Module 2: Lecture Questions

Module 2 - Lecture 6 (Thursday – 4/1/21)

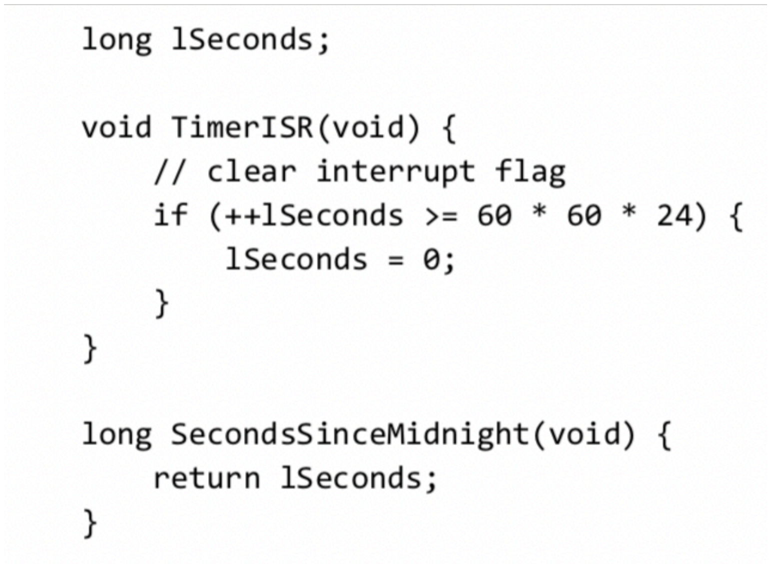
1. What is the only way to pass data into or out of an ISR?
2. What is shared data and when do problems arise because of it?
3. Explain how the ece3849_int_latency example program calculates CPU utilization.
4. Why will the value for CPU Utilization returned from ece3849_int_latency differ from the CPU utilization that we calculated using the RMS Theory?
5. What RMS Theory condition is not met, using the Earliest Deadline First Scheduling strategy?
6. Why not always use Earliest Deadline First (EDF) Scheduling? Under what conditions would you choose to use EDF. This is a nice link that outlines the pros and cons of EDF at the bottom.
   https://microcontrollerslab.com/earliest-deadline-first-scheduling/
7. The system with the characteristics below uses Earliest Deadline First scheduling.
   a. Use graphical method to determine the latency, response time and if it is schedulable for each task.
   b. How many ms do you need to graph to find the worst-case response time values?
   c. What is the CPU utilization of the system? What is the utilization upper bound for EDF that guarantees that tasks will be schedulable?

   https://en.wikipedia.org/wiki/Earliest_deadline_first_scheduling

Event	Period	Execution Time	Latency	Response Time	Relative Deadline	Schedulable ?
TaskA	4 ms	1 ms				
TaskB	6 ms	2 ms				
TaskC	8 ms	3 ms				

Module 2 - Lecture 7 (Monday – 4/5/21)

8. In the Time Example, we looked at solving our shared data problems in several ways. One option was to disable some or all of the interrupts. What are the pro's and con's of disabling some and all interrupts?
9. Under what conditions does priority inversion occur? Why is priority inversion to be avoided at all costs?
10. What is an atomic operation?
    a. Explain why x++ is not atomic. Explain how using this statement in a low-priority thread could create a shared data bug.
    b. What do I need to know to determine if x = y is an atomic operation?
    c. Why is it important to use atomic operations during critical shared data sections of the code? Explain two shared data bugs that are caused by non-atomic operations.
11. We saw that the following code had a shared data bug, when the function SecondsSinceMidnight was called from a higher priority interrupt



```
long lSeconds;

void TimerISR(void) {
    // clear interrupt flag
    if (++lSeconds >= 60 * 60 * 24) {
        lSeconds = 0;
    }
}

long SecondsSinceMidnight(void) {
    return lSeconds;
}
```

    a. Describe the cause and the outcome of the data bug.
    b. What are two fixes for this bug?

