Linux - Not real-time!

Date: 16.01.2015

Author(s): Michal Koziel

Abstract

A system is said to be real-time if it can respond to external triggers and perform periodic tasks with deterministic timing. The interrupt handling mechanism in the Linux kernel is efficient, but also fair to the least time critical tasks. Hardware interrupt handlers should in general only register events, to avoid blocking the processor longer than necessary, and defer related tasks to softirqs and tasklets. Both softirqs and tasklets are executed with higher priority than user and kernel background tasks. However, the Linux kernel can occasionally change the prioritization to favor less important tasks, dramatically increasing latency for handling of the real-time events.

Introduction

In this article I will explain what impact the interrupt handling in the Linux kernel may have on real-time performance. You should keep in mind that other kernel features may also have consequences, but they are not covered here. Although the Linux operating system (OS) is not an RTOS, does not mean that you cannot use Linux in your real-time application, as long as you are aware of some pitfalls. Other kernel features and available real-time patches or APIs that are not part of the standard Linux kernel, will be subject for another article.

All examples in this article are executed on Zedboard rev C (Zynq XC7Z020CLG484 with only one core enabled, no SMP) and Linux kernel 3.4.12 (petalinux 2014.2). The kernel timer frequency is changed from 100Hz to 1kHz. Softirq used in examples is assigned highest priority. Otherwise all default settings are kept as in the board support package Avnet-Digilent-ZedBoard-v2014.2-final.

Background

Linux is a complete, portable and scalable operating system. It is intended to deliver high overall performance emphasizing reliability and security. This makes Linux a good choice for server as well as desktop applications. Linux is now also widely used in embedded applications.

The availability of low cost 32-bit microcontrollers and processors combined with all the features of a mature OS is tempting many designers to migrate from simple bare-metal or micro kernel RTOS to embedded Linux. A reputation of high performing kernel soon develops cracks when a ported application is put up to the test. The result is usually high interrupt latency and jitter. Solutions often used are temporary as the problem is rather covered than fixed. The most commonly used solution is to modify the application or the Linux kernel to just meet the real-time requirements, even if the system architecture is compromised. Another solution is to boost the processing power by upgrading to a faster processor. Both are poor, short term solutions for the following reasons:

- *Tested and real-life external conditions are rarely the same:*
  A system can always be tuned up to pass tests in the lab by following a trial and error approach. Real-time performance in the Linux kernel is highly affected by variations in system load, as I will illustrate in this article.

- *Bug fixes and further application development:*
  Any change to a system, no matter how small, will affect real-time performance. For a non RTOS the impact is hard to predict, if at all possible.

- *Linux kernel and third-party software maintenance updates:*
  Even the most mature system will at some point require maintenance. A newer kernel or third-party software might have been redesigned extensively and any application tune-up have to be redone with unpredictable results. A third-party software upgrade with closed code makes this job much harder.

Interrupt handling in Linux kernel

Interrupt handling is one of the most important parts in an RTOS. I will here cover interrupt handling in the Linux kernel with respect to real-time performance.

IRQ and FIQ

A hardware interrupt request (IRQ) can preempt the kernel or a user process and trigger execution of an IRQ handler. Interrupt nesting was disabled in kernel version 2.6.35 so an IRQ handler cannot be preempted by another IRQ, not even of higher priority. Keep in mind that interrupts are not only disabled while executing an IRQ handler, but also in many parts of the kernel and drivers. Code executed while interrupts are disabled is a latency source for your IRQ handler.

The Linux kernel also supports fast interrupt requests (FIQ). Both IRQ and FIQ handlers run in interrupt context and are not possible to reschedule, thus sleeping or blocking in any other way within an interrupt handler is not allowed. The difference is that a FIQ can preempt an IRQ or a section in kernel code where IRQs are disabled. A FIQ handler cannot in addition call the kernel API or the device driver framework.
FIQs are rarely used, and usually not by the kernel itself. Typically, FIQs are used in video/audio and other real-time drivers where only a tiny part requires real-time response.

