Operating Systems 2

Interrupts Handling

Arkadiusz Chrobot

Department of Computer Science

April 4, 2020

Outline

Introduction

Interrupts are a vital part of every computer system. System calls are one of the examples of their applications. They are also used for handling exceptions and communication with I/O devices. The interrupts system is very hardware dependent. 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 discussed here.

Interrupts Overview

There are two main types of interrupts:

exceptions Those are high-priority interrupts associated with important events, like integer division by zero, that require immediate handling by the CPU, and cannot be ignored. Exceptions are usually synchronous, which means that they can occur only when a special part of code in performed. The kernel functions that handle exceptions are executed in the process context and make use of value returned by the `current` macro.

hardware interrupts Those 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 those interrupts must act quickly, thus they are performed in a special context called *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 (i.e. a computer system with only one CPU with only one core) 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 connect it with a special integrated circuit called *Programmable Interrupt Controller* or PIC for short. Each IRQ line has a unique number (usually a natural number) that allows the PIC to identify 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 its priority and notices the CPU, which performs a special kernel function called an *interrupt handler* or *interrupt service routine* that services the interrupt. The PIC allows kernel programmers to change the priority of interrupts. The described interrupt system uses a technique known as *vectored interrupt*, to quickly detect the source of the interrupt.

Hardware

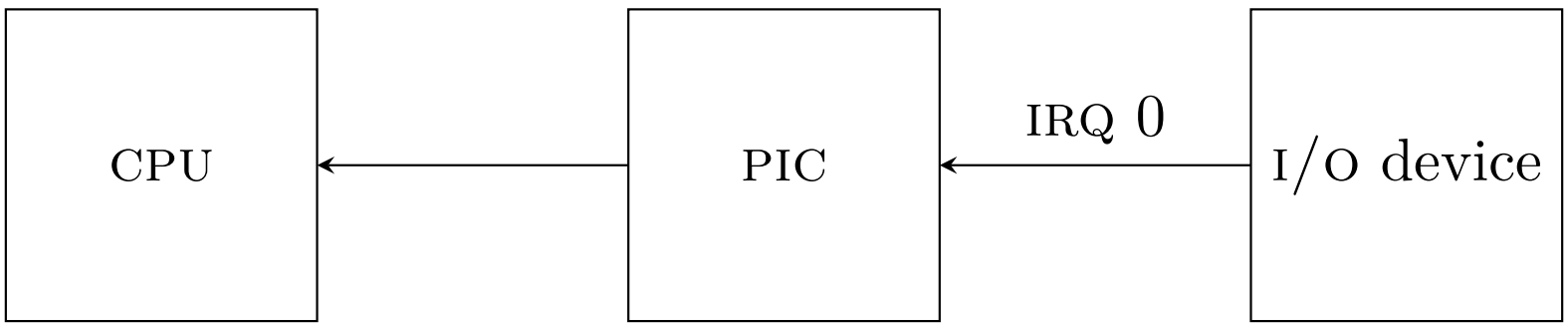

Figure 1 : Generic Hardware Structure for Interrupts

Hardware Structure

The vectored interrupts work correctly, only when each I/O device has its own IRQ line. Unfortunately, some of the contemporary computer systems (notably those based on x86 CPUs) have limited number of those 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 based computers the PIC changed from simple PIC to *Advanced Programmable Interrupt Controller* (APIC). Moreover, in multiprocessor systems each CPU has its own APIC called LAPIC.

Interrupt Servicing

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

high-level interrupt service routines it is a set of kernel functions responsible for actually processing the interrupt,

interrupt flow handling part of the kernel code that takes care of the handling 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 PICs in different computer systems or sometimes in the same system.

