Lecture #16

Cooperative & Preemptive Context Switching

18-348 Embedded System Engineering

Philip Koopman

Wednesday, 16-Mar-2016

Electrical & Computer ENGINEERING

Carnegie Mellon

Google Car

http://jalopnik.com/5851324/how-googles-self+driving-car-works

Where Are We Now?

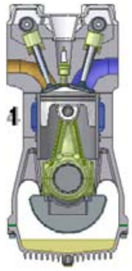
- **Where we've been:**
  - Interrupts
  - Response time calculations (non-preemptive)
- **Where we're going today:**
  - Things that look more like schedulers (cooperative)
  - Preemptive multi-tasking (overview and math, but not gory details)
- **Where we're going next:**
  - Concurrency, scheduling, interrupt practicalities
  - Analog and other I/O
  - Test #2
  - Final project

Preview

- **Tasking & context**
  - What's a task?
  - What does context switching involve
- **Cooperative tasking**
  - Non-preemptive approach
  - Tasks run to completion, then next task started
  - Can be round-robin or prioritized
- **Preemptive tasking**
  - Tasks pre-empted by other tasks

Why Does Latency/Real Time Matter?

- **Car engine spark plug timing**
  - http://upload.wikimedia.org/wikipedia/commons/a/a6/4-Stroke-Engine.gif
  - Poor timing results in:
    - Backfire
    - Loss of fuel economy



- **Control loop stability**
  - **December 11, 2000:** Hydraulic leak cripples an Osprey engine; a bug in the pilot's control software makes things worse. Four killed.

http://www.wired.com/wired/archive/13.07/osprey.html?pg=5

What's A Task?

For our purposes (in small embedded microcontrollers)…

- **A Task is an active flow of control within a computation**
  - Associated with a program counter value and set of register values
  - Multiple tasks can be active concurrently
  - Tasks are not supposed to interfere with other task's register values
  - Tasks share memory space and, in particular, can interfere with memory values

- **Bigger-system terminology:**
  - "thread" – generally a sequence of instructions being executed that shares a memory space with other threads
    - Sometimes called a "lightweight" process
  - "process" – generally a sequence of instructions being executed that has a different memory space from other processes
  - In small micros, a "task" is usually a "thread" because there is only one memory space
    - But our discussion on context switching mostly applies to processes as well

Preemptive vs. Non-Preemptive Tasks

- **Non-preemptive task**
  - Task runs to completion, even if other tasks want service
  - Example: the ISR routines from most of previous lecture
- **Preemptive task**
  - Task is suspended if some other task wants service, then restarted at a later time
  - Example: main program loop from previous lecture (preempted by ISRs)
- **Most real systems are a combination of both**
  - ISRs pre-empt non-ISR tasks, but are not preemptable themselves
    - Except if ISR for some reason clears I mask (unadvisable)
    - Except for non-maskable interrupt (usually don't worry about this case for timing)
  - Main tasks are preemptable
    - Except if main task sets I mask (should only do for short time)

"Tasking" As A Generalization Beyond ISRs

- **There is a pool of tasks**
  - Each task m can be running or suspended
  - There is a place in memory to save state for preemptable tasks m (the PCB)
    - PCB = "Process Control Block"
    - Really it's a task instead of a "process" but we'll use PCB for consistency with OS descriptions
    - Task state is: PC, flags, all registers
- **Tasks can be executed in various orders**
  - Depending on preemption policies and mechanisms
  - Depending on execution state
  - Depending on scheduling algorithm (more on this in later lectures)
- **Tricky part – can have multiple concurrent tasking approaches**
  - We're not going to go nuts with this, but it is reality
  - Example: cyclic executive main loop plus prioritized ISRs
  - Example: pre-emptive prioritized tasks plus non-preemptive ISRs
  - So let's look at "pure" versions to get an idea of the alternatives

Purely Non-Preemptive Tasking

- **Sometimes called “cooperative” tasking**
  - Cyclic executive is a degenerate case
  - In general, want a pool of tasks that are executed by a scheduler
  - Only one task runs at a time
  - No preemption, so no need for executive to save state:
    1. Task saves its own state before yielding control to scheduler once in a while ***OR***
    2. Task runs to completion

