Interrupt Handler: Top Half

Dongyoon Lee

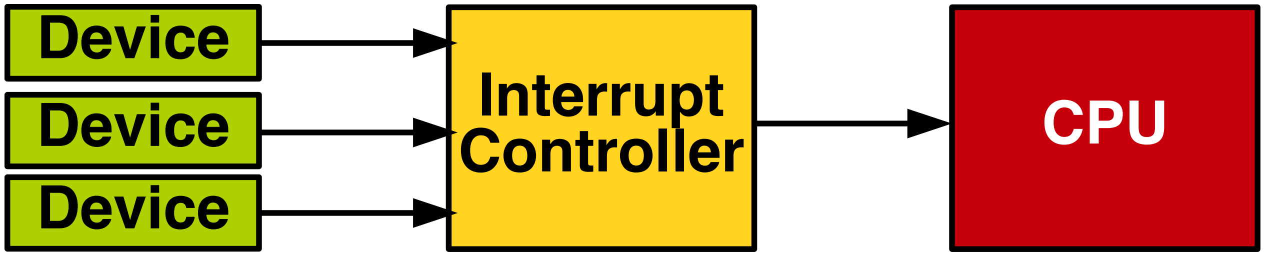
Summary of last lectures

- Tools: building, exploring, and debugging Linux kernel
- Core kernel infrastructure
    - syscall, module, kernel data structures
- Process management & scheduling
- Interrupt

Today: “interrupt handler: top half”

- A mechanism to implement abstraction and multiplexing
- Interrupt: asking for a service to the kernel
    - by software (e.g., `int`) or by hardware (e.g., keyboard)
    - synchronous exceptions and asynchronous interrupts
- Interrupt handling in Linux
    - top half + bottom half

Interrupt controller



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

Interrupt request (IRQ)

- **Interrupt line** or **interrupt request (IRQ)**
  - device identifier
- E.g., 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
  - True for most modern devices (PCIe)

Interface to hardware interrupt

- **Non-Maskable Interrupt (NMI)**
  - Never ignored, e.g., power failure, memory error
  - In x86, vector 2, prevents other interrupts from executing.
- **Maskable interrupt**
  - Ignored when `IF` in `EFLAGS` is 0
  - Enabling/disabling: - `sti` : set interrupt - `cli` : clear interrupt
- **INTA**
  - Interrupt acknowledgement
  - EOI (end of interrupt)

“Software” interrupt: INT

- Intentionally interrupts
    - x86 provides the `INT` instruction
    - Invokes the interrupt handler for the vector (0-255)
- Entering: `int N`
- Exiting: `iret`

Interrupt vector

- Telling `int N`
    - N-th interrupt handler (e.g., `int 0` → `vector 0`)

Interrupt descriptor table

- **IDT**
  - Table of 256 8-byte entries (similar to GDT)
  - Located anywhere in memory
- **IDTR register**
  - Stores current IDT
- **`lidt` instruction**
  - Loads IDTR with address and size of the IDT
  - Takes in a linear address

