ECE391
Computer System Engineering

Lecture 13

Dr. Zbigniew Kalbarczyk

University of Illinois at Urbana- Champaign

Spring 2021

Lecture Topics

- Linux interrupt system
  - data structures
  - handler installation & removal
  - invocation
  - execution

- Summary of Linux interrupt system

Aministrivia

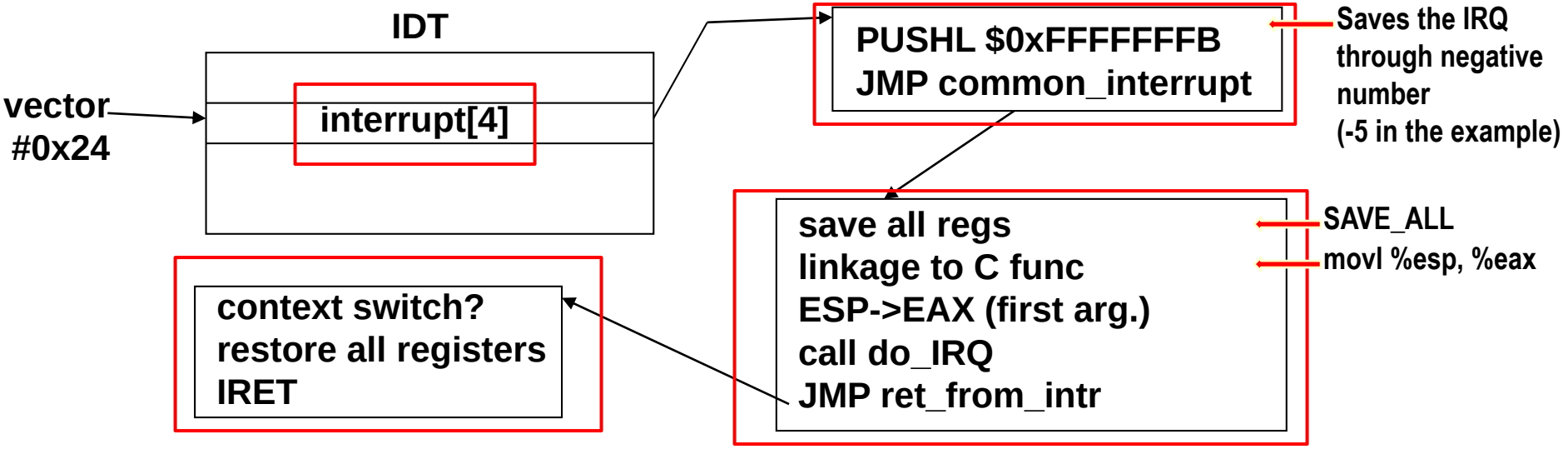
- **MP2 Checkpoint 2**
  - **Due by 5:59pm Monday, March 15**

