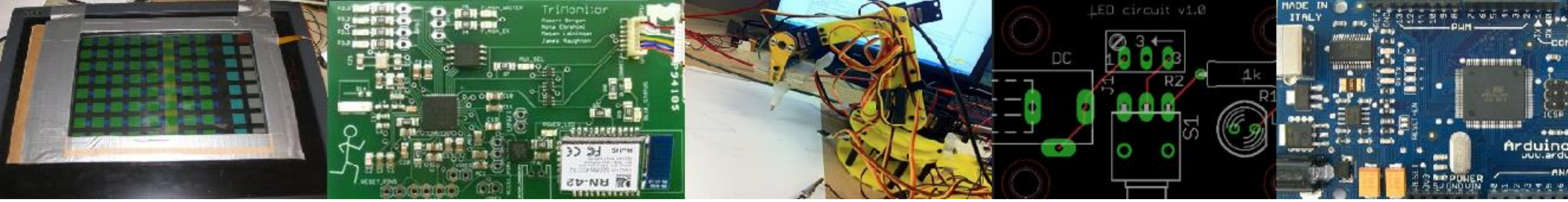


Projects (Finish up from last time)

What's reasonable

What's expected when

What is a reasonable project?

- **Required**
  - Entire embedded system with a realistic use
    - Design allows for that realistic use (not an unusable prototype)
  - Doable in the semester
  - Design and population of a PCB with a processor (or maybe FPGA).
  - Design and implementation of hardware interfaces
- **Suggested**
  - Not "barebones" programming
    - Use RTOS, Linux, or something else
    - Barebones might make sense if software fairly trivial
  - Stay away from switching power supplies

Projects in the past: Baby Monitor

- What it does
  - Watches for SIDS
  - Sets alarm on board and wirelessly to base station if problem detected.
- Main issues
  - Easy to use
    - Easy to place on baby correctly, easy to recharge, no complex interface
  - Resilient
    - Water-resistant, can be dropped, easily cleaned.
  - Cheap
    - Very low-cost target.
  - Testing that it works
    - Need baby?

Projects in the past: Triathlon monitor

- What it does:
  - Monitor location, heart rate, and strides
  - Display on watch, record to dump to PC later
- Issues
  - Weight
  - Waterproof
  - Getting data in (GPS, heart rate, strides)
  - Getting data out (to watch, stored and to PC)

Projects in the past:
Bike helmet (Hail-met)

- What it does:
  - Bluetooth
    - Voice, etc. Can make calls or listen to music
  - Turn signals and tail light
  - Basic heads-up display (signals, call info)
  - Solar recharge
- Issues
  - Weight
  - Waterproof
  - Safety

Hail-Met!

An all-in-one multi-purpose helmet for the modern commuter.

Intro

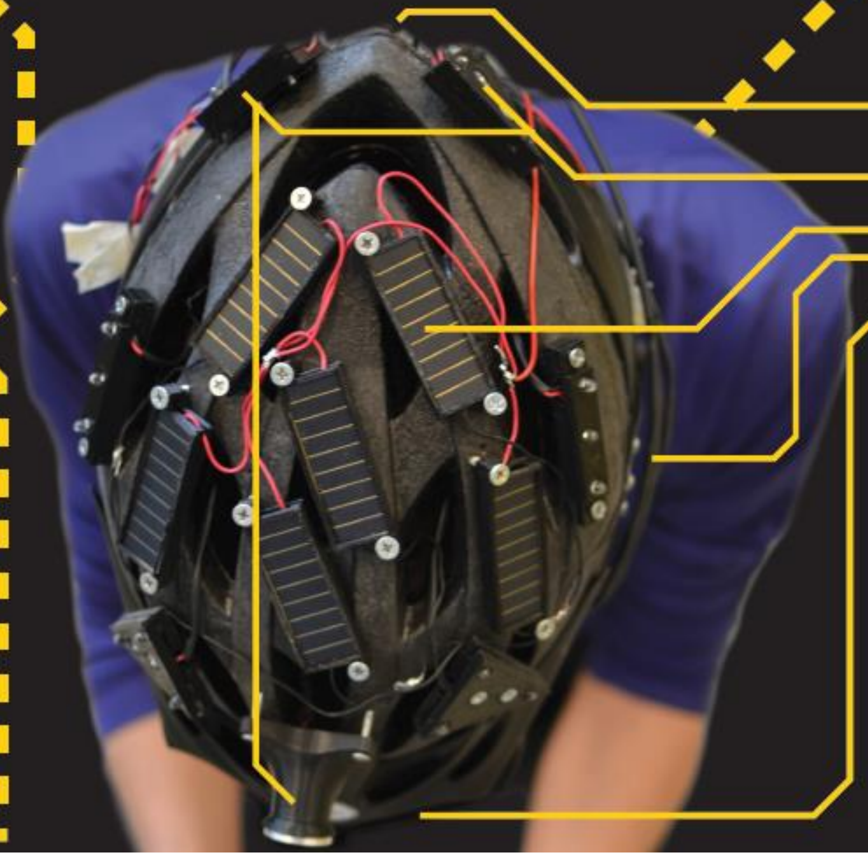
Besides protecting your head from direct contact with hard surfaces, traditional bike helmets don't offer many benefits to consumers. Advanced communication, entertainment, and night-time safety features come standard with every new car, but don't yet exist for cyclists.

Problem

Advanced communication, entertainment, and night-time safety features come standard with every new car, but don't yet exist for cyclists.

More modular solutions (like stand-alone detachable bike head lights) do exist but are easily stolen if they are not detached when the bike is in public.

Bicycle helmets are exposed to the elements (moisture, dust, etc.), so embedding electronic devices into a helmet is no easy feat.



Hardware

- PCB with Bluetooth, LiPo Battery, Charging, and LED Driver
- Turn Signals, Headlight and Taillight
- Solar panels
- Speakers
- Heads up display (HUD)

Custom PCB

Interfacing & Software Design

The Hail-met runs on an

Project dates next 30 days

- Sept. 6th:
  - Project ideas pdf *turned in* by **11pm** (on Gradescope already)
- Sept 7th:
  - All ideas posted, hopefully by noon.
- Sept. 8thth:
  - Forming groups 6:30-8:30pm **(IOE 1610)**
  - Each person will have a chance to speak for about 30 seconds about what they'd like to do
  - Wander around and form groups.
- Sept. 16th:
  - Draft proposal due (3% of project grade)
- Sept 22nd
  - Required meeting with staff to discuss proposal*
    - No class that day.
  - After meeting most groups can (and should) start ordering parts!
- Sept 29th
  - Formal proposal due. (12% of project grade).

*You should have met with various 473 staff members well before this!

Milestones: What are they? What's due?

- Report to us about how things are going and some demonstrations.
- What's due:
  - A short report (1-2 pages)
    - Due a few days before the meeting
  - Demonstrate to GSIs that you've meet your "objectively demonstrable" deliverables.
    - Needs to be done the day before the meeting at the latest.
- In general, these should take 30-60 minutes to write unless things are in really bad shape.
  - There is a docx template you are to use.

Milestone 1

(Can start ordering parts on Sept 22nd after meeting)

- Oct 13th
  - Device interfaces designed and working
    - Might need some refinement, but in good shape
  - Prototype largely working
    - Probably devboard (maybe breadboard, but avoid if possible!)
    - Can talk to all devices via interfaces
    - Can more-or-less do the task
  - Have identified solutions to each major issue.
- Started on PCB
  - Perhaps just barely, but at least have patterns done

Milestone 1 Example: Baby

- Have interface to GPS, flash (for storage), and variable resistor (for checking breathing) written useable.
- Not yet sure on chest cavity movement of infant.
  - Flash still a bit buggy—erasing not always working.
    - Otherwise demonstrated
  - GPS association working, (time is 3 minutes, but worst case probably acceptable for application)
    - Demonstrated

Milestone 1 Example: Baby

- Prototype on Arduino
  - Each interface works
  - When done together GPS (serial) interrupt messing with timer interrupt for resistor monitoring

PCB deadline

- All PCBs must be ordered by Nov 1st
  - **Worst case**, you need to respin the PCB later but we want them all out by this date.
    - Depending on budget for PCB, should have back in 7-10 days.
  - Should target earlier—hopefully most groups have PCB out by Oct 22nd

Milestone 2

- November 15th
  - PCB back and assembled, still may be debugging
  - Prototype working though more testing planned
  - Final plans for all “extras”
    - Enclosure (if needed)
    - Special documentation
    - ?????

