Introduction to Real-Time Operating Systems: Part 2

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

Posted 12th 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 may be deciding to use an RTOS for the first time, or perhaps you are thinking of moving to a new one. This is the second article in our series on RTOSes to help you to get started.

The previous article talked about the basics of real-time systems and about the "spectrum" of real-time, from "soft" to "hard". This article begins to answer the questions "how do we get real-time?" and "why isn't a general OS real-time?"

Firstly remember that "real-time" is not about performance but predictability. The better question might be, why isn't my normal desktop operating system real-time?

Pre-emption

The answer to this lies in something called "pre-emption". The standard dictionary definition goes something like "pre-empt: to acquire, to replace with something considered to be of greater value or priority" [1]. This isn't far off the RTOS definition: pre-emption is the ability to take a task off the CPU at any time and schedule in another, more important task, with as little delay as possible.

What causes a lack of ability for one task to pre-empt another?

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

This article will define & discuss the concept of pre-emption. Interrupt disabling and priority inversion will be discussed in following articles.

In the early days of Linux and Microsoft, for example, when a system call was made, as soon as the kernel code was executing, the kernel task in progress would not relinquish the processor until it had completed. This would mean that had the "car collision detect" event, talked about in the previous article, occurred whilst our MP3 player was running some filesystem API code in the kernel, we might have run someone over!

So why on earth was it designed like this in the first place, you may ask? Mostly for simplicity, but also because a desktop operating system is designed for a different purpose. To be pre-emptive a kernel has to add decision points into its code, where it can decide whether or not another task should be put on the CPU. As such, the kernel becomes more complex.

There is also more work to be done on a more frequent basis. Each time a task is pre-empted, the OS must save the task's state so that it can be restored later. Because the kernel has become more pre-emptible, this could potentially happen more often, hence a greater need for optimisation in this respect. The OS must also decide whether there are more important tasks than the one currently running. The scheduling algorithm will thus become more complex, requiring additional data structures, and checking algorithms, etc., to keep track of everything.

It must also decide if the system is in a state where pre-emption can occur. For example, if a task is holding a lock, perhaps it shouldn't be pre-empted. In this instance, the operating system is no longer completely pre-emptive. To get over this constraint, even if a task holds a lock, the OS could allow it to be pre-empted. But then how does the system account for effects like priority inversion (discussed in following articles) which could make the scheduling and other OS primitives like semaphores yet more complex to implement?

There would also be times when a process simply should not be pre-empted: for example, a process writing to an I2C device must never be interrupted by a process writing to another I2C device on the same bus. Now the driver developer needs to pay more attention to the locking primitives they are using. Do they use a pre-emptible lock or one that cannot be pre-empted? This leads to further complexity, and worse, complexity that is outside of the control of the core kernel developers. A third party driver that is written badly could still cripple a system's ability to pre-empt it by, for example, disabling pre-emption, holding a lock that cannot be pre-empted during a very long running operation, or even worse, doing the unthinkable: disabling interrupts.

So, one reason we started off with kernels that were not pre-emptible was simplicity. As hardware and software complexities grew, and the complexity of what was required of the system grew, most kernels moved towards a pre-emptible model. However, some are more so than others depending on how "fine grained" their pre-emption points are. For example, the Linux 2.6 kernel introduced more pre-emption points, but it could not be considered fully pre-emptible in the same way a true RTOS is, at least without

the PREEMPT-RT patch applied:

> *"PREEMPT.RT reworks the kernel "spinlock" locking primitives to maximize the pre-emptible sections inside the Linux kernel. (PREEMPT.RT was originally called the Sleeping Spinlocks Patch.)"*[2]

It is worth noting that the real-time aspect of Linux has been managed by the Linux Foundation since 2015, and over 80% of the patch has been now integrated directly into the mainline kernel with the remaining code being steadily maintained.

Another reason is that there is often a trade-off to be made between throughput and responsiveness. If the system wants to focus on high throughput, it is desirable that tasks are not taken on and off the CPU too often, otherwise the OS could be spending more time and effort context switching rather than getting actual work done, thus reducing throughput. However, if tasks are not switched often enough there will be a responsiveness issue. This is another reason why your average desktop OS is not an RTOS: it is trying to accomplish a different goal.

Now we know about the basics of pre-emption, what it is and why a general OS will not be as responsive as an RTOS. The next article will talk about why disabling interrupts is also a barrier to pre-emption and general system responsiveness. It will also demonstrate how RTOSes deal with this.

To receive an update when the next blog in this series is available please follow us on LinkedIn (http://www.linkedin.com/company/itdev/), Facebook (https://www.facebook.com/itdevltd)or Twitter (https://twitter.com/itdevltd).

References

1. https://www.merriam-webster.com/dictionary/preempt (https://www.merriam-webster.com/dictionary/preempt)
2. E. Brown, "Real-time Linux explained, and contrasted with Xenomai and RTAI", 10 02 2017. [Online]. Available: http://linuxgizmos.com/real-time-linux-explained/ (http://linuxgizmos.com/real-time-linux-explained/).

< prev (/blog/introduction-to-real-time-operating-systems-part-1) next > (/blog/introduction-to-real-time-operating-systems-part-3)

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)