Interrupt descriptor table

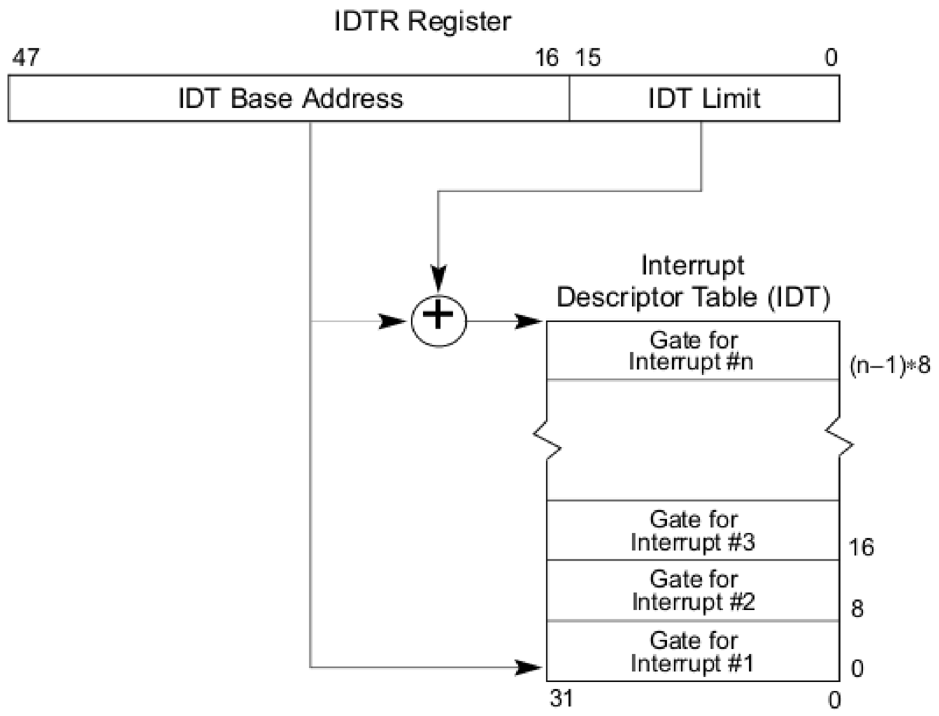
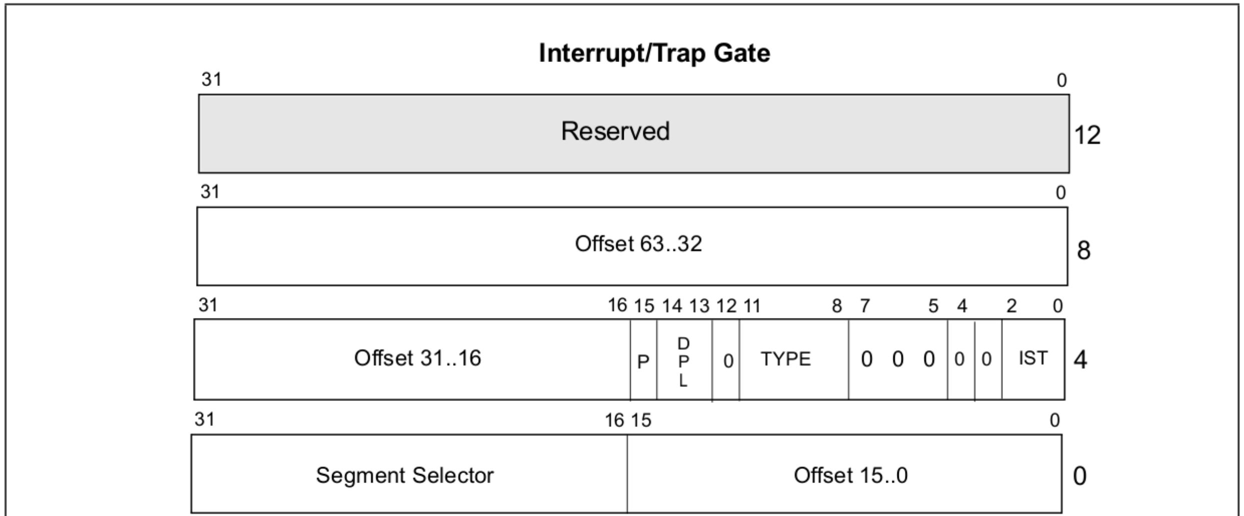


Figure 6-1. Relationship of the IDTR and IDT

Interrupt descriptor entry



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 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

Interrupt descriptor table

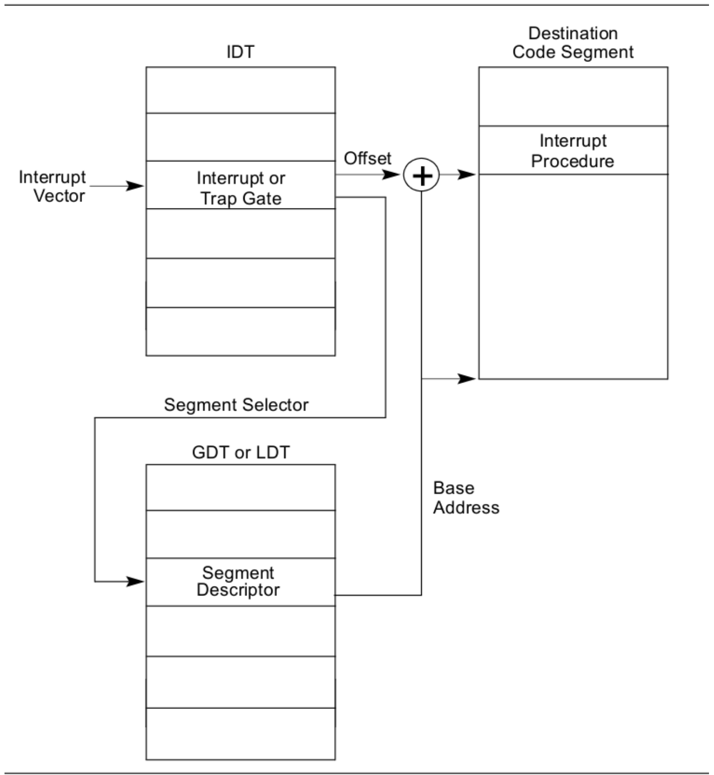


Figure 6-3. Interrupt Procedure Call

Predefined interrupt vectors

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

The INT instruction (1/2)