November 29th

- Everything should be done, just final testing and debug of a largely working system.
- Start on design expo things and final report by 12/2.
  - Setup, poster, etc.
  - Discussion in class about posters etc. on 12/1
  - Get much of the report written before the expo.

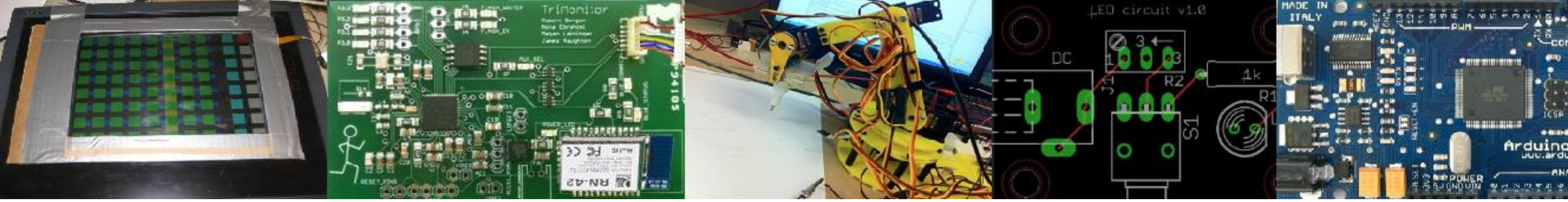
- Testing, testing, testing

Design Expo

- December 8th
  - Working and clearly presented demo
  - Hopefully exciting
    - Not all projects will be and that's okay.
  - Shirt and tie or equivalent required.

December 11th

- Final report due.
  - This report and the associated documentation is 15% of of your project grade.
  - With the proposal document done, you have a solid starting point.



EECS 473
Advanced Embedded Systems

Lecture 3 and 4

An introduction to real time systems and scheduling

Chunks adapted from work by
Dr. Fred Kuhns of Washington University
and Farhan Hormasji

Announcements

- HW0
  - Due today at 11pm on gradescope
    - I'll have the pdf posted of everyone's submissions on by Wednesday.

Group formation: Thursday night

- 1500 EECS from 6:30-8:30.
  - Yes, that overlaps with lab for some of you.
    - Sorry about that, but it’s really the only time that works.
    - We expect to have some extra hours Wednesday morning. And we’ll be there late on Thursday.
    - Please work with your lab partner to find a time that works.

Misc.

- Lab 1 inlab/postlab due before your lab starts.
- Lab 2 prelab due before your lab starts
  - Really important in this case

Outline

- Overview of real-time systems
  - Basics
  - Scheduling algorithms (RM, EDF, LLF & RR)
- Overview of RTOSes
  - Goals of an RTOS
  - Features you might want in an RTOS
- Learning by example: FreeRTOS
  - Introduction
  - Tasks
  - Interrupts
  - Internals (briefly)
  - What's missing?

What is a Real-Time System?

- Real-time systems have been defined as: "those systems in which the correctness of the system depends not only on the logical result of the computation, but also on the time at which the results are produced";
  - J. Stankovic, "Misconceptions About Real-Time Computing," ***IEEE Computer,*** 21(10), October 1988.

Soft, Firm and Hard deadlines

- The instant at which a result is needed is called a deadline.
  - If the result has utility even after the deadline has passed, the deadline is classified as **soft**, otherwise it is **firm**.
  - If a catastrophe ***could*** result if a firm deadline is missed, the deadline is **hard**.
- Examples?

Definitions taken from a paper by Kanaka Juvva, not sure who originated them.

Why is this hard?

Three major issues

1. We want to use as cheap ($$, power) a processor as possible.
   – Don't want to overpay
2. There are non-CPU resources to worry about.
   – Say two devices both on an SPI bus.
   – So often can't treat tasks as independent
3. Validation is hard
   – You've got deadlines you ***must*** meet.
   - How do you ***know*** you will?

Let's discuss that last one a bit more

Validating a RTS is hard

- Validation is simply the ability to be able to prove that you will meet your constraints
  - Or for a non-hard time system, prove failure is rare.

- This is a hard problem just with timing restrictions
  - How do you ***know*** that you will meet all deadlines?
    - A task that needs 0.05s of CPU time and has a deadline 0.1s after release is "easy".
    - But what if three such jobs show up at the same time?
  - And how do you know the worst-case for all these applications?
    - Sure you can measure a billion instances of the program running, but could something make it worse?
      - Caches are a pain here.

- And all that ***ignores*** non-CPU resource constraints!

We need some formal definitions to make progress here…

Real-Time Characteristics

- Pretty much your typical embedded system
  - Sensors & actuators all controlled by a processor.
  - The big difference is **timing constraints** (deadlines).
- Those tasks can be broken into two categories[1]
  - **Periodic Tasks**: Time-driven and recurring at regular intervals.
    - A car checking for a wall every 0.1 seconds;
    - An air monitoring system grabbing an air sample every 10 seconds.
  - **Aperiodic**: event-driven
    - That car having to react to a wall it found
    - The loss of network connectivity.

[1]Sporadic tasks are sometimes also discussed as a third category. They are tasks similar to aperiodic tasks but activated with some known bounded rate. The bounded rate is characterized by a minimum interval of time between two successive activations.

Some Definitions

- **Timing constraint:**
  - Constraint imposed on timing behavior of a job: hard, firm, or soft.
- **Release Time**:
  - Instant of time job becomes available for execution.
- **Deadline**:
  - Instant of time a job's execution is required to be completed.
- **Response time**:
  - Length of time from release time to the instant the job completes.

Example Using the Definitions: Periodic jobs

- Consider a processor where you have two different tasks.
  - We'll call them τ_1 and τ_2.
  - It is commonly the case that the period of a task is the same as the time you have to complete the task.
    - So if τ_1 needs to run every 7 seconds, you just need to finish a given instance of τ_1 before the next one is to start.

- Consider the following:

	Period
τ_1	7
τ_2	16

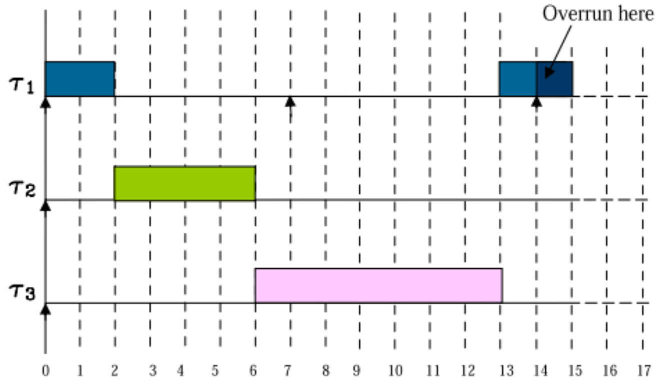
- Assuming all tasks are first released at time 0:
  - What is the deadline of the first instance of τ_1?
  - What is the release time of the second instance of τ_1?
  - What is the deadline of the second instance of τ_1?

Properties for Scheduling tasks

- **Priority**
  - If two tasks are both waiting to run at the same time, one will be selected. That one is said to have the higher priority.
- **Fixed/Dynamic priority tasks**
  - In priority-driven scheduling, assigning the priorities can be done statically (always the same) or they can be done dynamically (changing while the system is running)
- **Preemptive/Non-preemptive tasks**
  - Execution of a non-preemptive task is to be completed without interruption once it is started
  - Otherwise a task can be preempted if another task of higher priority becomes ready

Preemption

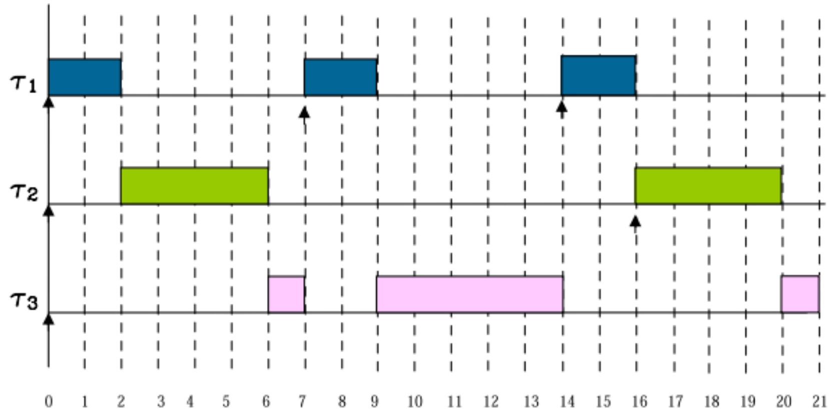
What it is and how it helps



Task 1's second instance misses its deadline!

