CS 5264/4224; ECE 5414/4414
(Advanced) Linux Kernel Programming
Lecture 10

Interrupts

February 27, 2025

Huaicheng Li

https://people.cs.vt.edu/huaicheng/lkp-sp25/

Acknowledgement: Credits to Dr. Changwoo Min for the original LKP lecture slides.

Interrupts: What? Why? and How?

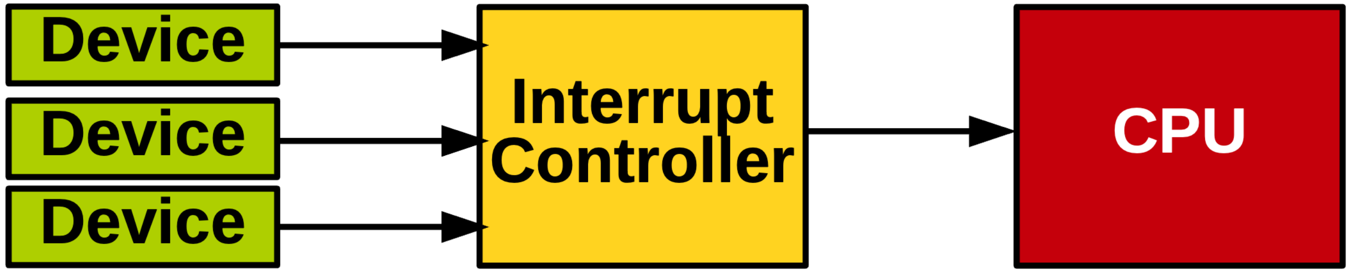
- A mechanism to implement abstraction and multiplexing
- Interrupt: asking for a service from the kernel
  - via software (e.g., "int 0x80") or by hardware (e.g., keyboard)
- Interrupt handling in Linux
  - how to track interrupts
  - how to handle them
    - » top half + bottom half

Interrupts

- Compared to the CPU, devices are slow
  - The kernel must be free to go and handle other work, dealing with the hardware only after the hardware has completed some work
- How to know the completion of hardware operations?
  - Polling: busy-waiting (e.g., in a while loop), periodically checking the hardware status
  - Interrupts: the hardware signals its completion to the processor
- Interrupt examples
  - Completion of disk read (e.g., the disk has read 4KB data and sent it to the host)
  - Key press on a keyboard
  - Network packet arrival (e.g., NIC receives one network packet)

Interrupt Controller

- Interrupts are electrical signals multiplexed by the interrupt controller
  - Sent to a specific pin of the CPU
- Once an interrupt is received, a dedicated function will be executed
  - interrupt handler (isr)
- The kernel/user space can be interrupted at (nearly) any time to process interrupts



Advanced PIC (APIC, I/O APIC)

