Lecture #17

Concurrency

18-348 Embedded System Engineering

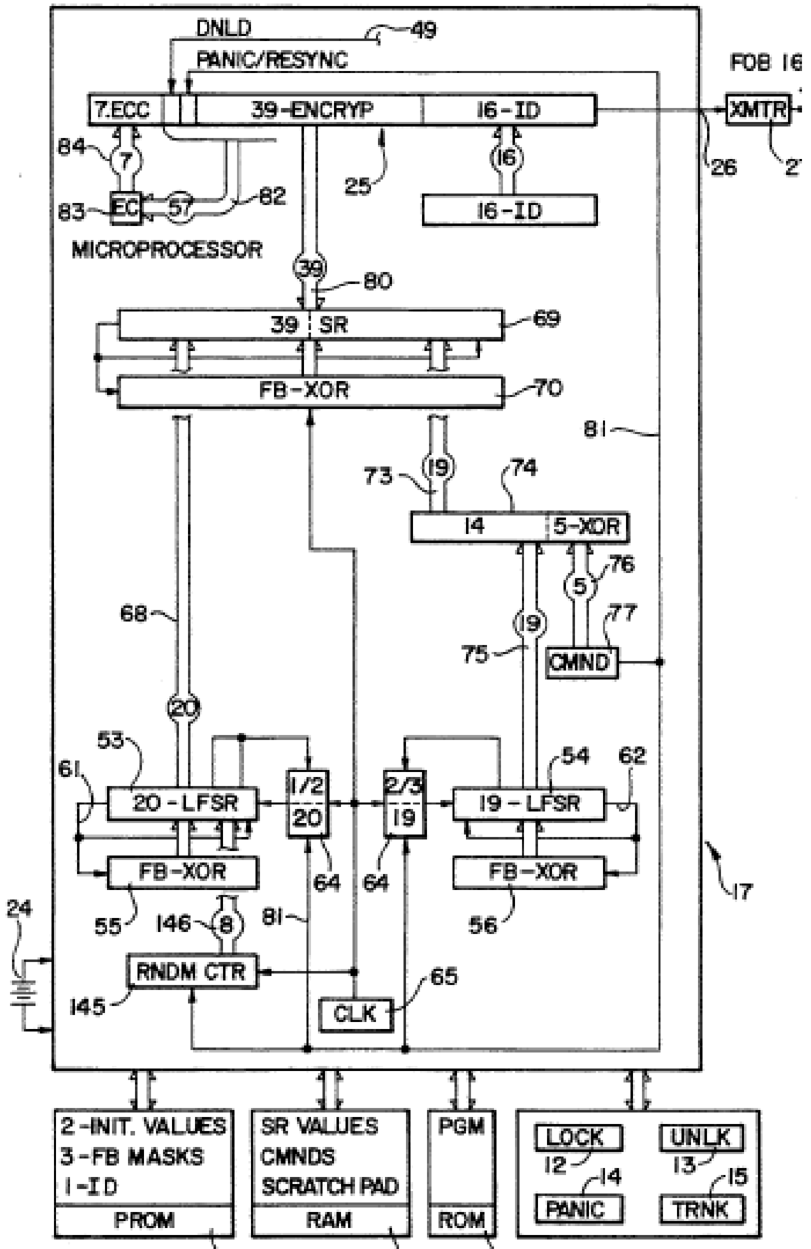
Philip Koopman

Monday, 21-March-2016

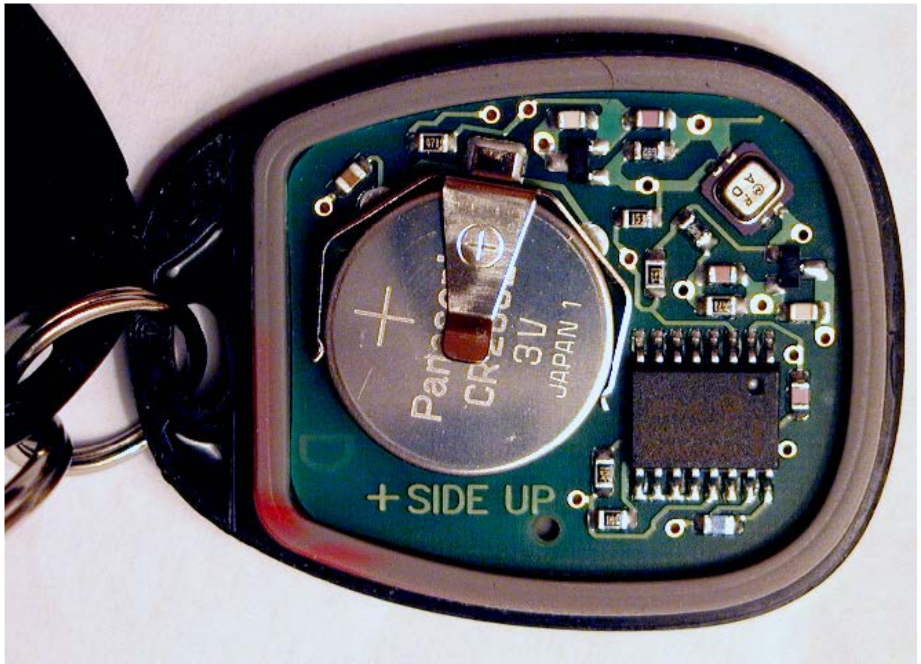
Electrical & Computer ENGINEERING

Carnegie Mellon

Example application – Remote Keyless Entry



Lear Encrypted Remote Entry Unit



KeeLoq®

- Lightweight block cipher
- 32-bit block size
- 64-bit key
- Sold by Microchip® Inc.

"Secure Learning" Key Derivation Procedure

[Biham et al. CRYPTO 2007]

- The manufacturer has a master secret
- For each car there is a unique identifier (known to the attacker)
- The XOR of these two gives the secret key used in this car.

Conclusion

- Finding **one** KeeLoq key leaks the **master secret**.

- **KeeLoq® is badly broken**
- Soon, cryptographers will all drive expensive cars*

Subaru is recalling more than 47,000 cars, including this Crosstrek, with faulty fobs and the ghostlike potential to start themselves, without human intervention.

http://money.cnn.com/2013/03/07/news/companies/subaru-recall/index.html?source=cnn_bin

NEW YORK (CNNMoney)

The Japanese automaker Subaru is recalling nearly 50,000 zombie-vehicles because they run the risk of starting themselves, without human intervention.

This is no small matter, according to the Subaru letter, which detailed what can happen when the car takes on a life of its own:

"The engine may inadvertently start and run for up to 15 minutes," the letter said. "The engine may continue to start and stop until the fob battery is depleted, or until the vehicle runs out of fuel. If the vehicle is parked in an enclosed area, there is a risk of carbon monoxide build-up which may cause asphyxiation."

First Published: March 7, 2013: 1:19 PM ET

Where Are We Now?

- **Where we've been:**
  - Interrupts
  - Context switching and response time analysis

- **Where we're going today:**
  - Concurrency

- **Where we're going next:**
  - Scheduling, real time system practicalities
  - Analog and other I/O
  - Robustness, safety
  - Bluetooth & CAN
  - Test #2
  - Last project

