
LAB 5: Scheduling Algorithms for Embedded Systems

Say you have a robot that is exploring an area. The computer controlling the robot has a number of tasks to do: getting sensor input, driving the wheels, and running a navigational program that may have a fairly high computational load. One key issue in such an environment is to ensure that all the tasks are done in a timely way. On a microcontroller you might use a timer to generate interrupts that regularly address the motors and sensors while having the navigational task running as the main program. But that fairly simple model breaks down in a number of situations. For example, what if the navigational program is actually a collection of programs each of which need to run at different intervals for different durations (perhaps image processing, routing, and mapping programs)? A real-time operating system (RTOS) provides tools that allow us to schedule these tasks.

FreeRTOS (obviously) is a free real-time operating system that allows us to easily schedule tasks. In this lab you will get a chance to use the RTOS and see the impacts of scheduling on the real world.

1. A Not-So-Quick Introduction to Scheduling Algorithms

In order to effectively use the RTOS we need to understand the scheduling algorithms that are available to us. And in order to do that, we need to define a vocabulary that lets us easily discuss the tasks and their requirements.

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
 - knowing these values for the tasks we are trying to schedule is crucial
- **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

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

- **Preemption** - this occurs when higher priority task is set to run while a lower priority task is already running. The higher priority task preempts the lower priority one and the higher priority task runs to completion (unless it is preempted by an even higher priority task)
- **Deadline** - a ending time for which a system must finish executing all of its scheduled tasks. Missing a deadline has different consequences, depending on the type of deadline that was missed.

- **hard deadline** - a deadline for which missing it would be a complete system failure (e.g. airbag deployment being late could kill the user)
- **soft deadline** - penalty if a soft deadline is missed. These deadlines are usually in place to optimize the system, not to prevent complete failure. (e.g. an MP3 player might create a “pop” if it misses the deadline, but the system would still function)
- **Priority** - importance assigned to each task; determines the order in which tasks execute/preempt each other; priorities are generally assigned in 2 major ways
 - **static scheduling policy** - priorities are assigned by the user at design time and remain fixed
 - **dynamic scheduling policy** - priorities may change during program execution and priority assignment is handled by the scheduler, not the user
- **Critical instant** - an instant during which all tasks in the system are ready to *start* executing simultaneously

Common Scheduling Policies

There are a number of fairly standard scheduling algorithms that see use in RTOSes. An ideal scheduling algorithm would be able to schedule any set of tasks that require 100% or less of the processor, would fail gracefully when the tasks aren’t schedulable and would have fixed task priorities (static scheduling policy) and otherwise be simple to work with. Of course, no such algorithm, so we have a number of different algorithms we can consider, each with its own set of advantages and disadvantages.

Rate-monotonic Scheduling (RMS)

Rate-monotonic scheduling (RMS) is a popular and easy to understand static policy which has a number of useful properties. Priorities are assigned in rank order of task period, so the highest priority is given to the task with the shortest period and the lowest priority is given to the task with the longest period.

However, when working with RMS, one must take into account the following assumptions:

1. The requests for all tasks for which hard deadlines exist are periodic.
2. Deadlines consist of runability constraints only, that is, each task must be completed before the end of its period.
3. The tasks are independent—each task does not depend on any other tasks.
4. Task run time is constant (or at least has a known worst-case bound).
5. Any non-periodic tasks are special (initialization or failure recovery, for example), and do not have hard critical deadlines.

Of course, these assumptions are often violated in real-world systems, i.e. some tasks can become aperiodic due to event-driven interrupts, and determining actual worst-case runtime can be quite difficult.

Below is an example of an RTOS running an RMS policy with three tasks.

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

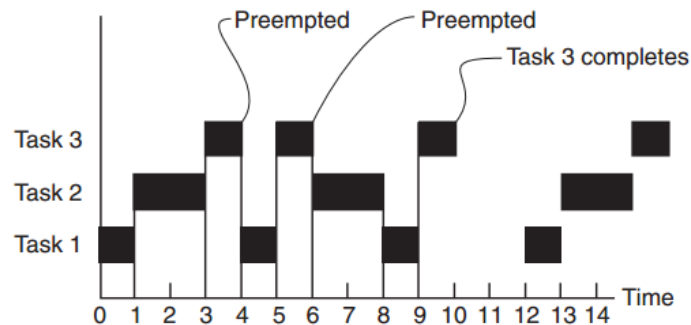


Table 1: A scheduling problem from [1]

Figure 1: “Critical instant” analysis, also from [1]

We want to know if these tasks are schedulable. First, notice that the total CPU utilization is $10/12$ (if it were greater than 1 we’d know the tasks were not schedulable). Next, we examine the critical instant—the time when all tasks try to start at the same time. In a static scheduling algorithm, this is the worst-case situation. Figure 1 shows what happens in the critical instant and we can see that each task easily meets its deadline and therefore the tasks are schedulable under RMS.