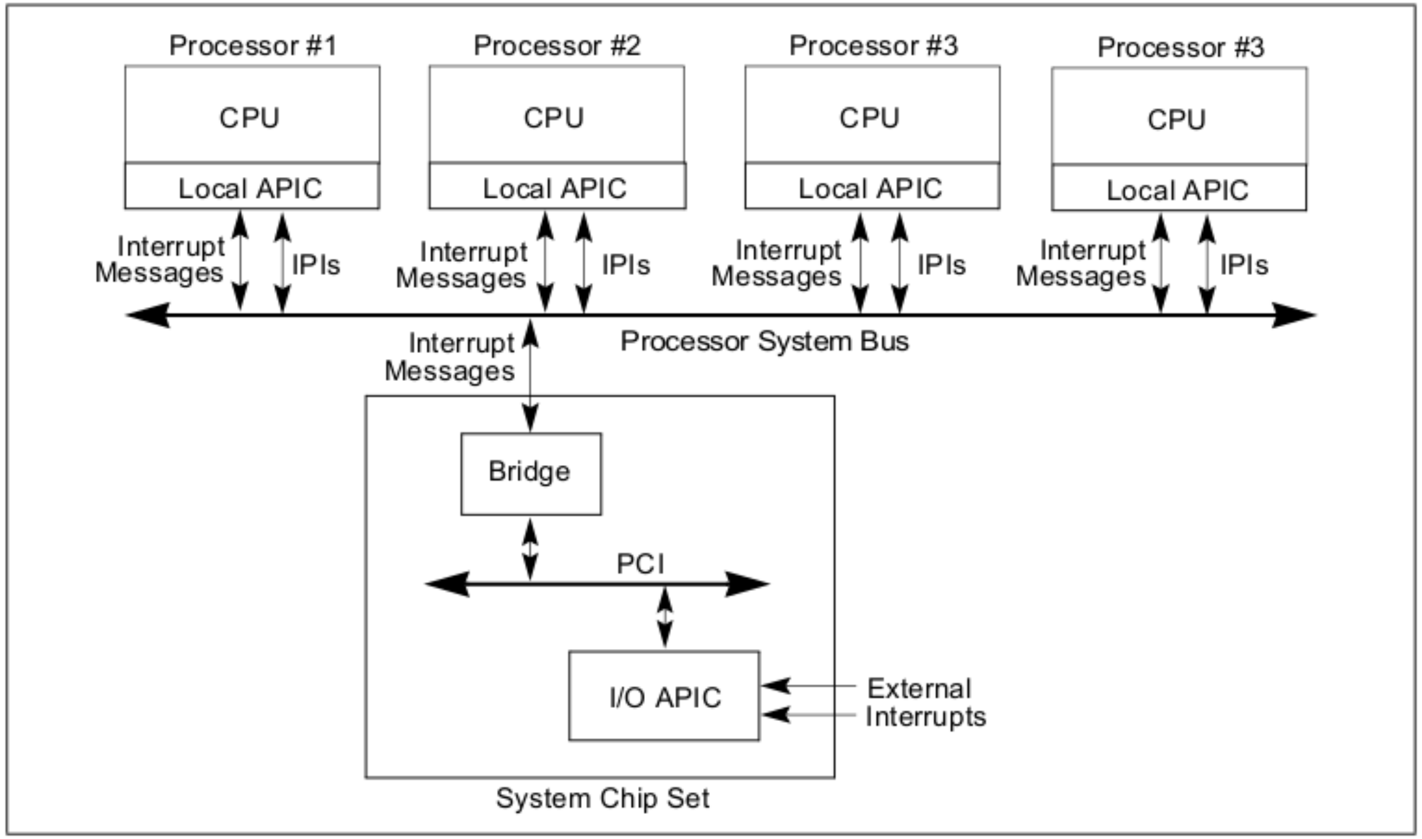
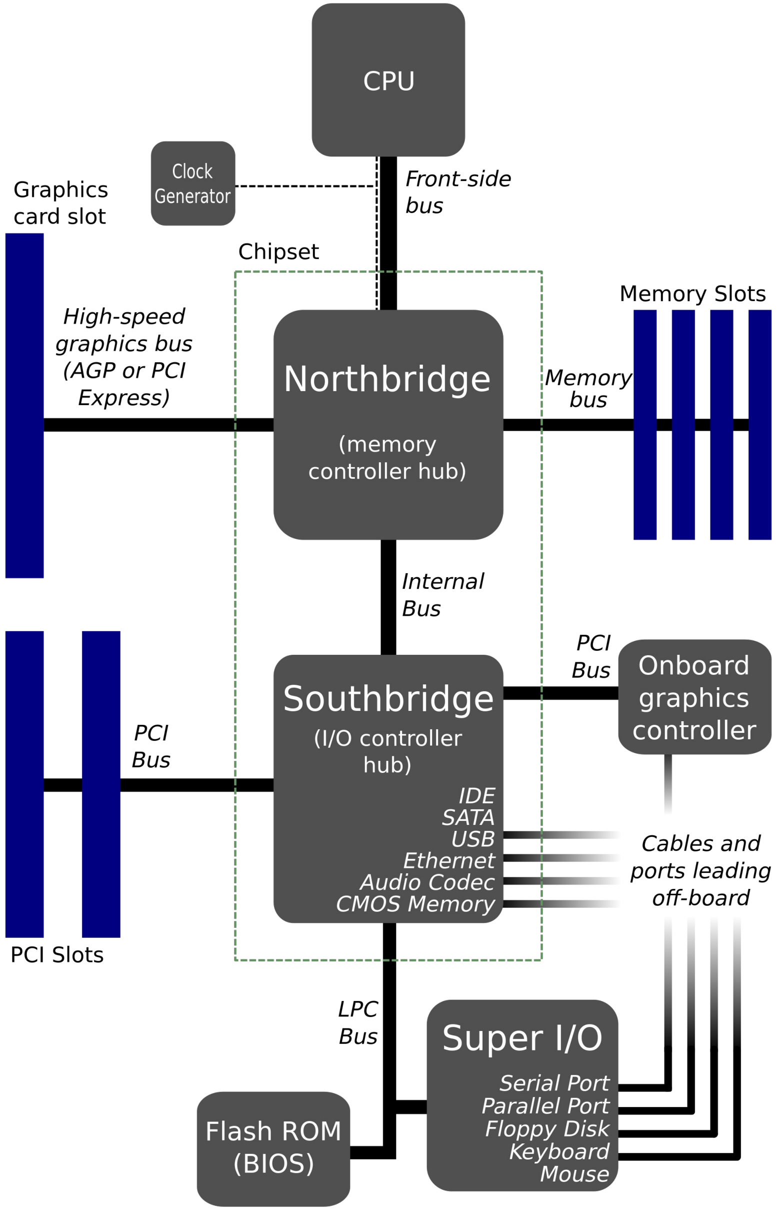
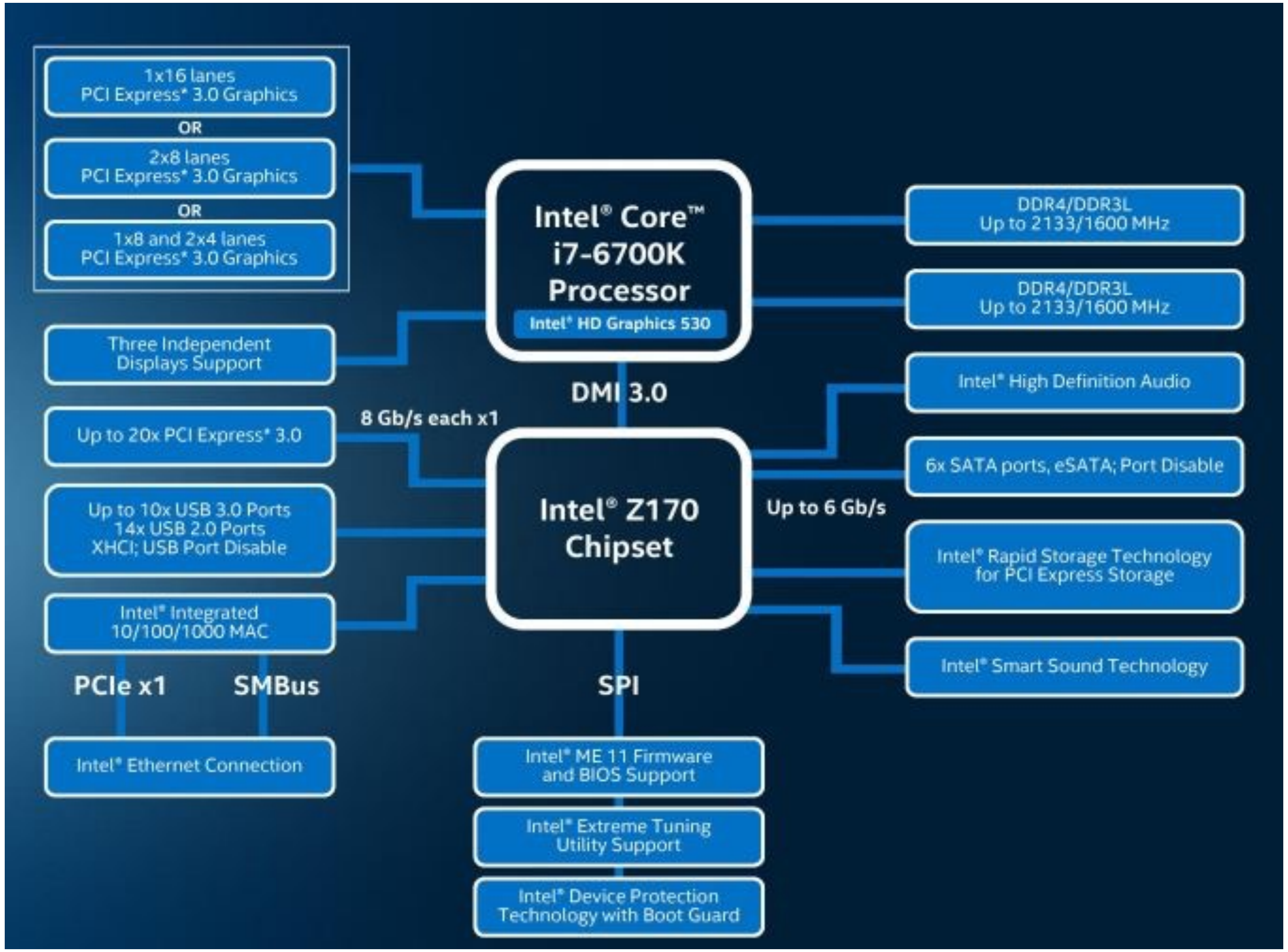


