Gabriele **Proietti Mattia**

Advanced Operating Systems and Virtualization

[5] Interrupts Management

Department of Computer, Control and Management Engineering "A. Ruberti", Sapienza University of Rome

gpm.name · proiettimattia@diag.uniroma1.it

A.Y. 2020/2021 · v3

Outline

1. **Introduction**
2. **IRQs and Inter-Processor Interrupts**
3. **The IDT and the Activation Scheme**
4. **Exception Handling**
    1. Fixups and Page Fault Handler
5. **Interrupts Handling**
    1. I/O Interrupts
    2. Inter-Processor Interrupts (IPIs)
6. **Software Interrupts (SoftIRQs)**
7. **Tasklets**
8. **Work Queues**

5.1

5. Interrupts Management

Introduction

DIAG

Interrupts and Exceptions

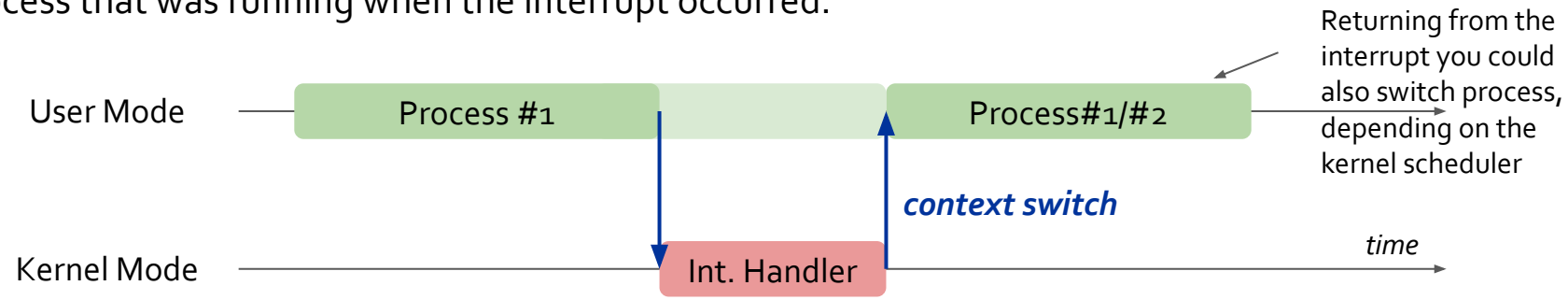
An interrupt is usually defined as an event that alters the sequence of instructions executed by a processor. These events are actually electrical signals generated by hardware circuits. There are two kinds of interrupts:

- **synchronous** interrupts are produced by the CPU control unit while executing instructions, they are generated after the execution of an instruction. They are usually generated by
  - programming errors, in this case the kernel delivers a signal to the program (e.g. `SIGKILL`)
  - unusual conditions in which the kernel must find a solution, e.g. the Page Fault (when you request a page that is not allocated) or a system call via sysenter
- **asynchronous** interrupts are generated by other hardware devices at arbitrary times with respect the CPU clock signals. They are usually generated by timers and I/O devices, like the keystrokes for instance.

Intel usually calls the former *exceptions* and the latter *interrupts.* In the slides we will refer to Interrupt Signals for referring to exceptions and interrupts.

The Role of the Interrupt Signals

When an interrupts signal arrives the CPU must necessarily stop what it's currently doing and switch to a new activity. This is done by saving the current value of the program counter (EIP + CS) in the Kernel Mode stack and places the address of the interrupt type into the program counter. This is called the **context switch** but differently to the switching to another process the code to which we switch is not a process but kernel code that **runs at expense** of the same process that was running when the interrupt occurred.



The Interrupt Handler is **lighter than a process**.

The Role of the Interrupt Signals

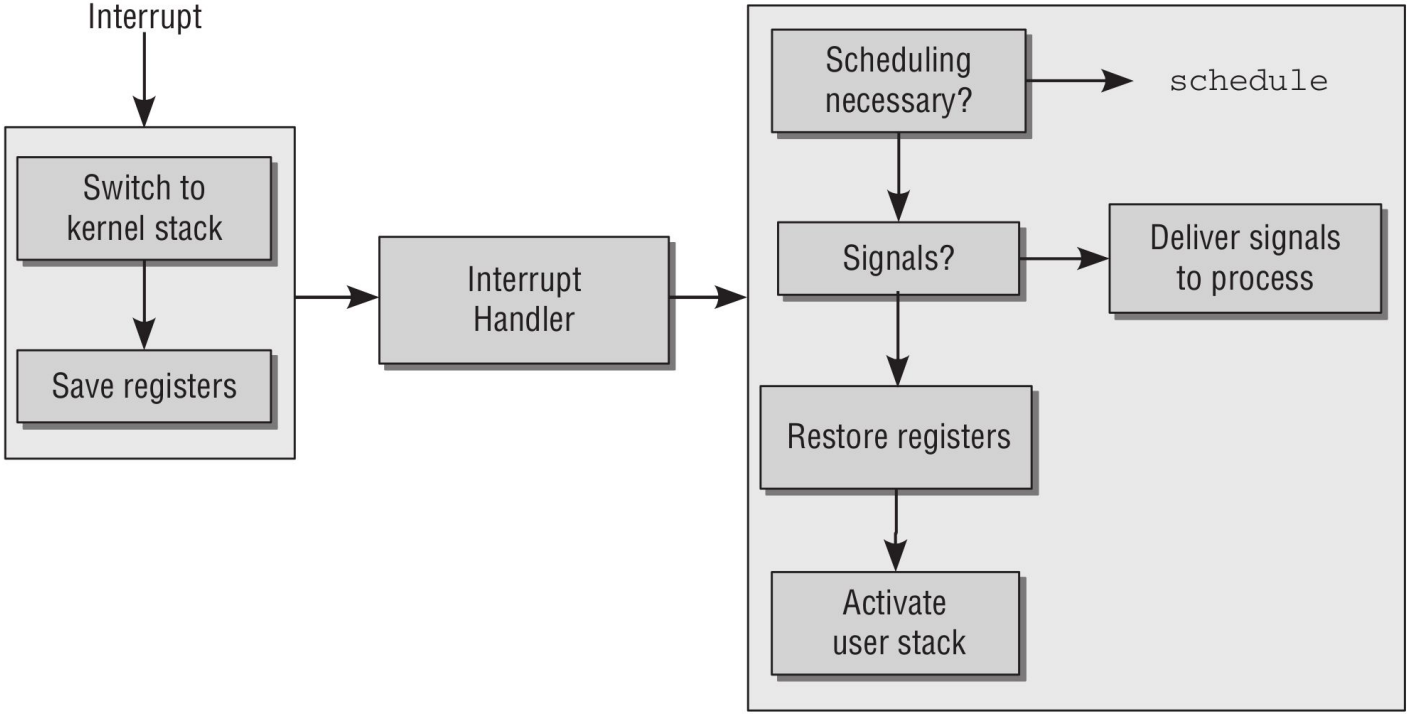


Figure 14-2: Handling an interrupt.

Mauerer, Wolfgang. *Professional Linux kernel architecture*. John Wiley & Sons, 2010.

The Role of the Interrupt Signals

Interrupt handling is one of the most sensitive tasks performed by the kernel because it must satisfy the following constraints:

1. *since the interrupts can come at any time* when the kernel has even to go on with the process it was executing so it must go out of the interrupt as soon as possible and **defer** the demanding work to be executed later. Interrupts has always a **critical** part and a **deferrable** part (top-half/bottom-half);
2. *since the interrupts can come at any time* the kernel might be handling one of them while another one occurs (**nesting**), this should be allowed much as possible since keeps I/O devices busy;
3. even if nesting is allowed (2) there should be present critical regions in which interrupts must be temporarily disabled and these regions must be used only in case of strict necessity.

Classification

The Intel Documentation classifies interrupts and exceptions as follows:

- **Interrupts**
  - *Maskable Interrupts*. All interrupts requests (IRQs) issued by I/O devices are maskable. When an interrupt is masked is temporarily ignored by the CPU.
  - *Non-maskable Interrupts*. A few critical events are non-maskable, like hw failures
- **Exceptions**
  - *Processor-detected exceptions* are generated when the CPU detects an anomalous condition, according to the `eip` (Instruction Pointer) value they are divided in:
    - **Faults** can generally be corrected, eip is the address that caused the fault so the instruction will be re-executed returning from the Int. Handler
    - **Traps** are reported immediately after the execution of trapping instruction, eip points to the next instruction after the trapping one. Traps were essentially used for debugging purposes
    - **Aborts** represents serious errors, the eip cannot be restored to a precise position returning from the Int. Handler, process will be terminated
  - *Programmed Exceptions* occurs at the request of the programmer, they are triggered by the instructions int, int3, into and bound. These interrupts are treated like traps and they are often called also software interrupts they are used for implementing syscall or debugging purposes.