Aministrivia

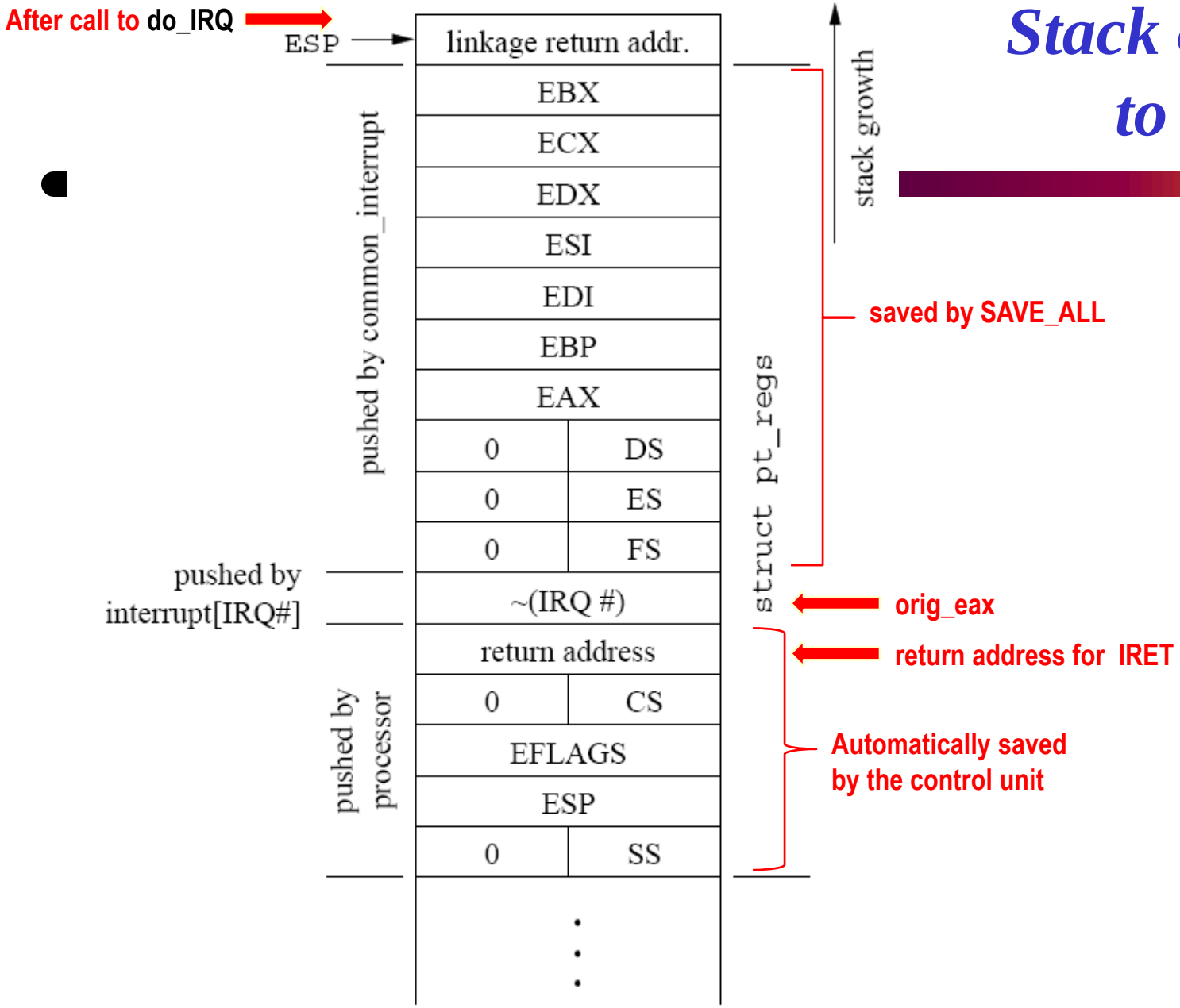
- **MP3 Teams**
  - Please submit this google form:
    https://forms.gle/FfeSuDFjo49VNjtd7
    before Thursday 3/11 at 11:59PM CT.
  - Every team must have exactly 4 members
  - Only one person from each group must submit the form

