Operating Systems 2

Interrupts Handling

Arkadiusz Chrobot

Department of Information Systems

April 8, 2024

Outline

Introduction

Interrupts are a vital part of every computer system. They are used for handling exceptions and communication with I/O devices. System calls are one of the examples of their applications. The interrupts implementation is very hardware-specific. This lecture gives a general overview of the interrupts handling in the Linux kernel. The more advanced topics, like inter-processor interrupts or interrupts balancing in a multiprocessor systems are not described here.

Interrupts Overview

Linux recognizes two main types of interrupts:

exceptions These are high-priority interrupts associated with important events (like integer division by zero, a page fault, a system call) that require immediate handling by the CPU, and cannot be ignored. Exceptions are usually synchronous, which means that they may occur only as a result of CPU instruction. The kernel functions that handle exceptions are executed in the process context and make use of value returned by the `current` macro.

hardware interrupts These interrupts are used by the I/O devices to signal that they require servicing by the CPU. They are asynchronous, which means they can occur at any time. Kernel functions that handle these interrupts must act quickly, thus they are performed in a special context called an *interrupt context*. Hardware interrupts are the main topic of this lecture.

Hardware Structure

The interrupt system needs a hardware support. The general design of such hardware in a uniprocessor computer system is shown in the figure 1. Each I/O device has a special physical line, called an *Interrupt Request Line* or IRQ line for short, that connects it with a special integrated circuit called *Programmable Interrupt Controller* or PIC for short. Every IRQ line has a unique number (usually a natural number) that identifies the source of the interrupt. The I/O device signals the interrupt by changing the state of the line. The PIC detects the change, determines the number of the interrupt and notifies the CPU, which performs a special kernel function called an *interrupt handler* or *interrupt service routine* that services the interrupt. The discussed interrupt system uses a technique known as a *vectored interrupt*, to quickly detect the source of the interrupt.

Hardware Structure

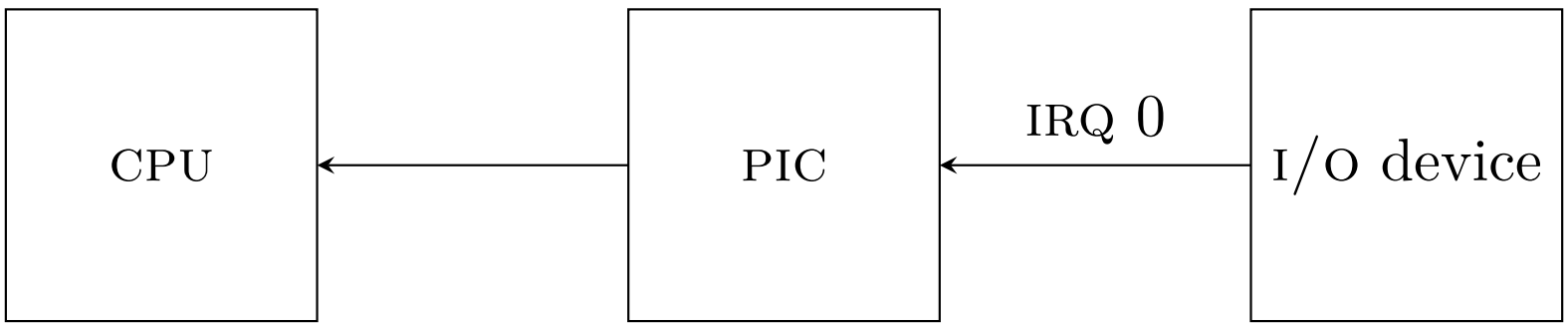

Figure 1: Generic Hardware Structure for Interrupts

Hardware Structure

The vectored interrupts are efficient, only when each I/O device has its own IRQ line. Unfortunately, some of the contemporary computer systems (notably these based on x86 CPUs) have limited number of these lines, so they allow devices to share some of the lines. This means that the vectored interrupt technique is combined with the *polling interrupt* technique. Each time when the interrupt is signaled on a shared line, the kernel has to check which of the devices has done it. This is a time-consuming task.