See next page for more

12. Your system is set up to interrupt when a fault is detected. The ISR increments a global fault counter, and main() handles the faults, decrementing the counter. The following code outlines the communication between the ISR and main(). Assume this code runs on an unspecified 32-bit CPU architecture: 32-bit **reads** and **writes** are atomic.

```
volatile uint32_t fault_count = 0;

void FaultDetectISR(void) {
  <clear interrupt flag>;
  fault_count++;
}

void main(void) {
  <init>;

  while (1) {
    while (fault_count > 0) {
      fault_count--;
      <handle one fault>; // does not use fault_count
    }
    // other unrelated code
  }
}
```

a. Find the shared data bug in this code. Explain the sequence of events that may cause an error.
b. Fix the shared data bug by disabling, then re-enabling interrupts. Exclude `<handle one fault>` from the critical section to minimize impact on interrupt latency.
c. Fix the shared data bug without disabling interrupts. Hints: You want to make the communication between the ISR and main() one-directional: if the ISR writes to `fault_count`, main() only reads it. This implies that `fault_count` cannot be decremented or modified in any other way in main(). Use local variables to replace the lost functionality.

See next page for more questions.

13. A timer ISR keeps track of time in milliseconds. Functions have been written to allow main() to read and adjust the time. This code runs on a **32-bit** CPU: 32-bit reads and writes are atomic.

```
volatile uint32_t time = 0;

void TimerISR(void) {           // periodic ISR, period = 1 ms
    <clear timer interrupt flag>; // <...> delineates pseudo-
code
    time++;
}

uint32_t GetTime() {            // called from main()
    return time;
}

void AdjustTime(uint32_t adj) { // called from main()
    time += adj;
}
```

   a. AdjustTime() is known to intermittently corrupt the `time`. Why does this happen? What is the worst case deviation from the correct time due to a single occurrence of the shared data issue? Exclude the special case when `time` wraps around.
   b. Why do shared data issues not occur in GetTime()?
   c. Fix AdjustTime() to resolve the shared data issue with the minimum changes to the existing code. .

Module 2 - Lecture 8 (Tuesday – 4/6/21)

14. For the problem in 13, assume that AdjustTime() is called at a rate slower than the TimerISR period. Implement a Mailbox such that TimerISR is the only thread writing to time.
15. A read-modify-write operation is occurring in a low priority task like main(). A high priority ISR can interrupt it at any time. What conditions need to be met in the ISR not to have a shared data problem?
16. Rewrite this non-atomic operation to be atomic using the bit-banding macro provided by TI. x &= ~(0x1 << 5);
17. In class we saw that by doing repeated Non-Atomic writes we could eliminate the shared data problem in the TimerISR / SecondsSinceMidnight example. Rewrite the TimerISR and SecondsSinceMIdnight function to use a sequence number instead of reading all the values. What advantages does this have?

18. A periodic ISR is sampling input data and recording a timestamp for each data point. The function getData() is provided for main() to read one data point and its timestamp. It is important that the timestamp and data point come from the same sample. Assume an `int` can be read or written atomically.

```
int t = 0;
int d;

void inputISR(void)
{
    <clear interrupt flag>;      // <...> delineates pseudo-code
    t = <time>;                  // assume time is always non-zero
    d = <data>;                  // sample data
}

int getData(int *pt, int *pd)
{
    if (t != 0) {
        *pt = t;
        *pd = d;
        t = 0;
        return 1;  // success
    }
    else
        return 0;  // no data available
}
```

(a) Find all the shared data bugs in the current implementation. Explain the errors they may cause.

(b) Is the following modified implementation free of shared data bugs that cause data corruption? How often must this getData() be called such that no data points are lost?

```
int getData(int *pt, int *pd)
{
    if (t != 0) {
        IntMasterDisable();
        *pt = t;
        *pd = d;
        t = 0;
        IntMasterEnable();
        return 1;  // success
    }
    else
        return 0;  // no data available
}
```

(c) Re-implement inputISR() and getData() to avoid disabling interrupts but still fix the data corruption problem (data point and timestamp mismatch). State any assumption you have made about thread timing.