Q1. Explain why, at time 3 task 3 is running. Also explain why it stops running at time 4.

The example below is a case in which two tasks are not RM schedulable.

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

Table 2: A set of tasks which are not RM schedulable

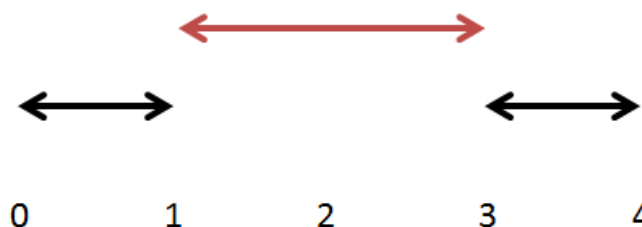


Figure 2: The critical instant analysis. Task 1 is in black, task 2 in red.

So, how do we know that this system isn't RM schedulable? It has a CPU utilization of 85.8%, so the next step is to check the critical instant. As we can see in figure 2, at time period 3, task 1 preempts task 2 and at time period 4 there are still 0.1 time units of execution time remaining for task 2 to complete, therefore task 2 has not met its deadline and the system isn't schedulable under RMS.

Q2. Using only single-digit periods and execution times identify set of no more than three tasks which are not RM schedulable but have a CPU utilization of less than 100% Show that they will fail to meet at least one deadline under critical instant analysis.

Sufficiency Condition for RMS

Although there isn't a straightforward way to determine for sure if a system is not RM schedulable, there is a sufficient but not necessary condition that guarantees that any system that meets this condition is RM schedulable. This check takes into account the processor's total utilization and the number of tasks to be scheduled (n).

$$\sum_{i=1}^n U \leq n(2^{1/n} - 1)$$

If this equation is satisfied, then the system is indeed RM schedulable. If this bound is not met, then we know nothing about the system's schedulability (it could still be RM schedulable) and must use the critical instant analysis. For two tasks (n=2) this means that we know any system that has a total CPU utilization less than 82.8% is schedulable. As always any system with a total CPU utilization greater than 100% is not schedulable. But for values between the two bounds we need to check the critical instant.

Below is an example of a system that does not meet the sufficiency condition, but is still clearly RM schedulable.

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

Table 3: A set of two tasks which doesn't meet the RMS sufficiency criteria, but is still RM schedulable

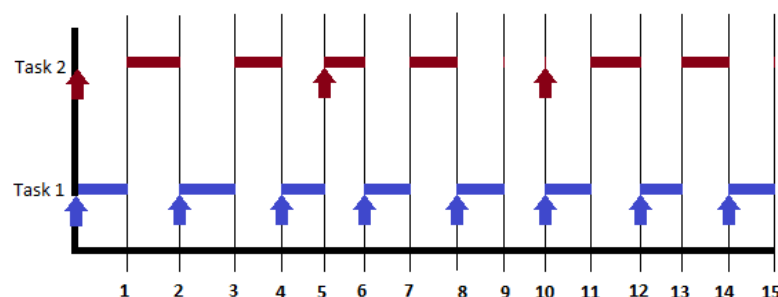


Figure 3: The critical instant analysis for the above tasks under RMS

The total utilization of the processor is 90% while the upper bound for 2 tasks, as defined by the sufficiency condition, is 82.8%, yet the system remains RM schedulable.

Q3. Say you have the following groups of tasks. For each group find the CPU utilization. Figure out which groups are RM schedulable. **Only** where needed are you to do (and show) the critical instant analysis.