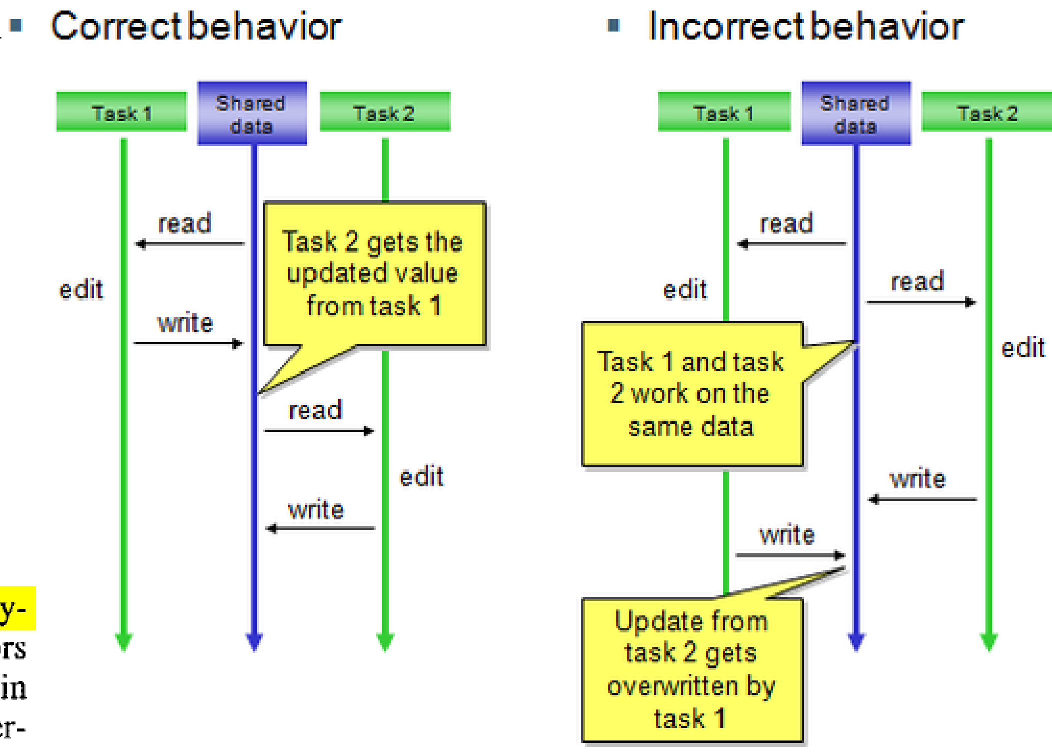
Preview

- **Buffer management**
  - Buffering and FIFOs
- **Reentrant code**
  - Making sure code can be executed by multiple threads concurrently
- **Atomic Actions**
  - Making sure that an operation can't be interrupted
- **Semaphores**
  - Mutex to implement mutual exclusion of critical regions

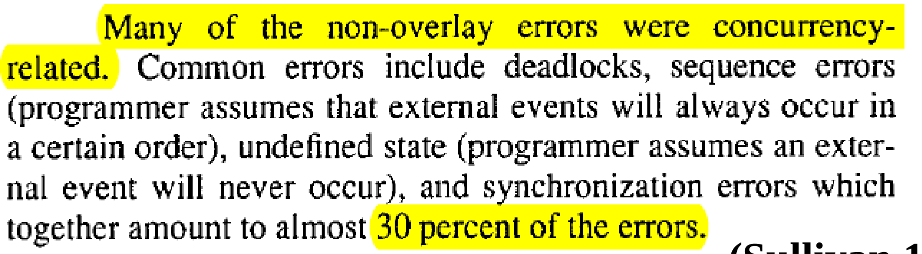
  - … and some example concurrency hazards …

Concurrency Problems In General

- **One CPU can have many tasks**
  - Tasks take turns, sharing the CPU and memory
  - CPU rapidly switches between tasks ("multi-tasking")
- **Concurrency defects often result from defective resource sharing**
  - Need to ensure that two tasks don't both try to change a global variable at the same time, e.g. via disabling interrupts to prevent task switching
  - Defects may be due to subtle timing differences, and are often difficult to reproduce



(Wind River)



Many of the non-overlay errors were concurrency-related. Common errors include deadlocks, sequence errors (programmer assumes that external events will always occur in a certain order), undefined state (programmer assumes an external event will never occur), and synchronization errors which together amount to almost 30 percent of the errors.

(Sullivan 1991, p. 7)

Buffer Management

- **Buffers are used to temporarily store data**
  - Used to collect pieces while they are being assembled
  - Used to hold assembled pieces while they are being disassembled
  - Used to hold incoming data until it can be processed
  - Used to hold outgoing data until it can be transmitted
  - Used to hold data too big to fit in registers during processing

- **Example: transmit buffer**

  uint8 buff[80];

  1. Put message to be transmitted into buff[]   (up to 79 chars plus null byte)
  2. Tell transmit routine to start transmitting at buff[0]
  3. Wait until transmission is completed
  4. Go to step 1 for next message

  - Don’t forget to check for over-running max length!!!!

Single Buffer Message Transmitting

- **To transmit multiple messages via Serial Port**

  volatile uint8 buff[80]; // message buffer

  volatile uint8 buff_owner=1; // who owns buffer? 1 is task 1; 2 is task 2

  // no concurrency issue -- task can't reclaim buffer until after other task uses it

- **Task 1 – transmit the next message:**

```
for(;;) // transmit messages forever
{ while(buff_owner == 2){sleep;} // wait for other task handoff
    buff[] = next message ;  // copy next message to buff
    buff_owner = 2;
}
```

- **Task 2 – actually send the bytes:**

```
while (still messages to transmit)
{  while(buff_owner == 1) {sleep;}  // wait until data ready
     send message byte at a time;
     buff_owner = 1;
}
```

- **But, only one task can get work done at time**

Double Buffering

- **Single buffering means one task is always waiting**
  - Task 1 produces data – waits for task 2 when done with a buffer
  - Task 2 consumes data – waits for task 1 when done with a buffer