5.2

5. Interrupts Management

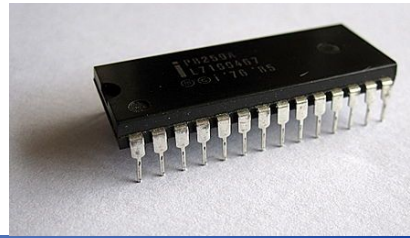
IRQs and Inter-Processor Interrupts

DIAG

IRQs and Interrupts

Each hardware device controller capable of issuing interrupts requests usually has a single output line designated as the Interrupt Request line (IRQ). All of these lines are connected to the input pins of a hardware circuit called the **Programmable Interrupt Controller** (PIC - usually Intel 8259A with the 8086 processor) which performs the following actions:

1. monitors the IRQ lines and if two or more lines are enabled selects the one that has lower pin number
2. If a raised signal occur on a IRQ line:
   a. the signal is converted to a vector
   b. stores the vector in an Interrupt Controller I/O port so the CPU can read it
   c. sends a signal to the INTR pin of the processor
   d. waits untils the CPU acknowledges the interrupt signal by writing into one of the PIC ports, this clears the INTR line
3. Go to 1



Intel 8259A

IRQs and Interrupts

The n-th IRQ line is associated with the vector n+32. Each IRQ line can be selectively disabled, disabled interrupts are not lost, since the PIC send them to the CPU as soon as they will be re-enabled. Enabling and disabling interrupts is not the same as masking and unmasking:

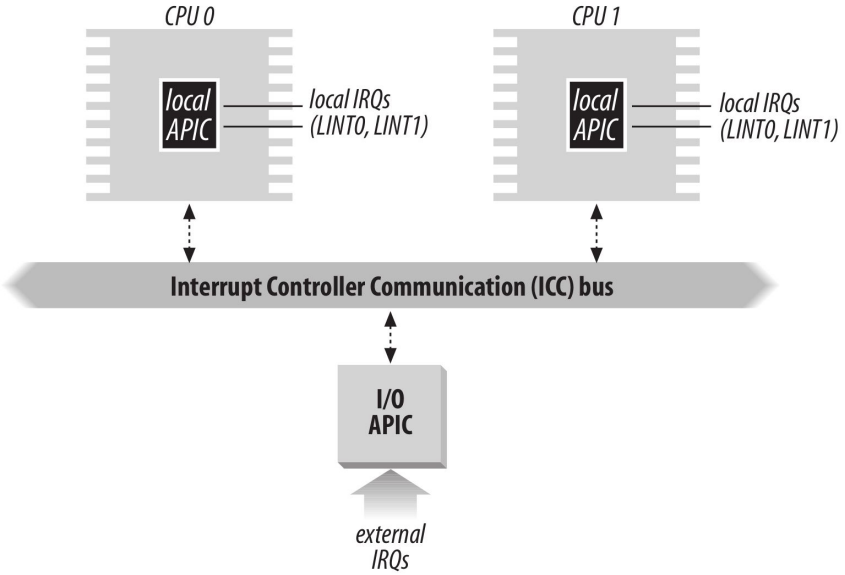
- **enabling/disabling** is done by communicating with the PIC
- **masking/unmasking** is done by clearing and setting the IF flag in the EFLAGS register, that is done with `cli` and `sti` assembly instructions

Multicore Processors

If we are on a single CPU the output line of the PIC can be connected to the INTR line of the CPU, directly. On multicore systems, Intel introduced a new PIC controller, called the **APIC** (Advanced Programmable Interrupt Controller), each microprocessor has a Local-APIC with:

- 32bit registers
- internal clock and local timer
- two additional lines **`LINT0`** and **`LINT1`** reserved for local APIC interrupts.

All the LAPICs are connected to an external I/O APIC.



Bovet, Daniel P., and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management*. " O'Reilly Media, Inc.", 2005.

Multicore Processors

The I/O APIC has 24IRQ lines, a 24-entry Interrupt Redirection Table, programmable registers and a message unit for sending and receiving APIC messages over the bus. The interrupt priority is not given by the lowest number but it is written in the redirection table, that in the end translates external interrupts into a message to one or more APIC units. External interrupts can be distributed in two ways:

1. **Static Distribution**. The IRQ signal is delivered to the LAPICs listed in the proper entry of the redirection table
2. **Dynamic Distribution**. The IRQ signal is delivered to the LAPIC of the processor that is executing the process with lowest priority. This is done by programming the TPR register inside the LAPIC.

A multi-APIC system also allows to generate inter-processor interrupts (IPI), by using the ICR register. The IPIs are a critical part of a SMP system, in Linux they are used for exchanging messages between the CPUs.

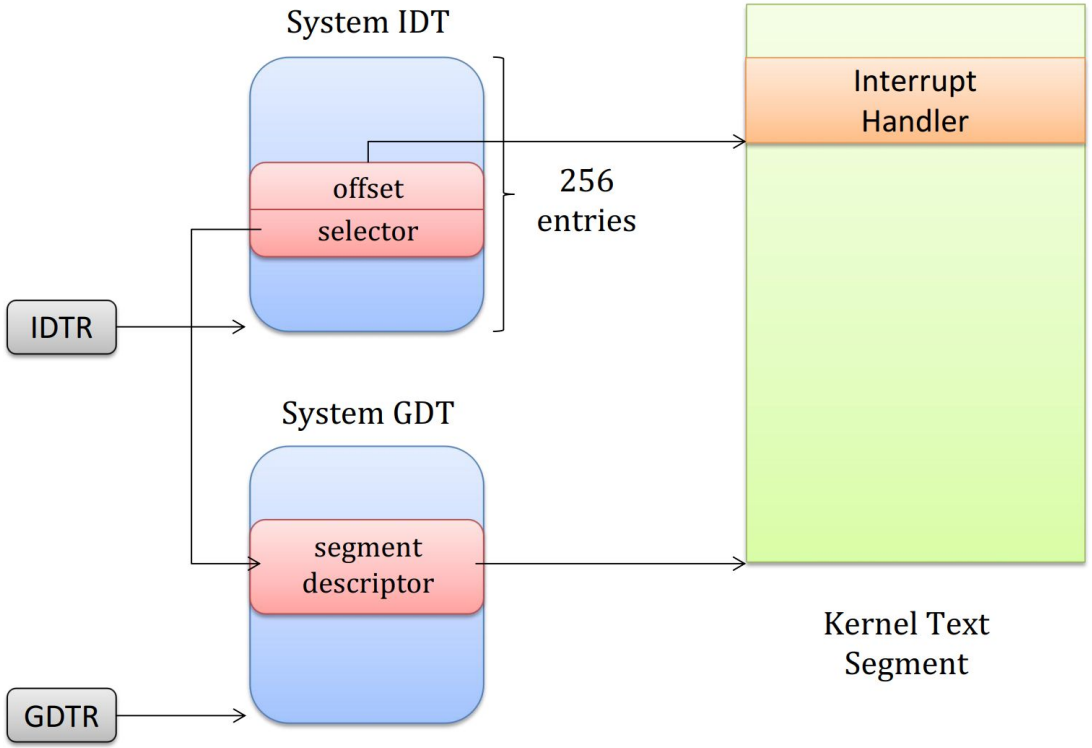
5.3

5. Interrupts Management

The IDT and the Activation Scheme

DIAG

Back to the beginning



The IDT

The Interrupt Descriptor Table associates each interrupt or exception vector with address of the corresponding interrupt or exception handler. Similarly to the GDT, each entry of the table corresponds to an interrupt or exception vector and consists of 8-byte descriptor (on x86 or 16-byte on x86_64). Thus we need 256*8=2048 bytes to store the table. The IDT is pointed by the IDTR register so it can be anywhere in memory.

As we already discussed there are different types of entries:

- **Interrupt Gate** entries, includes the segment selector and the offset inside the segment for the handler, while transferring the control the CPU **clears the IF flag** thus disabling the maskable interrupts. These gates are used to handle interrupts.
- **Trap Gate** entries, same as Interrupt Gates but the **IF flag is not cleared**. These gates are used to handle exceptions.
- *Task Gate* entries, includes the TSS selector of the process that must replace the current when an interrupt occurs. These gates were intended to be used for process switch, today they are no more used.

Traps

Differently from interrupts, trap management does not automatically reset the interruptible-state of a CPU core (IF), therefore critical sections in the trap handler must explicitly mask and then re-enable interrupts (cli and sti instructions).

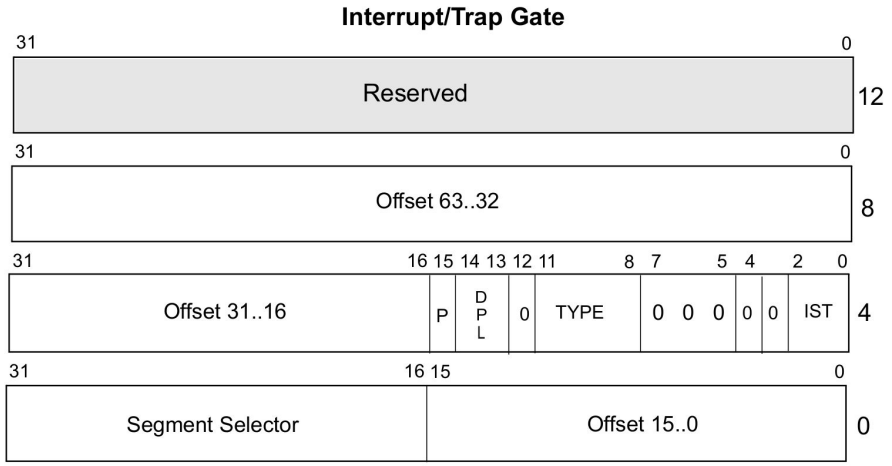
For SMP/multi-core machines this **might not be enough** to guarantee correctness (atomicity) while handling the trap. The kernel uses **spinlocks**, based on atomic test-and-set primitives:

- `cmpxchg`
- `xchg`
- use the prefix `lock` before the instruction (e.g. **`lock`** `incl`)

Interrupt/Gate Descriptor

V5.11

On 64-bit architecture



DPL	Descriptor Privilege Level
Offset	Offset to procedure entry point
P	Segment Present flag
Selector	Segment Selector for destination code segment
IST	Interrupt Stack Table

Figure 6-8. 64-Bit IDT Gate Descriptors

```
84  struct gate_struct {
85          u16             offset_low;
86          u16             segment;
87          struct idt_bits bits;
88          u16             offset_middle;
89  #ifdef CONFIG_X86_64
90          u32             offset_high;
91          u32             reserved;
92  #endif
93  } __attribute__((packed));
```

https://elixir.bootlin.com/linux/v5.11/source/arch/x86/include/asm/desc_defs.h#L84

```
69  struct idt_bits {
70          u16             ist     : 3,
71                          zero    : 5,
72                          type    : 5,
73                          dpl     : 2,
74                          p       : 1;
75  } __attribute__((packed));
```

https://elixir.bootlin.com/linux/v5.11/source/arch/x86/include/asm/desc_defs.h#L69

IDT Entries

Range	Use
0-19	Nonmaskable interrupts and exception
20-31	Intel-reserved
32-127	External interrupts (IRQs)
128 (0x80)	Programmed exception for system calls
129-238	External interrupts (IRQs)
239	Local APIC timer interrupt
240-250	Reserved by Linux for future use
251-255	Inter-processor interrupts