The Linux kernel programmers had to take into account all differences in the design of the interrupt hardware support among many computer systems or even different versions of the same system. For example in the x86 CPU-based computers the basic PIC was replaced with an *Advanced Programmable Interrupt Controller* (APIC). Moreover, in multiCPU systems each CPU has its own APIC called LAPIC.

Interrupt Servicing

To address these differences the Linux kernel programmers have split the part of kernel responsible for handling interrupts into three layers:

high-level interrupt service routines it is a set of kernel functions (ISRs) responsible for processing the interrupts,

interrupt flow handling a part of the kernel code that takes care of the differences between servicing level-triggered, edge-triggered, (see Fig. 2) per-CPU, and other types of interrupts,

chip-level hardware encapsulation this layer is a low-level layer that is responsible for handling various PICs in different computer systems or sometimes in the same system.

All these layers are linked by the most important data structure of the kernel interrupt handling subsystem: the interrupt descriptors array called `irq_desc`. Each element of the array stores pointers to ISRs and functions handling the PICs.

Interrupt Signalling

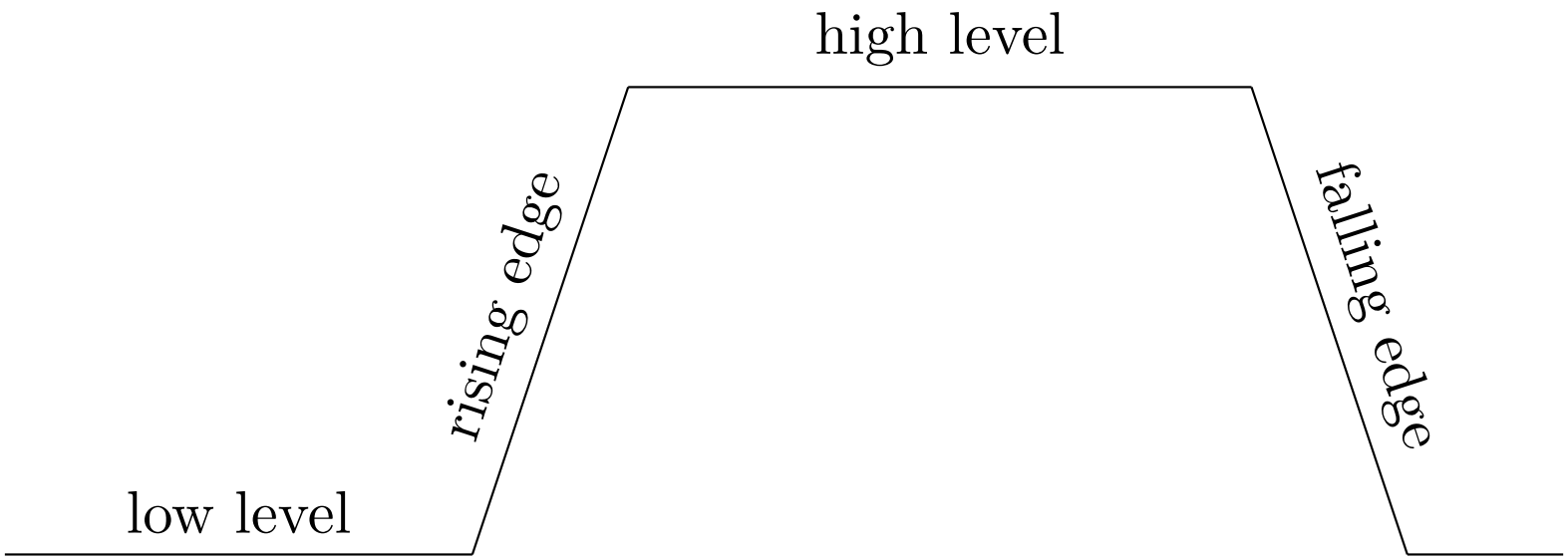

Figure 2: Binary Signal

Interrupt Processing

When the CPU receives the interrupt signal, it automatically switches to the system mode (if it wasn't already in that mode) and performs a CPU-specific kernel code written in assembly language, that saves the registers on the process kernel stack and prepares the environment for performing functions written in the C language. The assembly code is in the `entry.S` file specific to a given hardware platform (in case of x86 CPUs it is `entry_32.S` file for the 32-bit processors and `entry_64.S` file for the 64-bit processors). After the assembly code exits the control on most hardware platforms is passed to the `do_IRQ()` function. Exceptions are the computers based on Sparc, Sparc64 and Alpha CPUs, but they are not discussed in this lecture.