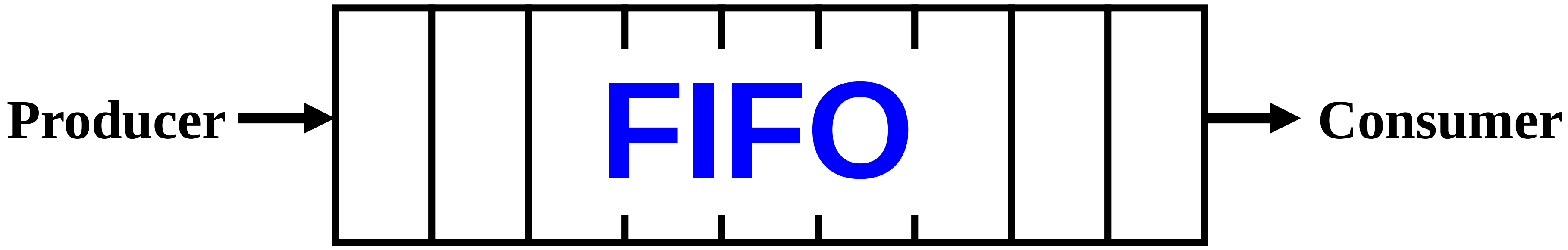
- **Double buffering idea:**
  - Two buffers   Buffer A and Buffer B

  - Phase 1:
    - Task 1 owns Buffer A   Task 2 owns Buffer B
    - Task 1 fills Buffer A   Task 2 consumes Buffer B
    - Wait until both Task 1 and Task 2 are done
  - Swap – each task trades buffers with the other task
  - Phase 2:
    - Task 1 owns Buffer B   Task 2 owns Buffer A
    - Task 1 fills Buffer B   Task 2 consumes Buffer A
    - Wait until both Task 1 and Task 2 are done
  - Swap – each task trades buffers with the other task
  - Go to Phase 1

FIFO (Queue)

- **To decouple producer and consumer, use a FIFO**
  - FIFO = “First In First Out”
  - Multiple items
  - Item inserted, waits until previous items removed, then that item is removed

- **Speeds are independent as long as they don’t get too far ahead/behind**
  - Producer can produce faster or slower than consumer
    - FIFO has fixed size, so too many items too quickly will overflow FIFO!
  - Consumer can consume faster or slower than producer
    - FIFO might get empty if consumer is slower than producer for too long



FIFO Implementation

- Usually implemented with circular accesses to an array
  - "head pointer" – the head of the queue = next item to be removed
  - "tail pointer" – the tail of the queue = most recent item inserted
  - One way to implement: when head == tail, FIFO is empty
    - Lab assignment uses another way, involving an empty/full flag

```
#define FIFOSIZE 10
volatile int fifo[FIFOSIZE];                 // one int per element
volatile uint8 head = 0, tail = 0;           // init to empty FIFO

bool insert(int x)   // insert; return 1 if success; 0 if fail
{ int newtail;
  // access next free element; wrap around to beginning if needed
  newtail = tail+1;   if (newtail >= FIFOSIZE) { newtail = 0; }

  // if head and tail are equal, fifo would overflow
  if (newtail == head) {return(FALSE)};  // FIFO is full

  fifo[newtail] = x;   // write data before updating pointer
  tail = newtail;      //   … otherwise remove might get stale data
  return(TRUE);
}
```

Reentrant Code

- **Reentrant code can have more than one thread executing it at a time**
  - i.e., can be "entered" more than once at a time
    - A bit different than "shared variables" – it's about the code, not just a data location
  - Originated in memory-limited mainframes to re-use subroutines…
    … still relevant for OS code, and for multi-threaded code
    … and can still be relevant for shared library code
    … and definitely relevant for small-memory-size embedded systems

- **Important for embedded systems for:**
  - ISRs that re-enable mask bit (don't do this if you can avoid it!)
  - Shared code, such as:
    - Math libraries with statically allocated memory
    - Exception handlers with statically allocated memory
    - Methods to handle data structures
  - Recursive code (don't do this if you can avoid it!)
  - Usually not important for ordinary "main loop" application code

  - **Question: are global variables reentrant?**

Example Reentrancy Problem

- **Compute nth Fibonacci number (1, 1, 2, 3, 5, 8, 13, 21, …)**
  - We're using this because it doesn't require exact timing to show the problem

```
uint16 fib(uint16 n)
{ uint16 sum;
  if (n < 2)  return (n);
  sum =  fib(n-1);
  sum += fib(n-2);
  return(sum);
}
```

- Produces this correct output ➔ ➔ ➔ ➔ ➔ ➔ ➔ ➔

```
N        fib(N)
0          0
1          1
2          1
3          2
4          3
5          5
6          8
7          13
8          21
9          34
10         55
11         89
12         144
13         233
14         377
15         610
16         987
17         1597
18         2584
19         4181
20         6765
```

Let’s Introduce A Reentrancy Problem

```
static uint16 sum; // temporary global holding variable

// compute nth fibonacci number using recursion
uint16 fib(uint16 n)
{ if (n < 2)  return (n);
  sum = fib(n-1);
  sum += fib(n-2);
  return(sum);
}
```