Top half and bottom half

Good practice for interrupt handling is writing small handlers for the most timing critical work. Linux strongly encourages splitting the interrupt handler into two parts, referred to as top half and bottom half. Top half is the interrupt handler itself which does what is absolutely necessary with respect to the hardware and timing and defers the rest of the work to be done outside the handler. Bottom half does the rest of the work, but executes at a later time at lower priority. There are three main mechanisms for the bottom half, softirqs, tasklets and workques. These are briefly described below.

Softirqs

Softirqs, short for soft interrupts, are used for the most time critical bottom half cases and are allocated and assigned priorities at compile time. Softirqs can be preempted by IRQs, but not by the kernel scheduler. Softirqs cannot sleep or copy data to user space, but are allowed to block. The Linux kernel supports up to 32 softirqs, where the first few are reserved typically for high priority tasklets, networking and timers, but varies among Linux distributions and processor architectures.

Tasklets

Tasklets are based on softirqs and run in soft interrupt context. The most important difference between tasklets and softirqs is that tasklets can be created at runtime and can be assigned only one of two priority levels.

Workques

Workques is simply a list of threads to be scheduled and executed in process context. Workques are used for the bottom half implementation.

Ksoftirqd

A situation where there is always a softirq handler ready to run might occur. Since softirq handlers are executed with higher priority than the kernel and user code, they will monopolize the cpu, only preempted by IRQs. To avoid starvation of lower priority processes the Linux kernel allows execution of only 10 softirq handlers in a row. The remaining softirqs are processed by a special thread, ksoftirqd, that is awakened and runs with user level process priority. Consequently, softirq priority is reduced to a lower level than most processes, thus increasing softirq latency.

Top half and bottom half in action

In Figure 1 illustrates the execution of the top and bottom half of a n IRQ handler. The top half pulls a hardware pin low once it starts execution, followed by raising a softirq flag and finally pulling the pin high on exit. The pin logic level is illustrated by the blue signal. The bottom half, implemented as a softirq handler, toggles another pin on entry and exit. The pin logic level is illustrated by the yellow signal. Most of the work is allocated to the bottom half resulting in much longer execution time. Figure 2 shows the variance of both signals over a period of 10s.
The signal triggering the IRQ is not shown, and consequently the IRQ latency is unknown. Although not to be ignored in a real case, this is of no relevance for illustration of the relation between the top and bottom half.
The small bottom half jitter is acceptable in this case, but after a while the picture is much different, as shown in Figure 3 and 4.

Figure 3 and 4 shows the same signal as in Figure 1, but over a period of 5 minutes. In Figure 3, the falling edge of the signal toggled by the top half is used as a trigger, while in figure 4 the falling edge of the signal toggled by the bottom half is used as a trigger. The latency and execution time of the bottom half are sporadically higher. The bottom half worst case latency increased from 10us to 75us, while the worst case execution time increased from around 70us to above 100us, for the 10s and 5 minutes periods respectively. The increase is caused by the occurrence of other IRQs. This case is not specific to the interrupt handling in the Linux kernel and can also be observed in bare-metal applications even with only two IRQs enabled. While this is fairly easy to resolve in a small bare-metal application, it can be frustrating to deal with in Linux. Figures 3 and 4 show observation during a relatively short period of time and with minimal system load. It is hard to predict the worst case latency and execution time knowing that in the Linux kernel even an interrupt handler of lowest priority can execute as long as it wants, thus delaying all other activity.