- Decide the vector number: e.g., `int 0x80`
- Fetch the interrupt descriptor for vector 0x80 from the `IDT`. The CPU finds it by taking the 0x80'th 8-byte entry starting at the physical address that the `IDTR` CPU register points to.
- Check if CPL < = DPL in the descriptor.

The INT instruction (2/2)

- Push user SS
- Push user ESP
- Push user EFLAGS
- Push user CS
- Push user EIP
- Clear some 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)**
  - function executed by the CPU in response to a specific interrupt
- Runs in **interrupt context (or atomic context)**
  - Opposite to process context (system call)
  - A task cannot sleep in an ISR because an interrupt context is not a schedulable entity

Two conflicting goals of ISR

1. **Interrupt processing must be fast**
   - We are indeed interrupting user processes executing (user/kernel space)
   - Some other interrupts may need to be disabled while processing an interrupt
2. **Sometimes it requires significant amount of work**
   - So it will take time
   - E.g., processing a network packet from the network card

Top half vs. bottom half

- In many modern OS including Linux, an interrupt processing is split into two parts:
- **Top-half:** run immediately upon receipt of the interrupt
  - Performs only the time-critical operations
    - E.g., Acknowledging receipt of the interrupt
    - resetting the hardware
- **Bottom-half:** less critical & time-consuming work
  - Run later with other interrupts enabled

Example: network packet processing

- **Top-half: interrupt service routine**
  - Acknowledges the hardware
  - Copies the new network packets into main memory
  - Makes the network card ready for more packets
  - Time critical because the packet buffer on the network card is limited in size → packet drop
- **Bottom-half: processing the copied packets**
  - Softirq, tasklet, work queue
  - Similar to thread pool in user-space

Registering an interrupt handler

```
/* 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

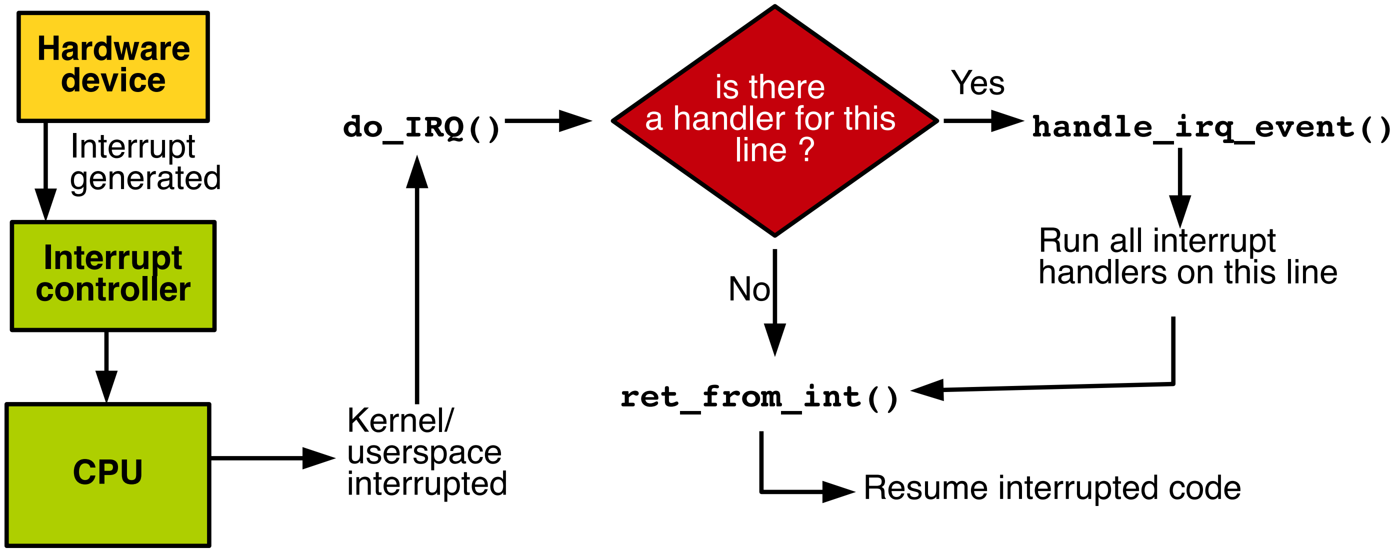
```
/* linux/include/linux/interrupt.h */