- MP3 will be released next week

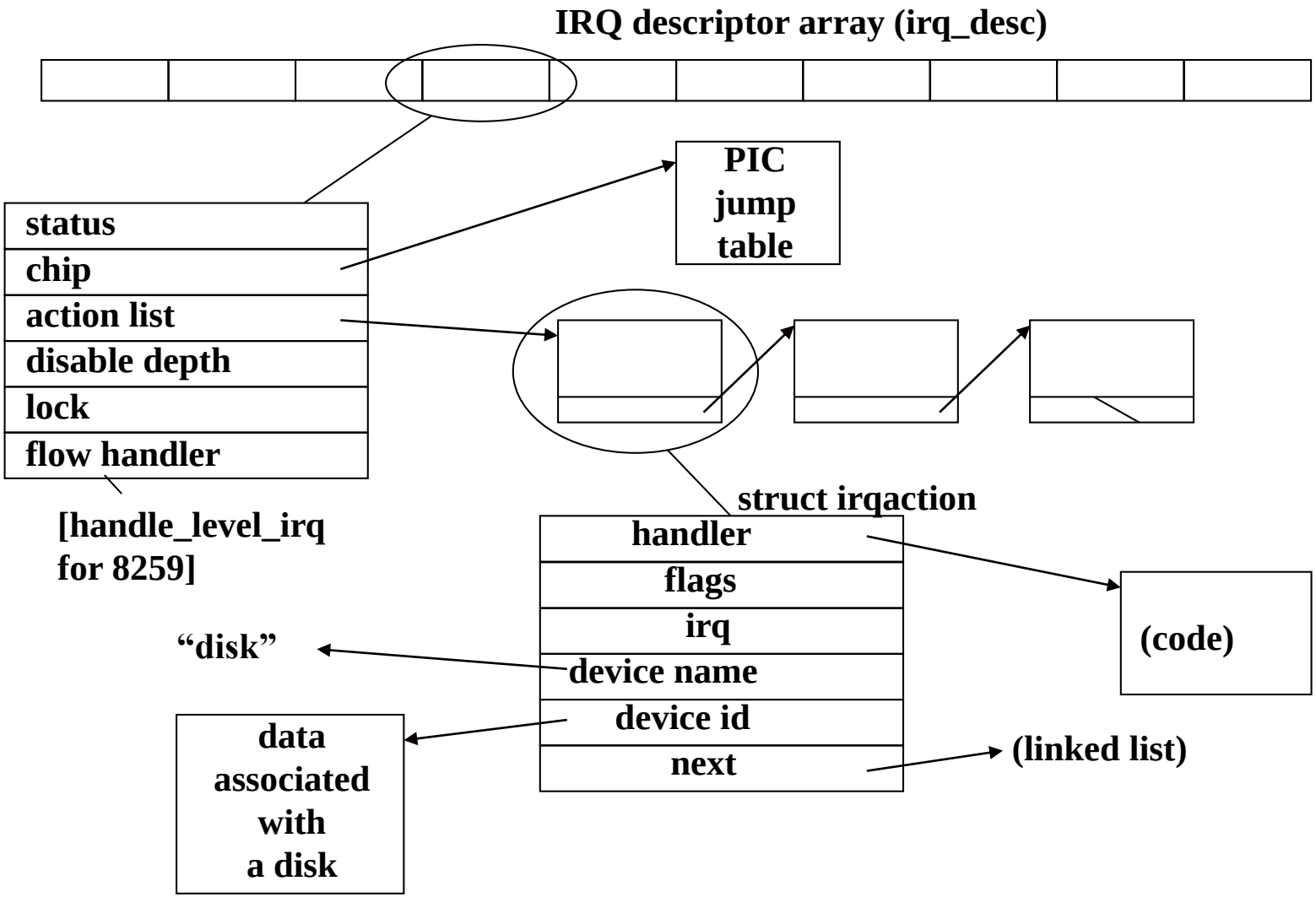
Interrupt Invocation



Stack on Entry to do_IRQ



Linux' Interrupt Data Structures



Linux do_IRQ

arch/i386/kernel/irq.c

```
fastcall unsigned int
do_IRQ (struct pt_regs* regs)
{
    struct pt_regs* old_regs;
    /* high bit used in ret_from_ code */
    int irq = ~regs->orig_eax;
    struct irq_desc* desc = irq_desc + irq;

    if ((unsigned)irq >= NR_IRQS) {
        printk (KERN_EMERG "%s: cannot handle IRQ %d\n", __FUNCTION__, irq);
        BUG ();
    }

    old_regs = set_irq_regs (regs);
    irq_enter ();

    /* for 8259A interrupts, handle_irq is set to handle_level_irq */
    desc->handle_irq (irq, desc);

    irq_exit ();
    set_irq_regs (old_regs);
    return 1;
}
```

fastcall convention used to pass arguments in registers; reduces the number of memory accesses required for the call

gets the interrupt number from the orig_eax filed in the regs structure

Record registers at time of interrupt

Increments counter of nested interrupt handlers

call interrupt flow handler handle_level_irq for all 8259A interrupts

Exit irq context and process softirqs if needed

Interrupt Invocation (cont.)