N	fib(N)
0	0
1	1
2	1
3	2
4	2
5	4
6	4
7	8
8	8
9	16
10	16
11	32
12	32
13	64
14	64
15	128
16	128
17	256
18	256
19	512
20	512

- **Problem is with variable sum**
  - fib(n-1) stores value in sum
  - fib(n-2) trashes sum with recursive call

Sometimes Subtle Code Changes Are All It Takes

- **With Code Warrior this code works (but is asking for trouble!)**
  - Luckily, C compiler doesn't update sum until after addition – but that is just luck!

```
uint16 fib(uint16 n)
{ static uint16 sum;
  if (n < 2)  return (n);
  sum = fib(n-1) + fib(n-2);
  return(sum);
}
```

- **BUT, this code fails:**
  - C compiler decides to update sum rather than keep it in a register before the add

```
uint16 fib(uint16 n)
{ static uint16 sum;
  if (n < 2)  return (n);
  sum = fib(n-1);
  sum += fib(n-2);
  return(sum);
}
```

General Rules To Avoid Reentrancy Problems

- **Assembly language**
  - All scratch variables have to go on the stack
  - No references to statically allocated memory (unless protected by semaphores)
    - This includes globals, static keyword variables, and I/O registers
- **Reentrancy problems are common in assembly language**
  - We know that using the stack for temp variables is a pain to write
  - BUT, if you use a “DS” defined variable, you risk reentrancy problems
- **Reentrant C programs must have at least:**
  - No global variables (globals also compromise modularity)
  - No use of keyword “static” for local variables
  - No use of pointers (some might be OK, but asking for trouble)
  - No reference to variables outside scope of current procedure
- **If you writing “good” C, reentrancy problems are unusual**
  - Mostly because putting values on stack is easy, and “static” keyword is rare
  - But they can still happen, and are very difficult to debug!

Atomic Actions

- **An "atomic action" is one that can't be stopped once it is started**
  - Execution of a single instruction, e.g.:

```
INC   3,SP
```

  - A sequence of actions completed in hardware, e.g.:

```
SWI               ; stacks many register values

<HW Interrupt>       ; stacks many register values

LDD   TCNT     ; load both bytes of TCNT; hardware
               ; prevents TCNT changing during the load
```

  - Execution of a non-interruptible piece of code, e.g.:

```
SEI            ; mask interrupts
LDAA  0,SP
ADDA  #7
STAA  0,SP
CLI            ; enable interrupts
```

Do C Compilers Generate Atomic Actions?

- **Which of these is an atomic action?**
  - `foo = foo + 1;`
  - `foo += 1;`
  - `foo++;`

- **Trick question – it all depends on the context, CPU, and compiler!**
  - For example, in this code:

    ```
    foo += 1;
    bar += foo;
    ```
  - Compiler might increment foo in memory
    OR
    load foo, increment, then store, keeping in register for adding to bar.
  - There is no guarantee of atomicity in source code!

HC12 With Code Warrior Examples

- `foo++;          // is atomic for uint8`

```
  16:      foo++;
   0011 6282           [3]      INC    3,SP
```

- `bar++;          // is NOT atomic for uint16`
  `               // HC12 doesn’t have 16-bit memory INC`

```
 18:      bar++;
   0014 ee80           [3]      LDX    0,SP
   0016 08             [1]      INX
   0017 6e80           [2]      STX    0,SP
```

- `baz += 2;       // is NOT atomic for any data size`

```
  20:      baz += 2;    //   what if we did:  baz++; baz++;
   0019 e682           [3]      LDAB   2,SP
   001b cb02           [1]      ADDB   #2
   001d 6b82           [2]      STAB   2,SP
```

Uses For Atomic Actions

- **Accessing changing hardware values**
  - E.g., getting all 16 bits of TCNT without a change between bytes
  - E.g., changing SCI parameters or TCNT parameters and ISR vectors all at once

- **Accessing values changed by ISRs**
  - E.g., getting time of day that is updated by ISR

- **Accessing a variable shared among tasks**
  - For example, a single counter of errors or events for a single task would be:

    ```
    events++;
    ```

  - But if shared among multiple tasks, would have to be:

    ```
    #define CLI() {asm cli;}
    #define SEI() {asm sei;}
    …
    SEI();  // be careful to minimize blocking time!
    events++;
    CLI();  // does this disable or enable interrupts?
            //  …. Are you 100% sure without looking it up?
    ```