Group	$T1$ Execution Time	$T1$ Period	$T2$ Execution Time	$T2$ Period	$T3$ Execution Time	$T3$ Period	CPU Utilization?	RM Schedulable?
A	2	5	1	10	2	4		
B	2	5	1	10	2	6		
C	2	6	1	10	2	8		
D	1	5	2	8	2	9		

Table 4: Four groups of tasks

Further reading

- A good overview of RMS: http://en.wikipedia.org/wiki/Rate-monotonic_scheduling
- The original RMS paper, including the sufficiency bounds analysis (which is fairly complex):
 - Liu, C. L.; Layland, J. (1973), "Scheduling algorithms for multiprogramming in a hard real-time environment"
- "Embedded Software Architecture", J.A. Cook, J. S. Freudenberg. This is an EECS 461 handout which you can probably find on-line. It covers a lot more than RMS.

First In First Out (FIFO)

First In First Out (FIFO) is, in some ways, the simplest scheduling algorithm. As the name suggests, this policy is based on a queue where the first process that enters the queue is the first process that is executed. Each process runs to completion (and no process can be preempted) before the scheduler executes the next process in the queue. And because there is no prioritization in FIFO, systems using this policy often have trouble meeting deadlines, even in systems with fairly low total CPU utilization. Below is an example of where FIFO's lack of prioritization causes a set of tasks with a very low CPU utilization to fail.

Task	Execution Time	Period
T1	2 sec	1 hr
T2	0.5 sec	2 sec

Table 4: A set of tasks which are not schedulable by FIFO. Note that there is no prioritization

In the example in table 4 can see that the total CPU utilization is clearly under 100% (it is just over 25% in fact), but it is not schedulable under FIFO. We can see this by looking at the critical instant. If T1 starts

executing just before T2 (say T1 arrived a fraction of a second earlier than T2) then T2 will not start until after its deadline has already passed. Among many limitations of FIFO, in general if any task has a period shorter than the total execution time of all tasks, then the tasks cannot be scheduled under FIFO.

Q4. Propose two groups of tasks that

- a. Are not schedulable under FIFO and have 1% CPU utilization.
- b. Are schedulable under FIFO and have 90% CPU utilization.

Q5. Consider the claim that “in general if any task has a period shorter than the total execution time of all tasks, then the tasks cannot be scheduled under FIFO.”

- a. Justify this claim in your own words.
- b. Does there exist a set of tasks that are unschedulable under FIFO that don’t meet that condition? This is, is this both a necessary and sufficient condition?

A common variation of FIFO is a priority-based version of FIFO. This is generally implemented with preemption. The idea is that higher priority tasks are always run first (and can preempt lower-priority tasks) but when two tasks both have the same priority, the one that arrived first is run.

Round Robin (RR)

Round Robin scheduling is one of the most widely used static scheduling algorithms, especially in operating systems. Processes are placed in a queue. An arbitrary interval of time is defined and the scheduler cycles through the queue, running each process for the length of that time interval until it is preempted by the next process in the queue. The scheduler keeps cycling through the queue until all processes have finished executing.

Because of its lack of prioritization, RR can also fail to successfully schedule a low-CPU utilization hard-deadline system, as shown in the following example.

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

Table 5: A set of tasks with low CPU utilization that isn’t schedulable by RR.

In the above example the total CPU utilization is around 68.3%, but it isn’t schedulable by RR. Assuming an infinitesimal time interval, in the critical instant T1 and T2 will each get half of the CPU. At one second, T2’s deadline will have arrived without the task being completed.

Q6. Does there exist a larger time interval for which the above tasks are schedulable? If so, give an example, if not explain why not.

Q7. For two tasks is there a total CPU utilization which is guaranteed to result in a schedulable system under RR? Explain why or why not. You are to assume an infinitesimal time interval.

Once again, it isn’t uncommon to have RR mixed with a priority scheme.

Earliest Deadline First (EDF)

Earliest Deadline First (EDF) is an optimal single-processor scheduling algorithm. It is a dynamic, priority-driven algorithm in which highest priority is assigned to the request that has the earliest deadline, and a higher priority request always preempts a lower priority one.

Since EDF is a dynamic scheduling policy, this means priority of a task is assigned as the task arrives (rather than determined before the system begins execution) and is updated as needed. This can be a very difficult thing to implement as the scheduler must track all deadlines and the amount of CPU time used by each process thus far.

