**Interrupt Management in RHEL**

**1. Overview**

Interrupt management in RHEL is a critical aspect of ensuring efficient system performance, particularly in real-time and high-performance environments. Here's an overview:

**1. Types of Interrupts**

* **Hardware Interrupts**: Triggered by hardware devices (e.g., keyboard, network cards) to signal the CPU for attention.
* **Software Interrupts**: Generated by software or operating system services to request CPU intervention.

**2. Key Features in RHEL for Interrupt Management**

* **APIC-Based Interrupt Handling**: RHEL uses Advanced Programmable Interrupt Controllers (APICs) to manage interrupts. APICs allow:
  + **Load-Balanced Interrupt Delivery**: Distributes interrupts across multiple CPUs to avoid overloading a single processor.
  + **Hierarchical APICs**: Ensures efficient routing of interrupts in complex systems.
* **IRQ Affinity**: Allows binding specific interrupts to particular CPUs, optimizing performance for real-time applications.
* **Interrupt Prioritization**: Ensures critical interrupts are handled promptly, improving system responsiveness.

**3. Real-Time Enhancements in RHEL**

For real-time workloads, RHEL provides additional features:

* **Deterministic Interrupt Handling**: Reduces latency by ensuring predictable interrupt response times.
* **Threaded Interrupt Handlers**: Separates interrupt handling into top halves (quick response) and bottom halves (deferred processing) for better performance.
* **Preempt-RT Kernel**: A real-time kernel patch that enhances interrupt handling by reducing kernel latencies.

**4. Best Practices**

* **Minimize Interrupt Latency**: Use IRQ affinity and prioritize critical interrupts.
* **Optimize Driver Design**: Implement efficient top-half and bottom-half mechanisms.
* **Monitor Interrupts**: Use tools like irqbalance and /proc/interrupts to analyze and optimize interrupt distribution.

RHEL's robust interrupt management ensures high efficiency and reliability, making it suitable for both general-purpose and real-time applications.

**2. What Is an Interrupt?**

[**Interrupts**](https://www.baeldung.com/cs/system-call-vs-system-interrupt)**, also known as traps, are calls made to the processor in order to stop the code that is currently executing.** The processor might not always satisfy this demand. Still, when it accepts it, the processor stops the code execution, halts the current activity, saves the state, and calls a special function to deal with the interrupt. We refer to this function as an interrupt handler or interrupt service routine, which we’ll cover in detail in a later section. After the processor handles the interrupt, it usually resumes operation.

**The reason to trigger this interrupt is usually another event that has priority.** For example, a hardware device might send an interrupt to let the processor know that there is a change in the state that requires attention. Another example is multitasking, where we can take advantage of interrupts to implement computing operations in real-time.

**3. Types of Interrupt and Interrupt Masking**

Before going deeper into the discussion, we can clarify the difference between synchronous and asynchronous interrupts.

**The CPU produces synchronous interrupts or exceptions when it encounters an error that the kernel needs to handle.** These are mostly programming errors such as overflows or page faults. The kernel handles them after the execution of the previous instruction and tries to recover from the error.

**Asynchronous interrupts, or simply interrupts, are those calls that can appear at arbitrary times from external hardware or the processor.**

**Interrupts can be nonmaskable or maskable.** On the one hand, nonmaskable interrupts are those few events that are so critical that the processor will always handle them, as these events cannot be ignored. On the other hand, maskable interrupts are the rest of the interrupts, those which are less critical. We can mask or unmask the maskable interrupts. If we masked them, the processor will either defer or ignore them.

**The masking operation depends on the processor architecture.** In general, we can enable and disable selectively some hardware interrupts in the internal interrupt mask register of the processor. Every interrupt is associated with a bit in this register. In some devices, having the bit indicates masking, while in others, having the bit indicates unmasking.

**4. Interrupt Handler**

As we described, the interrupt handler is a mechanism linked to a given interrupt. It is a piece of code that starts after one interrupt linked to it arrives.

**The interrupt handler, or interrupt service routine (ISR), implements the sequence of operations needed to support the device (including even device drivers) and avoid an error.**

When the interrupt handler returns, the processor will resume the operation it was carrying before the interrupt.

Historically, interrupts were implemented in hardware with electrical or logic conditions. A table of vectors handled these interrupts that were asynchronously invoked. After this, the system entered a special context with more privileges. When moving into a software implementation, there were similar mechanisms implemented. Callback functions at the software level replaced the table of vectors at the hardware level.