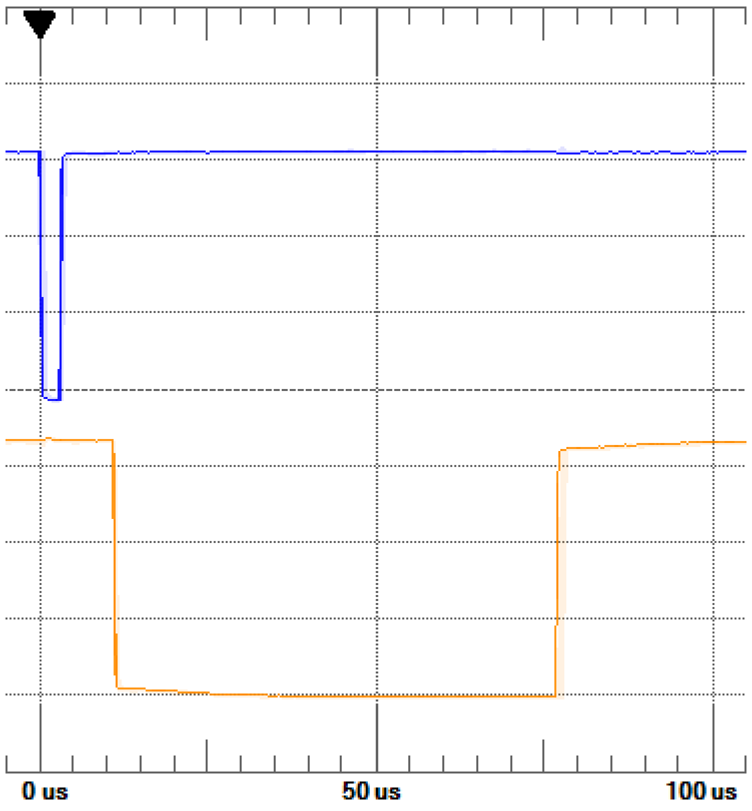

Figure 1. Top half and bottom half execution represented by blue and yellow signals respectively.

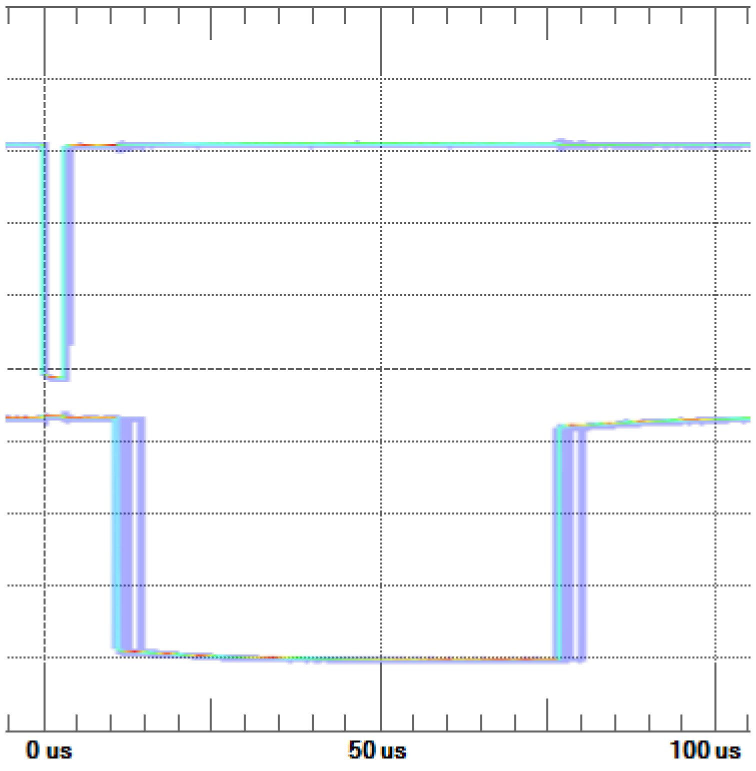

Figure 2. Variance for signals in Figure 1 over 10 seconds.