The schedulability test for EDF is quite straightforward:

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

This means that all deadlines are met if and only if CPU utilization does not exceed 100% using EDF scheduling.

Execution Time	Period
1 sec	8 sec
2 sec	5 sec
4 sec	10 sec

Table 6: Example taken from [4]

From table 6 we can see that the total CPU utilization of that system is 92.5%, meaning it is schedulable under the EDF policy.

- Q8.** Do the critical instant analysis for the above set of tasks under EDF.
- Q9.** If you were writing an EDF scheduler, how often would you have to at least consider updating which task should run? Justify your answer.
- At a fixed time interval like RR
 - Whenever a new task is added
 - Whenever a task is completed
 - B & C above
 - All of the above

Least Laxity First (LLF) [sometimes called Least Slack Time First (LST)]

The laxity (also referred to as “slack time”) of a process is defined as the maximum amount of time a task can wait and still meet its deadline. It is formally defined as:

$$(d - t) - c'$$

where d is the deadline of the task, t is the real time since the task first wished to start, and c' is the remaining execution time. In LLF scheduling, highest priority is given to the task with the smallest laxity. Like EDF, LLF can be used for processor utilization up to 100%. This is also an optimal scheduling algorithm for periodic real-time tasks.

When a task is executing, it can be preempted by another task whose laxity has decreased to be below that of the active task. However, a problem can arise with this scheme in which two tasks have similar laxities. One process will run for a short while and then be preempted by the other task, then priority will switch back, and so on. This causes many context switches to occur during the lifetime of these tasks.

EDF is generally simpler and can schedule exactly the same tasks, so why might we prefer LLF? The main reason is that LLF can often handle unschedulable tasks better than EDF. Once it is clear that a process can't be completed (slack goes negative) it can stop executing the task. This can be useful if your deadlines are soft rather than hard.

Example: <http://www.scribd.com/doc/79592652/22/Least-slack-time-first-LST>

Summary

Now that we examined several of the most popular RTOS scheduling algorithms, we will discuss the advantages and disadvantages between different algorithms. We will only look at the most common used algorithms: RMS, FIFO, RR, and EDF scheduling.

FIFO is the simplest scheduler in that all it requires is a very basic queuing algorithm. It requires no input from the user (such as assigning priorities, an estimation of run time, or period of the tasks) as everything is handled by the RTOS. However, due to FIFO's lack of prioritization, it can fail with a schedule that has a fairly low total CPU utilization. Thus, although FIFO is an easy algorithm to understand, its limited capabilities make it ill-suited for many real-world applications.

RR, like FIFO, also has the benefit of being simple to understand and being nearly as simple to implement. It has the advantage of being "fair" in that all tasks will get an equal share of the CPU and no one task can hog the processor. This fairness is extremely useful on a typical multi-user system (such as a Linux server) but much less relevant for a real-time OS. Its usefulness in a general user system means RR (or some close variant) is a commonly available option.

RMS's fixed prioritization and relative ease of use (once understood) makes it well-suited for use in many applications. The sufficiency condition allows users to quickly determine the schedulability of a group of tasks (although more checks will be needed should the sufficiency condition fail). However, it is not an optimal scheduling algorithm so there are cases where the total CPU utilization is less than 100% yet it won't find a schedulable solution. In addition, if the tasks aren't periodic, RMS won't be an available option.

Unlike the above schemes, EDF can schedule any set of tasks that don't exceed 100% CPU utilization. However, this comes at the cost of a more complex scheduling algorithm...

Real-World Difficulties

Although the aforementioned scheduling policies are commonly used, they aren't free from a few serious issues. Here are a few of the issues that may arise and methods for dealing with those problems.

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.

Priority inversion occurs when a higher priority task is forced to wait for the release of a semaphore that is owned by a lower priority task. The higher priority task cannot run until the lower priority task releases its semaphore, which usually occurs when the lower priority task has run to completion. This effectively then makes the lower priority task a higher priority, and vice versa for the higher priority task.

