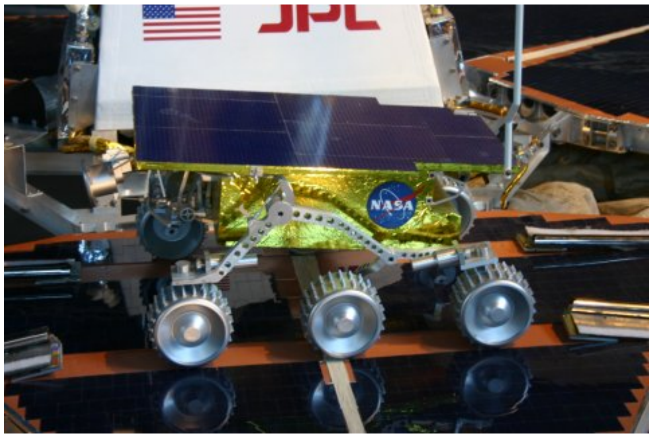
Introduction to Real-Time Operating Systems: Part 4

Call: +44 (0)23 8098 8890
E-mail: sales@itdev.co.uk (mailto:sales@itdev.co.uk)

Posted 26th October 2017, By James H (/company/about/team/james-h)



Introduction

In the embedded world, the use of a Real Time Operating System (RTOS) is commonplace, and with the advent of the Internet of Things (IoT) they are becoming more so. You might be deciding to use an RTOS for the first time, or perhaps you are thinking of moving to a new RTOS. This is the fourth article in our series on RTOSes to help you to get started.

The previous article explained the concept of pre-emption, why an RTOS differs from a general OS in this matter and the reasons for doing so. It asked “when and why can a task not pre-empt another?” and concluded that there were three reasons.

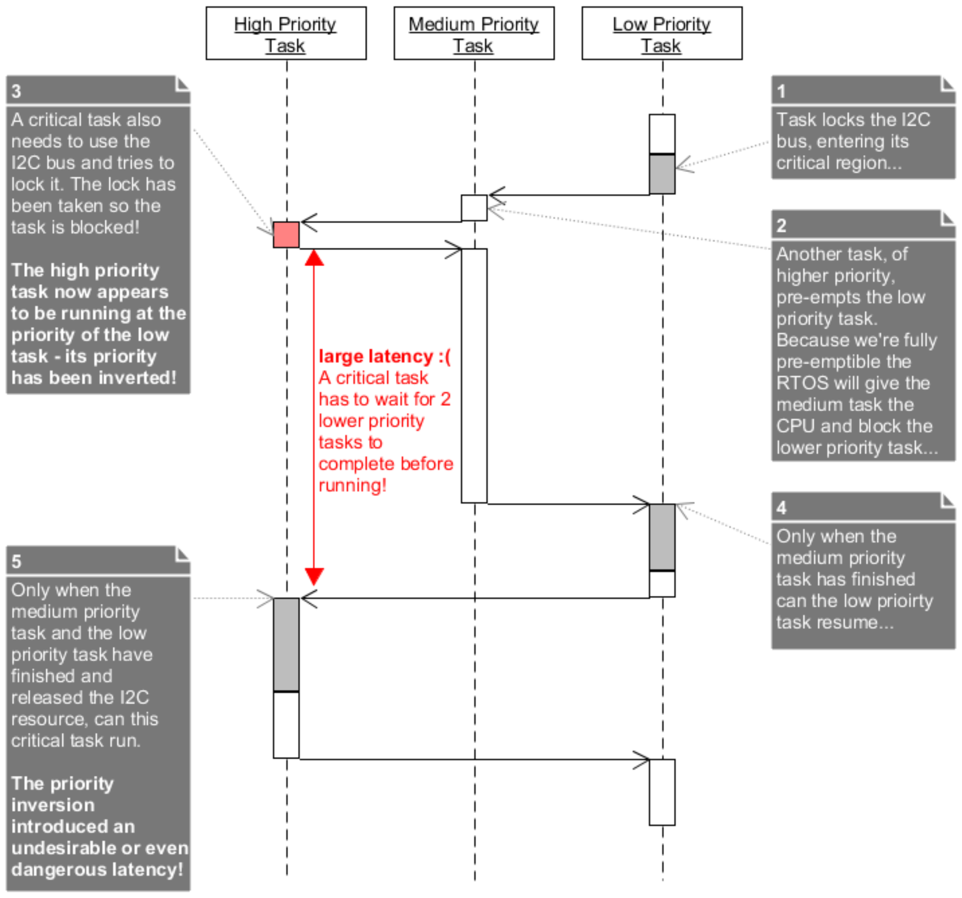
- Lack of pre-emption support
- Interrupts being disabled (for long periods)
- Locks and issues like priority inversion

This article will deal with the third reason, namely priority inversion and its solution, priority inheritance.