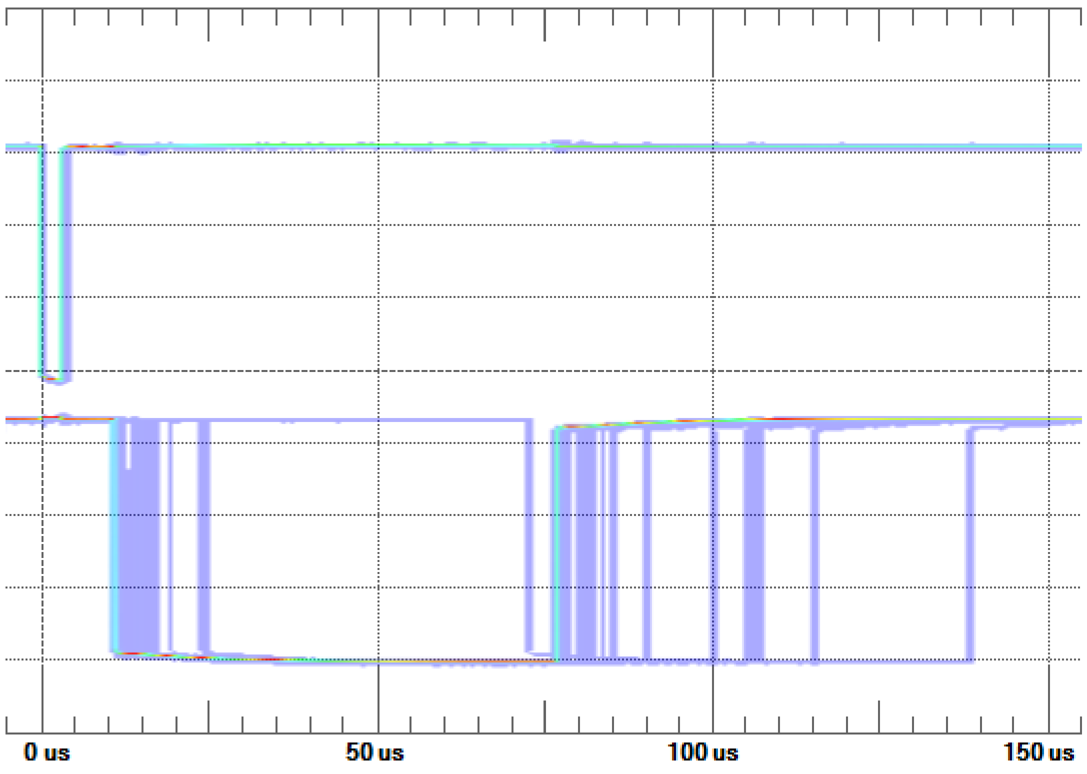

Figure 3. Variance for signals in Figure 1 over 5 minutes.