/**
 * 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);
```

Interrupt context

- Process context: normal task execution, syscall, and exception
- Interrupt context: interrupt service routine (ISR)
    - **Sleeping/blocking is not possible** because the ISR is not a schedule entity
    - No `kmalloc(size, GFP_KERNEL)`: use `GFP_ATOMIC`
    - No blocking locking (e.g., mutex): use spinlock
    - No `printk`: use `trace_printk`
- Small stack size
    - Interrupt stack: one page (4KB)

Interrupt handling internals in Linux



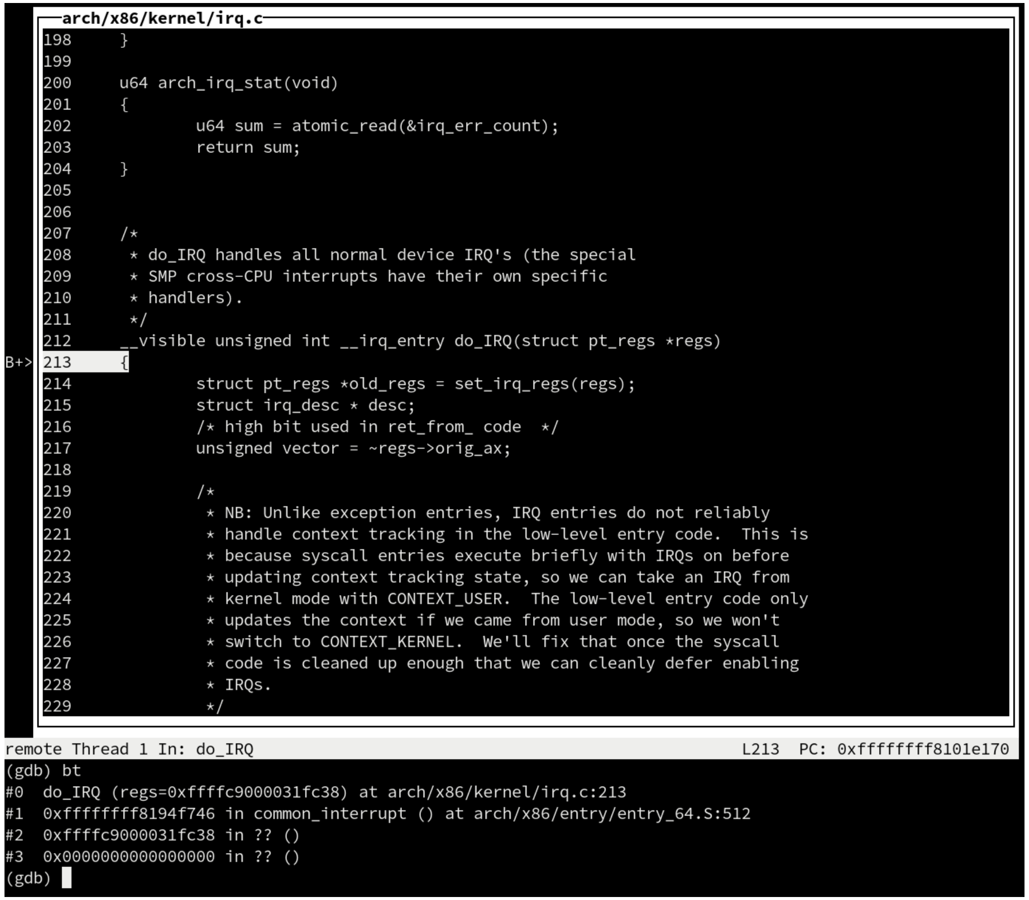
Interrupt handling internals in Linux

- Specific entry point for each interrupt line
    - Saves the interrupt number and the current registers
    - Calls `do_IRQ()`
- `unsigned int do_IRQ(struct pt_regs *regs)`
    - Acknowledge interrupt reception and disable the line
    - Calls architecture specific functions

Interrupt handling internals in Linux

- Call chain ends up by calling `generic_handle_irq_desc()`
  - Call 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
- `do_IRQ()` returns to entry point that call `ret_from_intr()`
  - Checks if reschedule is needed (`need_resched`)
  - Restore register values

`do_IRQ()` in QEMU/gdb