We mentioned earlier that being fully pre-emptible can introduce certain problems.

Priority Inversion

The diagram below demonstrates an issue called priority inversion.

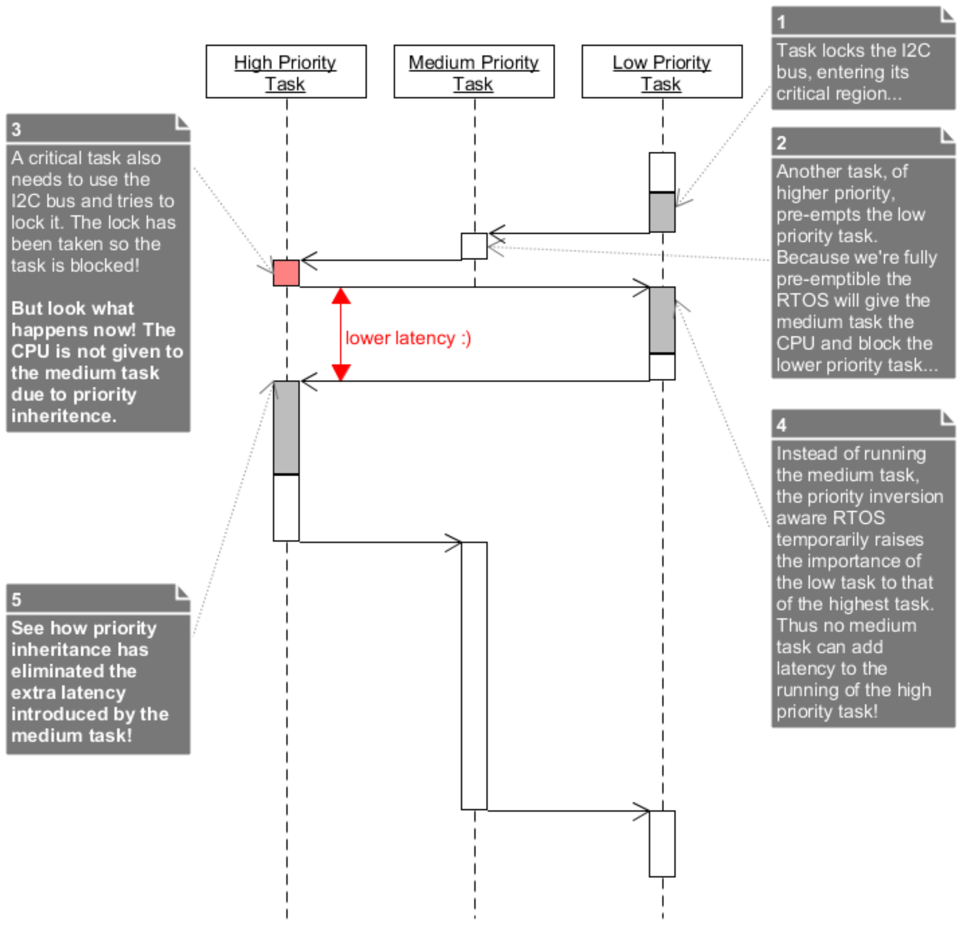

High Priority Task
Medium Priority Task
Low Priority Task
1
Task locks the I2C bus, entering its critical region...
2
Another task, of higher priority, pre-empts the low priority task. Because we're fully pre-emptible the RTOS will give the medium task the CPU and block the lower priority task...
3
A critical task also needs to use the I2C bus and tries to lock it. The lock has been taken so the task is blocked!
The high priority task now appears to be running at the priority of the low task - its priority has been inverted!
large latency :(
A critical task has to wait for 2 lower priority tasks to complete before running!
4
Only when the medium priority task has finished can the low prioirty task resume...
5
Only when the medium priority task and the low priority task have finished and released the I2C resource, can this critical task run.
The priority inversion introduced an undesirable or even dangerous latency!

In the above diagram we see that a low priority task has locked the I2C bus. This is an unlucky but very possible scenario. Whilst it is in the critical region it is pre-empted by a medium priority task. In the meantime another critical task also wants to use the I2C bus, but is prevented from doing so because the bus is locked. This means that the critical task is blocked waiting until the lock becomes available. However, once it blocks, the low priority task cannot run because the medium priority task will be given the CPU based on its priority.