	Priority	*Period*	*Computation time*
τ_1	1	7	2
τ_2	2	16	4
τ_3	3	31	7

Now add preemption:



All tasks are released at time 0.

Mohammadi, Arezou, and Selim G. Akl. "Scheduling Algorithms for Real-Time Systems." (2005)

Scheduling algorithms

- A scheduling algorithm is a scheme that selects what job to run next.
  - Can be preemptive or non-preemptive.
  - Dynamic or static priorities
  - Etc.

Two scheduling algorithms

- **Rate monotonic (RM)**
  - Static priority scheme
  - Simple to implement
  - Nice properties
- **Earliest deadline first (EDF)**
  - Dynamic priority scheme
  - Harder to implement
  - Very nice properties

RM and EDF *assumptions*

- No task has any non-preemptable section and the cost of preemption is negligible.
- Only processing requirements are significant; memory, I/O, and other resource requirements are negligible.
- All tasks are independent; there are no precedence constraints.

Terms and definitions

- **Execution time of a task** - time it takes for a task to run to completion
- **Period of a task** - how often a task is being called to execute; can generally assume tasks are periodic although this may not be the case in real-world systems
- **CPU utilization** - the percentage of time that the processor is being used to execute a specific scheduled task

$$U = \frac{e_i}{P_i}$$

  - where e_i is the execution time of task i, and P_i is its period
- **Total CPU utilization** - the summation of the utilization of all tasks to be scheduled

$$\sum_{i=1}^{n} \frac{e_i}{P_i}$$

RM Scheduling

- It is a **static**-priority preemptive scheme involving periodic tasks only.
  - Well, it mumbles about non-periodic tasks, but…
- Basic idea:
  - Priority goes to the task with the lowest period.

How well does RMS work?

- Surprisingly well actually.
  - Let *n* be the number of tasks.
  - If the total utilization is less than $n(2^{1/n}-1)$, the tasks are schedulable.
    - That's pretty cool.
      - At n=2 that's ~83.3%
      - At n=∞ that's about 69.3%
  - This means that our (extremely) simple algorithm will work if the total CPU utilization is less than 2/3!
    - Still, don't forget our assumptions (periodic being the big one)
- Also note, this is a sufficient, but not necessary condition
  - Tasks may still be schedulable even if this value is exceeded.
    - But not if utilization is over 100% of course…

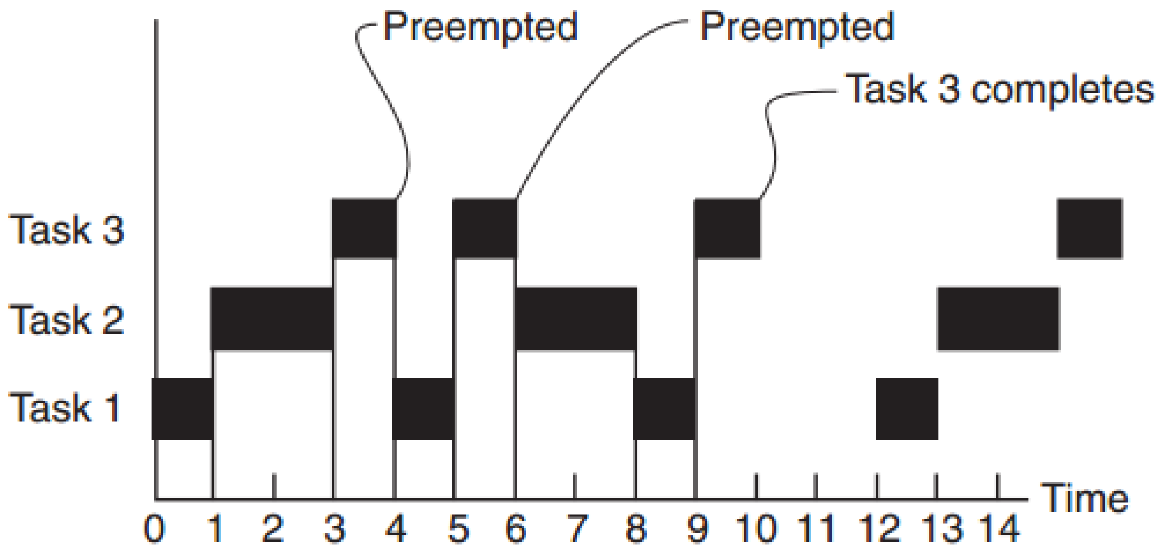
Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment
http://igm.univ-mlv.fr/~masson/pdfANDps/liulayland73.pdf

What if the sufficiency bound is not met?

- Critical instant analysis
  - The worst case for RMS is that all tasks happen to start at the exact same time.
    - If RM can schedule the first instance of each such task, the tasks are schedulable.
- With RM scheduling we can always jump to doing the critical instant analysis

Example #1

Task	Execution Time	Period	Priority
T1	1	4	High
T2	2	6	Medium
T3	3	12	Low



https://www.eecs.umich.edu/courses/eecs461/lecture/SWArchitecture.pdf

Example #2

Task	Execution Time	Period	Priority
T1	1	3	High
T2	1.5	4	Low

Example #3

Task	Execution Time	Period	Priority
T1	1	3	High
T2	2.1	4	Low

Example #4

Task	Execution Time	Period	Priority
T1	1	2	High
T2	2	5	Low

Easy?

- Fixed priorities are pretty easy.
  - Scheduler easy to write.
  - Don’t have to worry about updating priorities
    - Extra effort (Engineering and CPU)
    - Could lead to thrashing

EDF Scheduling

- Also called the deadline-monotonic scheduling algorithm
  - a priority-driven algorithm in which higher priority is assigned to the request that has earlier deadline *and* a higher priority request always preempts a lower priority one.
- Uses dynamic priority assignment in the sense that the priority of a request is assigned as the request arrives
  - May have to update other tasks' priorities (why?)
- We make all the assumptions we made for the RM algorithm, except that the tasks do not have to be periodic
- Same runtime complexity as RM scheduling if sorted lists are used
- EDF is an ***optimal*** uniprocessor scheduling algorithm

If all tasks are periodic and have relative deadlines equal to their periods, they can be feasibly scheduled by EDF *if and only if* $\sum_{i=1}^{n} C_i/P_i \leq 1$.

Mohammadi, Arezou, and Selim G. Akl. "Scheduling Algorithms for Real-Time Systems." (2005)

EDF issues

- Dynamic priorities
  - When do you need to recompute priorities?
  - How much time will you need to take?
    - O(?)
- When it fails
  - Which tasks will be dropped is hard to predict.
    - “This is a considerable disadvantage to a real-time systems designer.”
- Representing wide variety of time scales can be tricky.
  - Need to track how long until deadline, but some tasks may have deadline on the order of milliseconds, others hours or days.
- Shared resources are tricky.
  - Like priority inheritance problem for RM scheduling (covered shortly)

Wonderfully understated quote in green from Wikipedia.

LLF (Least Laxity First) Scheduling

- The laxity of a process is defined as the deadline minus remaining computation time
  - In other words, the laxity of a job is the maximal amount of time that the job can wait and still meet its deadline
- The algorithm gives the highest priority to the active job with the smallest laxity
- While a process is executing, it can be preempted by another whose laxity has decreased to below that of the running process
  - A problem arises with this scheme when two processes have similar laxities. One process will run for a short while and then get preempted by the other and vice versa, thus causing many context switches occur in the lifetime of the processes. (Thrashing)
- The least laxity first algorithm is an optimal scheduling algorithm for systems with periodic real-time tasks
  - Also useful with aperiodic tasks

round robin (no priorities)

Task	Execution Time	Period
T1	5 min	1 hr
T2	0.6 sec	1 sec

Finally…

Optimal?