- **Starting point – cyclic executive:**

```
// main program loop
for(;;)
{   poll_uart();
    do_task1();
    do_task2();
}
```

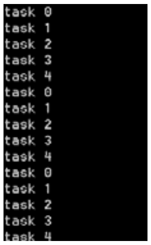
More Flexible Cooperative Task Approach

```
#include <stdio.h>            // compiles with gpp on desktop machine
void task0(void) { printf("task 0\n");}
void task1(void) { printf("task 1\n");}
void task2(void) { printf("task 2\n");}
void task3(void) { printf("task 3\n");}
void task4(void) { printf("task 4\n");}

// NTASKS is number of defined tasks
#define NTASKS 5
typedef void (*pt2Function)(void);  // pointer to function
pt2Function PCB[NTASKS];      // stores pointer to START of each task

int main(void) {
  int i;
  PCB[0] = &task0;
  PCB[1] = &task1;
  PCB[2] = &task2;
  PCB[3] = &task3;
  PCB[4] = &task4;

  // cooperative multitasking executive -- round robin
  for(;;) {
   for (i = 0; i < NTASKS; i++)
   { PCB[i](); }  // execute task i
   // optionally – could wait here so loop runs once every N msec
  }
return(0);
}
```



```
task 0
task 1
task 2
task 3
task 4
task 0
task 1
task 2
task 3
task 4
task 0
task 1
task 2
task 3
task 4
```

Round Robin Cooperative Tasking Latency

- **"Round Robin" = everyone takes one turn in some defined order**
- **For Round Robin, response time for N tasks is:**
  - (assume task reads inputs at start, processes, does outputs at end)
  - The task has just started, and will need to run again for new inputs
  - All other tasks will need to run
  - This same task will need to run again
  - Same time for all tasks … "i" doesn't appear on right hand side of equation

$$R_i = \sum_{j=0}^{j=N-1} \left(C_j \right)$$

- **Why assume task has just started?**
  - Assume first instruction of task samples input data
  - Task is not synchronized to inputs, so even a very infrequent task might just miss inputs

Multi-Rate Round Robin Approach

- **Simple brute force version**
  - Put some tasks multiple times in single round-robin list
  - But gets tedious with wide range in rates

- **More flexible version**
  - For each PCB keep:
    - Pointer to task to be executed
    - Period (number of times main loop is executed for each time task is executed) i.e., execute this task every k*th* time through main loop.
    - Current count – counts down from Period to zero, when zero execute task

```
typedef void (*pt2Function)(void);

struct PCB_struct
{ pt2Function Taskptr;   // pointer to task code
  uint8       Period;    // execute every kth time
  uint8       TimeLeft;  // starts at k, counts down
  uint8       ReadyToRun; // flag used later
};
PCB_struct PCB[NTASKS];  // array of PCBs
```

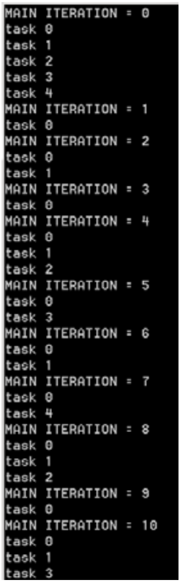
Multi-Rate Round Robin Cooperative Tasking

```
int main(void)
{ int i;
  // init PCB values
  PCB[0].Taskptr = &task0;   PCB[0].Period = 1;  PCB[0].TimeLeft = 1;
  PCB[1].Taskptr = &task1;   PCB[1].Period = 2;  PCB[1].TimeLeft = 1;
  PCB[2].Taskptr = &task2;   PCB[2].Period = 4;  PCB[2].TimeLeft = 1;
  PCB[3].Taskptr = &task3;   PCB[3].Period = 5;  PCB[3].TimeLeft = 1;
  PCB[4].Taskptr = &task4;   PCB[4].Period = 7;  PCB[4].TimeLeft = 1;

 // cooperative multitasking executive -- multirate
  for(int j=0; j < 100 ; j++) // normally an infinite loop
  { printf("MAIN ITERATION = %d\n",j);
    for (i = 0; i < NTASKS; i++)
    { if (--PCB[i].TimeLeft == 0) // decrement and check time
        // NOTE: “time” is in # iterations, not wall time
      { PCB[i].Taskptr();  // TimeLeft is zero – execute task
        PCB[i].TimeLeft = PCB[i].Period;  // re-init TimeLeft
      }
    }
  }
return(0);
}
```