Figure 10-2. Local APICs and I/O APIC When Intel Xeon Processors Are Used in Multiple-Processor Systems

Source: Intel software development manual (SDM)

- I/O APIC
  - system chipset (south bridge)
  - redistribute interrupts to local APICs
- Local APIC
  - inside a processor chip
  - has a timer, which raises timer interrupt
  - issues IPIs (inter-processor interrupt)


CPU
Clock Generator
Front-side bus
Graphics card slot
Chipset
Memory Slots
High-speed graphics bus (AGP or PCI Express)
Northbridge
(memory controller hub)
Memory bus
Internal Bus
PCI Bus
Onboard graphics controller
Southbridge
(I/O controller hub)
IDE
SATA
USB
Ethernet
Audio Codec
CMOS Memory
PCI Bus
Cables and ports leading off-board
PCI Slots
LPC Bus
Super I/O
Serial Port
Parallel Port
Floppy Disk
Keyboard
Mouse
Flash ROM (BIOS)


1x16 lanes PCI Express* 3.0 Graphics
OR
2x8 lanes PCI Express* 3.0 Graphics
OR
1x8 and 2x4 lanes PCI Express* 3.0 Graphics
Intel® Core™ i7-6700K Processor
Intel® HD Graphics 530
DDR4/DDR3L Up to 2133/1600 MHz
DDR4/DDR3L Up to 2133/1600 MHz
Three Independent Displays Support
DMI 3.0
8 Gb/s each x1
Up to 20x PCI Express* 3.0
Intel® High Definition Audio
6x SATA ports, eSATA; Port Disable
Up to 10x USB 3.0 Ports 14x USB 2.0 Ports XHCI; USB Port Disable
Intel® Z170 Chipset
Up to 6 Gb/s
Intel® Rapid Storage Technology for PCI Express Storage
Intel® Integrated 10/100/1000 MAC
Intel® Smart Sound Technology
PCIe x1
SMBus
SPI
Intel® Ethernet Connection
Intel® ME 11 Firmware and BIOS Support
Intel® Extreme Tuning Utility Support
Intel® Device Protection Technology with Boot Guard

Interrupt Request (IRQ)