All those 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

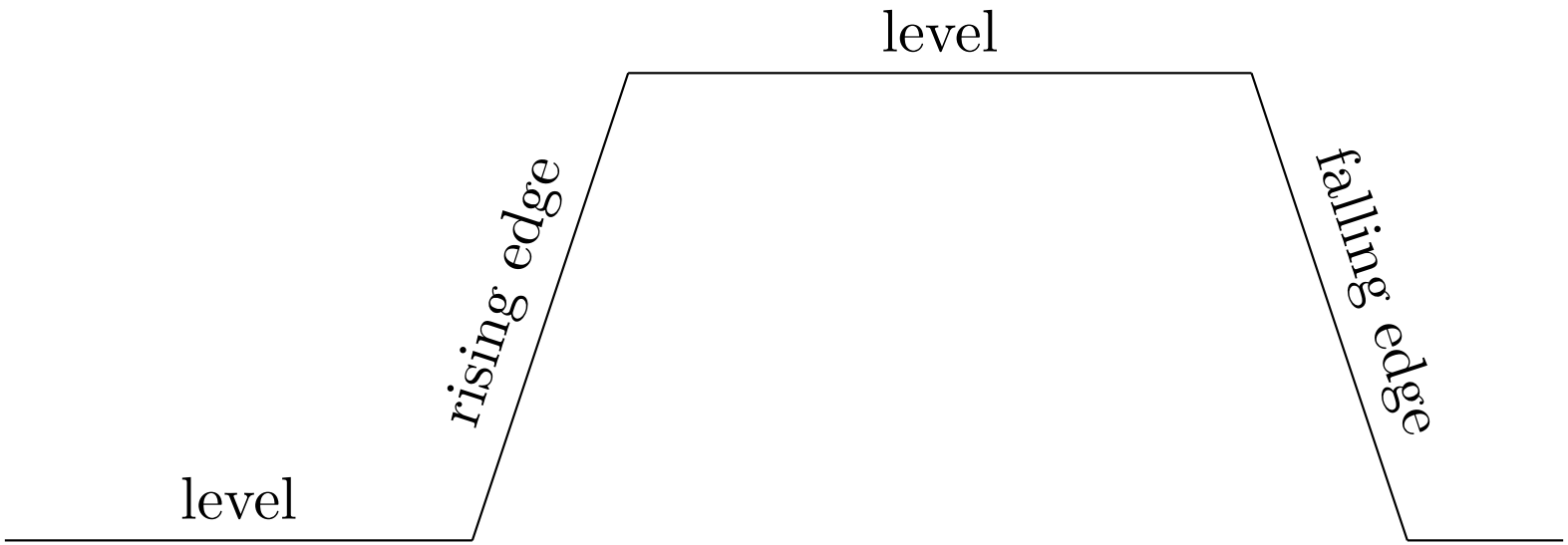

Figure 2 : Binary Signal

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 placed 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 flow on the most hardware platforms is passed to the `do_IRQ()` function. The exceptions are the computers based on Sparc, Sparc64 and Alpha CPUs, but they are not discussed in the lecture.

Interrupt Processing

The `do_IRQ()` Function

The implementation of the `do_IRQ()` function is also hardware platform specific, but its behaviour is generally similar. It first stores the values of registers from the kernel stack to a structure of the `struct pt_regs` type (which is CPU-specific). Then it uses the value of one of the registers to determine the number of the interrupt and uses it as an index in the `irq_desc` array. Next, the `do_IRQ()` function blocks the IRQ line associated with the interrupt by using the 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 whole interrupt system (or at least the interrupt system local for 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 sequence and expects that one of them will return the `IRQ_HANDLED` value, which means, that the interrupt has been serviced. If no `isr` has been registered for the interrupt the `do_IRQ()` function returns signaling an error. When the interrupt is handled the `add_interrupt_randomness()` function is called to supply the kernel entropy pool with new values (explained latter), unlocks the IRQ line or re-enables the whole (local) interrupt system and returns. Most of the activities are usually not preformed directly by the `do_IRQ()` function by delegated to other functions such as `handle_irq_event()` and `ret_form_intr()`.

Registering and Unregistering Interrupt Handler

From the device driver programmer the most important kernel functions associated with interrupt handling are those 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 flags that do not contradict.