Multi-Rate Cooperative Execution Sequence

- All tasks set ready to run at time 0
  - `PCB[0].Period = 1;`
    - Runs every iteration
  - `PCB[1].Period = 2;`
    - Runs iterations 0, 2, 4, 6, …
  - `PCB[2].Period = 4;`
    - Runs iterations 0, 4, 8, …
  - `PCB[3].Period = 5;`
    - Runs iterations 0, 5, 10, …
  - `PCB[4].Period = 7;`
    - Runs iterations 0, 7, 14, …
- Note that this is by iteration number rather than time
  - (Would doing it by time make any difference?)



```
MAIN ITERATION = 0
task 0
task 1
task 2
task 3
task 4
MAIN ITERATION = 1
task 0
MAIN ITERATION = 2
task 0
task 1
MAIN ITERATION = 3
task 0
MAIN ITERATION = 4
task 0
task 1
task 2
MAIN ITERATION = 5
task 0
task 3
MAIN ITERATION = 6
task 0
task 1
MAIN ITERATION = 7
task 0
task 4
MAIN ITERATION = 8
task 0
task 1
task 2
MAIN ITERATION = 9
task 0
MAIN ITERATION = 10
task 0
task 1
task 3
```

Time-Based Prioritized Cooperative Tasking

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

```
struct PCB_struct
{ pt2Function Taskptr;   // pointer to task code
  uint8       Period;    // Time between runs
  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
              // note – NOT  timer_ticks + Period  !!
       PCB[i].Taskptr();
       break;  // exit loop and start again at task 0
      }
    }
  }
```

Prioritized Cooperative Tasking

- **Idea – tasks have priority**
  - Tasks don't run all the time – only periodically
    - Especially useful for multiple periods and less than 100% CPU load
    - Or when task runs because some flag is set (e.g. "input is ready, run the task")
  - Any task, once running, runs to completion
  - When system is idle, nothing runs (except loop looking for next task to run)
- **Assume something sets the "ready to run" flag (e.g., a timer ISR):**
  - Executive loop only runs highest priority task, *not* all tasks
  - Main loop becomes:

```
for(;;)
 { for (i = 0; i < NTASKS; i++)
   { if (PCB[i].ReadyToRun)
     {PCB[i].ReadyToRun = 0; // OK, it's going to run
      PCB[i].Taskptr();
      break;  // exit loop and start again at task 0
       // what happens if "break" is omitted?
     }
   }
 }
```

Prioritized Cooperative Tasking Latency

- <u>This is the SAME scheduling math AS ISR execution!</u>
  - **For an ISR, the hardware interrupt request sets “ready to run”**
  - **For non-ISR, a timer or something else sets “ready to run”**
- **For prioritized approach, response time for N tasks is:**
  - Some lower priority task executes
  - Higher priority tasks need to execute one or more times
  - This is the same math as for interrupts …
    … because <u>interrupts are non-preemptive prioritized tasks</u>

$$R_{i,0} = \max\left[\max_{i<j<N}\left(C_j\right), B\right] \qquad ; i < N - 1$$

$$R_{i,k+1} = R_{i,0} + \sum_{m=0}^{m=i-1}\left(\left\lfloor \frac{R_{i,k}}{P_m} + 1 \right\rfloor C_m\right) \qquad ; i > 0$$

When Is It Ready To Run?