- Interrupt line or interrupt request (IRQ)
  - device identifier, i.e., who generates the interrupt?
- e.g., in 8259A interrupt lines
  - IRQ 0: system timer
  - IRQ 1: keyboard controller
  - IRQ 3, 4: serial port
  - IRQ 5: terminal
- Some interrupt lines can be shared among several devices
  - e.g., for modern PCIe devices

Exceptions

- Exceptions are interrupts issued by the CPU
  - software interrupt, as opposed to hardware interrupts
  - Examples:
    - » program faults: division-by-zero, page fault, general protection fault, etc.
    - » Voluntary exceptions: "int" instruction, e.g., for syscall invocations (in the old days)
- Exceptions are managed by the kernel in the same way as hardware interrupts

Hardware Interrupt Interface

- Non-Maskable Interrupt (NMI)
  - Never get ignored, e.g., power failure, memory error
  - On x86, vector 2, prevent other interrupts from executing
- Maskable interrupts
  - Ignored when "IF" bit in "EFLAGS" is 0
  - Instructions to enable/disable interrupts:
    - "sti": set interrupt
    - "cli": clear interrupt
- INTA
  - interrupt acknowledgement
  - End of Interrupt (EOI)

“Software” Interrupt: INT

- Intentional interrupts
  - “int” instructions on x86
  - invokes the interrupt handler for the vectors, N in [0-255]
    - » N-th interrupt handler
  - Entering: ”int N”
  - Exiting: “iret”

Interrupt Descriptor Table (IDT)

- IDT
  - Table of 256 8-byte entries (similar to GDT)
  - located in memory
- IDTR register stores current IDT
- "lidt" instruction to load IDT
  - loads IDTR with address and size of the ID
  - Takes in a linear address

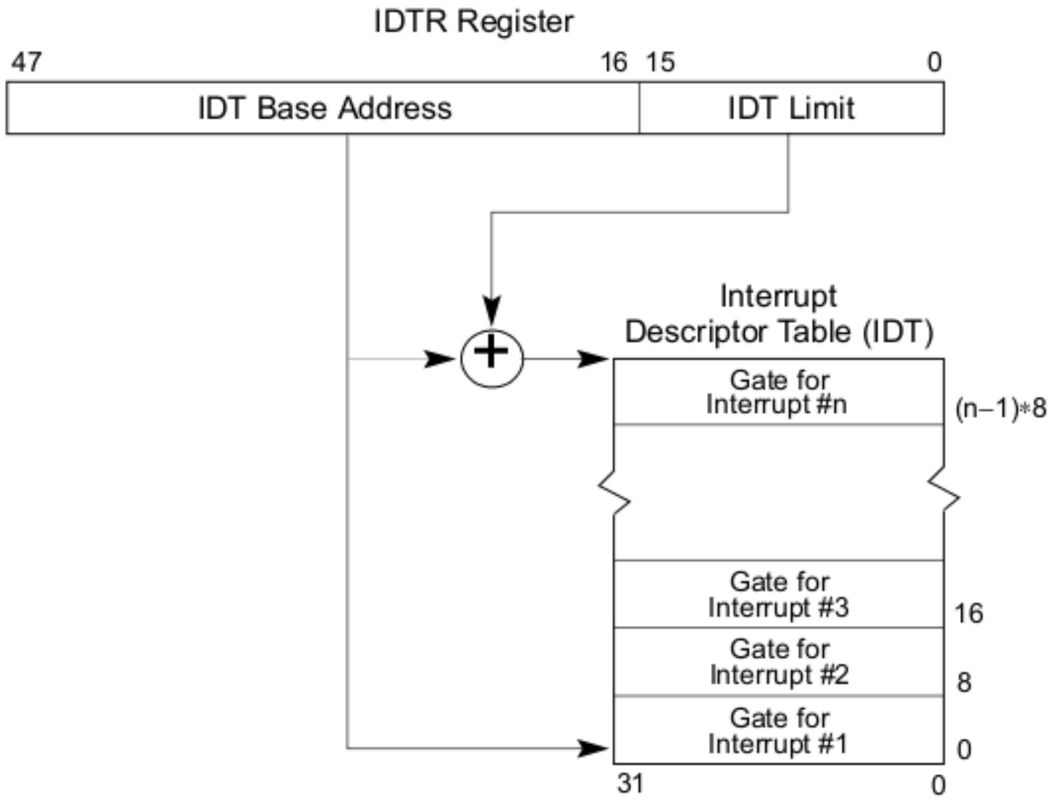
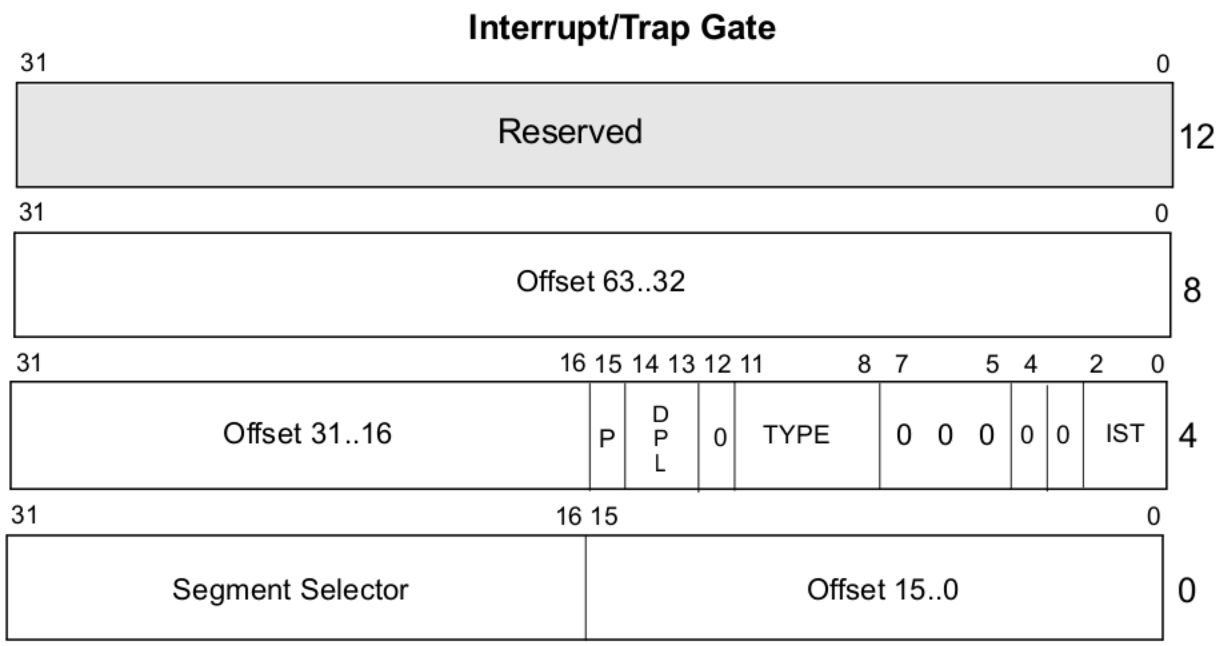