19. A periodic ISR reads position and speed sensors and stores the results in circular buffers. The function getxv(), called from main(), returns the latest position and speed from the buffers. Assume an `int` can be read or written atomically.

```
#define N 1000      // number of samples in the circular buffers
volatile int x[N];  // circular buffer for position
volatile int v[N];  // circular buffer for speed
volatile int i = 0; // index of the latest sample in each buffer

void ISR_xv(void) {              // periodic ISR
    <clear interrupt flag>;      // <...> delineates pseudo-code
    i++;                         // increment and wrap index
    if (i >= N) i = 0;
    x[i] = <read position>;      // store data in circular
buffers
    v[i] = <read speed>;
}
void getxv(int *px, int *pv) {   // called from main()
    *px = x[i];
    *pv = v[i];
}
```

a. In the function getxv(), **underline** all accesses to shared global variables and label them “read,” “write” or “read-modify-write.” Explain the shared data problem(s) that can occur. Explain why some of the accesses to shared data do not cause problems, stating any conditions, if required. Hint: Note the similarity to the Lab 1 ADC buffer.
b. Fix the shared data bug(s) in getxv() without disabling interrupts.

See next page for more questions.

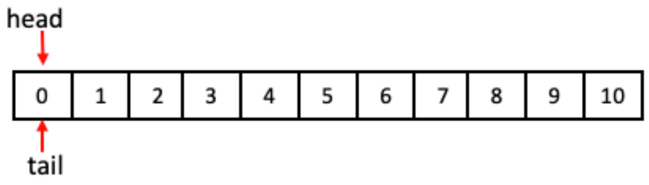
Module 2 - Lecture 9 (Thursday – 4/8/21)

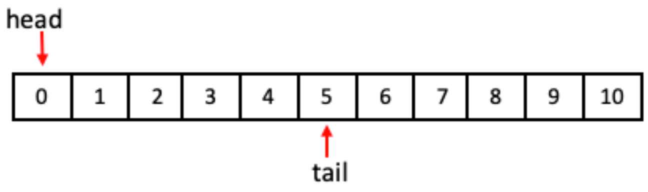
20. For the problem in 13, assume that AdjustTime() can be called at most every 100 usec. Explain how you might use a FIFO to fix the shared data problem such that TimerISR is the only thread writing to time.
    a. The TimerISR() is called every msec, assuming TimerISR() will read all the entries in the FIFO when it is called what is the minimum FIFO length that can be used?
    b. Using the fifo_get() and fifo_put() commands, update the code below to implement the fifo. Remember to add the required defines and global variables to implement the fifo. You do not have to write the fifo_get() and fifo_put() functions, just use them.
    c. If getTime() could be called from a higher priority interrupt than the TimerISR, would we have a shared data problem? Explain.

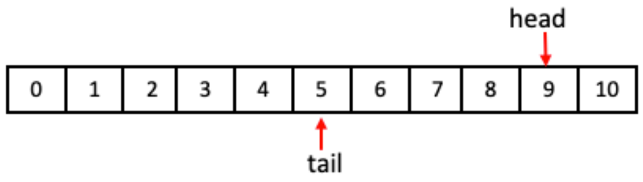
```
volatile uint32_t time = 0;

void TimerISR(void) {           // periodic ISR, period = 1 ms
    <clear timer interrupt flag>; // <...> delineates pseudo-code
    time++;
}

uint32_t GetTime() {            // called from main()
    return time;
}

void AdjustTime(uint32_t adj) { // called from main()
    time += adj;
}
```

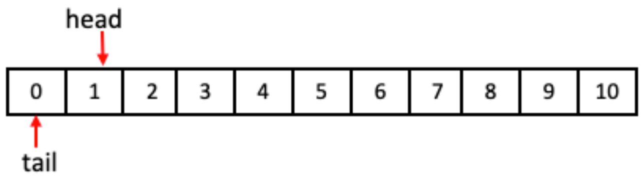
21. In the shared_data example demonstrated in class, the ISR wrote up to 5 characters every 10 msec into a FIFO. The main function read 1 character every 100 msec. When the ISR saw that the FIFO was full it just stopped writing until there was space in the FIFO.
    Not all applications can just stop sending, some will have to write 5 characters of data every 10 msec or data will be lost. Explain how you would modify the FIFO and the main() functionality such that it still reads from the FIFO every 100 msec and prints the new data to the screen but without data loss.
22. In class we discussed the shared data problem with the fifo_get() function. There were two possible solutions to the problem. What were they? Which one was preferred and why?
23. What does the keyword volatile mean? Why do global variables such as fifo_head and fifo_tail have to be volatile?

24. In the figures showing the fifo head and tail index locations for a FIFO holding characters. How many characters are stored in each FIFO? Is the FIFO full or empty?

a)


b)


c)


d)

head
0
1
2
3
4
5
6
7
8
9
10
tail