- **One way – some external mechanism**
  - For example, a “fat ISR” for serial input might be broken into two parts:
    1. A skinny ISR that grabs a data byte and sets part #2 to “ready to run”
    2. A processing routine that runs when it can; ISR part #1 sets its “ready to run” flag
  - For example, a timer interrupt
    1. An actual ISR that increments a tick counter and sets “ready to run” for part 2
    2. A part 2 routine that converts tick counter into seconds/minutes/hours/days
- **Another way – executive keeps track of elapsed time**
  - Sets “ready to run” when enough time has elapsed
    - – Periodically check time and set ready to run when appropriate

Task Scheduling Policies

- **Scheduler needs to have a task selection policy – answer to question “which task runs next?” varies:**
  - Round-Robin
  - Earliest Deadline
  - Highest Priority

  - More on this in a later lecture on scheduling
    - For now just assume scheduler knows which task goes next

- **Latency depends on scheduling policy!**

A Step Back To The Big Picture

- **In the last lecture we said “don’t do anything fancy in the main loop”**
  - Does all this stuff look like something fancy?

- **You should only use these techniques if you can compute latency**
  - You need to know fastest possible period of “ready to run”
  - You need math that expresses your approach
    - And we’ve given you the math
  - Make sure you understand the worst case when every task wants to run at once

- **It gets tricky to write a tasker/executive/scheduler** (pick your favorite term)
  - You can probably pull it off for non-preemptive tasking, but be careful
  - If you have a choice, use a proven tasker, not one you make yourself
    - Be skeptical of taskers from authors without deep knowledge of this topic
    - Be only slightly less skeptical of home-brew taskers from experts
  - If you can’t express latency with actual math (that you understand!) then don’t use the approach

Why Preemptive Tasks?

- **Preemptive task**
  - Executing task is suspended if some other task wants service
  - Suspended task is restarted at a later time
  - Example: an ISR preempts the main program loop

- **Without preemption, latency is bounded by longest single task execution**
  - Might even be lowest priority task that keeps everything else blocked

- **With pre-emption**
  - Longest single lower priority task is no longer a bound on responsiveness
  - Lets you get better latency – gets rid of that pesky max(C_j) term in response time

…but…

- Means you have to be very careful saving state of interrupted computation
- Makes it much easier to get concurrency problems (next lecture)
- Still leaves main loop blocking time *B* as a bound on responsiveness

General Idea Of Preemption

- **Interrupts are the simplest types of preemption**
  - Save main program state on stack
  - Execute interrupt
  - Restore main program state from stack
  - Return to main program

- **General preemptive scheduling approach**
  - N tasks are running concurrently
  - Have a place to store N copies of program state
    - NOT on the stack – that limits order of execution
  - Have a scheduler that saves and restores state:
    - Stop task K
    - Save state in the “task K save area”
    - Restore state from the “task J save area”
    - Restart task J from wherever it left off
  - Simplest approach is to have K stacks – one per task – and save states there

How Do You Know When To Preempt?

- **Hardware approach**
  - Hardware determines something of higher priority needs to execute
  - Works OK for ISRs preempting ordinary code
  - But, tricky for ISRs preempting other ISRs!
- **Software approach**
  - Software sees which task is highest priority, and starts it running
  - But, what lets the software to do the checking run?
  - Simple approach – run the executive on a periodic timer
    - Augment by having a “ready to run” setting event also call the executive
  - But, here’s the rub – you need to interrupt tasks, not run to completion
- **You need a context swap to make this work**
  - “Context” = all data required to define the state of the current task
  - Simplest example: all register values
  - Might include other things in more complex computing environments (for example, file I/O status)

Example Context Swap

- **Save everything that matters from old task**
  - old PC
  - old D
  - old X
  - old Y
  - old Condition Codes
  - old SP
- **Restore everything that matters before restarting new task**
  - new PC
  - new D
  - new X
  - new Y
  - new Condition Codes
  - new SP
- **BUT – everything (except SP) gets put there automatically when processing the interrupt that triggered the task switch!**