- An optimal real-time scheduling algorithm is one which may fail to meet a deadline only if no other scheduling algorithm could meet all of the deadlines.

Finally…

Of course, this ignores a lot

- We've only worried about the CPU as a resource.
  - What if one task is waiting on another to free a resource?

Waiting for a Serial Port

- Say I’ve got two tasks that can write to a serial port (using printf()* let’s say).
  - They are writing messages to a terminal.
  - If one is in the middle of a message, I can’t preempt it with another that can/will write to the serial port.
    - My message will be jumbled as one message hits the middle of the other.
- This can happen with all kinds of resources including other ports (e.g. I2C, USB) and timers
  - Common to have a shared data structure where if one task is writing, no task can use it.
    - Why might someone else not be able to read?

*The standard version of printf is not “reentrant”. This is a good thing to know (and understand)

How do we know to wait?

- We use a variable.
  - Any task that is going to use the locked resource (say serial port) checks that variable first.
  - If that variable is a “0” we wait until it’s a “1”.
  - Once it’s a “1”, we set it to “0” and then proceed to use the serial port.
  - Once we’re done with the serial port, we set it back to a “1”.
- The standard terminology for this is:
  - The variable is called a “binary semaphore”
  - Setting it to “0” is called locking. If it is a “0” we say the semaphore is locked.
  - Setting it to “1” is called unlocking.
- Anyone wanting to use the serial port is supposed to check the semaphore, and then lock it and unlock it in the same way.
  - The code that needs to be locked is often called a “critical section”.

Let’s do this in C

- Write code that uses
  - a global variable “sem_serial” as the lock.
  - Calls “printf()” to use the serial port

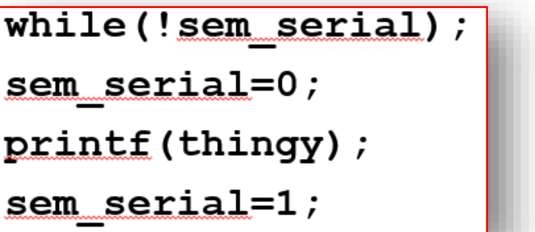
Proposed answer

```
while(!sem_serial);        // wait until it's available
                           //using 1 as available and
                           //0 as not. Notice the ;

sem_serial=0;
printf(thingy);
sem_serial=1;
```

Problems with the code we wrote (1/2)

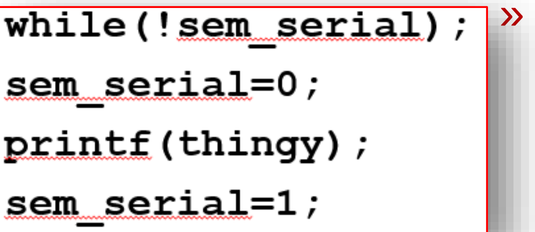
- Now we generally need to be a bit more tricky than this.
  - Imagine task "A" checks the semaphore and then, before setting the lock, gets preempted by another task "B" that locks the semaphore then A gets scheduled again.
    - "A" will have passed the check and will proceed to use the serial port. So we need a way to prevent this from happening.
  - Most processors have ways of dealing with that issue at the assembly level (test-and-set for example)
    - Libraries that handle multiple tasks almost always include function calls that use those assembly-level instructions correctly.



```
while(!sem_serial);
sem_serial=0;
printf(thingy);
sem_serial=1;
```

Problems with the code we wrote (2/2)

- The other issue is that our while loop is busy waiting.
  - Say that task A is the high priority task. Task A and B both use the serial bus.
    - B is running and locks the bus
    - A preempts B.
    - A will now wait in the loop until B finishes
      - But A is higher priority, so B can't run until A finishes. And A can't run until B finishes.
        - » **DEADLOCK!**



```
while(!sem_serial);
sem_serial=0;
printf(thingy);
sem_serial=1;
```

- Solution is that rather than a loop, we call a function and ask to lock the semaphore.
  - If we don't get the lock we give up control of the processor until the lock is available.
  - Tricky to write
    - Another reason to use libraries here rather than rolling your own!

Obligatory deadlock joke

- Interviewer:
  - "Explain deadlock and we'll hire you."
- Me:
  - "Hire me and I'll explain it to you."

But wait, there's more!

- We want some kind of semaphore_take() and semaphore_give() functions that handle the issues described
  - Uses atomic operations like test-and-set to handle race conditions.
  - The take function should cause the current task to give up the CPU
  - The give function should check to see if anyone is waiting on the semaphore.
- But that's still not enough.

Priority Inversion

- In a preemptive priority-based real-time system, sometimes tasks may need to access resources that cannot be shared.
  - The method of ensuring exclusive access is to guard the critical sections with binary semaphores.
  - When a task seeks to enter a critical section, it checks if the corresponding semaphore is locked.
  - If it is not, the task locks the semaphore and enters the critical section.
  - When a task exits the critical section, it unlocks the corresponding semaphore.
- This could cause a high priority task to be waiting on a lower priority one.
  - Even worse, a medium priority task might be running and cause the high priority task to not meet its deadline!

Some taken from Mohammadi, Arezou, and Selim G. Akl. "Scheduling Algorithms for Real-Time Systems." (2005)

Example: Priority Inversion

- Low priority task "C" locks resource "Z".
- High priority task "A" preempts "A" then requests resource "Z"
  – Deadlock, but solvable by having "A" sleep until resource is unlocked.
- But if medium priority task "B" were to run, it would preempt C, thus effectively making C and A run with a lower priority than B.
  – Thus priority *inversion*.

Finally…

Solving it: Priority Inheritance

- When a high priority task sleeps because it is waiting on a lower priority task, have it boost the priority of the blocking task to its own priority.
  - In general a mutex is a semaphore that is used for **mu**tual **ex**clusion.
- There are other solutions
  - For example, if there are only 2 priorities this can't happen
  - Windows has a rather interesting solution called "random boosting"

Net effect

- Handling locks and preemption and all that is hard.
  - It’s also fairly generic
    - That is, you’re solving the same problem each time.
  - So what you want is to just do it once.
    - Or better yet, borrow code from someone else who is really good at this stuff.
  - This is part of what an RTOS will generally provide.

Outline

- Overview of real-time systems
  - Basics
  - Scheduling algorithms (RM, EDF, LLF & RR)
- Overview of RTOSes
  - Goals of an RTOS
  - Features you might want in an RTOS
- Learning by example: FreeRTOS
  - Introduction
  - Tasks
  - Interrupts
  - Internals (briefly)
  - What's missing?

Goals of an RTOS?

- Well, to manage to meet RT deadlines (duh).
  - While that's all we ***need*** we'd ***like*** a lot more.
    - After all, we can meet RT deadlines fairly well on the bare metal (no OS)
      - But doing this is time consuming and difficult to get right as the system gets large.
    - We'd ***like*** something that supports us
      - Deadlines met
      - Locks work
      - Interrupts just work
      - Tasks stay out of each others way
      - Device drivers already written (and tested!) for us
      - Portable—runs on a huge variety of systems
      - Oh, and nearly no overhead so we can use a small device!
        - » That is a small memory and CPU footprint.

Detailed features we'd like (1/2)

Deadlines met

- Ability to specify scheduling algorithm
- Interrupts are fast
  - So tasks with tight deadlines get service as fast as possible
    - Basically—rarely disable interrupts and when doing so only for a short time.

Interrupts just work

- Don't need to worry about saving/restoring registers
  - Which C just generally does for us anyways.
- Interrupt prioritization easy to set.

Detailed features we'd like (2/2)

Locks work

- High-priority waiting on locks give up the CPU until lower-priority tasks finish their critical section
- Priority inheritance is done for us.

Device drivers

- Clean interfaces to I2C, USB, SPI, timers, and whatever else our processor supports already written for us.
  - Hopefully well tested and well documented.
- In my experience, this is often the least likely thing to be 100% there.
  - Mainly because much of the other stuff is easy to port.
  - This is very processor specific.

