blinkyRtos.c → producerConsumer.c
startup_gcc.c
ti_tm4c123g.ld
```

2. Delete all the interrupt and semaphore code that dealt with switches; modify the green-led task so that it's a minimal form similar to:

```
static void
_heartbeat( void *notUsed )
{
    uint32_t green500ms = 500; // 1 second
    uint32_t ledOn = 0;

    while(true)
    {
        ledOn = !ledOn;
        LED(LED_G, ledOn);
        vTaskDelay(green500ms / portTICK_RATE_MS);
    }
```

}

3. Rename your green thread to 'heartbeat' and have it run at idle priority. This thread should always run and keep the green LED blinking at around 1 Hz.

   Lab-report-question-1: How much effort did it take to get this basic heartbeat-only task working?

   - Look at my module I've added serialStubs.c. There are instructions in uartSerial.h serial. Once working this will give you a method to output printf() type data to your emulated serial port.
   - You can use any serial communications device to receive serial output from the tiva board. On Linux the device shows up as /dev/ttyACM0 (that's ACM<zero>), I use Kermit with the following config file below;

   Take Note: There are confusing interactions about when the serial communications disconnect and they are different on various platforms.

   I use Kermit on Linux and OSx and putty on Windows.

```
miller@trailtechu3:[~]
$ cat ~/kermACM0
set line /dev/ttyACM0
set speed 115200
set carrier off
set parity none
set handshake none
set stop-bits 1
set flow xon/xof

c
q

miller@trailtechu3:[~]
$ kermit ~/kermACM0
```

4. Create a Queue with 20 entries before you start the scheduler.
   - Use a pointer to a structure as your queue entry element.
   - Using his method you can pass data to a thread to manage it's behavior. If for instance you passed a pointer to a structure like the one below you can tune the operating characteristics of the task. This allows multiple tasks to actually run the same code.

```
struct task_characteristic {
    queueHandle_t my_queue;
    uint32_t          some_timing_paramter;
    char *            task_name;
}
```

   Lab Question-2: What are some trade-offs of running multiple tasks on the same code.

5. Add a consumer thread and a producer thread.
   - Look at the parameter that you pass from the TaskCreate call in main() to be received by the parameter passed to the thread when it's started.

- If the parameter you pass is to a structure like I gave above you can pass it ok but you will need to cast it back to your structure-pointer type after it's been received by the task.
- Pass the handle of the queue to each thread as part of its thread parameters, do not use global variables. (to do this you will need to cast the queue handle to a (void *) and pass as pvParameters)

  Important Note: You cannot allocate memory for your characteristics structure as an automatic in main(). For now create the structure as either 'static' in main() or as a global structure. You will have to update it in main() after you create your queues but that works ok.

6. Make the consumer thread block on queue receive so and momentarily light the blue led whenever a message is received. Use a frequency that works visually.

7. Make your producer thread block for a random delay then send a message, If the queue is full assert(). Try to hit about 10 messages per second for a starting average rate. If you assert you are probably holding the LED on too long and overflowing the queue.

8. Add a second producer that will also insert messages into the queue. When the consumer thread receives a consumer2 message light the red LED momentarily. Also time the random message generation rate to average about 10 Hz. Assert if the queue send fails. This second producer thread should use the same function as the first.

9. Re-arrange the priorities so that the consumer thread has a lower priority then the two producer threads. If the program does not assert after a while, increase the period of the LED indication in the consumer to slow it down.

   Lab Question – 3: Why do I expect an assert here?

Debugging Tools (UARTPrintf, assert, and gdb).

10. Using the provided handout (pdf file in the lab directory), get the debugger up and running with your now running program

11. In the top of the assert.c file add a printf so that assert conditions report where the assert failed.

```
#ifdef USB_SERIAL_OUTPUT
UARTprintf("Assertion Failed: %s at %s::%d\n",assertion, file, line);
#endif
```

12. After the UARTprintf Verify ort add the two lines in your assert code.
    - taskENTER_CRITICAL();
    - taskDISABLE_INTERRUPTS();

13. Add some code so you fail an assert() in a producer task. In theory any other threads should have been stopped as well by the two lines added above.
    - Do the other Threads stop executing as well?
    - Why or why not are the other threads stopping?

14. Try with the two lines above in the code or commented out.

    Lab Question – 4: Is there any difference between the two cases?

15. Take time to review the file CS466_lab03_Build_and_Debug_Basics.pdf.
    - Take a look at the .gdbinit file documented in the .pdf file and in your directory.
    - You will also need to create the global .gdbinit file in your $HOME directory,

16. Start GDB as in the .pdf file.
    - Take a look at the GDB cheat sheet file and try some commands that don't actually execute any code.
    - Display the first 16 32-bit values stored in ROM (at address 0x00000000)
    - Display the processor registers
    - Display what breakpoints are set
    - What data is in the first 8 bytes of memory?
    - Note that if you make and try to flash the image from the linux command line the flash will fail because openocd has reserved the ICDI USB interface to the Tiva.
    - Make a small modification to your program
    - At the (gdb) prompt type the command s

```
(gdb) make
(gdb) reload
(gdb) c
```

      Lab Question – 5: Describe what happened. (Assuming this instruction is correct)

      Lab Question – 6: What do the following commands do?

      a) (gdb) reload
      b) (gdb) I b
      c) (gdb) I r
      d) (gdb) l
      e) (gdb) mr
      f) p <variable>
      g) p {<var>, <var>, <var>}
      h) si) n

    - Take some time to set some breakpoints and step through your code.

CS566 students must complete the following steps for full credit,
CS466 students may complete it for extra credit.

In step 5, I had you use storage 'Not' automatically allocated in the main() function. Lets investigate that some.

17. Given the hypothesis that automatic variables allocated in main() will get corrupted, devise an experiment that will expose variables corrupted in main.
    - Does it happen?
    - Why does it happen?

18. Given that our runtime environment is laid out in the startup_gcc.c file, and that the corrupted stack space is allocated there as an array. Make modifications to startup_gcc so that automatic variables in main do not get corrupted.
    - Describe your solution and any limitations it imposes on you.