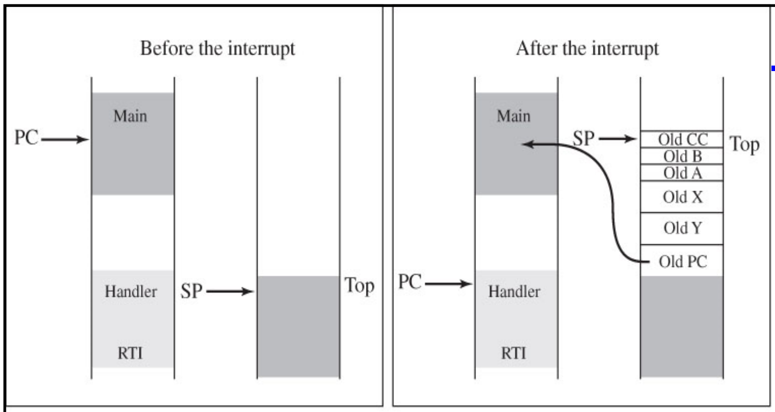


Figure 4.19
6812 stack before and after an interrupt.

Table 7-2. Stacking Order on Entry to Interrupts

Memory Location	CPU Registers
SP + 7	$RTN_H : RTN_L$
SP + 5	$Y_H : Y_L$
SP + 3	$X_H : X_L$
SP + 1	B : A
SP	CCR

RTI — Return from Interrupt — RTI

Operation:

$(M_{(SP)}) \Rightarrow CCR;\ (SP) + \$0001 \Rightarrow SP$
$(M_{(SP)} : M_{(SP+1)}) \Rightarrow B : A;\ (SP) + \$0002 \Rightarrow SP$
$(M_{(SP)} : M_{(SP+1)}) \Rightarrow X_H : X_L;\ (SP) + \$0004 \Rightarrow SP$
$(M_{(SP)} : M_{(SP+1)}) \Rightarrow PC_H : PC_L;\ (SP) - \$0002 \Rightarrow SP$
$(M_{(SP)} : M_{(SP+1)}) \Rightarrow Y_H : Y_L;\ (SP) + \$0004 \Rightarrow SP$

Description:

Restores system context after interrupt service processing is completed. The condition codes, accumulators B and A, index register X, the PC, and index register Y are restored to a state pulled from the stack. The X mask bit may be cleared as a result of an RTI instruction, but cannot be set if it was cleared prior to execution of the RTI instruction.

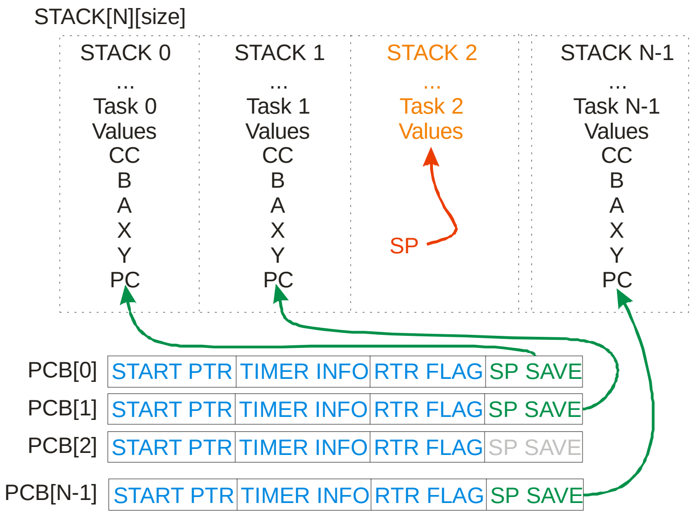
If another interrupt is pending when RTI has finished restoring registers from the stack, the SP is adjusted to preserve stack content, and the new vector is fetched. This operation is functionally identical to the same operation in the M68HC11, where registers actually are re-stacked, but is faster.