Detailed features we'd like: **Tasks stay out of each others way**

- This is actually remarkably hard
  - Clearly we need to worry about CPU utilization issues
    - That is what our scheduling algorithm discussion was to address
  - But we also need to worry about *memory* problems.
    - One task running awry shouldn't take the rest of the system down.
  - So we want to prevent tasks from harming each other
    - This can be ***key***. If we want mission critical systems sharing the CPU with less important things we have to do this.
    - Alternative it to have separate processors.
      - $$$$
- The standard way to do this is with page protection.
  - If a process tries to access memory that isn't its own, it fails.
    - Probably a fault.
    - This also makes debugging a LOT easier.
- This generally requires a lot of overhead.
  - Need some sense of process number/switching
  - Need some kind of MMU in hardware
    - Most microcontrollers lack this…
    - So we hit some kind of minimum size.

Further reading on page protection (short) http://homepage.cs.uiowa.edu/~jones/security/notes/06.shtml

Aside: What is an MMU?

- Memory Management Unit
  - Tracks what parts of memory a process can access.
    - Actually a bit more complex as it manages this by mapping virtual addresses to physical ones.
    - Keeps processes out of each other's memory.

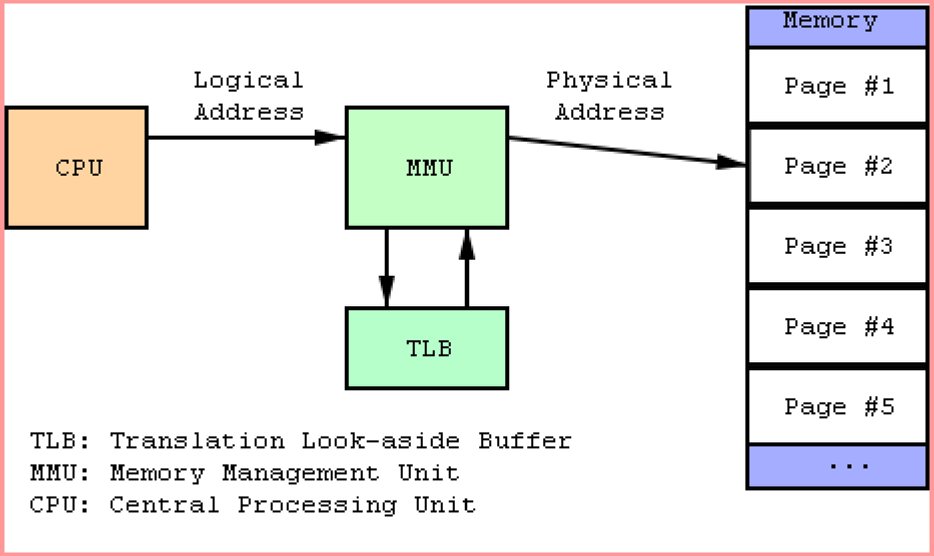

CPU
Logical Address
MMU
Physical Address
TLB
Memory
Page #1
Page #2
Page #3
Page #4
Page #5
...
TLB: Translation Look-aside Buffer
MMU: Memory Management Unit
CPU: Central Processing Unit

Figure from Wikipedia

Portable

- RTOS runs on many platforms.
  - This is potentially incomputable with the previous slide.
  - It’s actually darn hard to do even without peripherals
    - For example I have spent 10 hours debugging a RTOS that had a pointer problem that only comes up if the pointer type is larger than the int type (20 bit pointers, 16 bit ints, yea!)
    - Things like timers change and we certainly need timers.

Outline

- Quick review of real-time systems
- Overview of RTOSes
  - Goals of an RTOS
  - Features you might want in an RTOS
- Learning by example: FreeRTOS
  - Introduction
  - Tasks
  - Interrupts
  - Internals (briefly)
  - What's missing?

Learning by example: FreeRTOS

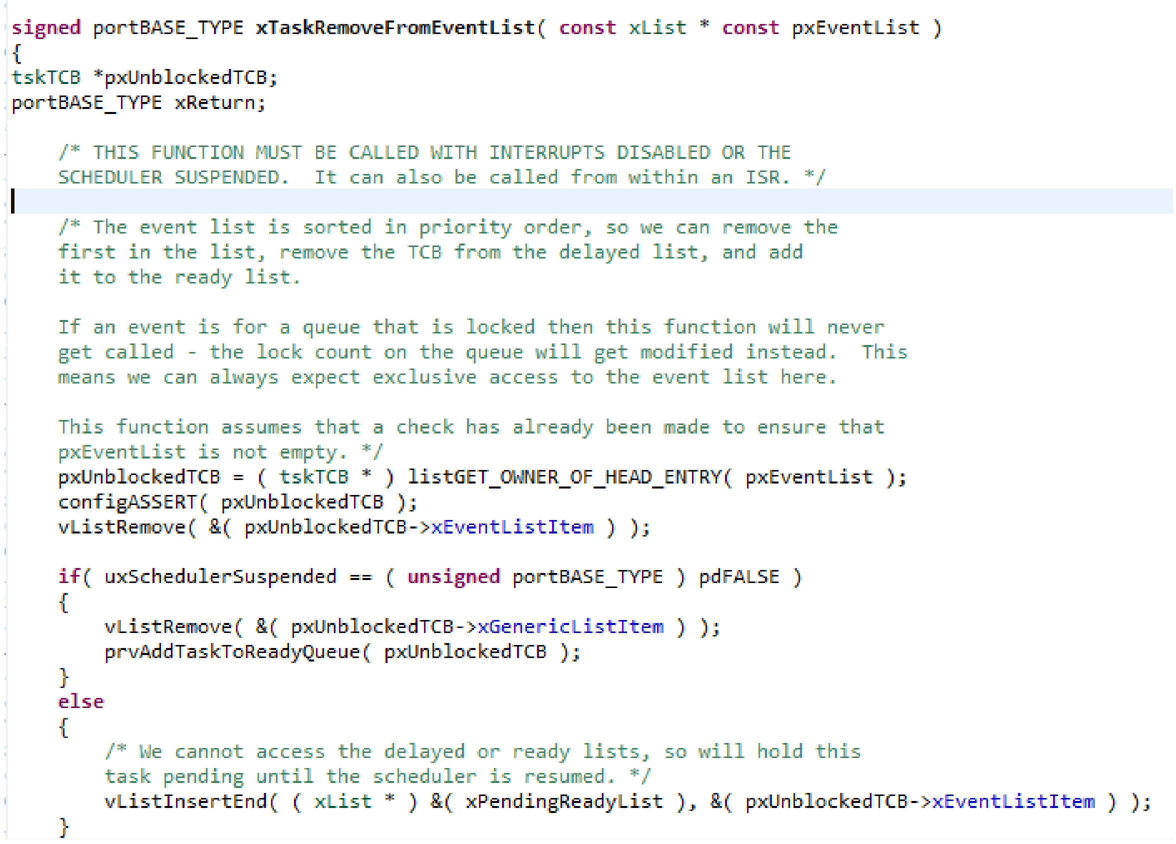
- Introduction taken from Amr Ali Abdel-Naby
  - Nice blog:
    - http://www.embedded-tips.blogspot.com

FreeRTOS Features

- Source code
- Portable
- ROM-able
- Scalable
- Preemptive and co-operative scheduling
- Multitasking
- Services
- Interrupt management
- Advanced features

Source Code

- High quality
- Neat
- Consistent
- Organized
- Commented



```
signed portBASE_TYPE xTaskRemoveFromEventList( const xList * const pxEventList )
{
tskTCB *pxUnblockedTCB;
portBASE_TYPE xReturn;

    /* THIS FUNCTION MUST BE CALLED WITH INTERRUPTS DISABLED OR THE
    SCHEDULER SUSPENDED.  It can also be called from within an ISR. */

    /* The event list is sorted in priority order, so we can remove the
    first in the list, remove the TCB from the delayed list, and add
    it to the ready list.

    If an event is for a queue that is locked then this function will never
    get called - the lock count on the queue will get modified instead.  This
    means we can always expect exclusive access to the event list here.

    This function assumes that a check has already been made to ensure that
    pxEventList is not empty. */
    pxUnblockedTCB = ( tskTCB * ) listGET_OWNER_OF_HEAD_ENTRY( pxEventList );
    configASSERT( pxUnblockedTCB );
    vListRemove( &( pxUnblockedTCB->xEventListItem ) );

    if( uxSchedulerSuspended == ( unsigned portBASE_TYPE ) pdFALSE )
    {
        vListRemove( &( pxUnblockedTCB->xGenericListItem ) );
        prvAddTaskToReadyQueue( pxUnblockedTCB );
    }
    else
    {
        /* We cannot access the delayed or ready lists, so will hold this
        task pending until the scheduler is resumed. */
        vListInsertEnd( ( xList * ) &( xPendingReadyList ), &( pxUnblockedTCB->xEventListItem ) );
    }
```

Portable

- Highly portable C
- 24+ architectures supported
- Assembly is kept minimum.
- Ports are freely available in source code.
- Other contributions do exist.