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, `IRFQ_PERCPU` – the ISR will be performed only on specific CPU in a multiprocessor hardware platform. `IRQF_ONESHOT` the IRQ will not be re-enabled immediately after the ISR exits, `IRQF_SHARED` the ISR is registered for a IRQ lined shared by several I/O devices and also by several other ISRes. In the past also two others where used. The `IRQF_DISABLE` flag, also called `SA_INTERRUPT` in older kernel versions, was used for registering ISRs that required disabling the whole interrupt system before they could be performed. This flag has been removed from the kernel since version 4.1. The `IRQF_SAMPLE_RANDOM` flag was used to indicate that the information about the frequency of the registered interrupt will supply the kernel entropy pool, which is used for implementing two cryptographically secure pseudorandom number generators.

Registering and Unregistering Interrupt Handler

Those generators are available to user-space software via two character device files called `/dev/random` and `/dev/urandom`. Those file can be read just like other 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 the use the `/dev/random` generator. Not all interrupts were used for supplying the kernel entropy for those generators. The timer interrupt is too regular to be a source for randomness. The network card interrupt seems to be a good candidate, but this source is vulnerable to tampering. Hence a flag was needed to indicate which interrupts will be sources of randomness. But, Theodore Ts'o, the author of those generators has added a patch to the kernel that made the flag obsolete.

Registering and Unregistering Interrupt Handler

Nowadays, all interrupts contribute to the entropy pool, but not directly. When an interrupt is handled the kernel reads several values from different places that are good sources of randomness, such as some of the CPU registers. 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.

[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 uses 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 by the kernel was added. It is so-called threaded interrupts and originates from the branch of the kernel code prepared for hard real-time systems. Today it is also available the main kernel branch. The main idea behind this new mechanism is that the ISR should exit as quickly 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 needed, 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 interrupt handlers or ISRs are a kernel functions written in a C language, which may also contains some assembly code. The definitions of those functions are part of device drivers responsible for handling the peripheral devices. Those 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 was used for registering the ISR. The `irqreturn_t` type is defined with the use of the `typedef` keyword and depending of the kernel version it is a `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 of those values the Linux kernel programmers added a `IRQ_RETVAL(x)` macro, which expands to `IRQ_HANDLED` when its argument is non-zero, 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_rest` type. However, not all ISRs used registers values. To spare the place on the process kernel stack, the parameter has been removed. Those ISRs that need 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 activities 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 those 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 *top half* and the other *bottom half*, although they are not necessarily equal. In the top half the most important activities associated with handling the interrupt, that cannot be postponed are performed. So, the top half is just another name for the ISR. The other activities are performed in the bottom half, which in reality is not a single mechanism, but a set of such mechanisms.

Interrupt Handlers

The bottom halves will be discussed in the next lecture. The threaded interrupts are another approach to solve those issues with interrupt handling. 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 CPU family it is 4KiB and for the Alpha family processors it is 8KiB. Moreover, there is a possibility to limit that size to only one page, which is useful in 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 reentrant interrupts, i.e interrupts that can interrupt servicing of other interrupts. However, in multiprocessor systems the ISR may use some synchronization methods if it shares resources with some other code.

Message Signalled Interrupts

Moder 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 those devices, which leads to many issues. To address them the hardware engineers introduced so-called *Message Signalled Interrupts* (MSI for short). Those 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 those 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 those interrupts.

Message Signalled 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` 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 Signalled Interrupts

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

`PCI_IRQ_LEGACY` the PCI device will used the interrupts signaled by the IRQ lined, 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 the any available interrupt kind,

`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. Prior to 4.8 kernel version the following (now obsolete) functions where used for the MSIs: `pci_enable_msix_range()`, `pci_enable_msi()`, `pci_disable_msi()`, `pci_enable_msix_exact()`, `pci_disable_msix()`

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 of 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 ISR, otherwise zero.

The `local_irq_disable()` and `local_irq_enable()` functions replaced functions `cli()` and `sti()` which globally disable or enabled all interrupts in the computer system, which was suboptimal. The `synchronize_irq()` function has to be called as many times as the `disable_irq()` function was invoked. The same goes for the `enable_irq()` and the `disable_irq_nosyc()` functions.

Questions

?

The End

Thank You for Your attention!