Hardware Handling of Interrupts/Exceptions

Process -> Interrupt Handler

The check if an interrupt arrived is done after the execution of every asm instruction. Then, if the check is positive, the following steps are executed:

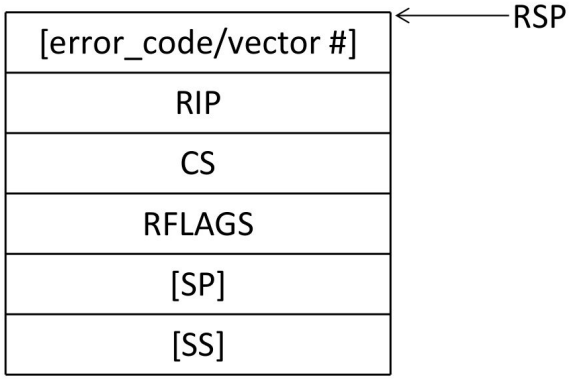
1. determining the vector i (0<= i <= 255) associated with the interrupt/exception
2. read the i entry in the IDT referred by IDTR
3. get the base address of the GDT from GDTR and read the segment descriptor for that
4. If CPL < segment DPL --> **General Protection Error**
5. If *programmed exception* and if gate DPL < CPL -> **General Protection Error**
6. If CPL is different from segment DPL (there is a context switch)
   a. the TR register is read to access to the TSS segment of the running process
   b. SS and ESP registers are loaded with proper values associated to the new privilege level (remind the TSS structure)
   c. In this new stack, the old SS and ESP are saved
7. If there was a **fault** CS and EIP are loaded with the logical address of the instruction that caused the exception so that it can be executed again
8. EFLAGS, CS and EIP are **saved** in the stack
9. If the exception carries an **error code** it is saved in the stack
10. CS and EIP are loaded with the reference to the Interrupt Handler from the i entry of IDT

Hardware Handling of Interrupts/Exceptions

Interrupt Handler -> Process

After the interrupt, the `iret` asm instruction is called. If you pushed an error code in the stack you need to pop it before executing `iret`. Then the `iret` instruction:

1. Loads the CS, EIP and EFLAGS registers with the values saved in the stack
2. Checks if the interrupted process had the same privilege of the Interrupt Handler (looking at the CPL of the handler and the current CPL). If so, `iret` concludes the execution
3. Loads SS and ESP from the stack returning to the stack of the old privilege level
4. Examines the content of DS, ES, FS and GS registers. If any of these registers contains a DPL lower than the CPL then the register is cleared, this is done for security reasons.



Interrupt Stack Frame

Nested Execution of Ex./Int. Handlers

Every interrupt or exception gives rise to a kernel control path or separate sequence of instructions that execute in Kernel Mode on behalf of the current process. These paths may arbitrarily nested by another interrupt handler, thus giving rise to a nested execution of kernel control paths.

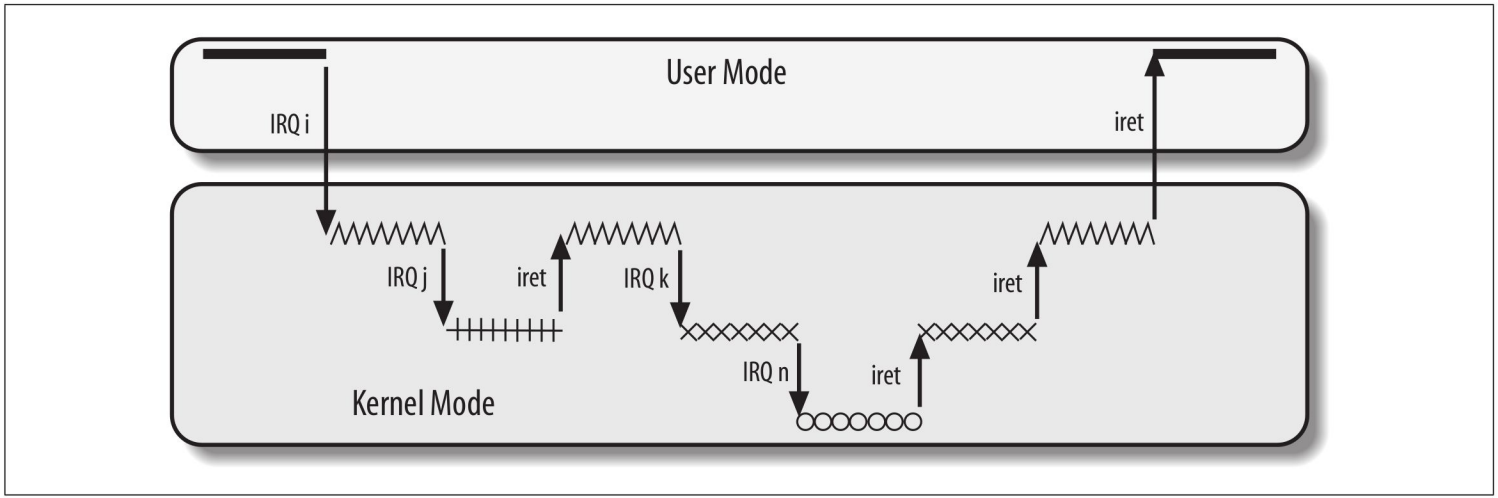


Figure 4-3. An example of nested execution of kernel control paths

Bovet, Daniel P., and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management*. " O'Reilly Media, Inc.", 2005.

Nested Execution of Ex./Int. Handlers

The price to pay for allowing nested kernel control path is that an interrupt handler **must never block**, i.e. no process switch can be done while an interrupt handler is running because we have saved the context in the Kernel Stack of the previous process.

In general, most exceptions are raised when in User Mode, however the **Page Fault** exception can occur in Kernel Mode, when a page is not in RAM (e.g. swapped). When there is a Page Fault, the current process must be put in sleep therefore the Page Fault exception never gives rises to further exceptions.

An interrupt handler **may preempt both** other interrupt handlers and exceptions but an exception handler never preempts an interrupt handler, the only "exception" is the Page Fault, but Interrupt Handlers never performs operations that gives rise to page faults.

In multiprocessor systems there are several parallel kernel control paths, so an exception may start on a CPU and end on another due to the process switch.

5.4

5. Interrupts Management

Exception Handling

DIAG

Exceptions Handling

Most of the exceptions are interpreted by Linux as error conditions. When one of them occurs, the kernel sends a **signal** to the process that caused the exception. But it is not always the case.

In some cases Linux exploits exceptions to manage hardware resources more efficiently, for example the Page Fault.

Table 4-1. Signals sent by the exception handlers

#	Exception	Exception handler	Signal
0	Divide error	`divide_error( )`	SIGFPE
1	Debug	`debug( )`	SIGTRAP
2	NMI	`nmi( )`	None
3	Breakpoint	`int3( )`	SIGTRAP
4	Overflow	`overflow( )`	SIGSEGV
5	Bounds check	`bounds( )`	SIGSEGV
6	Invalid opcode	`invalid_op( )`	SIGILL
7	Device not available	`device_not_available( )`	None
8	Double fault	`doublefault_fn( )`	None
9	Coprocessor segment overrun	`coprocessor_segment_overrun( )`	SIGFPE
10	Invalid TSS	`invalid_TSS( )`	SIGSEGV
11	Segment not present	`segment_not_present( )`	SIGBUS
12	Stack segment fault	`stack_segment( )`	SIGBUS
13	General protection	`general_protection( )`	SIGSEGV
14	Page Fault	`page_fault( )`	SIGSEGV
15	Intel-reserved	None	None
16	Floating-point error	`coprocessor_error( )`	SIGFPE
17	Alignment check	`alignment_check( )`	SIGBUS
18	Machine check	`machine_check( )`	None
19	SIMD floating point	`simd_coprocessor_error( )`	SIGFPE

Bovet, Daniel P., and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management*. " O'Reilly Media, Inc.", 2005.

Structure

v2.6

```
824  void __init trap_init(void)
825  {
836          set_intr_gate(0, &divide_error);
837          set_intr_gate_ist(2, &nmi, NMI_STACK);
838          /* int4 can be called from all */
839          set_system_intr_gate(4, &overflow);
840          set_intr_gate(5, &bounds);
841          set_intr_gate(6, &invalid_op);
842          set_intr_gate(7, &device_not_available);
843  #ifdef CONFIG_X86_32
844          set_task_gate(8, GDT_ENTRY_DOUBLEFAULT_TSS);
845  #else
846          set_intr_gate_ist(8, &double_fault, DOUBLEFAULT_STACK);
847  #endif
848          set_intr_gate(9, &coprocessor_segment_overrun);
849          set_intr_gate(10, &invalid_TSS);
870  #ifdef CONFIG_X86_32
871          set_system_trap_gate(SYSCALL_VECTOR, &system_call);
872          set_bit(SYSCALL_VECTOR, used_vectors);
873  #endif
881  }
```

https://elixir.bootlin.com/linux/v2.6.39.4/source/arch/x86/kernel/traps.c#L824

The standard structure of an exception is the following:

1. **save** the content of most registers in the kernel mode stack
2. **handle** the exception by mean of a C function
3. **exit** with `ret_from_exception()` function

The C functions which handle the exceptions are registered during `trap_init()`.

Remind the `system_call` handler.