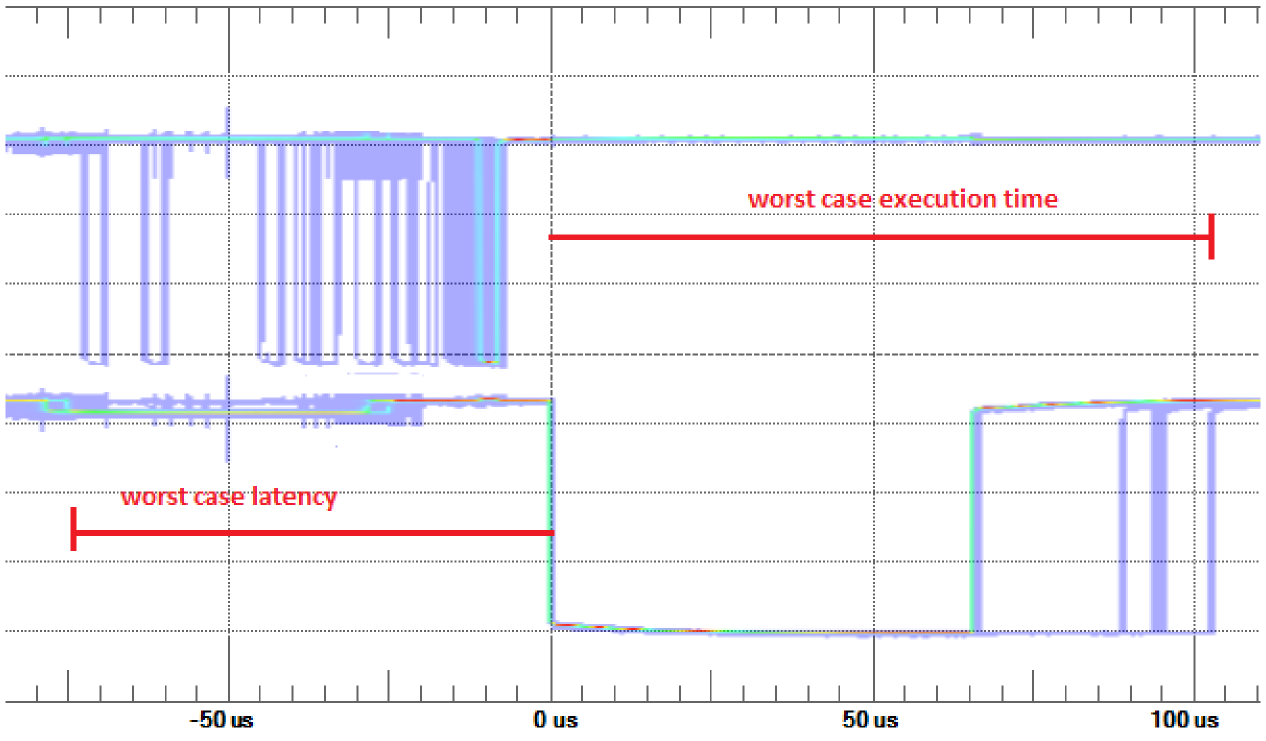

Figure 4. Signals as in Figure 3, but bottom signal is used as trigger. Worst case latency and execution time of bottom half observed over 5 minutes.

Impact of ksoftirqd

A Softirq can occationally be raised (tasklets can be scheduled) faster then they can be processed. Softirq flags are not cumulative, raising a softirq flag when it's already raised does not make the softirq execute more than once. This means that you have to keep track of how many times you raise the flag and how many times the softirq has been executed if you cannot afford to miss any softirqs. Note that a softirq flag can also be raised from any part of the kernel code, not only by an IRQ handler. Figure 5 shows an IRQ handler executed 16 times, each raising a softirq flag, followed by execution of only 14 softirq handlers. At two points, an IRQ handler raised a softirq flag when the flag was already raised. It happened once, just after the ksoftirqd thread was awaked, but before the softirq handler flag was taken down. The second time happened when two hardware interrupts where received during execution of the same softirq handler.