Thus, the low priority task must wait until the medium priority task completes before it can run and release the I2C bus. In this situation **the critical task is running as-if it had the priority of the low-priority task: its priority has been *inverted!*** We call this **unbounded priority inversion** because the low priority task could potentially be stopped indefinitely waiting on however many medium priority tasks are running in the system. In this example we only had one medium priority task, but if we had more, the inversion time could increase almost indefinitely!

This is a problem that only an RTOS faces. Remember, your desktop OS is probably more concerned with throughput than it is with real time responsiveness, so whilst it will support some pre-emption, it might, for example, not allow tasks holding locks to be pre-empted, therefore avoiding this kind of scenario. However, an RTOS has to be as responsive as possible, so the kind of latency we saw introduced above is unacceptable. How do we get around this?

Priority Inheritance


High Priority Task
Medium Priority Task
Low Priority Task
1 Task locks the I2C bus, entering its critical region...
2 Another task, of higher priority, pre-empts the low priority task. Because we're fully pre-emptible the RTOS will give the medium task the CPU and block the lower priority task...
3 A critical task also needs to use the I2C bus and tries to lock it. The lock has been taken so the task is blocked! But look what happens now! The CPU is not given to the medium task due to priority inheritence.
lower latency :)
4 Instead of running the medium task, the priority inversion aware RTOS temporarily raises the importance of the low task to that of the highest task. Thus no medium task can add latency to the running of the high priority task!
5 See how priority inheritance has eliminated the extra latency introduced by the medium task!

The answer comes from something called **priority inheritance**. When a higher priority task is blocked by a lower priority task, the RTOS *temporarily* raises the priority of the low task until the critical task can unblock. At this point the priority of the low task is restored and the critical task is scheduled.

You might notice that there is still some latency. It's not as much as before, but it's still there. We refer to this as **bounded priority** inversion. It is, unfortunately, unavoidable, but the basic priority inheritance scheme shown above minimises it to as little as possible. It is then up to the system designer to make sure that the time tasks spend in their critical regions are as small as possible.

The trouble is that not only has the RTOS had to become more complicated in order to support priority inheritance, those who develop for RTOSes also have to be aware of such issues, so the technical burden is increased and part of the burden has to be borne by the developer, who may or may not be as expert as the RTOS provider.

In fact, even NASA can get it wrong, as was demonstrated by the infamous 1997 Mars Pathfinder problem [1], which was caused by exactly this kind of issue. So if anyone ever tells you priority inheritance isn't rocket science, you'll know better (cue laughter). The Pathfinder problem was explained as follows.

> "The higher priority … task was blocked by the much lower priority ASI/MET task that was holding a shared resource. The ASI/MET task had acquired this resource and then been pre-empted by several of the medium priority tasks…" [2]

The example of increased latency in the previous article, caused by the interrupt masking, is also a type of priority inversion. It, however, cannot be solved using the above method.

So far this series has covered the "bread and butter" of real-time systems: the fundamental issues that they help solve. However, the scheduler isn't the only aspect of a good RTOS and the OS as a whole is not just the kernel but also the infrastructure supporting the user that sits on top of the kernel. The next article will discuss some other features that should be noted in an RTOS.

References

1. M. Jones, 07 12 1997. [Online]. Available: http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/authoritative_account.html (http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/authoritative_account.html).
2. M. Mike Jones, "Real-World (Out of This World) Story," [Online]. Available: https://www.microsoft.com/en-us/research/people/mbj/ (https://www.microsoft.com/en-us/research/people/mbj/).

Attributes

- Photo: "Mars Pathfinder Lander and Sojourner Rover (https://www.flickr.com/photos/ideonexus/3732031105)" by Ryan Somma (https://www.flickr.com/photos/ideonexus/) is licensed under CC BY-SA 2.0 (https://creativecommons.org/licenses/by-sa/2.0/)

< prev (/blog/introduction-to-real-time-operating-systems-part-3) next > (/blog/introduction-to-real-time-operating-systems-part-5)

(http://www.nmi.org.uk/)

(https://www.theiet.org/)

(http://accu.org/)

(http://www.ktn-uk.co.uk/)

Latest Blog Posts

(/blog/reflection-2020)

A Reflection on 2020

(/blog/reflection-2020)
Posted 29th January 2021, By Jon O
(/company/about/team/jon-o)
Having started 2021 much as we ended 2020, we thought it would be good to look back to before the pandemic and reflect on how the year turned out against ...more
(/blog/reflection-2020)

(/blog/universal-verification-methodology-design-reuse)

Universal Verification Methodology: design for reuse

(/blog/universal-verification-methodology-design-reuse)
Posted 5th January 2021, By Tom J
(/company/about/team/tom-j)