The "Modern" Initial IDT

V5.11

```
75  static const __initconst struct idt_data def_idts[] = {
76          INTG(X86_TRAP_DE,               asm_exc_divide_error),
77          INTG(X86_TRAP_NMI,              asm_exc_nmi),
78          INTG(X86_TRAP_BR,               asm_exc_bounds),
79          INTG(X86_TRAP_UD,               asm_exc_invalid_op),
80          INTG(X86_TRAP_NM,               asm_exc_device_not_available),
81          INTG(X86_TRAP_OLD_MF,           asm_exc_coproc_segment_overrun),
82          INTG(X86_TRAP_TS,               asm_exc_invalid_tss),
83          INTG(X86_TRAP_NP,               asm_exc_segment_not_present),
84          INTG(X86_TRAP_SS,               asm_exc_stack_segment),
85          INTG(X86_TRAP_GP,               asm_exc_general_protection),
86          INTG(X86_TRAP_SPURIOUS,         asm_exc_spurious_interrupt_bug),
87          INTG(X86_TRAP_MF,               asm_exc_coprocessor_error),
88          INTG(X86_TRAP_AC,               asm_exc_alignment_check),
89          INTG(X86_TRAP_XF,               asm_exc_simd_coprocessor_error),
90
91  #ifdef CONFIG_X86_32
92          TSKG(X86_TRAP_DF,               GDT_ENTRY_DOUBLEFAULT_TSS),
93  #else
94          INTG(X86_TRAP_DF,               asm_exc_double_fault),
95  #endif
96          INTG(X86_TRAP_DB,               asm_exc_debug),
97
98  #ifdef CONFIG_X86_MCE
99          INTG(X86_TRAP_MC,               asm_exc_machine_check),
100 #endif
101
102         SYSG(X86_TRAP_OF,               asm_exc_overflow),
103 #if defined(CONFIG_IA32_EMULATION)
104         SYSG(IA32_SYSCALL_VECTOR,       entry_INT80_compat),
105 #elif defined(CONFIG_X86_32)
106         SYSG(IA32_SYSCALL_VECTOR,       entry_INT80_32),
107 #endif
108 };
```

https://elixir.bootlin.com/linux/v5.11/source/arch/x86/kernel/idt.c#L184

```
30  /* Interrupt gate */
31  #define INTG(_vector, _addr)                            \
32          G(_vector, _addr, DEFAULT_STACK, GATE_INTERRUPT, DPL0, __KERNEL_CS)
33
34  /* System interrupt gate */
35  #define SYSG(_vector, _addr)                            \
36          G(_vector, _addr, DEFAULT_STACK, GATE_INTERRUPT, DPL3, __KERNEL_CS)
37
38  /*
39   * Interrupt gate with interrupt stack. The _ist index is the index in
40   * the tss.ist[] array, but for the descriptor it needs to start at 1.
41   */
42  #define ISTG(_vector, _addr, _ist)                      \
43          G(_vector, _addr, _ist + 1, GATE_INTERRUPT, DPL0, __KERNEL_CS)
44
45  /* Task gate */
46  #define TSKG(_vector, _gdt)                             \
47          G(_vector, NULL, DEFAULT_STACK, GATE_TASK, DPL0, _gdt << 3)
48
```

https://elixir.bootlin.com/linux/v5.11/source/arch/x86/kernel/idt.c#L31

We have

- **Interrupt Gate,** has DPL0
- System Gate had DPL3 (<= v2.6)
- **System Interrupt Gate,** has DPL3
- Trap Gate had DPL0 (<= v2.6)
- **Task Gate** has DPL3

The Double Fault Exception

The only task gate is referring to the Double Fault exception, because it denotes a serious kernel misbehaviour. Therefore, exception handler **does not trust** the value in **esp** register.

A Double Fault occurs when an exception is unhandled or when an exception occurs while the CPU is trying to call an exception handler. Normally, two exception at the same time are handled one after another, but in some cases that is not possible. For example, if a page fault occurs, but the exception handler is located in a not-present page, two page faults would occur and neither can be handled. A double fault would occur.

A double fault will always generate an error code with a value of zero. The saved instruction pointer is undefined. A double fault cannot be recovered. The faulting process must be terminated.

https://wiki.osdev.org/Exceptions

Exception Handlers

The generic exception handler handler_name is composed by the following assembly instructions:

```
handler_name:
    pushl $0 /* only for some exceptions */
    pushl $do_handler_name
    jmp error_code
```

Three operations:

1. push **`$0`**, the control unit does not put in the stack the hw error code
2. push **`$do_handler_name`**, that is the address of the exception handler
3. jump to **`error_code`**, that is the **same** for every exception, the block performs a set of operations in order to prepare the call to **`do_handler_name`**. The invoked function receives its arguments on registers rather than in the stack (as **`__switch_to()`** that we will later).

do_*handler_name*()

The name of the functions which implements exception handling always starts with the prefix do_. Most of these functions in the end call invoke do_trap() to store the hardware error code and the exception vector in the process descriptor and then send a suitable signal (see Table earlier).

```
current->thread.error_code = error_code;
current->thread.trap_no = vector;
force_sig(sig_number, current);
```

The signal is handled in user mode, if the programmer defined a signal handler, otherwise in Kernel Mode and the kernel usually kills the process.

The exception handler **must determine** if the error happened in User Mode or in Kernel Mode and in this latter case if it was due to an invalid argument passed to a system call, because in this particular case the kernel uses a **Fix-Up code**. In any other case the kernel call the function `die()` which prints a dump in the screen and kills the process (remember the kernel *oops*).

5.4.1

5. Interrupts Management
4. Exception Handling

Fixups and Page Fault Handler

DIAG

Accessing Memory and System Calls

In general, there may be the case that when a user space process calls a system call it passes a parameter to a memory area. When this pointer is passed to Kernel Space the kernel may check it in on of the two ways:

- check if the address belongs to the process address space
- check if the address is lower than PAGE_OFFSET

The first, more time consuming, was used by the earlier versions of the kernel, from 2.2 the second check is performed. Obviously this is a very coarse checking so the idea is to the defer as later as possible the true check. The check is done by `access_ok()` macro.

```
int access_ok(const void * addr, unsigned long size)
{
    unsigned long a = (unsigned long) addr;
    if (a + size < a ||
        a + size > current_thread_info()->addr_limit.seg)
        return 0;
    return 1;
}
```

The Page Fault

`access_ok()` only performs a coarse check but if it passes then the address can be still not be valid for that process, therefor a Page Fault can be raised when:

1. the kernel attempts to address a page belonging to the process address space but the frame **does not exist** or it is **read-only**;
2. the kernel addresses a page belonging to its address space but the corresponding entry in the Page Table has **not** been yet **initialized**;
3. there is **bug** in the kernel or an hardware **error**;
4. a system call service routine attempts to read or write into a memory area whose address has been passed as a system call parameter by it does not belong to the process address space.

In the first case the kernel checks if the linear address belongs to the process, in the second case it is again easy to recognize by looking at the Master Kernel Page Table entry. But how the other two cases?

The Exception Tables

v2.6

Only a small group of functions and macros are used to access the process address space within the kernel (e.g. `get_user()`, ...), thus if the exception is caused by an invalid parameter the instruction that caused it **must be included** in one of the functions.

For this reason, the addresses of these functions are put in a exception table and the `do_page_fault()` handler will look at the table: if it includes the address of instruction that triggered the exception the error is caused by a system call parameter, otherwise by a more serious bug.

The kernel exception table is stored in the `__ex_table` section of the kernel and each entry contains:

- `insn` that is the address of an instruction that accesses the process address space
- `fixup` that is the address of the assembly code which solves the problem. These instructions in general are put in the `.fixup` section of the kernel code segment

The Exception Tables

v2.6

The `do_page_fault()` executes the following statements:

```
if ((fixup = search_exception_tables(regs->eip))) {
    regs->eip = fixup->fixup;
    return 1;
}
```

Generating the Exception Table

The GNU Assembler allows to specify the content of a section with the label ".section" and "a" stands for add to the kernel binary image.

```
.section __ex_table, "a"
    .long faulty_instruction_address, fixup_code_address
.previous
```

In the section `.fixup`

Bovet, Daniel P., and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management*. " O'Reilly Media, Inc.", 2005.

The Kernel Exception Table

v2.6

Example

```
__get_user_1:
    [...]
1:  movzbl (%eax), %edx
    [...]
__get_user_2:
    [...]
2:  movzwl -1(%eax), %edx
    [...]
__get_user_4:
    [...]
3:  movl -3(%eax), %edx
    [...]
bad_get_user:
    xorl %edx, %edx
    movl $-EFAULT, %eax
    ret
.section __ex_table,"a"
    .long 1b, bad_get_user
    .long 2b, bad_get_user
    .long 3b, bad_get_user
.previous
```

This is the example of the exception table for the `get_user` functions, obviously the exceptions table are also defined for other functions, like for example `strlen_user(string)`.

In general, are expanded the one on the right.

The "x" in th of .section te that the cod table code.

```
354  #define __get_user_size(x, ptr, size, retval, errret)                  \
355  do {                                                                   \
356          retval = 0;                                                    \
357          __chk_user_ptr(ptr);                                           \
358          switch (size) {                                                \
359          case 1:                                                        \
360                  __get_user_asm(x, ptr, retval, "b", "b", "=q", errret);\
361                  break;                                                 \
362          case 2:                                                        \
363                  __get_user_asm(x, ptr, retval, "w", "w", "=r", errret);\
364                  break;                                                 \
365          case 4:                                                        \
366                  __get_user_asm(x, ptr, retval, "l", "k", "=r", errret);\
367                  break;                                                 \
368          case 8:                                                        \
369                  __get_user_asm_u64(x, ptr, retval, errret);            \
370                  break;                                                 \
371          default:                                                       \
372                  (x) = __get_user_bad();                                \
373          }                                                              \
374  } while (0)
375
376  #define __get_user_asm(x, addr, err, itype, rtype, ltype, errret)      \
377          asm volatile("1:        mov"itype" %2,%"rtype"1\n"             \
378                       "2:\n"                                            \
379                       ".section .fixup,\"ax\"\n"                        \
380                       "3:        mov %3,%0\n"                           \
381                       "  xor"itype" %"rtype"1,%"rtype"1\n"              \
382                       "  jmp 2b\n"                                      \
383                       ".previous\n"                                     \
384                       _ASM_EXTABLE(1b, 3b)                              \
385                       : "=r" (err), ltype(x)                            \
386                       : "m" (__m(addr)), "i" (errret), "0" (err))
```

https://elixir.bootlin.com/linux/v2.6.39.4/source/arch/x86/include/asm/uaccess.h#L354

Fixup activation steps

Recap

1. access to invalid address e.g. from get_user()
2. MMU generates exception
3. CPU calls `do_page_fault`
4. do page fault calls `search_exception_table`()
5. `search_exception_table` looks up the address of `current->eip` in the exception table and returns the address of the associated fault handle code, the fixup.
6. `do_page_fault` modifies its own return address to point to the fault handle code and returns.
7. execution continues in the fault handling code:
   a. EAX becomes `-EFAULT` (== -14)
   b. DL becomes zero (the value we "read" from user space)
   c. execution continues at local label 2 (address of the instruction immediately after the faulting user access).