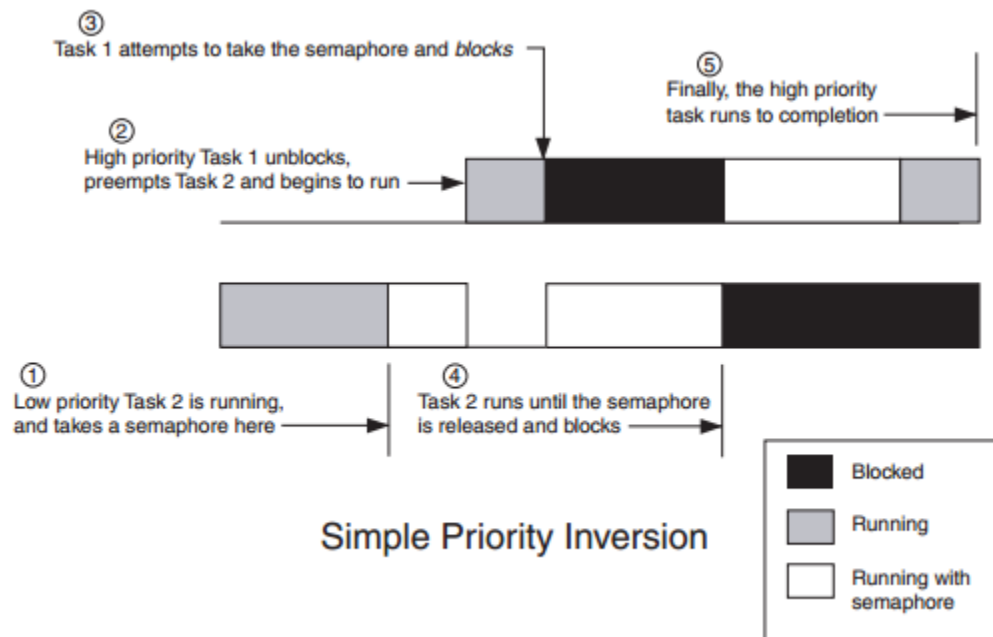


Figure 4: taken from [1]

Priority Ceiling Protocol

In priority ceiling protocol, each task is assigned a static priority, and each resource is assigned a ceiling priority greater than or equal to the maximum priority of all the tasks that use it. During run time, a task assumes a priority equal to the static priority or the ceiling value of its resource (whichever is larger). If a task requires a resource, the priority of the task will be raised to the ceiling priority of the resource; when the task releases the resource, the priority is reset.

