Back to RTOS

- Scheduling
  - Deadline
  - Laxity
  - Rate Monotonic
- Shared Code in Multiprocessing
- Share Resources: Deadlock avoidance
- Process Synchronization and Communication
- Memory Management
Dynamic Non-preemptive Scheduling

Char arrives

serial_isr signals serial task

Music task is never more than one OS time slice away

serial_isr

music_isr

serial

music

OS

music time slice...signal music task

os time slice

os time slice

os time slice

music time slice

deadline

CSE466 Autumn '00- 2
Dynamic Preemptive Scheduling

Pre-emptive: hi priority tasks preempt low priority task.
Advantage: Lower latency, faster response for high priority tasks.
Disadvantage: Potential to starve a low priority task
Tiny: no priority, round robin only. No starvation.
Priority Inversion: when T2 disables interrupts
Priority Function: any suggestions?
Scheduling Algorithms (Priority Functions)

- Egalitarian: Round Robin
  Problem: We may have unused compute resources even though we don’t meet some deadlines … it can be non-optimal.
  Example: system with music task and n non-critical tasks.
    § if deadline < time_tick * n + music_task then we have a chance to miss the deadline.
    § If music is higher priority than worst case is: time_tick + music_task

- Theory: for a system with N periodic tasks. The system is schedulable if:
  \[ \sum \frac{C_i}{P_i} \leq 1 \]
  where Ci is seconds or computation per event i and Pi is period of event i
  where \( \frac{C_i}{P_i} \) is the fraction of time spent dealing with i
  § Let Ci = .5 and Pi = 2 \( \frac{C_i}{P_i} = \frac{.5}{2} = .25 \) (1/4 of the time)

Rate monotonic scheduling: Priority \( \propto \) Frequency
At run time: Highest priority task always preempts lowest priority task.
Proven to be optimal by Liu and Layland.
Real systems rarely fit this model perfectly
Other Scheduling Algorithms (Priority Functions)

- Earliest Deadline First
  Keep list of runnable processes sorted by deadline
  Algorithm always runs the first item on the list

- Least Laxity
  Like earliest deadline first, but considers the expected run time of the task. Priority = Deadline – RunTime. Sort the list according to this criteria.
  Run the first item on the list

- Static Priority
  Various combinations
  - Static priority with least laxity as the tie breaker

- Engineering Challenge
  worst case analysis to satisfy yourself that your system will always meet your deadline given the scheduling policy
Reentrant functions...sharing code not data

- Are there shared functions that we would like to have?
  - deq? enq? next (same for head or tail)?
  - C Library Routines!!

- Can task switching clobber local variables (parameters and automatics)?
  - What happens when this function is interrupted by the OS?

```c
unsigned char next(unsigned char current, unsigned char size) {
    if (current+1 == size) return 0;
    else return (current+1);
}
```

- it depends on where the parameters, automatics, and spill values are stored... this one is probably okay!

3 places for parameters
a. Registers
b. fixed locations
c. stack...but not the hardware stack!
Implementation Example: Reentrant, Encapsulated Queue

typedef struct qstruct {
    unsigned char head;
    unsigned char tail;
    unsigned char *array;
    unsigned char size;
} fifo;

fifo Q;
unsigned char array[QSIZE];

void producer(void) _task_ 0 {
    unsigned char i;
    bit fail;
    initq(&Q, array, QSIZE);
    os_create_task(1);
    while (1) {
        do {
            disable();
            fail = enq(&Q,i);
            enable();
        } while (fail);
        i++; // simulated data
    }
}

void consumer(void) _task_ 1 {
    bit fail;
    unsigned char i;
    while (1) {
        os_wait();
        disable();
        fail = deq(&Q,&i);
        enable();
        if (fail)...else use(I);
    }
}

Shared functions are okay if we disallow task switch during calls.
why? re-entrant stack not protected by Tiny OS.
What about C-libraries (subroutine calls?)

is this okay for timing if we don’t use it in Tone Gen ISR (overhead)?
Examples of Reentrant functions

int sum(tree) {
    if (!tree) return 0;
    return sum(tree->left) + sum(tree->right) + tree->val;
}

reason for reentrancy: re-use code
The key to reentrancy: relative addressing

Other examples of reentrancy:
two tasks share a function, ISR and task share a function
Reentrancy in Keil C51

- In C51, most parameter passing is done through registers (up to three parameters). Then fixed memory locations are used. Register method is reentrant, the other isn’t.