5.5

5. Interrupts Management

Interrupts Handling

DIAG

Interrupts Classification

As we discussed, most exceptions are handled simply by sending a unix signal to the process that caused the exception. So the action to be taken will be executed as soon as the process receives the signal, this does not hold for interrupts since they can also arrive long after the process to which they are related so a signal does not make sense.

The interrupt handling changes according to the type of the interrupt raised:

- **I/O Interrupts** are received every time that an I/O device requests attention to the kernel. The interrupt handler must query the device to setup proper actions;
- **Timer Interrupts**. The LAPIC timer has issued an interrupt, this notifies the kernel that some time has passed
- **Inter-processor Interrupts** (**IPI**). A CPU issued an interrupt to another CPU. On multicore systems, we must ensure, for instance, that different cores synchronize with each other in some circumstances

5.5.1

5. Interrupts Management

5. Interrupts Handling

I/O Interrupts

DIAG

I/O Interrupts Handling

An I/O Interrupt Handler must be enough flexible to service several devices at the same time, but the IRQ lines are in general shared by multiple devices, so reading only the IRQ line number it will be not sufficient to understand which device issued it.

There are two different situations:

1. **IRQ Sharing**. The interrupt handler executes several interrupt service routines (ISRs), each routine is related to each device that shares the IRQ line. Every ISRs is always executed.
2. **IRQ Dynamic Allocation**. An IRQ line is associated with a device at the last possible moment, for instance only when device is in use. In this way two devices cannot obviously use the same line at the same time.

In any case, when handling an interrupt not every action that you need to perform has the same priority, this because when you handle an interrupt on a line all the other signals are ignored. Therefore the handling must be as quick as possible.

I/O Interrupts Handling

Linux differentiates in three categories the actions that should be carried out in a interrupt handler:

1. **Critical**. Actions like acknowledging the IRQ to the PIC, updating the data structure shared by the device and the CPU. These action are critical and quick, they must be executed immediately with maskable interrupts disabled (you cannot be interrupted).

2. **Non-Critical**. Actions like updated the data structure accessed only by the CPU (e.g. understanding which key has been pressed on the keyboard). These action can finish quickly and they are executed immediately but with the interrupts enabled (you can be interrupted).

3. **Non-Critical Deferrable**. Actions like copying a buffer content into the process address space (e.g. sending the keyboard line buffer to the terminal handler process). These actions may be delayed for a long time interval, the user process will wait (e.g. sleep) for the data. These actions are executed by means of separate functions (*SoftIRQs* and *Tasklets*).

Flow of operation

In any case, the Interrupt Handler performs the following operations:

1. save the IRQ value and the register in the kernel mode stack
2. send the ack to the PIC, thus allowing further interrupts on that line
3. execute the ISRs for all the devices that shares that IRQ line
4. terminate with `ret_from_intr()`

Table 4-3. An example of IRQ assignment to I/O devices

IRQ	INT	Hardware device
0	32	Timer
1	33	Keyboard
2	34	PIC cascading
3	35	Second serial port
4	36	First serial port
6	38	Floppy disk
8	40	System clock
10	42	Network interface
11	43	USB port, sound card
12	44	PS/2 mouse
13	45	Mathematical coprocessor
14	46	EIDE disk controller's first chain
15	47	EIDE disk controller's second chain

Bovet, Daniel P., and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management*. " O'Reilly Media, Inc.", 2005.

Interrupt Handling

v2.6

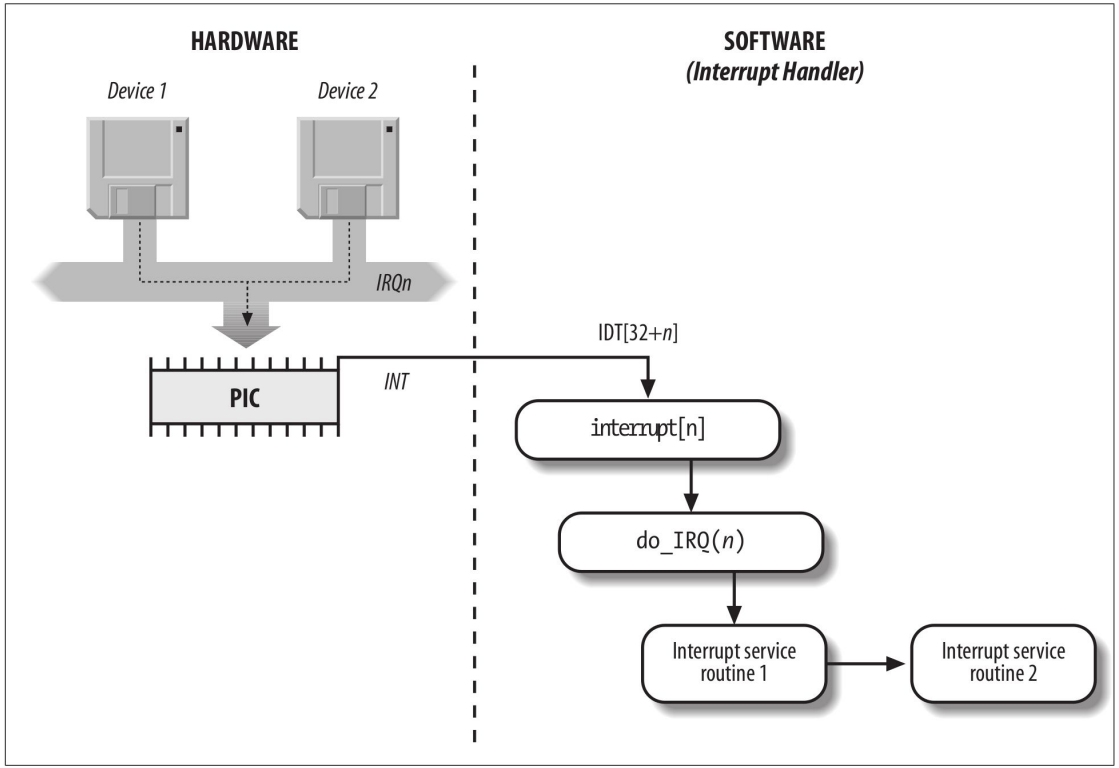


Figure 4-4. I/O interrupt handling

Bovet, Daniel P., and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management*. " O'Reilly Media, Inc.", 2005.

IRQ Data Structures

v2.6

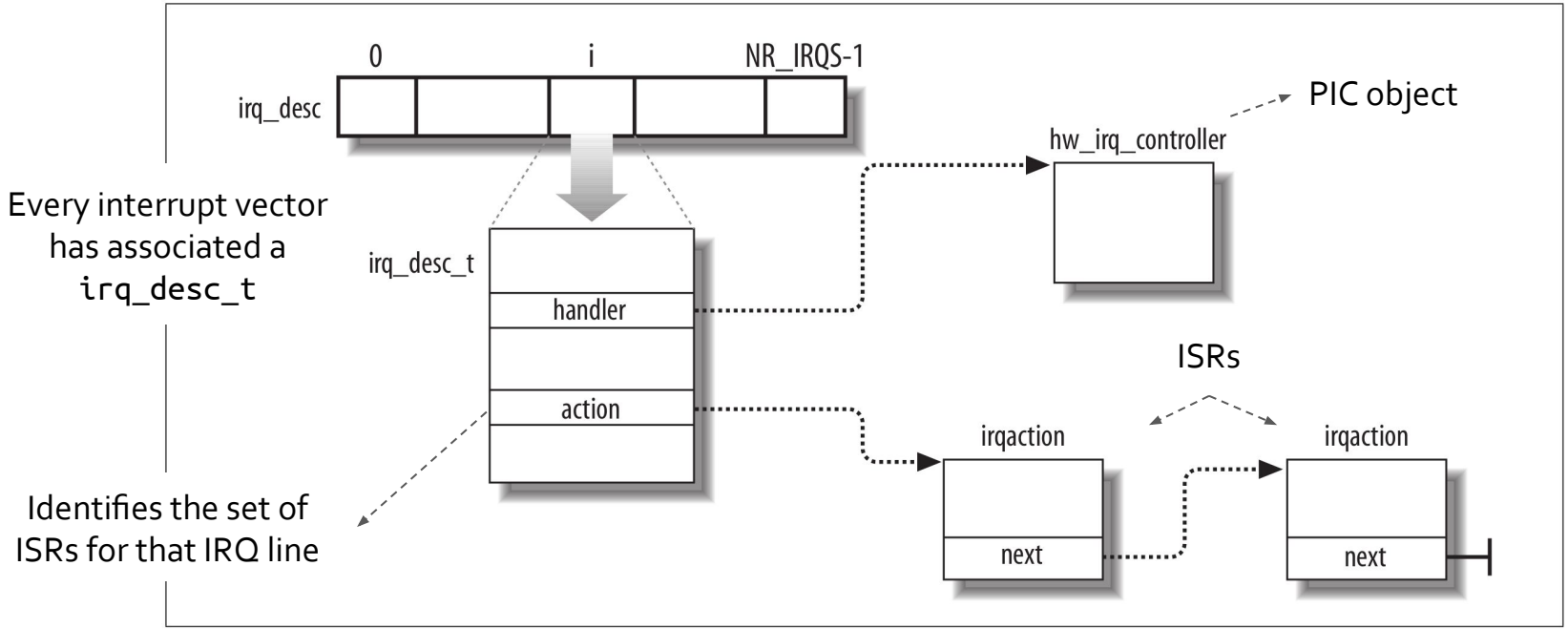


Figure 4-5. IRQ descriptors

Bovet, Daniel P., and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management*. " O'Reilly Media, Inc.", 2005.

Multiple Kernel Mode stacks

The struct that represents a process (that we will later) is couple with a data structure which represents the Kernel Mode stack. If it is declared (at compile time) as 8KB size, the stack will be used for any kind of kernel activity, otherwise if we declare it as 4KB size three types of stacks will be used (a C union):

- the **exception** stack, used for handling exceptions (including system calls). One for each process;
- the **hard IRQ** stack, used for handling interrupts. One for each CPU;
- the **soft IRQ** stack, used for handling deferrable activities (SoftIRQs and Tasklets). One for each CPU.