Launched by Motorola
freescale™
semiconductor

ARM

intel

ATMEL

RENESAS
Everywhere you imagine.

Scalable

- Only use the services you only need.
  - FreeRTOSConfig.h
- Minimum footprint = 4 KB
- Version in lab on Pi is ~24 KB including the application and data for the OS and application.
  - Pretty darn small for what you get.
  - ~6000 lines of code (including a lot of comments, maybe half that without?)

Preemptive and Cooperative Scheduling

- Preemptive scheduling:
  - Fully preemptive
  - Always runs the highest priority task that is ready to run
  - Comparable with other preemptive kernels
  - Used in conjunction with tasks

- Cooperative scheduling:
  - Context switch occurs if:
    - A task/co-routine blocks
    - Or a task/co-routine yields the CPU
  - Used in conjunction with tasks/co-routines

Multitasking

- No software restriction on:
  - # of tasks that can be created
  - # of priorities that can be used
  - Priority assignment
    - More than one task can be assigned the same priority.
    - RR with time slice = 1 RTOS tick

Services

- Queues
- Semaphores
  - Binary and counting
- Mutexes
  - With priority inheritance
  - Support recursion

Interrupts

- An interrupt can suspend a task execution.
- Interrupt mechanism is port dependent.

Advanced Features

- Execution tracing
- Run time statistics collection
- Memory management
- Memory protection support
- Stack overflow protection

Device support in related products

- Connect Suite from High Integrity Systems
  - TCP/IP stack
  - USB stack
    - Host and device
  - File systems
    - DOS compatible FAT

Licensing

- Modified GPL
  - Only FreeRTOS is GPL.
  - Independent modules that communicate with FreeRTOS through APIs can be anything else.
  - FreeRTOS can’t be used in any comparisons without the authors’ permission.

A bit more

- System runs on “ticks”
  - Every tick the kernel runs and figures out what to do next.
    - Interrupts have a different mechanism
  - Basically hardware timer is set to generate regular interrupts and calls the scheduler.
    - This means the OS eats one of the timers—you can’t easily share.

OK, onto tasks!

Outline

- Quick review of real-time systems
- Overview of RTOSes
  - Goals of an RTOS
  - Features you might want in an RTOS
- Learning by example: FreeRTOS
  - Introduction
  - Tasks
  - Interrupts
  - Internals (briefly)
  - What's missing?

Tasks

- Each task is a function that must not return
  - So it’s in an infinite loop (just like you’d expect in an embedded system really, think Arduino).
- You inform the scheduler of
  - The task’s resource needs (stack space, priority)
  - Any arguments the tasks needs
- All tasks here must be of void return type and take a single void* as an argument.
  - You cast the pointer as needed to get the argument.
    - I’d have preferred var_args, but this makes the common case (one argument) easier (and faster which probably doesn’t matter).

Example trivial task with busy wait (bad)

```
void vTask1( void *pvParameters )
{
const char *pcTaskName = "Task 1 is running\r\n";
volatile unsigned long ul;

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );

        /* Delay for a period. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /* This loop is just a very crude delay implementation.  There is
            nothing to do in here.  Later examples will replace this crude
            loop with a proper delay/sleep function. */
        }
    }
}
```

Task creation

```
portBASE_TYPE xTaskCreate(
  pdTASK_CODE pvTaskCode,
  const char * const pcName,
  unsigned short usStackDepth,
  void *pvParameters,
  unsigned portBASE_TYPE uxPriority,
  xTaskHandle *pvCreatedTask
  );
```

Create a new task and add it to the list of tasks that are ready to run. **xTaskCreate()** can only be used to create a task that has unrestricted access to the entire microcontroller memory map. Systems that include MPU support can alternatively create an MPU constrained task using xTaskCreateRestricted().

- **pvTaskCode:** Pointer to the task entry function. Tasks must be implemented to never return (i.e. continuous loop).
- **pcName:** A descriptive name for the task. This is mainly used to facilitate debugging. Max length defined by tskMAX_TASK_NAME_LEN – default is 16.
- **usStackDepth:** The size of the task stack specified as the number of variables the stack can hold - not the number of bytes. For example, if the stack is 16 bits wide and usStackDepth is defined as 100, 200 bytes will be allocated for stack storage.
- **pvParameters**: Pointer that will be used as the parameter for the taskbeing created.
- **uxPriority:** The priority at which the task should run. Systems that include MPU support can optionally create tasks in a privileged (system) mode by setting bit portPRIVILEGE_BIT of the priority parameter. For example, to create a privileged task at priority 2 the uxPriority parameter should be set to (2 | portPRIVILEGE_BIT).
- **pvCreatedTask:** Used to pass back a handle by which the created task can be referenced.
- **pdPASS**: If the task was successfully created and added to a ready list, otherwise an error code defined in the file errors.h

From the task.h file in FreeRTOS

Creating a task: example

```
int main( void )
{
    /* Create one of the two tasks.  Note that a real application should check
    the return value of the xTaskCreate() call to ensure the task was created
    successfully. */
    xTaskCreate(    vTask1,  /* Pointer to the function that implements the task. */
                    "Task 1",/* Text name for the task.  This is to facilitate
                             debugging only. */
                    1000,    /* Stack depth - most small microcontrollers will use
                             much less stack than this. */
                    NULL,    /* We are not using the task parameter. */
                    1,       /* This task will run at priority 1. */
                    NULL );  /* We are not going to use the task handle. */

    /* Create the other task in exactly the same way and at the same priority. */
    xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, NULL );

    /* Start the scheduler so the tasks start executing. */
    vTaskStartScheduler();
```

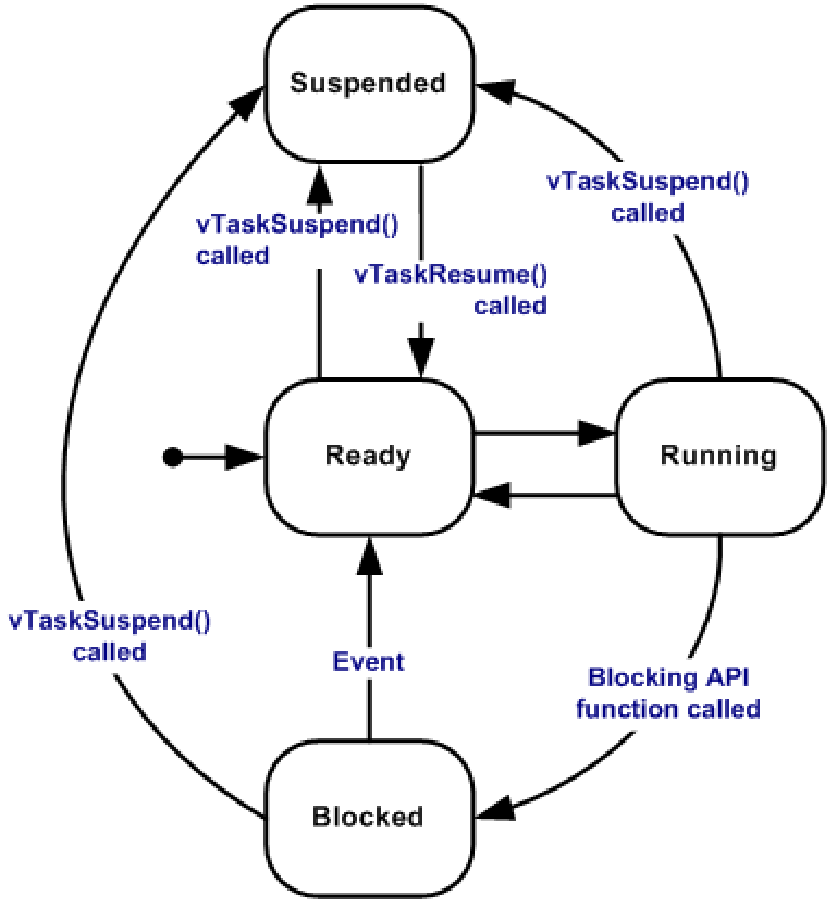
OK, I’ve created a task, now what?