Figure 6-1. Relationship of the IDTR and IDT



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 6-7. 64-Bit IDT Gate Descriptors

Interrupt Descriptor Entry

- Offset is a 32-bit value split into two parts pointing to the destination IP or EIP
- Segment selector points to the destination CS in the kernel
- Present flag indicates that this is a valid entry
- Descriptor Privilege Level (DPL) indicates the minimum privilege level of the caller to prevent users from calling hardware interrupts directly
- Size of gate can be 32 bits or 16 bits

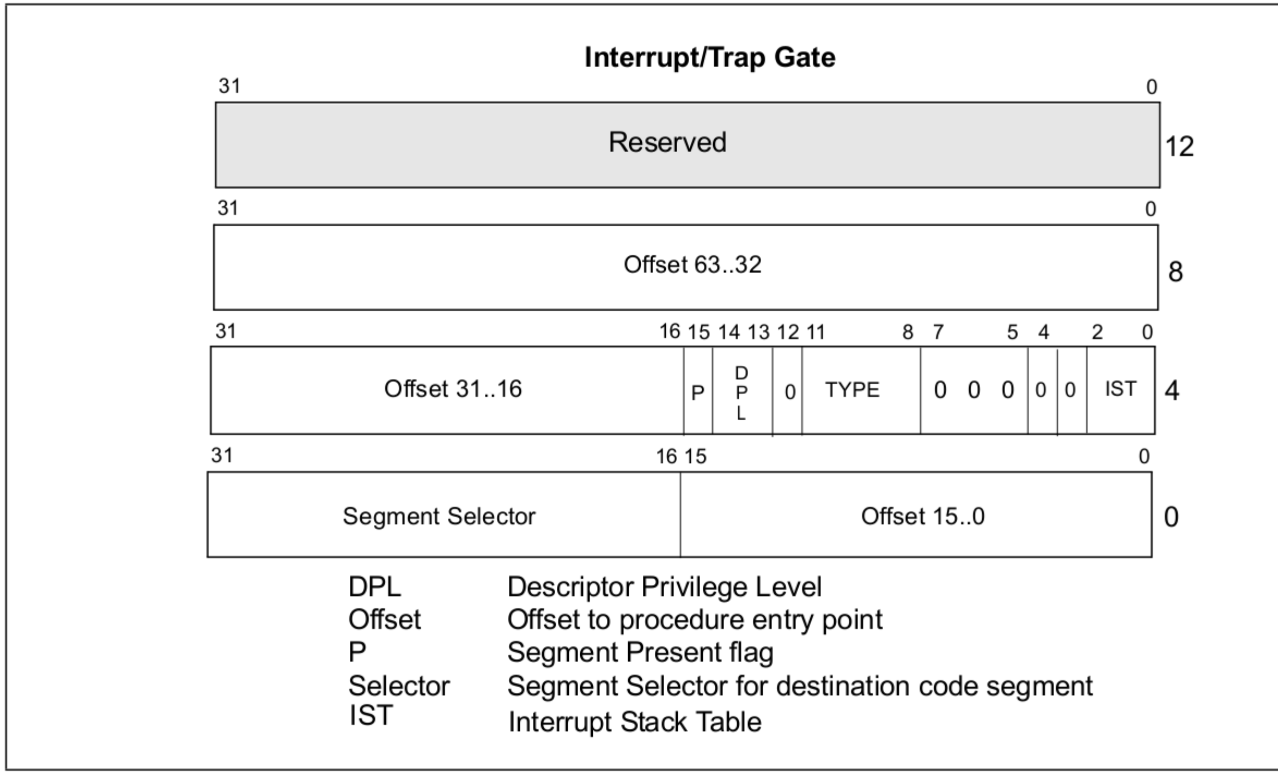


Figure 6-7. 64-Bit IDT Gate Descriptors

Predefined Interrupt Vectors

- 0: Divide Error
- 1: Debug Exception
- 2: Non-Maskable Interrupt
- 3: Breakpoing Exception (e.g., int 3)
- 4: Invalid Opcode
- 13: General Protection Fault
- 14: Page Fault
- 18: Machine (abort)
- 32-255: User Defined Interrupts