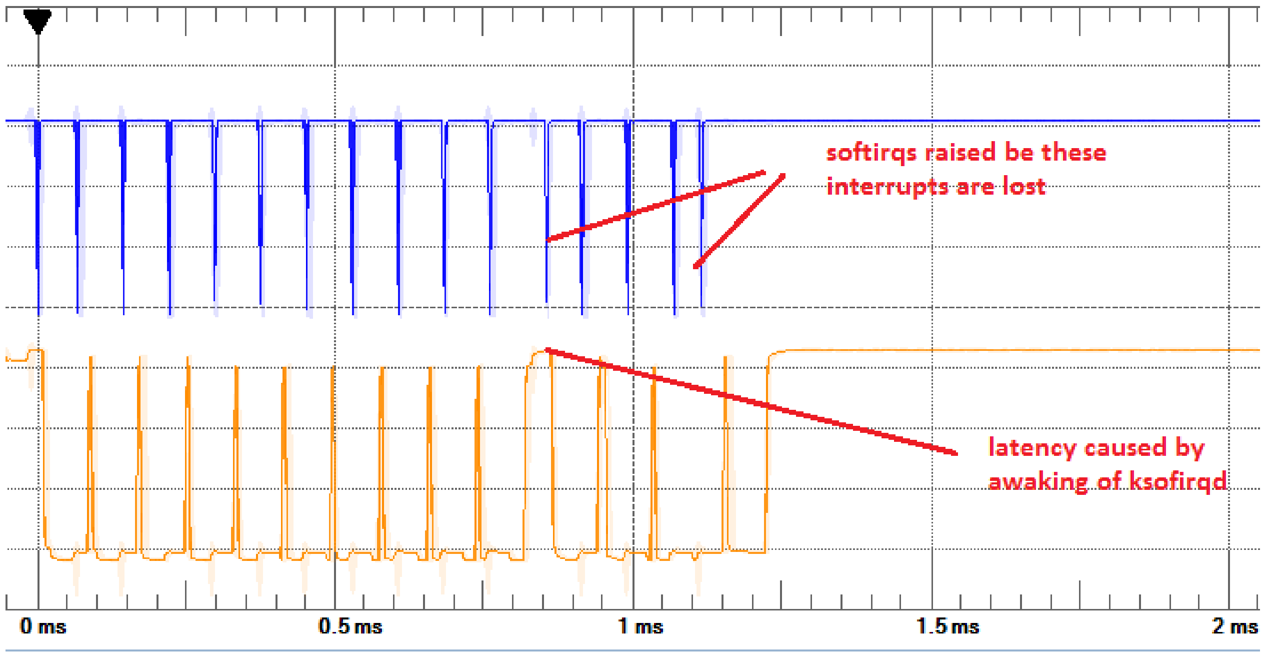

Figure 5. Loss of softirqs cased by frequent IRQs. 16 IRQs (blue) and 14 softirqs (yellow) are handled.

To avoid loosing softirqs you can increment a counter in the IRQ handler and decrement it in the softirq handler. If the counter value is positive and the softirq flag is not raised on softirq handler exit, the softirq handler can reactivate itself by raising its flag. Remember that this might awake the ksoftirqd thread if done too many times in a row. Figure 6 shows the same case as in Figure 5, but with the counter implemented. Now the same number of IRQ and softirq handlers is executed. The increased latency of the last few softirq handlers is nothing but expected due to the total amount of work in this period of time. The latency introduced by the ksofirqd thread is negligible compared to the latency for the last softirq. Increasing the processing power would sure reduce the latency, in particular for the last few softirqs. This situation is however observed under minimal system load. Under high system load, things get much worse.

Figure 7 shows how a spinning high priority user process can dramatically increase the latency of softirqs. Note that the IRQ handlers seem to be unaffected by the load, which can be misleading if only IRQ and not also softirq latency is investigated. In this case, the ksoftirqd thread was already awakened after processing of the third softirq handler. This shows that other softirqs can contribute to awaking of the ksoftirqd thread. Here, the ksoftirqd have only one softirq handler to process before it goes back to sleep resuming fast softirq handling. The ksoftirqd thread is however again awakened after handling 10 softirqs in a row. This time it is less lucky and it takes long before all softirq handlers are processed.