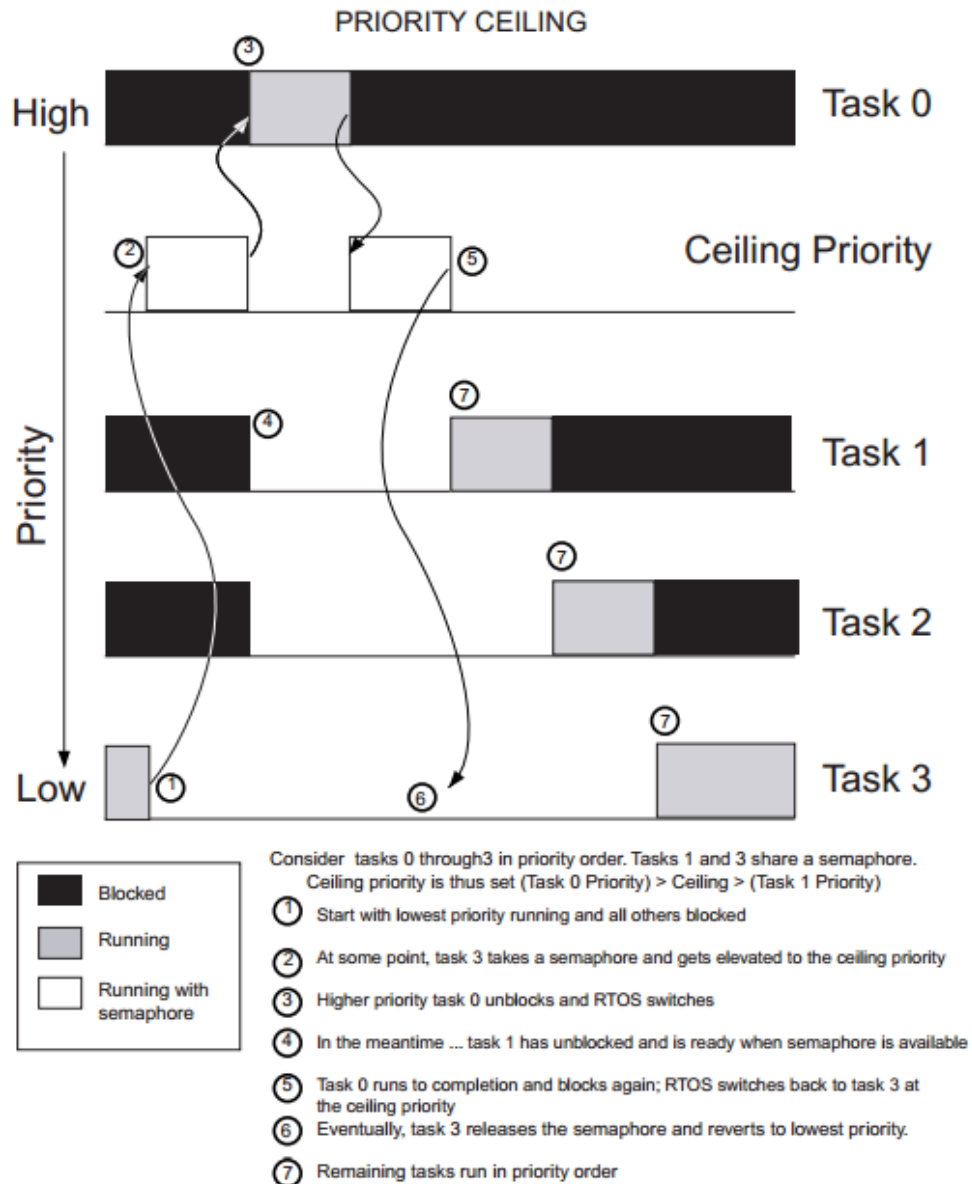


Figure 5: taken from [1]

Priority Ceiling Emulation

The highest priority of any task that will lock a resource is kept as an attribute of that resource. Whenever a task is granted access to that resource, its priority is temporarily raised to the maximum priority associated with the resource. When the task has finished with the resource, the task is returned to its original priority.

Examples: <http://www.eetimes.com/design/embedded/4024970/How-to-use-priority-inheritance>

LLF

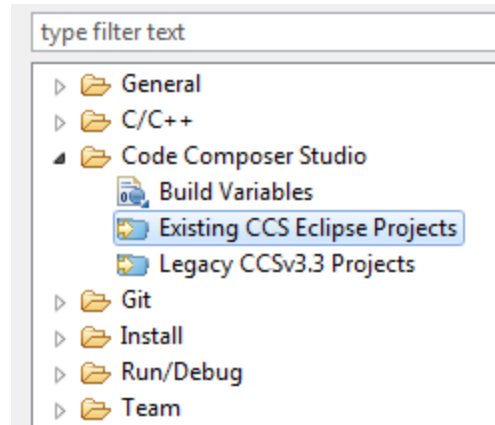
In an LLF scheduler, when a task is executing it can be preempted by another task whose laxity has decreased to be less than that of the active task. However, a problem can arise when two tasks have similar laxities. Task one will run for a short while and then be preempted by task two, and after a short while be preempted by task one and so on. This causes many context switches to occur during the lifetime of these tasks.

Sources

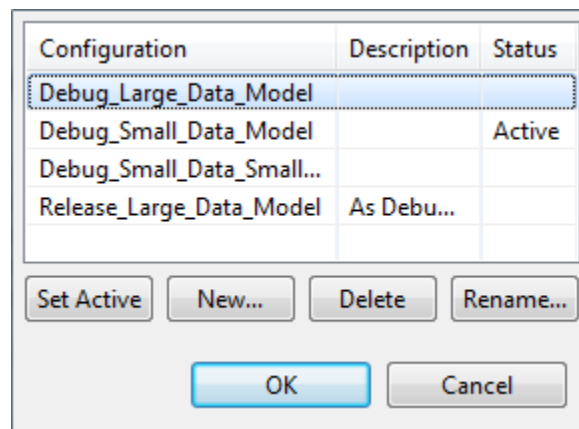
1. http://www.idsc.ethz.ch/Courses/embedded_control_systems/Exercises/SWArchitecture08.pdf
 - o EECS 461
 - o Images taken from here
2. http://cn.el.yuntech.edu.tw/course/95/real_time_os/present%20paper/Scheduling%20Algorithms%20for%20Multiprogramming%20in%20a%20Hard-.pdf
 - o Liu and Layland paper
3. Mohammadi, Arezou, and Selim G. Akl. "Scheduling Algorithms for Real-Time Systems." (2005)
4. http://en.wikipedia.org/wiki/Earliest_deadline_first_scheduling#Example