INT Instruction

- Fetch the interrupt descriptor for a vector (e.g., 0x80) from the IDT
  - IDT base addr + 0x80 * 8bytes
- Check that CPL <= DPL in the descriptor
- Save ESP and SS in a CPU-internal register
- Load SS and ESP from TSS (Task State Segment)
- Plush user SS, ESP, EFLAGS, CS, EIP
- Clear certain EFLAGS bits
- Set CS and EIP from IDT descriptor's segment selector and offset

Interrupt Service Routine (ISR)

- Interrupt handler or interrupt service routine (ISR)
  - functions executed by the CPU in response to a specific interrupt
- In Linux, a normal C function matching a specific prototype to pass in the handler information
- Runs in *interrupt context* (or atomic context)
  - Opposite to process context (system call)
  - A task cannot sleep in an ISR b/c … (?)

ISR Design Goals

- Interrupt processing should be *fast*
  - *minimizing disrupting user process execution (user/kernel space)*
  - *get back to other interrupts which might arrive during an ongoing interrupt processing*
- *Interrupt processing might involve much work to do*
  - *it takes time …*
  - *e.g., processing a network packet from the NIC*

Top-half vs. Bottom-half

- In Linux (and many other OSes), an interrupt process is split into two parts
- Top-half: run immediate upon receiving the interrupt
  - only handle time-critical operations, e.g., ack and reset interrupt
- Bottom-half: less critical & time-consuming work
  - Run later with other interrupts enabled
- An example: network packet processing
  - Top-half
    - » acknowledge the hardware, "hey, I received your signal"
    - » Copy packet to main memory
    - » Set the NIC to a status to receive more packets
    - » Critical: packet buffer on NIC is limited, might lead to packet drop if not processed timely
  - Bottom-half
    - » softirq, tasklet, workqueue
    - » Similar to thread pool in user-space

Registering an Interrupt Handler

```c
/* linux/include/linux/interrupt.h */

/**
 *  This call allocates interrupt resources and enables the
 *  interrupt line and IRQ handling.
 *
 *  @irq: Interrupt line to allocate
 *  @handler: Function to be called when the IRQ occurs.
 *        Primary handler for threaded interrupts
 *  @irqflags: Interrupt type flags
 *        IRQF_SHARED - allow sharing the irq among several devices
 *        IRQF_TIMER - Flag to mark this interrupt as timer interrupt
 *        IRQF_TRIGGER_* - Specify active edge(s) or level
 *  @devname: An ascii name for the claiming device
 *  @dev_id: A cookie passed back to the handler function
 *        Normally the address of the device data structure
 *        is used as the cookie.
 */

int request_irq(unsigned int irq, irq_handler_t handler,
    unsigned long irqflags, const char *devname, void *dev_id);
```

Freeing an Interrupt Handler

```
/* linux/include/linux/interrupt.h */

/**
 *  Free an interrupt allocated with request_irq
 *
 *  @irq: Interrupt line to free
 *  @dev_id: Device identity to free
 *
 *  Remove an interrupt handler. The handler is removed and if the
 *  interrupt line is no longer in use by any driver it is disabled.
 *  On a shared IRQ the caller must ensure the interrupt is disabled
 *  on the card it drives before calling this function. The function
 *  does not return until any executing interrupts for this IRQ
 *  have completed.
 *
 *  Returns the devname argument passed to request_irq.
 */
const void *free_irq(unsigned int irq, void *dev_id);
```

Writing an Interrupt Handler

```
/* linux/include/linux/interrupt.h */

/**
 * Interrupt handler prototype
 *
 * @irq: the interrupt line number that the handler is serving
 * @dev_id: a generic pointer that was given to request_irq()
 *      when the interrupt handler is registered
 *
 * Return value:
 *      IRQ_NONE: the interrupt is not handled (i.e., the expected
 *        device was not the source of the interrupt)
 *      IRQ_HANDLED: the interrupt is handled (i.e., the hanlder was
 *        correctly invoked)
 *      #define IRQ_RETVAL(x)   ((x) ? IRQ_HANDLED : IRQ_NONE)
 *
 * NOTE: interrupt handlers need not be reentrant (tread-safe)
 *      - When a given interrupt handler is executing, the corresponding
 *      interrupt line is disabled on all cores while.
 *      - Normally all other interrupts are enables, os other interrupts
 *      are serviced.
 */
typedef irqreturn_t (*irq_handler_t)(int irq, void *dev_id);
```