CCR Details:

S	X	H	I	N	Z	V	C
Δ	⇓	Δ	Δ	Δ	Δ	Δ	Δ

General Multi-Tasking State Layout

- **Picture shows Task 2 currently running**
  - All other tasks are inactive, with state saved where shown
  - SP is the CPU stack pointer, currently pointing to top of Task 2 stack area



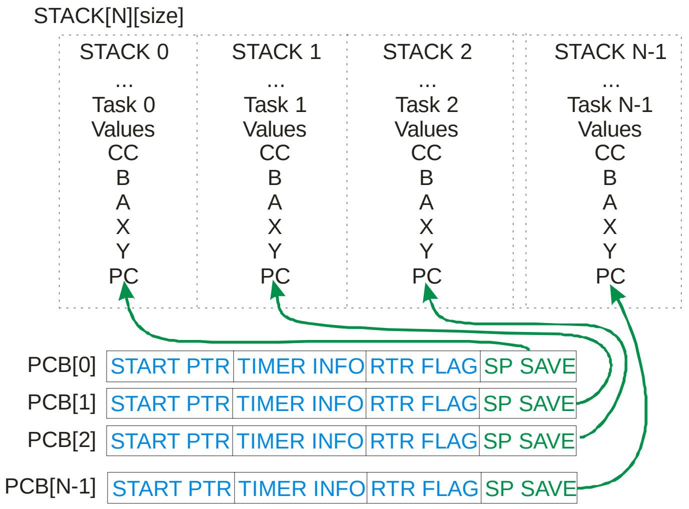
STACK[N][size]

STACK 0	STACK 1	STACK 2	STACK N-1
...	...	...	...
Task 0	Task 1	Task 2	Task N-1
Values	Values	Values	Values
CC	CC		CC
B	B		B
A	A		A
X	X	SP	X
Y	Y		Y
PC	PC		PC

PCB[0]	START PTR	TIMER INFO	RTR FLAG	SP SAVE
PCB[1]	START PTR	TIMER INFO	RTR FLAG	SP SAVE
PCB[2]	START PTR	TIMER INFO	RTR FLAG	SP SAVE
PCB[N-1]	START PTR	TIMER INFO	RTR FLAG	SP SAVE

State While Switching Tasks

- **Timer ISR has pushed Task 2 state onto Stack 2**
  - ISR saves the SP value into PCB[2].SPSAVE
  - To start task 1, just put PCB[1].SPSAVE into SP and do an RTI



STACK[N][size]

STACK 0	STACK 1	STACK 2	STACK N-1
...	...	...	...
Task 0	Task 1	Task 2	Task N-1
Values	Values	Values	Values
CC	CC	CC	CC
B	B	B	B
A	A	A	A
X	X	X	X
Y	Y	Y	Y
PC	PC	PC	PC

PCB[0]	START PTR	TIMER INFO	RTR FLAG	SP SAVE
PCB[1]	START PTR	TIMER INFO	RTR FLAG	SP SAVE
PCB[2]	START PTR	TIMER INFO	RTR FLAG	SP SAVE
PCB[N-1]	START PTR	TIMER INFO	RTR FLAG	SP SAVE

Sketch of Preemptive Context Switching

- **Use timer interrupt to do task switching**
  - Assume that no other ISRs are active
    - (This is a poor assumption in general, but it simplifies this discussion)
  - Assume every task has a separate stack space

- **Each TCNT rollover switches to next round-robin task**
  - Some details when task starts or terminates; not our concern here

- **ISR outline for round-robin preemption**
  - Acknowledge TOF interrupt (of course)
  - Look at static “current_task” variable to figure out which task is running
  - Save SP into PCB[current_task]
  - Set new current_task based on scheduling policy
  - Get new SP from PCB[current_task]
  - Execute RTI – restarts processor with new task where it left off

- **ISR outline for prioritized preemption**
  - As above, except pick highest priority active task, not current_task+1
  - Incorporate timer logic as for cooperative tasking case

What’s Response Time For Preemptive Tasks?

- **Make some assumptions:**
  - Task i never tries to execute again until previous execution runs to completion
  - Ignore overhead of timer ISR that does context swapping

$$R_{0,0} = 0$$

$$-----------------$$

$$R_{i,0} = C_o \qquad ; \mathit{for}\ i > 0$$

$$R_{i,k+1} = \sum_{j=0}^{j=i-1} \left(\left\lfloor \frac{R_{i,k}}{P_j} + 1 \right\rfloor C_j \right)$$

- Highest priority task has zero response time (as long as there is no blocking)
- Start the recursion with task 0, which could always execute first
- Similar to cooperative, but *higher priority tasks get to preempt task i*…
  …. so completion time may include additional preemptions!

