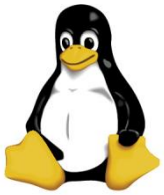

Implementing Real-Time Linux

By: Klaas Hofman

Abstract

Using Linux as a Real-Time Operating System (RTOS) might sound strange to someone familiar with the Linux kernel. In an RTOS, important tasks are guaranteed to be executed as soon as possible, at the expense of less important tasks. However, this is not how Linux has been designed.

The Linux kernel features a process scheduler named Completely Fair Scheduler (CFS), which as its name implies, tries to distribute time slices as fairly as possible over running processes.

RT-Linux, short for Real-Time Linux, attempts to move away from such time-sharing approach and implements real-time behavior. A crucial part, discussed in this article, involves reducing latencies wherever possible, in order to ensure that critical jobs are run without much delay.

Embedded Systems

If we take a look at typical embedded Linux applications we see a lot of examples that require the opposite behavior to real-time, namely optimized network throughput. To obtain better throughput you have to make sure that the task which is responsible for processing the network traffic gets enough time to get the job done. One approach to do that is to increase the scheduling latency and thus reducing the responsiveness of the system. Hence throughput and real-time are contradictory requirements.

Implementations

There are 2 different approaches to implement real time in Linux.
One approach is to combine Linux with an RTOS and a software layer that delegates all the real-time work to the real-time OS and the non-time-critical work to Linux. Sometimes the RTOS is implemented in hardware (for example in an FPGA).
Another approach is to make changes directly in the existing kernel.

For the rest of the article we will discuss this second approach, more specifically the Linux RT patches, or in short RT-Linux.

Understanding Latency

In a real-time application the delay between some (hardware) event and its handling by the software, called latency, is an important factor. For a Linux system the latency can be defined as the sum of the following terms:

kernel latency = interrupt latency + interrupt handler duration + scheduler latency + scheduler duration.

To guarantee that critical tasks are executed in time, a real-time OS will try to keep worst case kernel latency as low as possible. In practice a 'normal' (non-real-time) version of the Linux kernel will have a kernel latency in the order of milliseconds, RT-Linux in the order of microseconds. Let us have a look at the individual latencies and how RT-Linux tries to minimize them.

Interrupt latency is created because a new interrupt cannot be handled immediately, for example because a device driver might have temporarily disabled interrupts to prevent concurrent access, implemented by the spinlock_irq()-call. RT-Linux solves this by replacing the spinlock-calls by regular mutexes, this way concurrency is resolved by the mutexes and interrupts are not disabled.

Interrupt handler duration. In Linux interrupt handlers are split up into 2 parts. A top half that is executed as soon as the interrupt was triggered by the hardware, and a bottom-half that will be executed when all pending top-halves were finished. Top-halves have interrupts disabled. The execution time of the bottom half cannot be defined. If you need a real-time interrupt use the top-half, bottom-half handlers cannot be used. RT-Linux solves this by using threaded-interrupt handlers. These handlers only check in their top-half if there is a bottom-half registered and all the 'handler-code' is moved to the bottom half. While this approach adds a little extra latency the interrupt handlers are now fully preempt-able and have their interrupts enabled hence reducing the interrupt latency. Also the threaded-interrupt handlers are controlled by software so it is possible to use priorities amongst interrupt handlers to further reduce latency in case there is extra load.

Scheduler Latency. This is the latency when switching from one process to the other. An important aspect is the preemption, the possibility of the OS to immediately schedule out a low priority process in favor of a higher priority task. Whether your kernel code is preempt-able or not depends on configuration, but when it is turned off the scheduler will keep running kernel code until an entire time slice was passed or another interrupt came in. This may leave the schedule of your real-time task unbounded. RT-Linux solves this by making the entire kernel code preempt-able.

Latency created by the scheduler and other non-deterministic factors.

The combination of virtual memory, on-demand data-load-and-execute and hardware caches leads to a great uncertainty in the execution-time of applications. Not to mention the fact that many standard libraries are not designed with real-time constraints in mind. There is no easy way to solve this besides quantifying this latency. Fortunately this latency is typically much smaller than the other ones.

Conclusion

While the number of use cases where RT-Linux is a solution are limited and while active development on RT-Linux has recently stalled (although there seem to be a revival on using RT-Linux in drones !), experimenting with RT-Linux is a very interesting way to get a better understanding of the Linux OS and especially the process scheduler. This knowledge can be of use also in a non-real-time context: identifying and troubleshooting network-performance bottlenecks, device driver development, hardware CPU selection, etc.

References

[1] Realtime in embedded systems, Free Electrons, http://free-electrons.com/docs/realtime/

[2] Linux Kernel Development, Robert Love, Addison Wesley, 2010

[3] Moving interrupts to threads, https://lwn.net/Articles/302043/