- Task will run if there are no other tasks of higher priority
  - And if others the same priority will RR.
- But that begs the question: “How do we know if a task wants to do something or not?”
  - The previous example gave *always* wanted to run.
    - Just looping for delay (which we said was bad)
    - Instead should call **`vTaskDelay(x)`**
      - Delays current task for X “ticks” (remember those?)
    - There are a few other APIs for delaying…

Now we need an “under the hood” understanding

Task status in FreeRTOS

- **Running**
  - Task is actually executing
- **Ready**
  - Task is ready to execute but a task of equal or higher priority is Running.
- **Blocked**
  - Task is waiting for some event.
    - **Time**: if a task calls vTaskDelay() it will block until the delay period has expired.
    - **Resource**: Tasks can also block waiting for queue and semaphore events.
- **Suspended**
  - Much like blocked, but not waiting for anything.
  - Tasks will only enter or exit the suspended state when explicitly commanded to do so through the vTaskSuspend() and xTaskResume() API calls respectively.



Mostly from http://www.freertos.org/RTOS-task-states.html

Tasks: there's a lot more

- Can do all sorts of things
  - Change priority of a task
  - Delete a task
  - Suspend a task (mentioned above)
  - Get priority of a task.
- Example on the right
  - But we'll stop here…

```
void vTaskPrioritySet(
xTaskHandle pxTask,
unsigned uxNewPriority
);
```

Set the priority of any task.

- **pxTask:** Handle to the task for which the priority is being set. Passing a NULL handle results in the priority of the calling task being set.
- **uxNewPriority:** The priority to which the task will be set.

Outline

- Quick review of real-time systems
- Overview of RTOSes
  - Goals of an RTOS
  - Features you might want in an RTOS
- Learning by example: FreeRTOS
  - Introduction
  - Tasks
  - Interrupts
  - Internals (briefly)
  - What’s missing?

Interrupts in FreeRTOS

- There is both a lot and a little going on here.
  - The interface mainly uses whatever the native environment uses to handle interrupts
    - This can be **very** port dependent. In Code Composer Studio you'd set it up as follows:

```
#pragma vector=PORT2_VECTOR
interrupt void prvSelectButtonInterrupt( void )
```

  - That would cause the code to run on the PORT2 interrupt.
    - Need to set that up etc. Very device specific (of course).

More: Deferred Interrupt Processing

- The best way to handle complex events triggered by interrupts is to **not** do the code in the ISR.
  - Rather create a task that is blocking on a semaphore.
    - When the interrupt happens, the ISR just sets the semaphore and exits.
      - Task can now be scheduled like any other. No need to worry about nesting interrupts (and thus interrupt priority).
      - FreeRTOS does support nested interrupts on some platforms though.
  - Semaphores implemented as one/zero-entry queue.

Semaphore example in FreeRTOS

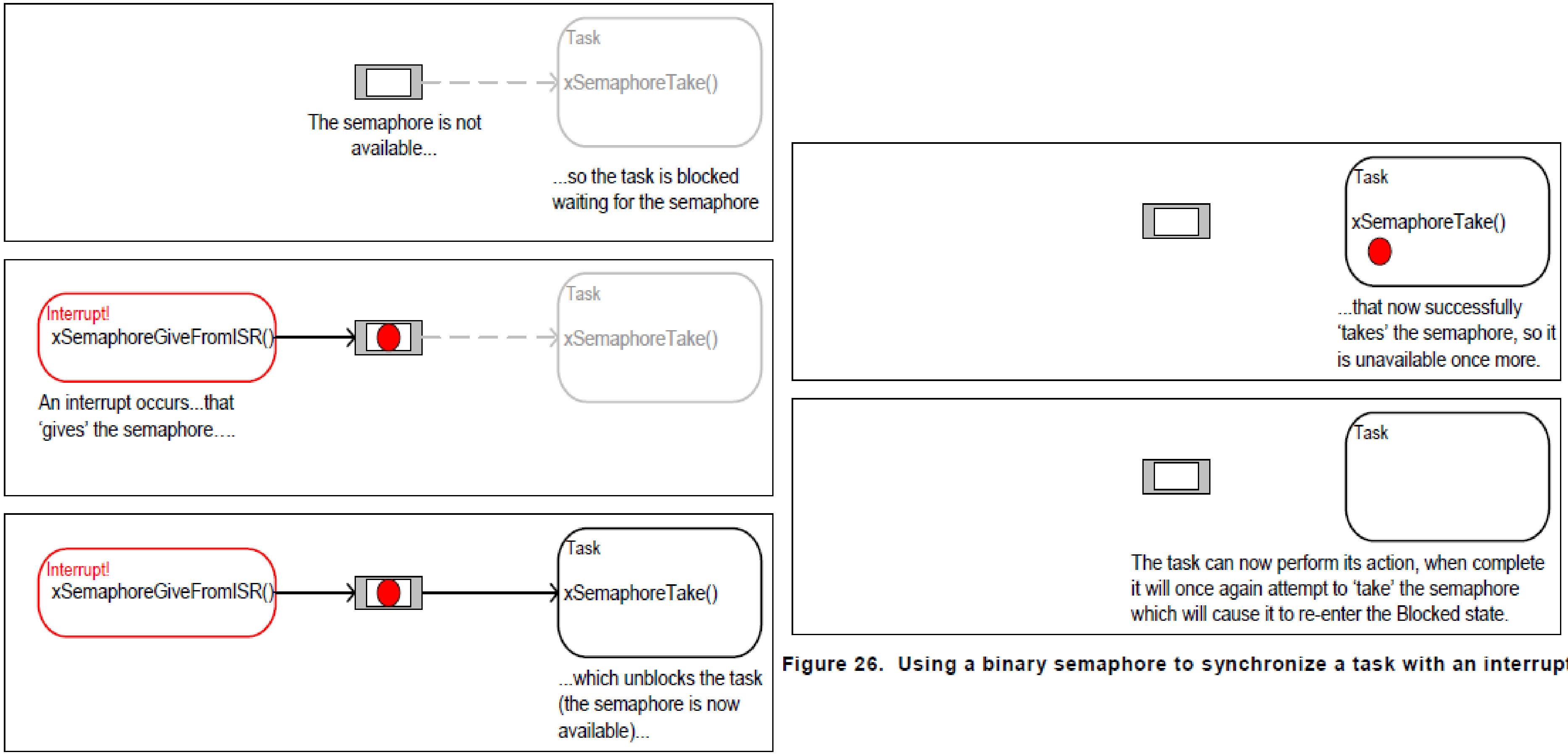

Task
xSemaphoreTake()
The semaphore is not available...
...so the task is blocked waiting for the semaphore
Interrupt!
xSemaphoreGiveFromISR()
Task
xSemaphoreTake()
An interrupt occurs...that 'gives' the semaphore....
Interrupt!
xSemaphoreGiveFromISR()
Task
xSemaphoreTake()
...which unblocks the task (the semaphore is now available)...
Task
xSemaphoreTake()
...that now successfully 'takes' the semaphore, so it is unavailable once more.
Task
The task can now perform its action, when complete it will once again attempt to 'take' the semaphore which will cause it to re-enter the Blocked state.

Figure 26. Using a binary semaphore to synchronize a task with an interrupt

Figure from ***Using the FreeRTOS Real Time Kernel*** (a pdf book), fair use claimed.

Semaphore take

```
xSemaphoreTake(
             xSemaphoreHandle xSemaphore,
             portTickType xBlockTime
              )
```

- *Macro* to obtain a semaphore. The semaphore must have previously been created.

- **xSemaphore** A handle to the semaphore being taken - obtained when the semaphore was created.

- **xBlockTime** The time in ticks to wait for the semaphore to become available. The macro portTICK_RATE_MS can be used to convert this to a real time. A block time of zero can be used to poll the semaphore.