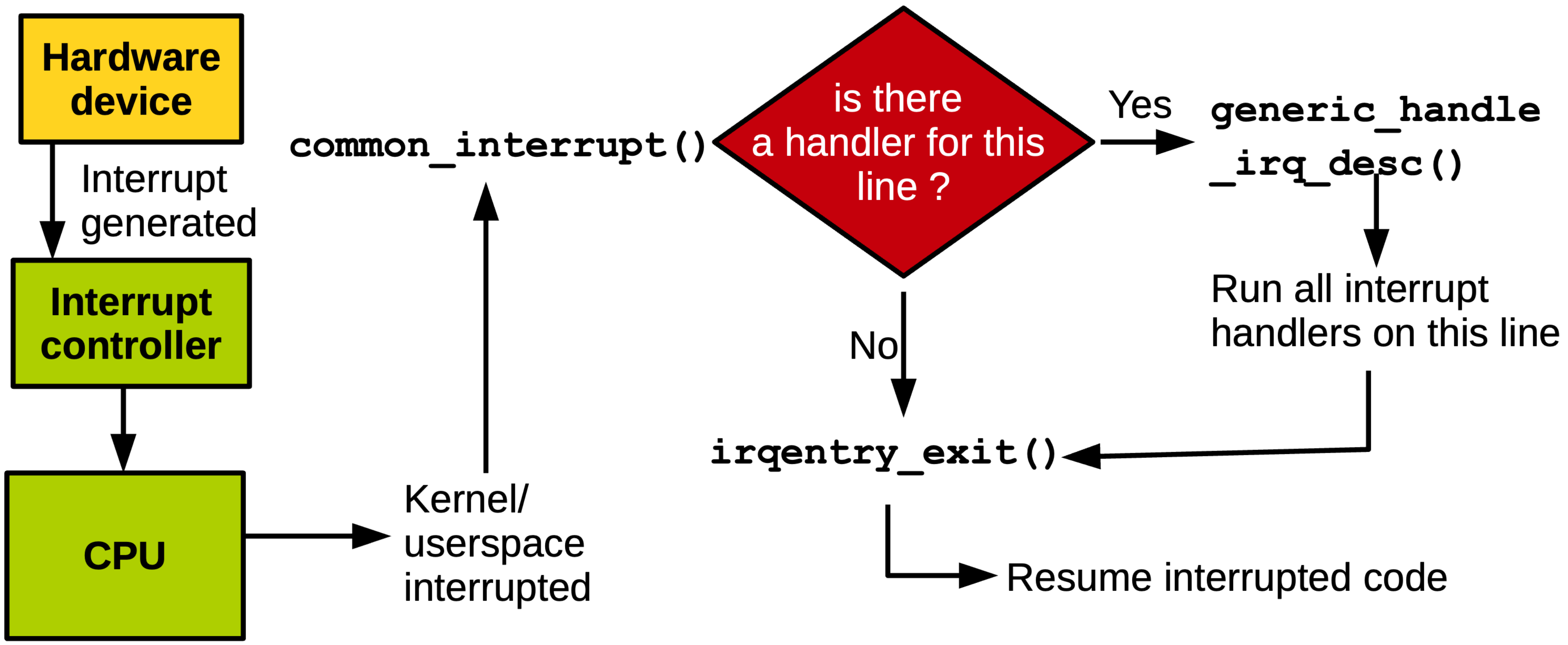
Shared Handlers

- The IRQF_SHARED flag must be set in the flags argument to request_irq()
- The dev_id argument must be unique to each registered handler
  - A pointer to any per-device structure is sufficient (e.g., struct device {})
  - When the kernel receives an interrupt, it invokes sequentially each registered handler on the line
    - » Therefore, it's important that the handler is capable of distinguishing whether it generates a given interrupt

Interrupt Context

- Process context: normal task execution, syscall, and exception
- Interrupt context: ISR
  - Sleeping/blocking is not possible b/c ISR is not a schedulable entity
  - No kmalloc(size, GFP_KERNEL), use "GFP_ATOMIC" instead
  - No blocking locking (e.g., mutex), use "spinlock()" instead
  - No printk(), use trace_printk instead
- Small stack size, one page, e.g., 4KB

Interrupt Handling in Linux



- Specific entry point for each interrupt line
  - Saves the interrupt number and current registers
  - Calls common_interrupt()
- common_interrupt(struct pt_regs *reg, u32 vector)
  - Ack interrupt, disable the line
  - Calls architecture specific functions
- Call chain ends up by calling generic_handle_irq_desc()
  - cal the handler if the line is not shared
  - otherwise iterate over all the handlers registered on that line
  - disable interrupts on the line again if they were previously enabled
- common_interrupt() returns to entry point that call irqentry_exit()
  - checks if reschedul is needed (*need_resched*)
  - restore register values