**Nowadays, we’ve got a multitude of interrupt handlers in most computers.** For example, when we press a key or move the mouse, an interrupt occurs, and we invoke the interrupt handler. Then, the interrupt handler might copy the information that the device provides and put it into the memory of the computer for further use.

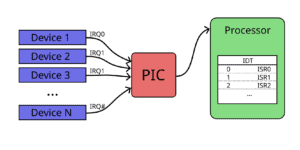
Even if many interrupt handlers are implemented in our systems, they’re intrinsically complex. Interrupt handlers execute in an atypical context with many temporal constraints, and since they’re asynchronous, they’re difficult to debug. Moreover, handling these interrupts is a delicate task. Since they occur at any time, the kernel will try to get rid of it as soon as possible. Moreover, new interrupts can arrive while handling other interrupts.

**5. Association Between Interrupt, Processor, and Handler**

Now that we know some terminology and concepts relating to interrupt handling, we can dive into how the interrupt, the processor, and the handler are connected.

**Most systems use Interrupt Request (IRQ), which the external hardware requests when it needs the processor’s attention.** The Programmable Interrupt Controllers (PIC) are responsible for collecting the IRQ and determining the order for passing the interrupt signals to the processor. Each CPU has IRQ assigned, and the PIC listens to the IRQ. Recent Linux systems distribute the IRQ among the cores with the Advanced PIC (APIC). We need to tweak the IRQ if we want to mask some interrupts.

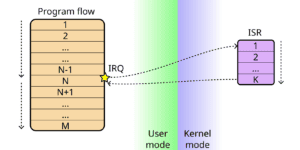
In the following figure, we can see how the architecture is between these components:



We can see in the previous image that there is yet another step between a device invoking an interrupt and its handling: the Interrupt Descriptor Table (IDT). **In the IDT, there is an association between each interrupt and the ISR (interrupt handler) that the processor will call.** The kernel knows, with the IDT, which interrupt handle will deal with each interrupt.

**6. Process of Handling Interrupts in Linux**

The processor of our system just received an interrupt from an input device! We’ll go through the different steps that the computer will follow:



**First of all, the processor needs to stop what it was doing immediately and handle the IRQ that just appeared in the PIC**. Before invoking the interrupt handler, the processor checks if we have masked the interrupt or not. If it is masked, nothing happens. If not, the processor needs to handle the interrupt. It also checks whether the interrupt comes from an authorized source and aborts if not.

Then, the processor saves the current state to, hopefully, resume its operation once it handles the interrupts. The processor switches from user mode to kernel mode and goes into a particular memory address indicated by the IDT. In this memory location, it can find the interrupt handling code or instructions branching to the interrupt handling code. Either way, the interrupt handler receives exclusive control to act accordingly and respond to the interrupt.

Once the interrupt handler has finished, the processor can reload the state in which it was before receiving the interrupt and resume the program execution.

**7. Design of Interrupt Handlers**

We have to design interrupt handlers carefully because their execution impacts the system’s performance. Since during a part of the interrupt handling operation, no other interrupts can arrive, the handling has to be fast. Moreover, the interrupt handling code should be efficient to avoid blocking interrupts for too much time or too many times.

**There are two parts in the interrupt handlers: “top half” and “bottom half”.** The “top half” is the minimum necessary part of the interrupt handler, where communication with the hardware occurs. The “bottom half” performs the rest of the operations, such as copying data into memory.

**This division exists because only during the execution of the “top half” the processor is in interrupt mode, and in general, no other interruptions can happen.** Very high-level interrupts can still happen in some CPUs, even in interrupt mode. However, we should minimize the time the kernel spends in the “top half” to avoid missing other interrupts. The “bottom half” represents the second part of the handler solution.

Thus, even if there aren’t clear guides on how to split the work between the two halves, the “top half” should only do the time-sensitive and hardware-related part of the work. The “bottom half” should take care of the rest.

Since the implementation of interrupt handlers is architecture dependent, it falls outside the scope of this article. However, there are plenty of resources online for common architectures, such as [arm](https://linuxburps.blogspot.com/2013/10/linux-interrupt-handling.html) and [x86](https://pdos.csail.mit.edu/6.828/2005/lec/lec8-slides.pdf).

**8. Conclusion**

In this article, we’ve talked about interrupt handling in Linux. We started with the definitions and classification. Then, we discussed the different steps in Linux once a device invokes an interrupt, and Linux needs to handle it. Finally, we closed by discussing how to design interrupt handlers.