Interrupt Handler Activation

v2.6

Interrupt


```
pushl $n-256
jmp common_interrupt
```

```
common_interrupt:
    SAVE_ALL
    movl %esp,%eax
    call do_IRQ
    jmp ret_from_intr
```

As in the exceptions, when the CPU receives the interrupts it starts executing the code found in the corresponding gate of the IDT. For doing this, there is a context switch and registers must be saved and restored.

Saving the registers is the first task of the interrupt handler, as in the exceptions there is a stub code that is executed before the true interrupt handler, that is `do_IRQ`.

Modern Handler Activation

In modern Linux kernel versions the `do_IRQ` has been removed in favour of a more efficient implementation of handlers. The execution path is optimized for different types of interrupts, that are:

- **Level** type - the signal voltage is above a certain threshold (e.g. > 5V)
- **Edge** type - the signal is positive or negative
- **Simple** type - simple interrupt handling with no chip interaction
- **Fast EOI** type - the signal allows a fast End-of-Interrupt interaction
- **Per CPU** type - interrupt is per cpu

The rest of the rationale behind the interrupts remains more or less the same.

https://www.kernel.org/doc/html/latest/core-api/genericirq.html
http://ww2.cs.fsu.edu/~rosentha/linux/2.6.26.5/docs/DocBook/genericirq/ch04s03.html

5.5.2

5. Interrupts Management

5. Interrupts Handling

Inter-Processor Interrupts (IPIs)

DIAG

Interrupts in multi-core machines

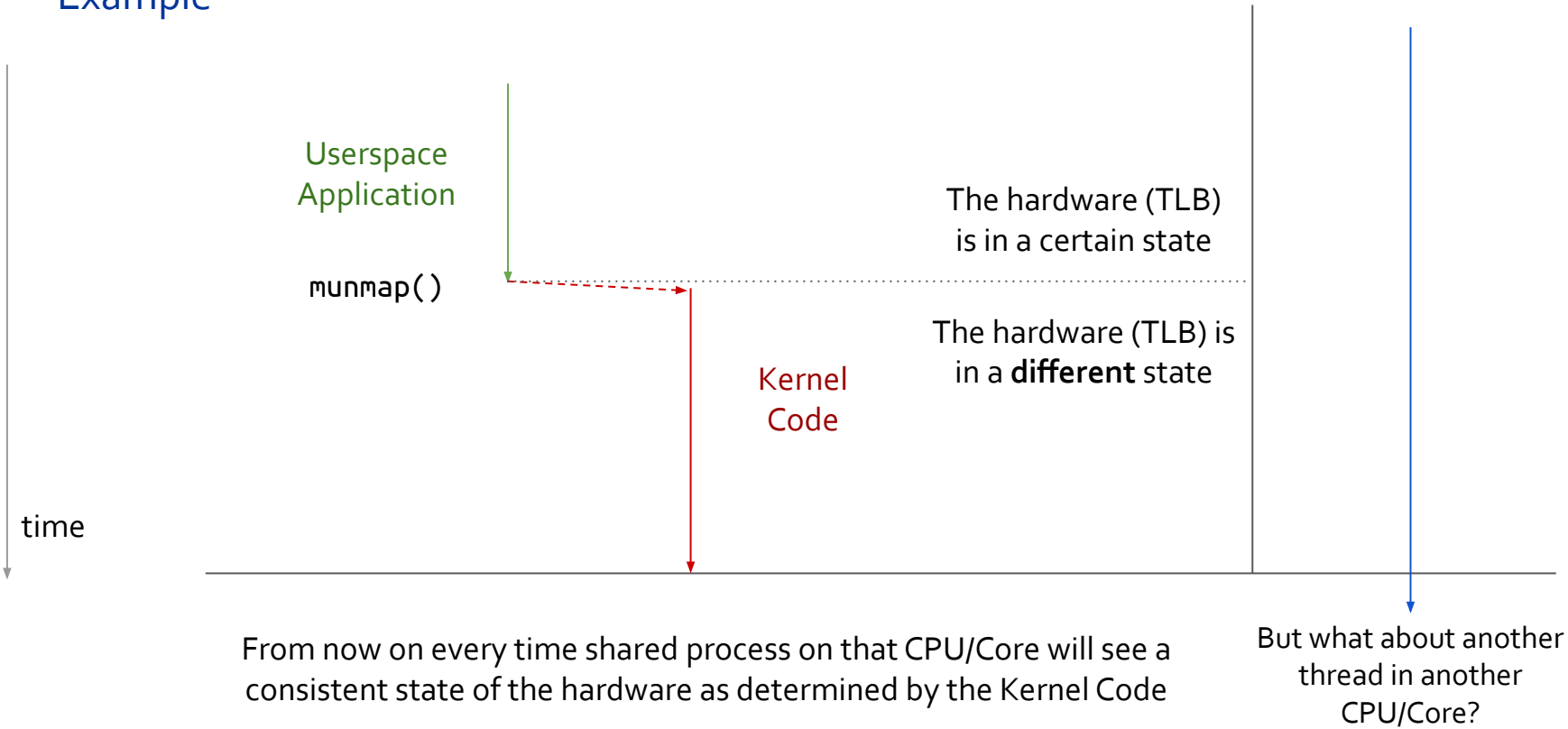
On single core machines, interrupt/trap events are managed by running operating system code on the single core in the system. This is sufficient to ensure consistency also in multithreaded applications. In multi-core systems, an interrupt/trap event is delivered to only one core:

- other cores might be running other threads of the same application, though. This can lead to race conditions or inconsistent state, due to the replication of hardware
- the hardware is time-shared across threads

We need a way to propagate an interrupt/trap event to other cores, if needed. The same problem holds for synchronous requests from userspace implemented without using traps (e.g., via the vDSO).

Memory Unmapping

Example



IPIs

IPIs are interrupts also used to trigger the execution of specific operating system functions on other cores. IPI are used to enforce cross-core activities (e.g. request/reply protocols) allowing a specific core to trigger a change in the state of another. IPIs are generated at firmware level, but are processed at software level:

- **synchronous** at the sender
- **asynchronous** at the receiver

At least two priority levels are available:

- **High priority** leads to immediate processing of the IPI at the recipient (a single IPI is accepted and stands out at any point in time)
- **Low priority** generally lead to queueing the requests and process them in a serialized way

IPIs Vectors

We have already seen the registers to trigger IPIs and the underlying (L)APIC architecture.

Usages of IPIs are:

- waking up additional cores
- execution of the same function across all the CPU cores (cross-core kernel synchronization)
- change of the hardware state across multiple cores in the system (e.g. TLB)

The **immediate handling** of the IPI is allowed when there's no need to share data across cores, as for example the *system halt* due to a kernel panic.

The kernel provides a set of macros and functions to easily trigger IPIs, the different IPI kinds of interrupt are referred as IPIs Vectors.

IPIs Vectors

V5.11

The Linux kernel makes use of three kinds if inter-processor interrupts:

- `CALL_FUNCTION_VECTOR (0xfb)` - it is sent to all CPUs but the sender forcing that CPU to run a function passed by the sender. The handler is called `call_function_interrupt()`. This is used for example for halting the system. This is interrupt is usually triggered by the function `smp_call_function()`;
- `RESCHEDULE_VECTOR (0xfc)` - when a CPU receives this type of interrupt, the corresponding handler named `reschedule_interrupt()` limits itself to acknowledging the interrupt, the rescheduling is done when returning from the interrupt;
- `INVALIDATE_TLB_VECTOR (0xfd)`- it is sent to all the CPU but the sender forcing them to invalidate the TLB, the handler is `invalidate_interrupt()`.

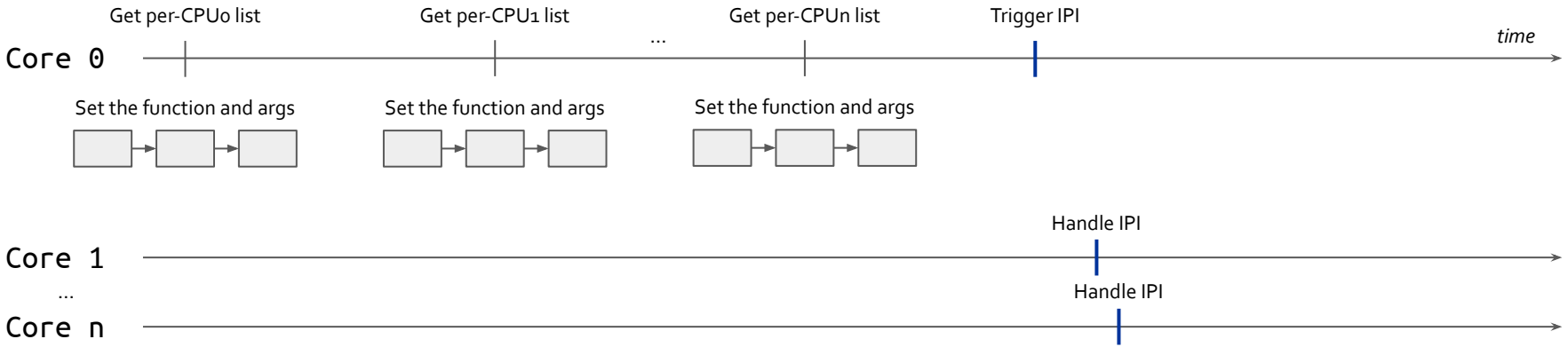
These vectors are issued by wrapper C functions that in the end call `default_send_IPI_all()`, `default_send_IPI_allbutself()`, `default_send_IPI_self()`, `default_send_IPI_mask( )`.

Registering IPI Functions

V5.11

IPIs are used to scheduled multiple cross-core tasks, but a single vector exists (`CALL_FUNCTION_VECTOR`). There is the need to register a specific action associated with the firing of an IPI. Older version of the kernel were relying on a global data structure protected by a lock, but this solution hampers scalability and performance.