- Local (automatic) variables and temporary values in functions are also mapped to fixed memory locations (w/ overlaying)...definitely not reentrant.

- How can we solve this: declare functions to be reentrant as in:
  ```c
  unsigned char next(unsigned char current, unsigned char size) reentrant {
      if (current+1 == size) return 0;
      else return (current+1);
  }
  ```

- BUT...the stack used for reentrant functions is NOT the same as the hardware stack used for return address, and ISR/TASK context switching. There is a separate “reentrant” stack used for that, which is not protected by the TINY OS. It’s a different region of memory, and a fixed memory location is used for the reentrant stack pointer. So this works for FULL and for recursion (no OS).

- Conclusion...you can have shared functions in TINY if you:
  - convience yourself that all parameters are passed through registers
  - convince yourself are no local variables that use fixed memory locations (compiler can allocate those to registers too)
  - be sure not to change HW settings non-atomically
  - or... you disable context switching in shared functions by disabling T0 interrupts

  Think of shared functions as critical sections. Consider impact on interrupt latency?
Sharing Resources

What would be an example of a shared resource in a simple 8051-like application (other than RAM variables)?

What if you have 64 control lines to manage, with no memory mapped I/O?

can Tiny OS help with this?
can memory mapped I/O help with this?
How about these?

- Is this reentrant? ... note: we don’t care about order
  
  ```
  void setLatch(addr, data) {port1 = data, port2 = addr, E = 1, E = 0}
  ```

<table>
<thead>
<tr>
<th>Thread 1 (x,y)</th>
<th>Thread 2 (a,b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Port1 = x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Port1 = a</td>
</tr>
<tr>
<td></td>
<td>Port2 = b</td>
</tr>
<tr>
<td>Port2 = y</td>
<td></td>
</tr>
</tbody>
</table>

- How can we get atomicity here?
- Deadlock scenario:
  1. thread-1 requests and gets port1
  2. thread-2 requests and gets port2 then requests port1
  3. thread-1 requests port2

- Can TinyOS help with this?
Deadlock

- Preemptable v. Nonpreemptable Resources
  - CPU – preemptable
  - Memory – Preemtable
  - Incoming packet processing on a network interface – non preemptable
  - Control of an external device like a disk drive, printer, display

- Deadlock: two threads each have a partial set of non-preemptable resources needed to complete their tasks and are waiting for the resources held by the other.

- Preconditions for Deadlock to occur
  - Mutual exclusion: each resource is either currently assigned to exactly one thread or is available
  - Hold and wait: A process currently holding resources granted earlier can request a new resource without giving up the other
  - Non-preemptive: Only the holder a resource can give it up
  - Circular Wait: see above
Solutions to Deadlock

q Forggedaboutit
Statisticallikelihood say once if fifty years. Statistical likelihood of a disk crash is once in 10…so worry about the disk. Also consider the consequences of its occurrence!

q Detection and Recovery
Look for cycles in the request chain, then break the chain if it happens. Make sure that in any chain, there is always a thread that can tolerate being killed!

q Prevention
Prevent at least one of the four preconditions from occurring!
- Mutual exclusion: spool I/O so that thread can continue
- Hold and wait: Request at resources at once.
- Circular wait: order resources numerically and allow a process to wait only on resources numerically higher than all that it currently holds.
Safety – flip side of deadlock

q No two threads access a non-sharable resource at the same time (critical section protection)

q Access to a resource
   Request Resource
   Use Resource
   Release Resource

q Safety is guaranteed if Request Resource is atomic. Just disable interrupts!
   But we don’t want user tasks to do that.
   Causes reduced predictability of system performance (latency).
   A user thread could crash with interrupts disabled

q How to protect a critical section without disabling interrupts
Coming Up

- A little more on OS
  - Real Time Scheduling Algorithms
  - Synchronization: Semaphores and Deadlock avoidance
  - Interprocess Communication
  - Concept of shared resources: Devices and Drivers

- Future
  - Linux and the Cerfboards
  - Networking
  - Product Safety
  - Java/Object Oriented Programming for Embedded Systems

- Design Meeting (Product Ideas…)