2. Inlab

Download the lab5_start.zip file from the web. To import the projects into Code Composer Studio, select “Existing CCS Eclipse Projects”. Choose “select archive file” and lab5_start.zip. Two projects will show up in the “Discovered projects” box; highlight both projects. After clicking “finish,” two projects will show up in the Project Explorer box. If you don’t see the Project Explorer box, click “View->Project Explorer” from the menu.



Right-click the Lab5_start folder in the Project Explorer box and click “Build Configurations->Manage...”. In the window that pops up, highlight Debug_Small_Data_Model and click Set Active. Press “OK” and exit the window.



Connect the MSP430 USB Debug Interface to your computer and the ribbon cable to the board. Set the MSP430 board switch in the upper left corner to “FET”. Click the hammer symbol  to build Lab5_start (it may take a while for your first build). Now press the “bug”  to enter debug mode and click the green arrow  in the debug pane to run the code on the board.

A. Taking some basic measurements

Now that you have the basics working, create a short task which uses a for loop to generate some delay. Create the following as the only task running:

```
void taskDT(void *par)
{
    volatile unsigned int i;
    for(;;)
    {
        P5OUT|=0x3;           // set both GPIOs
        for(i=0;i<2450;i++);  //delay high
        P5OUT&=~0x3;         // now clear them
        for(i=0;i<100;i++);   //delay low
    }
}
```

Also, add the following code in the main (right after the hardware setup function call).

```
P5SEL&=~0x03;    // set pin associated with GPIOs P5.0 and
                  // P5.1 to be the GPIO rather than
                  // something else.
P5DIR|=0x03;     // Make the GPIO an output.
```

Run the task as both priority 0 and priority 1 and run the longer delay as both 2450 and 1000. Take measurements for the period of the signal (use the measurement feature of the scope to get stats on the period of the signal)¹. The pins are labeled as 5.0 and 5.1 on port X.Y. Do be sure to reset stats when needed.

Q1. Answer the following questions

- Take a look at [http://processors.wiki.ti.com/index.php/Digital_I/O_\(MSP430\)](http://processors.wiki.ti.com/index.php/Digital_I/O_(MSP430)), section 9.2 of <http://www.ti.com/lit/ug/slau049f/slau049f.pdf> and http://eecs.umich.edu/courses/eecs498-brehob/Labs/MSP430F5438_include.h. Briefly explain what the short code in main is doing.
- Could that code in main go into the task or not? Briefly justify your answer.
- Why is the variable “i” declared as an volatile? When would this matter?
- A single increment of “i” seems to take about how long? This number will be important later...
- What is the impact of changing the priority from 0 to 1? Why do you think that is happening?
- How long is a clock tick? (Look in FreeRTOSConfig.h.)
- On this machine `ints` are 16 bits. What’s the largest value you can have in an unsigned 16-bit number?

¹ You should be aware that you are measuring the signal that is displayed. That sounds obvious, but it means that the level of “zoom” you have can impact your measurements. Also, be sure to “clear statistics” between measurements. Section 14 of <http://cp.literature.agilent.com/litweb/pdf/75019-97051.pdf> is relevant here.

B. Modeling CPU-tasks

Now you are going to create two tasks as shown below. As expected, each will run periodically and each will use a certain amount of CPU time and “sleep” until its time to start has occurred. Note that the following task pair is RM schedulable.

Task	Execution Time	Period
T1	20ms	50ms
T2	10ms	30ms

Rather than doing anything useful, we are running the for loop as the CPU load. In particular we want you to create the two tasks as follows:

- Task1 should run every 50ms and use about 2ms of CPU time (using a for loop) while task2 should run every 30ms and use about 10ms of CPU time. Notice that this is RM schedulable. You'll want the task with the shorter period to have the higher priority. Use priorities 1 and 2.
- Have each task control a different I/O pin (P5.0 and P5.1 by preference). It should raise the I/O pin before it starts the for loop and lower it when it exits.
- Getting a task to be periodic isn't really possible with the `vTaskDelay()` function discussed in class. Instead you will need to use `vTaskDelayUntil()` which is documented at <http://www.freertos.org/vtaskdelayuntil.html>.
- *If you get motivated* (in other words, optionally), try to write a single task that does this and use the parameter option for tasks to get this to work.