Interrupt Processing

The `do_IRQ()` Function

The implementation of the `do_IRQ()` function is also hardware-specific, but its behaviour is generally always the same. First, it stores the values of registers from the process kernel stack to a structure of the `struct pt_regs` type (which is also CPU-specific). Then it uses the interrupt number, which is stored in one of the registers, as an index in the `irq_desc` array. Next, the `do_IRQ()` function blocks the IRQ line associated with the interrupt by using some functions from the chip-level hardware encapsulation layer that are pointed by the interrupt descriptor. For example, in case of 32-bit x86 hardware platforms it uses the `mask_and_ack_8259A()` function. For some of the interrupts it has to disable the entire interrupt system (or at least the local interrupt system of the CPU that services the interrupt). After blocking the IRQ line the function checks if there is any ISR registered for this interrupt. If so, it calls the routine.

Interrupt Processing

The `do_IRQ()` Function

If there is more than one ISR registered for the specific interrupt, the `do_IRQ()` function invokes them in succession and expects that one of them will return the `IRQ_HANDLED` value, meaning that the interrupt has been serviced. If no ISR has been registered for the interrupt the `do_IRQ()` function signals an error when returning. When the interrupt is handled the `add_interrupt_randomness()` function is called to supply the kernel entropy pool with new values (explained latter), the IRQ line is unlocked or the whole (local) interrupt system is re-enabled and the `do_IRQ()` exits. Most of these operations are usually not preformed directly by the `do_IRQ()` function by delegated to other functions such as `handle_irq_event()` and `ret_form_intr()`.

Interrupt Processing

Entropy Pool

The entropy poll contains values that are used for *seeding* two cryptographically secure pseudorandom number generators implemented in the kernel. These generators are available to user-space software via two character device files called `/dev/random` and `/dev/urandom`. These files can be read just like regular text files. The first one blocks the read operation when the requested amount of entropy is not available and the second one never blocks. They both generate secure pseudorandom numbers, but in very rare cases (usually when the numbers are needed during the initialization of the kernel) it is more safe to use the `/dev/random` generator. In modern kernels, all interrupts contribute to the entropy pool, but not directly. When an interrupt is handled the kernel reads several values from different resources that are good sources of randomness, such as some of the CPU registers.

Registering and Unregistering Interrupt Handler

For the device driver programmer the most important kernel functions associated with interrupt handling are these that allow her or him to register or unregister the ISR. The former has the following prototype:

```
int request_irq(unsigned int irq, irq_handler_t
handler, unsigned long irqflags, const char *name, void
*dev)
```

The function returns zero on success and the `-EBUSY` value on failure. It not only registers the ISR but also activates the IRQ line associated with the interrupt. The function takes five arguments. The first one is the number of the interrupt, the second is the address of the ISR, the third one is a flag, or a result of the bitwise OR of non-conflicting flags.

Registering and Unregistering Interrupt Handler

In the `linux/interrupt.h` header file are defined many flags for registering the interrupt handlers. The most interesting ones are the following: `IRQF_TIMER` — the ISR will handle a timer interrupt, `IRQF_PERCPU` — the ISR will be performed only by a specific CPU in a multiprocessor hardware platform, `IRQF_ONESHOT` — the IRQ line will not be re-enabled immediately after the ISR exits, `IRQF_SHARED` — the ISR is registered for an IRQ line shared by several I/O devices and also by several other ISRs.

Registering and Unregistering Interrupt Handler

The forth argument of the `request_iqr()` function is a string that is a name of the device that will signal the interrupt. The name is used in `proc/interrupts` file[1]. It is a text file that contains statistics about all handled (or not) interrupts, including the IRQ line number, the CPU number, the interrupt type, the PIC name, the device name, etc. Some other statistics can be found in the `proc/irq` directories. Finally, the fifth argument can be `NULL` if the IRQ line is not shared. If it is, than it should be an address that uniquely identifies the ISR, for example an address of a structure associated with the device driver that contains the ISR. This address is required for correctly unregistering the ISR. It is also used by the interrupt handler itself to detect if the device that it should handle is the interrupt's source.