Top Concurrency Bug In The Field

```
uint32 timer; // assume initialized to current time

void main(void)
{ … initialization …
   for(;;)
    { do_task1();
      do_task2();
   }
}

void do_task1()
{   . . . .
  x = timer;  // Sometimes x doesn't get a clean value!
     . . . .
}

void interrupt 16 timer_handler(void) // TOI
{ TFLG2 = 0x80;
  timer++;
}
```

More Bad Code in a 3rd Party Library

```
volatile uint16_t * p_timer_lower16 = …;
volatile uint16_t * p_timer_upper16 = …;

uint32_t timerRead(void)
{
    return (*p_timer_upper16 << 16) | *p_timer_lower16;
}
```

What happens when lower 16 bits rolls over from 0xFFFF to 0x0000?

Fix To The Top Concurrency Bug In The Field

```
volatile uint32 timer; // assume initialized to current time

void main(void)
{ … initialization …
   for(;;)
    { do_task1();
      do_task2();
   }
}

void do_task1()
{   . . . .
  DisableInterrupts();
  x = timer; // Access to timer is made atomic
  EnableInterrupts();
     . . . .
}

void interrupt 16 timer_handler(void) // TOI
{ TFLG2 = 0x80;
  timer++;
}
```

What About Really Long Atomic Actions?

- **Masking interrupts to update a 16 bit counter is probably OK**
  - But not to do a really long computation on a shared data structure!
  - For example:
    - Inserting a linked list element
    - Computing an FFT on a data set
    - Access to an A/D converter or the UART to send multiple bytes
  - Why? Because the longest interval of masked interrupts is Blocking Time **B**

- **For longer accesses to shared data structures, need a semaphore**
  - Semaphore is a way to ensure only one task accesses data at a time
    - *Even if* task switches occur during the data access
    - In the general case there might be multiple users of multiple shared resources
  - Special case is **Mutex** (“**Mut**ual **Ex**clusion”) – single shared resource

  - **Critical section** – a region in the code where it accesses a shared resource
    - Needs protection (often via a mutex) to avoid concurrency problems

Conceptual Build-Up To Implementing A Mutex

- **Starting point below**
  - Don’t want to do this – interrupts disabled for too long

```
Mystruct foo;  // foo is shared by multiple tasks

… somewhere in a task …
  SEI();
  foo.a = <newval>;
  foo.b = <newval>;
  foo.c = <newval>;
  …
  foo.zy = <newval>;
  foo.zz = <newval>;
  CLI();
```

Add A Flag To Control Access (a Mutex)

- Add a binary variable indicating "free" (0) or "locked" (1)
  - *THIS VERSION DOESN'T WORK (why?)*

```
Mystruct foo;  // foo is shared by multiple tasks
uint8  foo_mutex = 0; // 0 means nobody using
                      // 1 means in use (locked)

… somewhere in a task …
  while(foo_mutex==1){;}   // wait while it is busy
  foo_mutex = 1;           // acquire resource

  foo.a = <newval>;
…
  foo.zz = <newval>;
  foo_mutex = 0;           // free resource
```

Try Making Mutex Update Atomic

- *THIS VERSION DOESN'T WORK EITHER (why?)*

```
volatile Mystruct foo;  // foo is shared by multiple tasks
volatile uint8  foo_mutex = 0; // 0 means nobody using
                               // 1 means in use (locked)

… somewhere in a task …
  while(foo_mutex==1){;}    // wait while it is busy
  DisableInterrupts();
  foo_mutex = 1;            // acquire resource
  EnableInterrupts();

   foo.a = <newval>;
…
   foo.zz = <newval>;
  foo_mutex = 0;            // free resource
```

Try Making Test And Set Of Mutex Atomic

- ***THIS VERSION DOESN'T WORK EITHER (why?)***

```
volatile Mystruct foo;  // foo is shared by multiple tasks
volatile uint8  foo_mutex = 0;
                      // 0 means nobody using
                      // 1 means in use (locked)

… somewhere in a task …
  DisableInterrupts();
  while(foo_mutex==1){;}   // wait while it is busy
  foo_mutex = 1;           // acquire resource
  EnableInterrupts();

   foo.a = <newval>;
…
   foo.zz = <newval>;
  foo_mutex = 0;           // free resource
```

So What We Need Is A Second Test of Mutex

- ***THIS VERSION SHOULD WORK***

```
volatile Mystruct foo;  // foo is shared by multiple tasks
volatile uint8  foo_mutex = 0; // 0 means nobody using
                      // 1 means in use (locked)

… somewhere in a task …
   uint8 initial_value;
   do {     DisableInterrupts();
            initial_value = foo_mutex;
            foo_mutex = 1;
            EnableInterrupts();
      } while (initial_value == 1);

  foo.a = <newval>;
…
  foo.zz = <newval>;
  foo_mutex = 0;            // free resource
```