From Kernel 5.0, there is a per-CPU linked list of registered functions and associated data to process. Concurrent access relies on the lock-free list.



5.6

5. Interrupts Management

Software Interrupts (SoftIRQs)

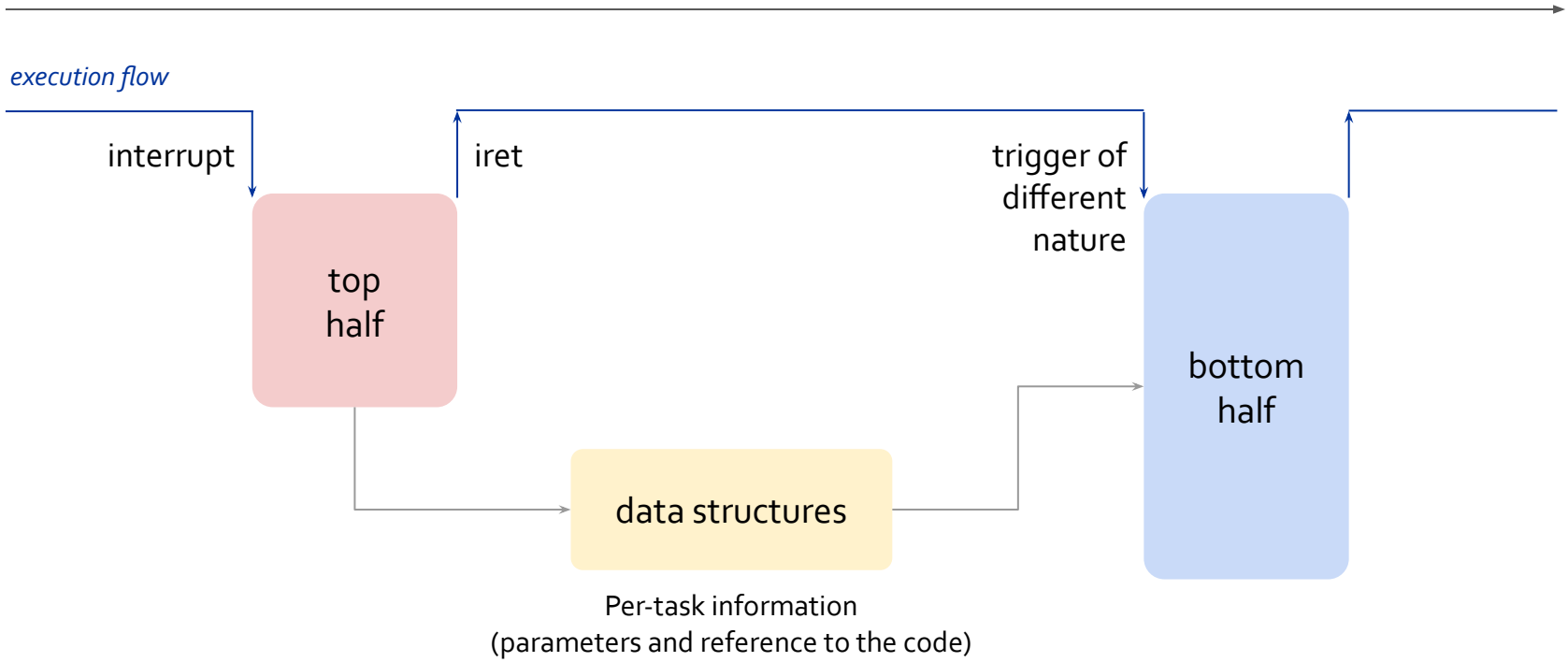
DIAG

Deferred Work

We already introduced the necessity of deferring the non-critical work when handling an interrupt. The basic idea behind this strategy takes the name of top-half/bottom-half:

- the **top-half** executes a minimal amount of work which is mandatory to later finalize the whole interrupt management. The top-half code is managed according to a non-interruptible scheme and it is in charge of scheduling the bottom-half task by queuing a record into a proper data structure;
- the **bottom-half** finalizes the work to be done for completing the interrupt handling. The bottom-half executes the deferred work with interrupts enabled.

Top/Bottom Halves



SoftIRQs and Tasklets

Linux uses two kinds of non-urgent and interruptible kernel functions:

- the ***deferrable functions***, that are softIRQs and tasklets;
- those executed by means of some work queues.

Tasklets are built on top the SoftIRQs and the term softirq which often appears in the kernel source refers to both of them. The main differences between these two kinds of deferrable functions are:

- *SoftIRQs* are **statically** allocated, they can run **concurrently** on several CPUs (even if they are of the different type, they are **reentrant** functions and must explicitly **protect** their data structures with *spinlocks*;
- *Tasklets* are initialized at **runtime** (for instance when mounting a kernel module), they do **not need to worry** about race conditions on data structures since they are strictly controlled by the kernel. Tasklets of the same type are always **serialized**, they cannot run concurrently, they do **not** need to be **reentrant**.

Main Steps

The main steps carried out on a deferrable function are the following.

1. **Initialization**. Define a new deferrable function, this is done when the kernel boots or when a module is loaded.
2. **Activation**. Marks the function as "pending" to be run the next time the kernel schedules a round of executions of deferrable functions.
3. **Masking**. Selectively disable a function so that it will be not executed even if activated.
4. **Execution**. Executes a pending deferrable function with other functions of the same type.

SoftIRQs are also called *software interrupts* but keep in mind that they are different from the programmed exceptions.

SoftIRQs

V5.11

Linux uses a limited number of softirqs. In general, Tasklets are used because are easier to write and they do not need to be reentrant.

```
531  /* PLEASE, avoid to allocate new softirqs, if you need not _really_ high
532     frequency threaded job scheduling. For almost all the purposes
533     tasklets are more than enough. F.e. all serial device BHs et
534     al. should be converted to tasklets, not to softirqs.
535   */
536
537  enum
538  {
539          HI_SOFTIRQ=0,
540          TIMER_SOFTIRQ,
541          NET_TX_SOFTIRQ,
542          NET_RX_SOFTIRQ,
543          BLOCK_SOFTIRQ,
544          IRQ_POLL_SOFTIRQ,
545          TASKLET_SOFTIRQ,
546          SCHED_SOFTIRQ,
547          HRTIMER_SOFTIRQ,
548          RCU_SOFTIRQ,    /* Preferable RCU should always be the last softirq */
549
550          NR_SOFTIRQS
551  };
```

High priority tasklets

Transmit/Receive packets to/from NICs

Normal tasklets

https://elixir.bootlin.com/linux/latest/source/include/linux/interrupt.h#L531

/proc/softirqs

```
~$ cat /proc/softirqs
                    CPU0       CPU1       CPU2       CPU3       CPU4       CPU5       CPU6       CPU7
          HI:          5          0          0          0          0          0          0          0
       TIMER:     332519     310498     289555     272913     282535     279467     282895     270979
      NET_TX:       2320          0          0          2          1          1          0          0
      NET_RX:     270221        225        338        281        311        262        430        265
       BLOCK:     134282         32         40         10         12          7          8          8
BLOCK_IOPOLL:          0          0          0          0          0          0          0          0
     TASKLET:     196835          2          3          0          0          0          0          0
       SCHED:     161852     146745     129539     126064     127998     128014     120243     117391
     HRTIMER:          0          0          0          0          0          0          0          0
         RCU:     337707     289397     251874     239796     254377     254898     267497     256624
```

SoftIRQs

V5.11

Data Structures

The vector softirq contains the description of each available SoftIRQ

```
55  static struct softirq_action softirq_vec[NR_SOFTIRQS] __cacheline_aligned_in_smp;
```

The `softirq_action` data structure contains the pointer to the softirq function (`void*`).

```
564  struct softirq_action
565  {
566          void    (*action)(struct softirq_action *);
567  };
```

Initialization

The initialization of softirqs is done at boot time with the function `open_softirq()`

```
486  void open_softirq(int nr, void (*action)(struct softirq_action *))
487  {
488          softirq_vec[nr].action = action;
489  }
```

https://oxax.gitbooks.io/linux-insides/content/Interrupts/linux-interrupts-9.html

SoftIRQs

V5.11

Activation

The SoftIRQs are activated by means of the function `raise_softirq()`

```
470  void raise_softirq(unsigned int nr)
471  {
472          unsigned long flags;
473
474          local_irq_save(flags);
475          raise_softirq_irqoff(nr);
476          local_irq_restore(flags);
477  }
```

https://elixir.bootlin.com/linux/latest/source/kernel/softirq.c#L470

Save the state of IF flag of EFLAGS register and disable interrupts on local CPU

Mark the interrupt as pending (in the local cpu bitmap) and wake softirqd if not in interrupt context

```
453  inline void raise_softirq_irqoff(unsigned int nr)
454  {
455          __raise_softirq_irqoff(nr);
456
457          /*
458           * If we're in an interrupt or softirq, we're done
459           * (this also catches softirq-disabled code). We will
460           * actually run the softirq once we return from
461           * the irq or softirq.
462           *
463           * Otherwise we wake up ksoftirqd to make sure we
464           * schedule the softirq soon.
465           */
466          if (!in_interrupt())
467                  wakeup_softirqd();
468  }
```

https://elixir.bootlin.com/linux/latest/source/kernel/softirq.c#L453

Restore the saved state of IF flag of EFLAGS register and re-enable interrupts on local CPU

SoftIRQs

Activation

The checks for pending softirq should be performed periodically but without too much overhead. Here's a list of significant points in the kernel in which the check is done:

- when softirqs are enabled on local CPU;
- when `do_IRQ()` finished processing;
- after a timer interrupt on LAPIC;
- after a `CALL_FUNCTION_VECTOR`;
- when a *ksoftirqd/n* kernel thread is wakened.