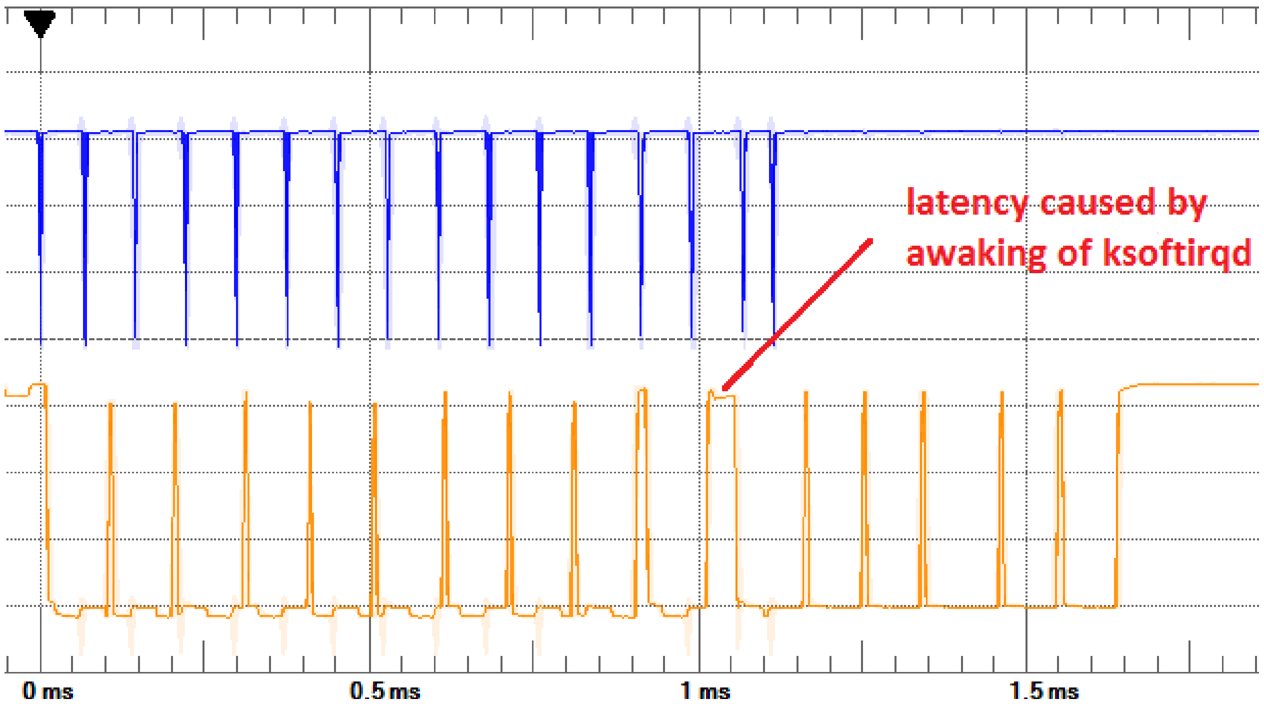

Figure 6. Execution of 16 IRQ handlers and 16 softirq handlers under low system load.

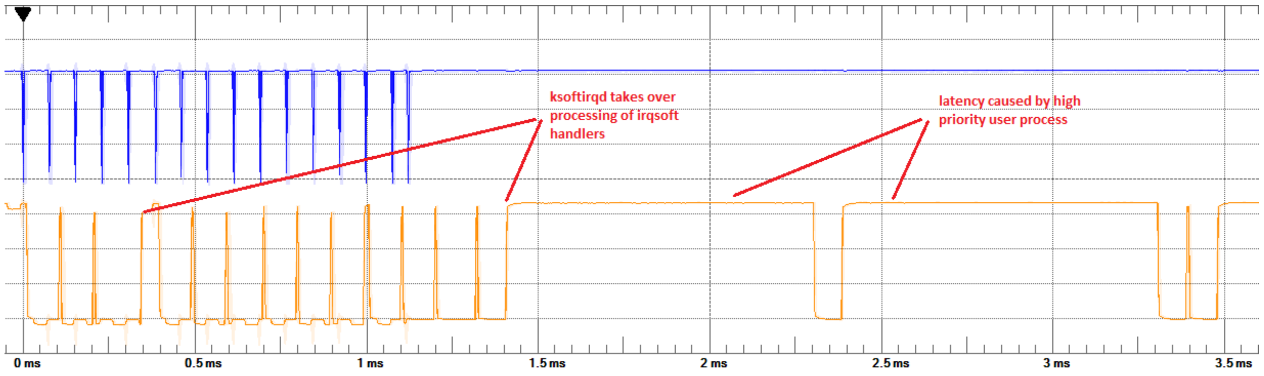

Figure 7. Execution of 16 IRQ handlers and 16 softirq handlers under high system load caused by a high priority user process.

Conclusion

The Linux kernel has a refined interrupt handling mechanism. As I illustrate in this article, the Linux philosophy of fairness is still dominating. The dynamic change of softirq priority makes interrupt handling following the top and bottom half approach highly unpredictable. As I mentioned in the article, even an IRQ handler may suffer from high latency due to interrupt disabling by the kernel or another driver. As in any other system, to be sure that latency and execution time for your real-time code will not exceed allowed values, you have to be aware of all potential latency sources. Interrupt disabling should be a well known source, handling of softirqs is clearly demonstrated to be another significant latency source. Keep in mind that once you need to execute your real-time code outside an IRQ handler, your code executes in a fair, but unpredictable environment. Linux can however still be used in real-time applications where the increased latency of softirq handlers is not an issue.