- Q2.** Answer the following questions using the scope once you have your two tasks running:
- a. Measure both the period and +width of both I/O pins over 500 or more samples. Record the mean, min, max and standard deviation of the four measurements. Recall the footnote above about measurements.
 - b. Explain why one of the tasks is a lot more consistent than the other.
 - c. Explain the variation seen. Why does the *period* of one of the tasks vary? Why it is both above and below the desired value?
 - d. Do you believe that both tasks are always completing before their deadline (before they are asked to start again). Can you be sure?
 - e. What is the difference between `vTaskDelay()` and `vTaskDelayUntil`

- Q1.** Change T1's period to 40ms. Explain the answers to the following questions to your GSI:
- a. Are both tasks are meeting their deadlines?
 - b. How, if at all, can you observe if deadlines are being met?

Having gotten that done, let's try to get a better picture of what's going on. Right now all we can see is when the task *actually* starts running to when it stops running. What we want to see is the

response time of the task². So what we are going to do is create a (new) high-priority task that does nothing other than set an output high and then wake up the lower priority task (task1) using a semaphore. To do this we'll need three functions³:

- **vSemaphoreCreateBinary(xSemaphoreHandle xS)**
 - This is a macro that properly creates a semaphore. If xS isn't NULL the semaphore was created successfully.
- **xSemaphoreTake(xSemaphoreHandle xS, portTickType xBlockTime)**
 - This macro tries to take the semaphore. If it's not available, it will delay until the semaphore is available or xBlockTime (in ticks) has passed. It will return pdPASS if the semaphore was taken and pdFALSE if not. If you don't want to wake up without the semaphore, set xBlockTime to be fairly large.
- **xSemaphoreGive(xSemaphoreHandle xSemaphore)**
 - Makes the semaphore available for others to take.

Here are a few helpful hints:

- Notice the above three functions need an include file (see the footnote).
- Be sure your semaphore is a global.
- Task1 should be waiting on the semaphore (and not otherwise delaying), the new (third) task should have a priority higher than the others.

62. Show the GSI how you can detect if a deadline gets missed. Explain why you couldn't easily do so in the previous part and what changed. You may need to play around with the various values for a bit.

C. Interrupts

Let us briefly look at interrupts. In FreeRTOS interrupts are largely handled using the platform's tools: in this case it's Code Composer Studio that's in the driver's seat. This does greatly reduce the portability of the code, but frankly if you are dealing with interrupts you've probably got portability problems already. Here is how CCS defines a function that is to be called when an interrupt on a port2 button happens:

```
#pragma vector=PORT2_VECTOR
interrupt void bob( void )
{
```

This will cause bob() to be called when an interrupt is generated from any button in port2. Take a look at http://eecs.umich.edu/courses/eecs498-brehob/Labs/MSP430F5438_include.h again and search port P2OUT. Now look at section 12 of <http://www.ti.com/lit/ug/slau208k/slau208k.pdf> and set things up so that line 4 of port 2 will generate an interrupt on the rising edge. That line is connected to the joystick's press button (just push the joystick in). Write code that simply loops for 2ms before it returns

² **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 to job completion.

³ <http://www.freertos.org/a00121.html> and related pages (linked from that one) detail these three functions.

in the ISR and set break point in your interrupt code to make sure your interrupt is happening.

- Q3.** Looking at the scope, capture an instance of the interrupt code jumping in the way of task2 and show it to your GSI.

Once you have interrupts working (you may need help...) remove task1 and make it so the interrupt causes (low priority) task 2 to run rather than using any CPU time itself. You shouldn't use the **xSemaphoreGive()** function. Instead use the function shown in the lecture notes for programming a deferred interrupt.

- Q4.** Demonstrate to your GSI this working code and explain why we might want handle interrupts in a task rather than in the ISR.

3. Post lab

- Q3.** Consider the desire to use EDF. Look over the task scheduler. How could you use **void vApplicationTickHook(void)** to accomplish this. To answer this question you'll want to look at the task diagram near the end of the RTOS lecture...