This Is Called The “Test-and-Set” Approach

- ***THIS VERSION SHOULD WORK***

```
volatile Mystruct foo;  // foo is shared by multiple tasks
volatile uint8  foo_mutex = 0; // 0 means nobody using
                      // 1 means in use (locked)

… somewhere in a task …
  GetMutex(&foo_mutex);

  foo.a = <newval>;
…
  foo.zz = <newval>;
  ReleaseMutex(&foo_mutex);           // free resource
```

Test And Set Primitives

```
#define BUSY 1
#define IDLE 0      // Every mutex must be initialized to IDLE

uint8 SwapAtomic(uint8 volatile *mutex, uint8 v)
{ uint8 res;
  DisableInterrupts();
     res = *mutex;  // atomically swap  inp and *mutex
     *mutex = v;
   EnableInterrupts();
  return(res);
}

void GetMutex(uint8 volatile *mutex)
{ uint8 val;
  do
  { val = SwapAtomic(mutex, BUSY);  // grab for the mutex
  } while (BUSY == val);            // success if val==0
}

Void ReleaseMutex(uint8 volatile *mutex)
{ *mutex = IDLE;    // no need for atomicity here (why?)
}
```

Test And Test And Set

- **Cached multi-core and multi-processor systems**
  - "Test & test & set" is more efficient for multi-processors with shared data bus
  - Test variable and only attempt Test&Set if it is currently unlocked
    - Reduces bus traffic by avoiding writes if lock is already set
    - Reduces bus traffic by reading from cache until it is invalidated by other write
  - http://en.wikipedia.org/wiki/Test_and_Test-and-set

```
void GetMutex(uint8 volatile *mutex)  // test and test and set
{ uint8 val;
  do
  { val = *mutex;
    if(BUSY != val)  // if it's busy here, skip the swap
    { val = SwapAtomic(mutex, BUSY);  // grab for the mutex
        // This might still fail if another task grabs it first
    }
  } while (BUSY == val);              // success if val==IDLE
}
```

Other Test And Set Considerations

- **Cooperative Multitasking**
  - Don’t want to sit forever waiting for Mutex – we won’t get it!
  - Need to return to scheduler loop whenever Mutex is busy
- **Preemptive Multitasking**
  - Might want to “yield” after every test to avoid burning CPU time
    - “yield” returns control to tasker, relinquishing rest of CPU time for now
    - Improves CPU efficiency

```
void GetMutex(uint8 volatile *mutex)  // yield version
{ uint8 val;
  do
  { val = *mutex;
    if(BUSY == val)
    { Yield();   // if it’s busy, yield to another task
    } else
    { val = SwapAtomic(mutex, BUSY); // grab for the mutex
        // This might still fail if another task grabs it first
    }
  } while (BUSY == val);              // success if val==IDLE
}
```

Mutex Hazards

- **Deadlock**
  - Task A needs resources X and Y
  - Task B needs resources X and Y

  - Task A acquires mutex for resource X
  - Task B acquires mutex for resource Y

  - Task A waits forever to get mutex for resource Y
  - Task B waits forever to get mutex for resource X

- **Livelock**
  - Tasks release resources when they fail to acquire both X and Y, but…
    just keep deadlocking again and again

- **Dealing with these situations is covered in other courses**
  - Operating Systems (we'll talk about "priority inversion" in a later lecture)
  - Real time databases

Bad Code involving Global Data

```
void IsrSpiTxComplete(void)
{
    SpiUpdateStats();
    InterruptDisable(SPI_TX_COMPLETE);
}

void SpiUpdateStats(void)
{
    // Update SPI stats in shared statistics structure
    MutexPend(SpiStatsMutex);
    SpiStats.TxOK++;
    MutexPost(SpiStatsMutex);
}
```

What if mutex is unavailable? (We're in an ISR!)

Example Concurrency Problem #1

- **Most robust embedded systems have an error log**
  - Keeps track of problems for engineering analysis with one log for system
  - Assume multiple tasks with a shared error log
  - Consider this error log code:

```
void MakeErrorLogEntry(uint8 problem_code; uint32 time)
{ error_log[error_ptr].code = problem_code;
  error_log[error_ptr].time = time;
  error_ptr += 1;
  if (error_ptr >= LOGSIZE)
  { error_ptr -= LOGSIZE; }
}
```

- **What are the potential problems with this code?**
  - What are ways to fix it?
  - This is really just a FIFO – why are there reentrancy problems?

