Interrupt Handling on x86 (RT) and Boot Interrupt Quirks

Alexander Graf, Olaf Dabrunz, Stefan Assmann
SUSE Linux Products GmbH
Maxfeldstr. 5, 90403 Nürnberg, Germany
{agraf,od,sassmann}@suse.de
January 13, 2009

Abstract

This paper details how the Linux kernel manages interrupts on the x86 platform and explains threaded interrupt handling, as used in the real-time Linux kernel. Threaded interrupt handling has been troubled by interactions with some hardware mechanisms since its introduction in real-time Linux. For that reason we describe mechanisms, sometimes called “boot interrupts”, that route interrupts coming from secondary buses to the primary interrupt controller in the system, when the system is considered to use the legacy interrupt controller (PIC). These boot interrupts are responsible for unexpected interrupts on many systems that actually use non-legacy interrupt controllers (APICs) when interrupts are handled in threaded mode. Solutions on how to make boot interrupts disappear on many chipsets are presented.

1 Introduction

During the execution of a program, an event that needs attention might happen on a device [1]. For example, this may be when a new package arrives on the network, when a key is typed on the keyboard, or when the sound card is done playing a chunk of data. When an interrupt occurs, the kernel’s interrupt service routine is executed. Non-real-time Linux handles an interrupt completely inside the service routine.

Real-time Linux, on the other hand, tries to reduce latencies for prioritized tasks, ensuring that a process is not interrupted for too long. Interrupt handling was redesigned in real-time Linux, in order to allow tasks to be assigned a higher priority than the interrupt handler. This rewrite introduced masking on interrupt lines, leading to unexpected side effects. This document will explain how, when, and why such unexpected interrupts occur and proposes how to circumvent them.

2 Interrupt Handling

2.1 Vanilla

The non-realtime kernel as is handles interrupts completely in one go. While the interrupt is being handled, no process can run on that CPU. After the interrupt is handled completely, it is turned off on the device, interrupt reception enabled again, and the scheduler triggered, allowing the program to continue. This is the cause of major latencies that real-time Linux attempts to address.

2.2 Real-time

In order to reduce these latencies, real-time Linux introduces the concept of “threaded interrupt handling”. When an interrupt arrives, vanilla Linux blocks the CPU until handling the interrupt is finished. Threaded interrupt handling, on the other hand, only blocks the CPU while waking up a thread to do the processing. This thread is responsible for handling the interrupt, but a higher prioritized process can now run on the same CPU, even though the handler is is not finished yet. Because the interrupt is still active on the device and could trigger again as soon as interrupts are reenabled, the interrupt line the device is on needs to be disabled until the interrupt is completely processed. (See Figure 1 for a comparison of these two approaches.)
3 Interrupt Routing

3.1 Internal Bridge

When a device generates an interrupt, it sends a signal on an interrupt line. This signal is then processed by an Interrupt Controller. The x86 architecture knows two of these Interrupt Controllers: the PIC and the APIC. The PIC (Programmable Interrupt Controller) has been used in PCs since the very beginning. By coupling two interrupt controllers that support 8 lines each, a total of 15 interrupts \(^ 1\) is supported by using the PIC. The PIC is also coupled with one specific CPU, thus binding interrupt processing to one CPU. It is also not able to do software configurable interrupt priorities.

In order to circumvent these restrictions, Intel invented the Advanced Programmable Interrupt Controller (APIC). The APIC is split into an IO-APIC, usually resident on the southbridge, and a LAPIC that resides within the CPU. The device sends an interrupt to one of the 24 lines of the IO-APIC, which in turn sends a message to all LAPICs in the system that an interrupt was triggered. [2]

In order to support legacy operating systems that are not able to use the APIC, the interrupt arrives on the PIC and the APIC alike. During bootup, the APIC is completely masked \(^ 2\) while the PIC is used to receive interrupts. As soon as an APIC aware operating system boots up, it masks the PIC and unmasks the interrupt lines it wants to receive on the APIC. (Figure 2 shows interrupt routing in the latter case.)

---

\(^1\) One interrupt line is used for coupling the devices, so 8+8-1 makes 15 available lines.

\(^2\) When an interrupt line is masked, no incoming interrupt on that line will be recognized by the interrupt controller. Masking an interrupt controller completely means that all interrupt lines on this controller are masked.
3.3 Boot Interrupts

Devices from an additional PCI Bridge are not linked to the legacy PIC. But bootloading code does not know anything about APICs, as it relies on the interrupts to arrive on the legacy PIC. To still be able to boot from such devices, PCI Bridges forward an interrupt request to the southbridge when the IO-APIC is masked. This behavior is called boot Interrupt in the Intel documentation[4, 3] and shown in Figure 4. Other bridge vendors implement boot interrupts as well, but may refer to them differently or not at all.

Figure 4: Boot Interrupt (Normal)

Linux does not usually utilize the PIC, disabling it and using only the APIC to receive device interrupts. As described before, real-time Linux needs to mask the interrupt on the APIC in order to continue processing while the interrupt is not handled. To achieve this, it masks the interrupt line it is handling in a thread on the IO-APIC. Thanks to boot interrupts, this interrupt is now deferred to the southbridge which then issues yet another interrupt. So even though the device is not supposed to issue any interrupts, it is actually triggering an interrupt on a line that the operating system does not even associate with the device. (Figure 5 visualizes this invalid boot interrupt.)

Figure 5: Boot Interrupt (APIC)

With invalid interrupts coming in, Linux shuts down the interrupt line on the first APIC, due to several interrupt requests that no device known to Linux on that line handles. Usually this hits a line where the disk or network controller is on, rendering the machine useless.

4 Dealing with Boot Interrupts

4.1 Disabling

Recent PCI Bridges can turn off boot interrupts in PCI control registers. This way no boot interrupts are issued and no unexpected interrupts caused by them occur. Our patchset[6, 7] for (real-time) Linux implements the switching mechanism on bridges we found it works on.

Disabling boot interrupts in the PCI control registers is very chipset specific, thus the patches need to be updated for every new bridge.

4.2 Rerouting

PCI Bridges which do not support boot interrupt disabling require software workarounds. The rerouting patch in our patchset always masks an interrupt on a non-primary IO-APIC that is known to implement boot interrupts. Because it is now always masked, an interrupt on the non-primary IO-APIC will cause a boot interrupt. The patch also includes logic to find out, on which interrupt line this Boot Interrupt will end up and teaches this to Linux, rerouting all lines to the corresponding primary IO-APIC lines.

While this approach introduces increased interrupt line sharing, it is the only approach known to us to work reliably on PCI bridges that do not support boot interrupt disabling.

5 Conclusions

Real-time Linux uses interrupt masking to allow asynchronous interrupt handling. Due to legacy compatibility, this masking breaks APIC interrupt delivery, requiring it to be turned off. Thanks to the patchset mentioned, this is possible to do in Linux now, enabling real-time Linux on more hardware platforms.

References


[3] Intel® 6700PXH 64-bit PCI Hub Datasheet: Section 2.15.2, July 2004, INTEL INC.