```
arch/x86/kernel/irq.c
    198     }
    199
    200     u64 arch_irq_stat(void)
    201     {
    202             u64 sum = atomic_read(&irq_err_count);
    203             return sum;
    204     }
    205
    206
    207     /*
    208      * do_IRQ handles all normal device IRQ's (the special
    209      * SMP cross-CPU interrupts have their own specific
    210      * handlers).
    211      */
    212     __visible unsigned int __irq_entry do_IRQ(struct pt_regs *regs)
B+> 213     {
    214             struct pt_regs *old_regs = set_irq_regs(regs);
    215             struct irq_desc * desc;
    216             /* high bit used in ret_from_ code  */
    217             unsigned vector = ~regs->orig_ax;
    218
    219             /*
    220              * NB: Unlike exception entries, IRQ entries do not reliably
    221              * handle context tracking in the low-level entry code.  This is
    222              * because syscall entries execute briefly with IRQs on before
    223              * updating context tracking state, so we can take an IRQ from
    224              * kernel mode with CONTEXT_USER.  The low-level entry code only
    225              * updates the context if we came from user mode, so we won't
    226              * switch to CONTEXT_KERNEL.  We'll fix that once the syscall
    227              * code is cleaned up enough that we can cleanly defer enabling
    228              * IRQs.
    229              */
remote Thread 1 In: do_IRQ                                                  L213   PC: 0xffffffff8101e170
(gdb) bt
#0  do_IRQ (regs=0xffffc9000031fc38) at arch/x86/kernel/irq.c:213
#1  0xffffffff8194f746 in common_interrupt () at arch/x86/entry/entry_64.S:512
#2  0xffffc9000031fc38 in ?? ()
#3  0x0000000000000000 in ?? ()
(gdb)
```

Initializing IDT

```
/* 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);
```

Initializing IDT

```
/* 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)
```

- Let’s see how `gate_desc` is initialized during boot.

Initializing IDT

```
/* 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();
}
```

Initializing IDT

```
/* 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);
}
```

Initializing IDT

```
/* 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);
    }
}
```

Initializing IDT

```
/* 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);
}
```

/proc/interrupts

```
$ cat /proc/interrupts
# Int line
# |        Num of occurrence per CPU
# |        |                 Int controller
# |        |                 |        Edge/level
# |        |                 |        |            Device name
# |        |                 |        |            |
           CPU0       CPU1
  0:         34          0   IO-APIC   2-edge      timer
  1:         26          8   IO-APIC   1-edge      i8042
  8:          0          0   IO-APIC   8-edge      rtc0
  9:          0          0   IO-APIC   9-fasteoi   acpi
 12:        156          0   IO-APIC  12-edge      i8042
 14:          0          0   IO-APIC  14-edge      ata_piix
 15:        116         24   IO-APIC  15-edge      ata_piix
 19:          5         68   IO-APIC  19-fasteoi   virtio0
 21:       3142        533   IO-APIC  21-fasteoi   ahci[0000:00:0d.0], snd_intel8x0
 22:         27          0   IO-APIC  22-fasteoi   ohci_hcd:usb1
NMI:          0          0   Non-maskable interrupts
LOC:       5031       4373   Local timer interrupts
PMI:          0          0   Performance monitoring interrupts
TLB:         27         89   TLB shootdowns
```

Interrupt control

- Kernel code sometimes needs to disable interrupts to ensure atomic execution of a section of code
  - By disabling interrupts, you can guarantee that an interrupt handler will not preempt your current code.
  - Moreover, disabling interrupts also disables kernel preemption.
- Note that disabling interrupts does not protect against concurrent access from other cores
  - Need locking, often used in conjunction with interrupts disabling
- The kernel provides an API to disable/enable interrupts

Disabling interrupts on the local core

- Disable and enable IRQ

```
local_irq_disable();
/* interrupts are disable ... */
local_irq_enable();
```

- What happend if local_irq_disable() is called twice?

```
local_irq_disable(); /* interrupt is disabled */
local_irq_disable(); /* no effect */
/* ... */
local_irq_enable();  /* interrupt is eanbled! */
local_irq_enable();  /* no effect */
```

Disabling interrupts on the local core

- Use `local_irq_save()`

```
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 a specific interrupt line

```
/** 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);
```

Status of the interrupt system

```
/* 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)))
```

Further readings

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

Next lecture

- Interrupt handler: bottom half