SoftIRQs

V5.11

Execution - do_softirq()

If there is a pending softirq then the function `do_softirq()` is invoked. The function:

1. checks if invoked in a interrupt context or softirqs are disabled, if yes returns
2. executes `local_irq_save()`
3. checks if there are pending softirq
4. calls `do_softirq_own_stack` this function switches to needed and calls `__do_softirq()`
5. calls `local_irq_restore()`

```
233  asmlinkage __visible void do_softirq(void)
234  {
235          __u32 pending;
236          unsigned long flags;
237
238          if (in_interrupt())
239                  return;
240
241          local_irq_save(flags);
242
243          pending = local_softirq_pending();
244
245          if (pending && !ksoftirqd_running(pending))
246                  do_softirq_own_stack();
247
248          local_irq_restore(flags);
249  }
```

https://elixir.bootlin.com/linux/latest/source/kernel/softirq.c#L233

SoftIRQs

V5.11

Execution - `__do_softirq()`

The `__do_softirq()` reads the bit mask of the local CPU and executes the deferrable functions corresponding to every set bit. While executing a softirq, another softirq may pop up and in order to avoid that `__do_softirq()` never regain control to user processes it only executes a **fixed number of iterations**, the remaining softirqs will be handled by ksoftirqd daemon.

The function `__do_softirq()`:

1. initializes the iteration counter to 10
2. copies the bitmap of the local cpu
3. disable softirqs in the local cpu, since softirq are executed serially
4. clears the bitmap of local cpu
5. executes `local_irq_enable()`
6. executes the action for every set bit
7. executes `local_irq_disable()`
8. copies the bitmap of the local cpu and decrease the iteration counter
9. if another softirq has been activated and iterations > 0 jump to 4.
10. if there are more softirq invokes `wakeup_softirqd()`
11. Re-enable softirqs in the local cpu

ksoftirqd

In modern kernel versions, each CPU has its own `ksoftirqd/n` kernel thread (where n is the logical number of the CPU). Each `ksoftirqd` runs the `ksoftirq()` function which essentially executes the following loop:

```
for(;;) {
    set_current_state(TASK_INTERRUPTIBLE);
    schedule();
    /* now in TASK_RUNNING state */
    while (local_softirq_pending()) {
        preempt_disable();
        do_softirq();
        preempt_enable();
        cond_resched();
    }
}
```

Bovet, Daniel P., and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management*. " O'Reilly Media, Inc.", 2005.

ksoftirqd

```
[gpm@fedora-xps ~]$ ps -aux | grep ksoft
root          13  0.0  0.0      0     0 ?        S    Apr05   0:00 [ksoftirqd/0]
root          19  0.0  0.0      0     0 ?        S    Apr05   0:00 [ksoftirqd/1]
root          24  0.0  0.0      0     0 ?        S    Apr05   0:00 [ksoftirqd/2]
root          29  0.0  0.0      0     0 ?        S    Apr05   0:00 [ksoftirqd/3]
root          34  0.0  0.0      0     0 ?        S    Apr05   0:00 [ksoftirqd/4]
root          39  0.0  0.0      0     0 ?        S    Apr05   0:00 [ksoftirqd/5]
root          44  0.0  0.0      0     0 ?        S    Apr05   0:00 [ksoftirqd/6]
root          49  0.0  0.0      0     0 ?        S    Apr05   0:00 [ksoftirqd/7]
```

5.7

5. Interrupts Management

Tasklets

DIAG

Tasklets

Tasklets are the preferred way to implement deferrable functions in I/O drivers or in kernel modules. As already introduced, tasklets are built on top of two softirqs named `HI_SOFTIRQ` and `TASKLET_SOFTIRQ`, they differ only in priority since several tasklets may be associated with the same softirq, each carrying its own function.

For using a tasklet you need to allocate the `tasklet_struct` by means of the macro `DECLARE_TASKLET` and then call one of the following functions to enable/disable it:

- `tasklet_enable(struct tasklet_struct *tasklet)`
- `tasklet_hi_enable( struct tasklet_struct *);`
- `tasklet_disable(struct tasklet_struct *tasklet)`
- `void tasklet_schedule(struct tasklet_struct *tasklet)`

Unless a tasklet reactivates itself, every tasklet activation triggers **at most one** execution of the tasklet function. Management of tasklets is such that a tasklet of the same kind cannot be run concurrently on two different cores

How Tasklets are run

Tasklets are run using Soft IRQs. Enable functions are mapped to Soft IRQs lines:

- tasklet_enable() mapped to `TASKLET_SOFTIRQ`
- tasklet_hi_enable() mapped to `HI_SOFTIRQ`

No real difference between the two, except that `do_softirq()` processes `HI_SOFTIRQ` before `TASKLET_SOFTIRQ`. All non-disabled Tasklets are executed, before the corresponding SoftIRQ action completes.

Remember that they are run with HardIRQs enabled.

Modern API

In the latest version of the kernel the Tasklets API is deprecated in favour of Threaded IRQs.

```
595  /* Tasklets --- multithreaded analogue of BHs.
596
597     This API is deprecated. Please consider using threaded IRQs instead:
598     https://lore.kernel.org/lkml/20200716081538.2sivhkj4hcyrusem@linutronix.de
599
```

https://elixir.bootlin.com/linux/latest/source/include/linux/interrupt.h#L595

The Tasklet API (but also the SoftIRQ) will be removed because the top-half of an IRQ is executed in a kernel thread by using the function `request_threaded_irq()` for allocating a IRQ line.

For further information see https://lwn.net/Articles/302043/.

5.8

5. Interrupts Management

Work Queues

DIAG

Work Queues

v2.6

The *worker queues* have been introduced in Linux 2.6. They are similar to the deferrable functions, but they are run by ad-hoc kernel-level worker threads.

Worker Queues always run in **process context** and they can perform blocking operations but this does not mean that they can access user address space (as the deferrable functions). Executing in process context is the only way for performing blocking operations (e.g. accessing data to disk), remind that no process switch can occur in interrupt context.

A work queue is defined by the `workqueue_struct` whose field `worklist` points to a doubly linked list of pending functions.

APIs

v2.6

Creating a queue

The function `create_workqueue("foo")` allows to create a new work queue and also creates n worker threads (where n is the number of CPUs). You can use the function `create_singlethread_workqueue()` for creating a work queue with only one thread. You can destroy the queue with the function `destroy_workqueue()`.

After creating a queue you can use:

- `queue_work()` for inserting a function (packaged in a work_struct) to the queue
- `queue_delayed_work()` for inserting a function that will be executed when after the passed time delay

A worker thread continuously loop inside the function `worker_thread()` that most of the time is sleeping if there is no function to be executed.

Sometimes it may be necessary to wait until all pending functions are executed, in that case, the function `flush_workqueue()` can be used.

APIs

v2.6

The predefined work queue

In most cases, creating a whole set of worker threads in order to run a function is overkill. Therefore, the kernel offers a predefined work queue called *events*, which can be freely used by every kernel developer.

To use the predefined queue you can use the following functions:

Table 4-14. Helper functions for the predefined work queue

Predefined work queue function	Equivalent standard work queue function
`schedule_work(w)`	`queue_work(keventd_wq,w)`
`schedule_delayed_work(w,d)`	`queue_delayed_work(keventd_wq,w,d)` (on any CPU)
`schedule_delayed_work_on(cpu,w,d)`	`queue_delayed_work(keventd_wq,w,d)` (on a given CPU)
`flush_scheduled_work()`	`flush_workqueue(keventd_wq)`

Bovet, Daniel P., and Marco Cesati. *Understanding the Linux Kernel: from I/O ports to process management*. " O'Reilly Media, Inc.", 2005.

kworkers

```
[gpm@fedora-xps ~]$ ps -aux | grep kworker
root           6  0.0  0.0      0     0 ?        I<   Apr05   0:00 [kworker/0:0H-events_highpri]
root          21  0.0  0.0      0     0 ?        I<   Apr05   0:00 [kworker/1:0H-events_highpri]
root          26  0.0  0.0      0     0 ?        I<   Apr05   0:00 [kworker/2:0H-events_highpri]
root          31  0.0  0.0      0     0 ?        I<   Apr05   0:00 [kworker/3:0H-events_highpri]
root          36  0.0  0.0      0     0 ?        I<   Apr05   0:00 [kworker/4:0H-events_highpri]
root          41  0.0  0.0      0     0 ?        I<   Apr05   0:00 [kworker/5:0H-events_highpri]
root          46  0.0  0.0      0     0 ?        I<   Apr05   0:00 [kworker/6:0H-events_highpri]
root          51  0.0  0.0      0     0 ?        I<   Apr05   0:00 [kworker/7:0H-events_highpri]
root         137  0.0  0.0      0     0 ?        I<   Apr05   0:00 [kworker/6:1H-events_highpri]
root         152  0.0  0.0      0     0 ?        I<   Apr05   0:00 [kworker/3:1H-events_highpri]
root         233  0.0  0.0      0     0 ?        I<   Apr05   0:00 [kworker/4:1H-events_highpri]
root         310  0.0  0.0      0     0 ?        I<   Apr05   0:00 [kworker/5:1H-kblockd]
root         328  0.0  0.0      0     0 ?        I<   Apr05   0:00 [kworker/7:1H-events_highpri]
root         376  0.0  0.0      0     0 ?        I<   Apr05   0:00 [kworker/0:1H-kblockd]
root         467  0.0  0.0      0     0 ?        I<   Apr05   0:00 [kworker/2:1H-events_highpri]
root         556  0.0  0.0      0     0 ?        I<   Apr05   0:00 [kworker/1:1H-events_highpri]
root       70810  0.0  0.0      0     0 ?        I    Apr05   0:00 [kworker/5:0-events_power_efficient]
root       72123  0.0  0.0      0     0 ?        I    Apr05   0:00 [kworker/1:2-cgroup_destroy]
root       73440  0.0  0.0      0     0 ?        I    Apr05   0:00 [kworker/6:1-events]
root       73920  0.0  0.0      0     0 ?        I<   Apr05   0:01 [kworker/u17:0-rb_allocator]
root       75463  0.0  0.0      0     0 ?        I    Apr05   0:00 [kworker/u16:4-i915]
root       75552  0.0  0.0      0     0 ?        I<   Apr05   0:02 [kworker/u17:1-i915_flip]
...
```

Advanced Operating Systems and Virtualization

[5] Interrupts Management

LECTURER

Gabriele **Proietti Mattia**

BASED ON WORK BY

http://www.ce.uniroma2.it/~pellegrini/

gpm.name · proiettimattia@diag.uniroma1.it

DIAG