Is There A Concurrency Problem With This Code? (#2)

- Assume `timer_ticks` is number of TCNT overflows recorded by ISR

```
struct PCB_struct
{ pt2Function Taskptr;   // pointer to task code
  uint8       Period;    // execute every kth time
  uint8       NextTime;  // next time this task should run
};
…  init PCB structures etc. …

for(;;)
  { for (i = 0; i < NTASKS; i++)
    { if (PCB[i].NextTime < timer_ticks)
      {PCB[i].NextTime += PCB[i].Period; // set next run time
       PCB[i].Taskptr();
       break;  // exit loop and start again at task 0
      }
    }
  }
```

Is There A Concurrency Problem With This Code? (#3)

- Assume `timer_ticks` is number of TCNT overflows recorded by ISR

```
struct PCB_struct
{ pt2Function Taskptr;   // pointer to task code
  uint32       Period;    // execute every kth time
  uint32       NextTime;  // next time this task should run
};
…  init PCB structures etc. …

for(;;)
  { for (i = 0; i < NTASKS; i++)
    { if (PCB[i].NextTime < timer_ticks)
      {PCB[i].NextTime += PCB[i].Period; // set next run time
       PCB[i].Taskptr();
       break;  // exit loop and start again at task 0
      }
    }
  }
```

What’s The Problem With The Skinny ISR Example? (#4)

```
volatile uint64 timer_val; // assume initialized to current time
uint8  seconds, minutes, hours;
uint16 days;

void main(void)
{ … initialization …
   for(;;)
   { update_tod();
     do_task1();
     do_task2();
   }
}

void update_tod()
{  seconds = (timer_val>>16)%60;
   minutes = ((timer_val>>16)/60)%60;
   hours =   ((timer_val>>16)/(60*60))%24;
   days =    (timer_val>>16)/(60*60*24);
}

void interrupt 16 timer_handler(void) // TOI
{ TFLG2 = 0x80;
  timer_val += 0x10C6;  // 16 bits fraction; 48 bits intgr
}   // blocking time of ISR no longer includes division operations!
```

Skinny ISR Fix

- **Version 1 –**
  not good enough for 32-bit+ size, but might work for 16 bits
  - Only works if transfer is atomic; risky solution (don't do this)

```
void update_tod()
{ timer_tmp = timer_val;    // timer_val could still change here!
  seconds = (timer_tmp>>16)%60;
  minutes = ((timer_tmp>>16)/60)%60;
  hours =   ((timer_tmp>>16)/(60*60))%24;
  days =    (timer_tmp>>16)/(60*60*24);
}
```

- **Version 2 – ought to work OK**

```
void update_tod()
{ DisableInterrupts();  // be careful to minimize blocking time!
  timer_tmp = timer_val;    // timer_val can't change now
  EnableInterrupts();
  seconds = (timer_tmp>>16)%60;
  minutes = ((timer_tmp>>16)/60)%60;
  hours =   ((timer_tmp>>16)/(60*60))%24;
  days =    (timer_tmp>>16)/(60*60*24);
}
```

Review

- **Buffer management**
  - Understand how Single buffer, double buffer, FIFO work
  - Study suggestion: write the code to manage the “head” pointer for FIFO and test everything out
- **Reentrant code**
  - Making sure code can be executed by multiple threads concurrently
  - Know rules for reentrant code; be able to spot a rule violation
- **Atomic Actions**
  - Making sure that an operation can’t be interrupted
  - Know how to make a piece of code atomic
- **Mutexes**
  - Mutex to implement mutual exclusion of critical regions
  - Know how to implement and use TestAndSet

Answers To Concurrency Problems #1

- If two threads try to make a log entry, they will write to same index error_ptr
- Context switch could happen right before +1, causing one valid entry and one blank entry
- One thread might increment error_ptr between other thread writing code and time, leading to mis-matched code/time pairs
- Both threads might get past the >= LOGSIZE check and both subtract LOGSIZE, giving an invalid pointer

Answers To Concurrency Problems #2 & #3

- **This one is tricky – timer_ticks could increment partway through the loop**
  - When timer_ticks increments, a higher priority task could become eligible for execution
  - BUT, if the value of "i" in the loop is greater than that high priority task, it will be ignored until some other task is selected for execution or all values of "i" have been tried.
- **Solutions:**
  - Check for timer_ticks incrementing and re-trigger loop each time
  - OR: just chalk it up to blocking time – because it isn't much longer than case where the lower priority task just started execution before timer_ticks incremented

Answers To Concurrency Problems #4

- **Timer_val could increment during execution of update_tod**