IDT Initialization

```c
/* linux/arch/x86/include/asm/desc_defs.h */
struct gate_struct {
    u16     offset_low;
    u16     segment;
    struct idt_bits bits;
    u16     offset_middle;
#ifdef CONFIG_X86_64
    u32     offset_high;
    u32     reserved;
#endif
} __attribute__((packed));

typedef struct gate_struct gate_desc;

/* linux/arch/x86/kernel/traps.c */
DECLARE_BITMAP(system_vectors, NR_VECTORS);
```

```
/* linux/arch/x86/include/asm/idtentry.h
/*
 * Build the entry stubs with some assembler magic.
 * We pack 1 stub into every 8-byte block.
 */
    .align 8
SYM_CODE_START(irq_entries_start)
    vector=FIRST_EXTERNAL_VECTOR
    .rept (FIRST_SYSTEM_VECTOR - FIRST_EXTERNAL_VECTOR)
    UNWIND_HINT_IRET_REGS
0 :
    .byte   0x6a, vector
    jmp asm_common_interrupt
    nop
    /* Ensure that the above is 8 bytes max */
    . = 0b + 8
    vector = vector+1
    .endr
SYM_CODE_END(irq_entries_start)
```

```c
/* linux/init/main.c */

asmlinkage __visible void __init start_kernel(void)
{
    /* ... */
    early_irq_init();
    init_IRQ();
    /* ... */
}

/* linux/arch/x86/kernel/irqinit.c */
void __init init_IRQ(void)
{
    int i;
    for (i = 0; i < nr_legacy_irqs(); i++)
        per_cpu(vector_irq, 0)[ISA_IRQ_VECTOR(i)] = irq_to_desc(i);

    BUG_ON(irq_init_percpu_irqstack(smp_processor_id()));

    x86_init.irqs.intr_init();
}
```

```c
/* linux/arch/x86/kernel/idt.c */
void __init idt_setup_apic_and_irq_gates(void)
{
    int i = FIRST_EXTERNAL_VECTOR;
    void *entry;

    idt_setup_from_table(idt_table, apic_idts, ARRAY_SIZE(apic_idts), true);

    for_each_clear_bit_from(i, system_vectors, FIRST_SYSTEM_VECTOR) {
        entry = irq_entries_start + 8 * (i - FIRST_EXTERNAL_VECTOR);
        set_intr_gate(i, entry);
    }
}
```

```c
/* linux/arch/x86/kernel/irqinit.c */
void __init native_init_IRQ(void)
{
    /* Execute any quirks before the call gates are initialised: */
    x86_init.irqs.pre_vector_init();

    idt_setup_apic_and_irq_gates();
    lapic_assign_system_vectors();

    if (!acpi_ioapic && !of_ioapic && nr_legacy_irqs())
        setup_irq(2, &irq2);
}
```

```c
/* linux/arch/x86/kernel/idt.c */
static void set_intr_gate(unsigned int n, const void *addr)
{
    struct idt_data data;

    BUG_ON(n > 0xFF);

    memset(&data, 0, sizeof(data));
    data.vector = n;
    data.addr   = addr;
    data.segment    = __KERNEL_CS;
    data.bits.type  = GATE_INTERRUPT;
    data.bits.p = 1;

    idt_setup_from_table(idt_table, &data, 1, false);
}
```

Interrupt Control

- Kernel code sometimes need to disable interrupts to ensure atomic execution
  - By disabling interrupts, it guarantees that an interrupt handle will not preempt your code
  - Disabling interrupts also disables kernel preemption
- Disabling interrupts does not protect against concurrent access from other cores
  - Need locking, often used in conjunction with interrupt disabling
- The kernel provides APIs to disable/enable interrupts
  - local_irq_disable()
  - local_irq_enable()
  - can be called multiple times

Disabling Interrupts on the Local Core

- Use local_irq_save()

```c
unsigned long flags;
local_irq_save(flags);    /* disables interrupts if needed */
/* ... */
local_irq_restore(flags); /* restore interrupt status to the previous */

/* nesting is okay */
unsigned long flags;
local_irq_save(flags);
{
  unsigned long flags;
  local_irq_save(flags);
  /* ... */
  local_irq_restore(flags);
}
local_irq_restore(flags);
```

Disabling Specific Interrupts

```
/** disable_irq - disable an irq and wait for completion
 *  @irq: Interrupt to disable
 *
 *  Disable the selected interrupt line.  Enables and Disables are
 *  nested.
 *  This function waits for any pending IRQ handlers for this interrupt
 *  to complete before returning. If you use this function while
 *  holding a resource the IRQ handler may need you will deadlock.
 *
 *  This function may be called - with care - from IRQ context. *//
void disable_irq(unsigned int irq);

/** disable_irq_nosync - disable an irq without waiting
 *  @irq: Interrupt to disable */
void disable_irq_nosync(unsigned int irq);

/** enable_irq - enable handling of an irq
 *  @irq: Interrupt to enable
 *
 *  Undoes the effect of one call to disable_irq().  If this
 *  matches the last disable, processing of interrupts on this
 *  IRQ line is re-enabled. */
void enable_irq(unsigned int irq);
```

Interrupt Status

```c
/* linux/include/linux/preempt.h */

/*
 * Are we doing bottom half or hardware interrupt processing?
 *
 * in_irq()       - We're in (hard) IRQ context
 * in_interrupt() - We're in NMI,IRQ,SoftIRQ context or have BH disabled
 * in_nmi()       - We're in NMI context
 * in_softirq()   - We have BH disabled, or are processing softirqs
 * in_serving_softirq() - We're in softirq context
 * in_task()      - We're in task context
 */
#define in_irq()                (hardirq_count())
#define in_interrupt()          (irq_count())
#define in_nmi()                (preempt_count() & NMI_MASK)
#define in_softirq()            (softirq_count())
#define in_serving_softirq()    (softirq_count() & SOFTIRQ_OFFSET)
#define in_task()               (!(preempt_count() & \
                                 (NMI_MASK | HARDIRQ_MASK | SOFTIRQ_OFFSET)))
```

References

- LWN: Debugging the kernel using Ftrace – part 1
- 0xAX; Interrupts and Interrupt Handling