What About Real Systems?

- **Math for real systems gets hairy**
  - Might have combination of round-robin and prioritized regular tasks
    - Generally preemptable
    - But, often use SEI/CLI in main task and thus have blocking time
  - Also have ISRs
    - Generally not preemptable
    - Each ISR causes blocking of higher priority ISRs while it is running
  - So this means the math has to include:
    - Some preemptable tasks
    - Some non-preemptable tasks
    - Account for timer tick that switches tasks
  - We’re not going to attempt it here… but realize that it isn’t as simple as just applying a single formula!

How Does This Relate To An RTOS?

- **RTOS = Real Time Operating System**
  - More discussion of these later, but, a key part is the scheduler
- **RTOS scheduling**
  - Provides a pre-built framework to support a scheduling policy
    - Handles task control blocks, task switching, etc.
    - Hopefully has been well designed, validated, tested so it really works
  - Usually we mean preemptive when we say “RTOS”
    - But cooperative or other models can also be a valid RTOS
  - Usually supports one or more prioritization schemes
    - More on this in later lectures
- **A key RTOS design parameter is blocking time B**
  - What’s the maximum amount of time for a task switch?
  - If you use a timer tick, depends on size of timer tick
  - Regardless of what triggers the tasker, worst case time interrupts are masked adds to blocking time

Should You Build Your Own Scheduler?

- **Or, put another way, should you build your own RTOS?**
  - Short answer: probably not.
- **It is *really difficult* to get an RTOS to be correct!**
  - Especially if you are hired to build an embedded gizmo, not write an OS!
  - Best bet – get one that someone else has put a lot of effort into
    - More on this in a later lecture
- **Even if you are an expert, and you do write your own…**
  - What happens when you leave the company?
    - Win the lottery; Get a better job; Get run over by a beer truck; … whatever
  - What happens when there is a bug found?
    - And you’re asleep
    - And you’re on vacation
    - And you’ve left the company (see above)
- **“Make/buy” decision advice:**
  - Real time operating systems should be bought, not made
  - Unless you are in a company that sells an RTOS as a product
  - In management speak, only build an RTOS if that is a critical core competency
  - If you are an embedded system company, RTOS is probably not a core competency (unless you actually sell an RTOS!)

End of Semester Project Preview

- **Get an idea for a project**
  - Soon we’ll ask for an idea sketch
- **Do something SIMPLE (two weeks worth of lab work) that incorporates:**
  - Three or more of: {D/A, A/D, PWM, SCI, CAN, SPI, counter/timer, fixed point multiply, mutex, priority inversion solution}
  - Use one or more interrupts; plus use COP properly; well defined scheduling
  - Worst case timing analysis; statechart design; test plan
  - Use parts in lab or inexpensive parts (<$5 total) we can get via Tech (requires our approval; more might be OK but avoid big-budget projects)
  - Something substantially different than an existing lab project (but can re-use pieces of a lab such as a timer service ISR)
  - *Killer* complexity is not desired
    - Be sure you have a safe, easier demo in case the project implodes
  - *Clever* is nice
  - *Cool* is nice
  - *On time is more important* than cool or clever
  - ***Actually works* is mandatory; so is *well documented***
  - (Note that this list lets you do an RTOS project if you want:
    - e.g., counter/timer, mutex, priority inheritance)

Review

- **Tasking & context**
  - What’s a task?
  - What does context switching involve?

- **Cooperative tasking**
  - Non-preemptive approach
    - What’s the response equation?
  - Tasks run to completion, then next task started
  - Can be round-robin or prioritized
    - What are the response equations?

- **Preemptive tasking**
  - Tasks pre-empted by other tasks
    - What’s the response equation? What does it mean? Why is it different than cooperative response equation?