- Signature for do_IRQ

```
fastcall unsigned int do_IRQ
    (struct pt_regs* regs);
```

  - **`fastcall`** macro in Linux tells gcc to pass args in EAX, EDX, ECX
  - EAX points to saved registers (**`regs`** argument)
  - saved registers already on stack
  - note that processor clears IF when it takes an interrupt
  - EFLAGS stored by processor include original IF value

Comments on `do_irq`

- Value pushed by irq-specific code re-converted to find irq #
- Start with a sanity check
- **`set_irq_regs`** calls record registers at time of interrupt
  - per-CPU storage

Comments on `do_irq` (cont.)

- **`irq_enter` / `irq_exit`**
  - necessary for proper priority
    - e.g., second hard interrupt occurs
    - after new interrupt handled, should return to first
    - must delay processing of soft interrupts
  - **`irq_exit`** processes soft interrupts when appropriate

Comments on `do_irq` (cont.)

- Central component
  - call interrupt flow handler
  - **`handle_level_irq`** for all 8259A interrupts

- Return value ignored (probably intended to indicate interrupt handled)

Linux handle_level_irq

kernel/irq/chip.c

```
fastcall void
handle_level_irq (unsigned int irq, struct irq_desc* desc)
{
    unsigned int       cpu = smp_processor_id ();
    struct irqaction* action;
    irqreturn_t        action_ret;

    spin_lock (&desc->lock);
    mask_ack_irq (desc, irq);

    if (desc->status & IRQ_INPROGRESS)
        goto out_unlock;
    desc->status &= ~(IRQ_REPLAY | IRQ_WAITING);
    kstat_cpu (cpu).irqs[irq]++;

    /*
     * If its disabled or no action available
     * keep it masked and get out of here
     */
    action = desc->action;
    if (!action || (desc->status & IRQ_DISABLED)) {
        desc->status |= IRQ_PENDING;
        goto out_unlock;
    }

    desc->status |= IRQ_INPROGRESS;
    desc->status &= ~IRQ_PENDING;
    spin_unlock (&desc->lock);

    action_ret = handle_IRQ_event (irq, action);
    if (!noirqdebug)
        note_interrupt (irq, desc, action_ret);

    spin_lock(&desc->lock);
    desc->status &= ~IRQ_INPROGRESS;
    if (!(desc->status & IRQ_DISABLED) && desc->chip->unmask)
        desc->chip->unmask (irq);

out_unlock:
    spin_unlock (&desc->lock);
}
```

Critical section begins

Call PIC's mask_ack function

If interrupt already in progress, do nothing

Atomically decide whether to execute handlers

Set status for execution: in-progress and no longer pending

Critical section ends

Handler execution done via handle_IRQ_event

Critical section begins

When done, remove in-progress flag and unmask on PIC (unless disabled)

Critical section ends

Comments on `handle_level_irq`

- Critical section starts
  - to read descriptor status and action (handler) list
  - IF=0 at this point (set by processor when taking interrupt)
- Immediately call PIC's mask_ack function (via a wrapper function)
- If interrupt already in progress, do nothing
  - Interrupt already re-masked and re-acknowledged on PIC
- Remove software replay
- Kernel statistics track # of interrupts seen (see /proc/interrupts)

Comments on `handle_level_irq` (cont.)

- Atomically decide whether to execute handlers immediately
  - do so if handler defined and not disabled by software
  - if not, skip to end
    - mark as pending and end interrupt handling
    - replayed after last nested **`enable_irq`** call

- Set status for execution: in-progress, and no longer pending

Comments on `handle_level_irq` (cont.)

- Handler execution done via **`handle_IRQ_event`**
  - done without descriptor lock
    - allows handlers to use infrastructure
    - e.g., **`enable_irq`**/**`disable_irq`**, **`request_irq`**/**`free_irq`**
  - usually done with IF=1 (changed inside **`handle_IRQ_event`**)
    - done without descriptor lock
    - otherwise this code can deadlock with self

- When done, remove in-progress flag and unmask on PIC (unless disabled)