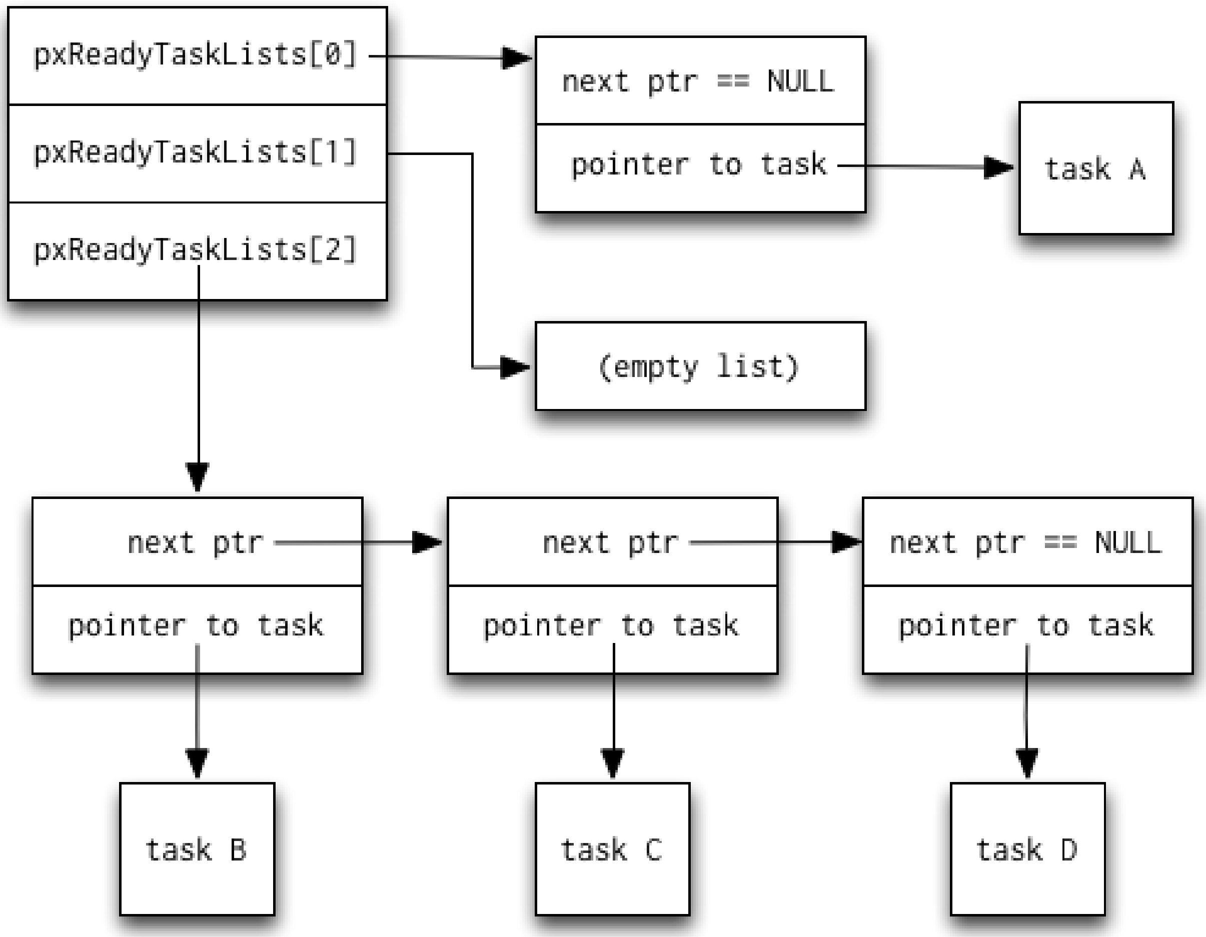
- TRUE if the semaphore was obtained.

- There are a handful of variations.
  - Faster but more locking version, non-binary version, etc.

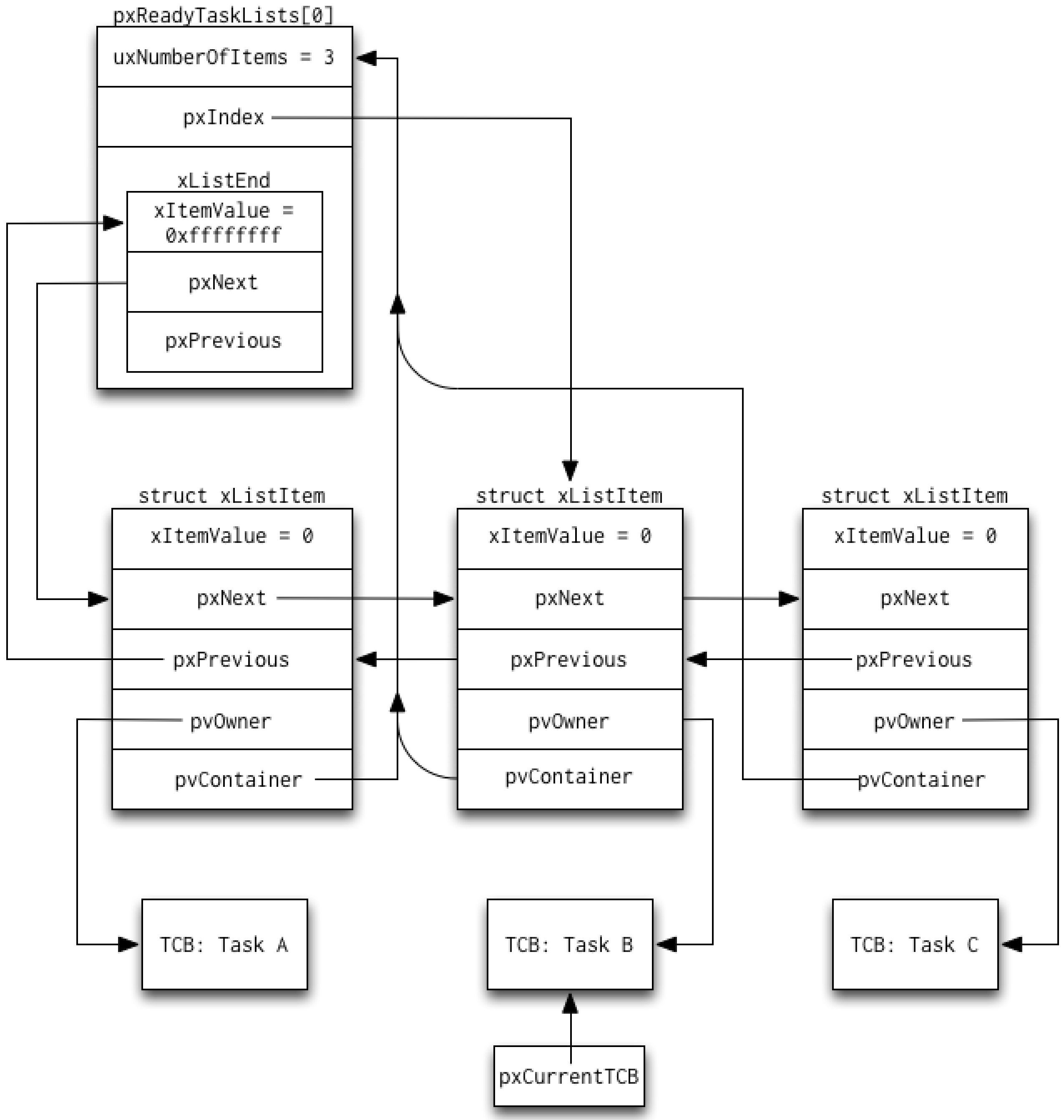
Outline

- Quick review of real-time systems
- Overview of RTOSes
  - Goals of an RTOS
  - Features you might want in an RTOS
- Learning by example: FreeRTOS
  - Introduction
  - Tasks
  - Interrupts
  - Internals (briefly)
  - What’s missing?

Common data structures



This figure and the next are from http://www.aosabook.org/en/freertos.html


pxReadyTaskLists[0]
uxNumberOfItems = 3
pxIndex
xListEnd
xItemValue = 0xffffffff
pxNext
pxPrevious
struct xListItem
xItemValue = 0
pxNext
pxPrevious
pvOwner
pvContainer
struct xListItem
xItemValue = 0
pxNext
pxPrevious
pvOwner
pvContainer
struct xListItem
xItemValue = 0
pxNext
pxPrevious
pvOwner
pvContainer
TCB: Task A
TCB: Task B
TCB: Task C
pxCurrentTCB