[1] The content of the file can be displayed on the screen by issuing the `cat /proc/interrupt` command.

Registering and Unregistering Interrupt Handler

To unregister the ISR the device driver programmer can use the `free_irq()` function which has the following prototype:

```
void free_irq(unsinged int irq, void *dev)
```

The first argument for the function is the IRQ line number, the second can be `NULL` if the line is not shared. Otherwise it must be the same address, which was given as the fifth argument to the `register_irq()` function.

Registering and Unregistering Interrupt Handler

Threaded Interrupts

In the 2.6.29 kernel release a new way of servicing interrupts has been added. It is so-called *threaded interrupts* and originates from the kernel code branch for hard real-time systems. The main idea behind this new solution is that the ISR should exit as soon as possible, because it cannot sleep. The rest of the work associated with handling the interrupt is delegated to a special kernel thread. To register a thread and the ISR for handling a specific interrupt the `request_threaded_irq()` function is used, which has the following prototype:

```
int request_threaded_irq(unsigned int irq,
irq_handler_t handler, irq_handler_t thread_fn,
unsigned long flags, const char *name, void *dev)
```

The function takes the same arguments as the `request_irq()` function, except for the additional third one, which is a pointer to a kernel thread function. The ISR should be registered with the `IRQF_ONESHOT` flag.

Interrupt Handlers

In the Linux kernel the interrupt handlers or ISRs are kernel functions written in the C language, which may also contain some assembly code. The definitions of these functions are part of device drivers responsible for handling the peripheral devices. These drivers are usually implemented as kernel modules. The prototype of the ISR must follow this pattern:

```
static irqreturn_t intr_handler(int irq, void *dev)
```

The name of a real ISR should be different than the one in the pattern. By the first parameter of the function is passed the interrupt number. The value of the second parameter is important only when the ISR is registered for a shared IRQ line. It is the same unique address which is used for registering the ISR. The `irqreturn_t` type is defined with the help of the `typedef` keyword and, depending of the kernel version, it is the `int` or `void` type. It was introduced for backward compatibility reasons.

Interrupt Handlers

The ISR can return one of the following values: `IRQ_NONE` — the interrupt has not been serviced by the ISR, `IRQ_HANDLED` — the interrupt has been serviced by the ISR. To simplify returning these values the Linux kernel programmers added the `IRQ_RETVAL(x)` macro, which expands to `IRQ_HANDLED` when its argument is a non-zero number, and to `IRQ_NONE` when it is zero. The ISRs associated with threaded interrupts can return a third value called `IRQ_WAKE_THREAD` which causes the kernel to activate the thread associated with the handling of the specific interrupt. In the past the ISR had another parameter which was a pointer to the structure of the `struct pt_regs` type. However, not all ISRs needed registers values, so the parameter has been removed. These ISRs that require values of registers can obtain the address of the registers structure with the use of the `get_irq_regs()` function.

Interrupt Handlers

The most important thing about Linux ISRs is that they are performed in the *interrupt context*. That means that their behaviour undergoes several limitations. Primarily they must act quickly, because the IRQ could interrupted some very important operations in kernel or in user-space. The ISRs are not associated with any process, hence they cannot invoke any functions that could cause the process to sleep. For example the ISR cannot call the `register_irq()` function to register another ISR. To put it simply the ISRs cannot sleep. To address these limitations the interrupt handling code in Linux kernel is split into two parts, just like in the case of other modern operating systems. The first part is called a *top half* and the other a *bottom half*, although they are not necessarily equal. In the top half are performed the most important operations associated with handling the interrupt, that cannot be postponed. So, the top half is just another name for the ISR. The other operations are performed in the bottom half, which actually is not a single mechanism, but a collection of mechanisms.

Interrupt Handlers

Most of bottom halves will be discussed in the next lectures, but threaded interrupts are an example of such a mechanism. The ISRs do not use the value returned by the `current` macro. As it was mentioned already, they are not associated with any process, so they do not need the descriptor of the current process, but they use the kernel process stack, just like other kernel functions. It should be reminded that the size of the stack is limited to only two pages, so in case of x86 CPUs it is 8KiB and for the Alpha processors it is 16KiB. Moreover, there is a kernel patch that limits that size to only one page, which is useful for the MPP (Massively Parallel Processing) systems. In that case however the ISRs get a separate stack for their use only.

The ISRs don't have to be reentrant, because the Linux kernel doesn't support nested interrupts, i.e interrupts that may occur while their earlier instances are serviced. However, in multiprocessor systems the ISR may use some synchronization methods if it shares resources with some other code.

Message Signaled Interrupts

Modern devices that use such buses as USB, PCI, PCI-Express need a lot of interrupts, that are assigned to them dynamically (some of the interrupts are assigned statically for historical reasons). This means that a lot of IRQ lines has to be shared between these devices, which leads to many issues. To address them the hardware engineers introduced so-called *Message Signaled Interrupts* (MSI for short). These interrupts are not signalled by changing the state of a physical IRQ line but by storing a short message (a few bytes) to a specific memory address. The first version of this solution was introduced in the PCI 2.2 standard. In the PCI 3.0 standard the possibility of individually masking these interrupts was added. This version of the standard also allows the devices to have several individually configured interrupts. This solution is called MSI-X. Starting from the 4.8 kernel version, Linux provides an API for using these interrupts.

Message Signaled Interrupts

The API consists of three functions:

`pci_alloc_irq_vectors()` the function allocates interrupt vectors for the PCI device. It takes four arguments. The first one is an address of the `struct pci_dev` type structure associated with the device, the second one is the minimal number of vectors (if required), the third one is the maximal number of vectors. The last argument is one or more flags. On success it returns zero, otherwise the `-ENOSPC` value.

`pci_irq_vector()` the function associates an interrupt number with the PCI device. It takes two arguments, the address of the `struct pci_dev` type structure and the interrupt number. It returns zero on success and non-zero otherwise.

`pci_free_irq_vectors()` the function frees the allocated interrupt vectors. It returns no value and as an argument takes the address of a `struct pci_dev` type structure.

Message Signaled Interrupts

The following flags can be passed to the `pci_alloc_irq_vectors()` function:

`PCI_IRQ_LEGACY` the PCI device will use the interrupts signaled by the IRQ line, instead of MSI (default mode),

`PCI_IRQ_MSI` the PCI device will use the basic MSI,

`PCI_IRQ_MSIX` the PCI device will use the MSI-X,

`PCI_IRQ_ALL_TYPES` the PCI device will use any available kind of the interrupt,

`PCI_IRQ_AFFINITY` in a multiprocessor system the function will spread the interrupts to all available CPUs.

The `pci_irq_vector()` function is used for obtaining an interrupt number for which an ISR can be registered with the use of the `request_irq()` or `request_threaded_irq()` function.

Interrupts Control

The following kernel macros and functions are used for controlling the interrupts system:

`local_irq_disable()` switches off a local interrupts system,

`local_irq_enable()` switches on a local interrupts system,

`local_irq_save(unsigned long flags)` saves the current state of the interrupts and then disables them,

`local_irq_restore(unsigned long flags)` restores the given state of the interrupts,

`disable_irq_nosync(unsigned int irq)` disables a given IRQ line and immediately returns,

`disable_irq(unsigned int irq)` disables a given IRQ line and ensures no interrupt handler is running for that line before returning,

`enable_irq(unsigned int irq)` enables the line switched off by the `disable_irq_nosync()` function,

Interrupts Control

`synchronize_irq(unsigned int irq)` enables the IRQ line disabled by the `disable_irq()` function,

`irqs_disabled()` returns nonzero if local interrupts system is disabled,

`in_interrupt()` returns zero in process context and nonzero in interrupt context,

`in_irq()` returns nonzero if invoked in an ISR, otherwise zero.

The `synchronize_irq()` function has to be called as many times as the `disable_irq()` function was invoked. The same applies for the `enable_irq()` and the `disable_irq_nosyc()` functions.

Questions

?

The End

Thank You for Your attention!