Linux handle_IRQ_event

kernel/irq/handle.c

```
irqreturn_t
handle_IRQ_event (unsigned int irq, struct irqaction* action)
{
    irqreturn_t  ret;
    irqreturn_t  retval = IRQ_NONE;
    unsigned int status = 0;

    /* call specific to ARM processor (to disambiguate timer ticks) */
    handle_dynamic_tick (action);

    if (!(action->flags & IRQF_DISABLED))
        local_irq_enable_in_hardirq ();

    do {
        ret = action->handler (irq, action->dev_id);
        if (ret == IRQ_HANDLED)
            status |= action->flags;
        retval |= ret;
        action = action->next;
    } while (action);

    if (status & IRQF_SAMPLE_RANDOM)
        add_interrupt_randomness (irq);
    local_irq_disable ();

    return retval;
}
```

call translates basically to STI
(local means "on this CPU")

Walk through list; call each handler
(return value 1 means handled)

Turn interrupts back off (IF=0 / CLI)

Comments on `handle_IRQ_event`

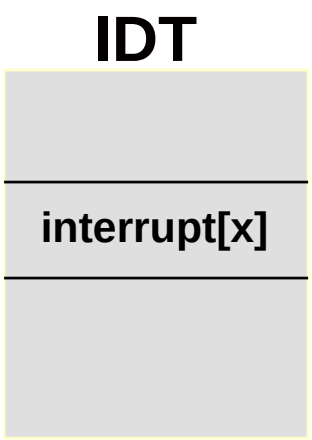
- Set IF=1 unless the first handler asked to be executed with IF=0
  - indicated by IRQF_DISABLED flag
  - call translates basically to STI (local means “on this CPU”)
- Walk through list
  - call each handler
  - return value 1 means handled
- Generate random numbers if desired
- Turn interrupts back off (IF=0 / CLI)

Summary on Descriptor Flags

- IRQ_PENDING—interrupt raised by hardware; waiting to be executed
- IRQ_INPROGRESS—some processor is executing handlers
- IRQ_DISABLED—interrupt disabled in software; postpone execution
- IRQ_REPLAY—software replay of previously postponed execution

Interrupt Descriptor Table (IDT)

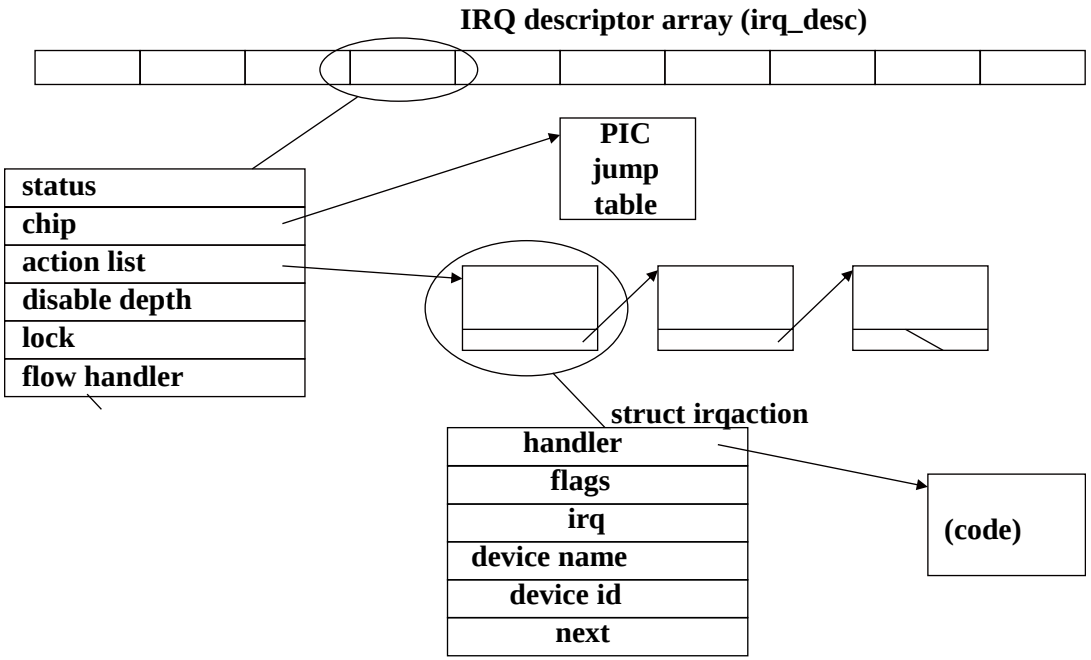
- Associates the interrupt line with the int. handler routine.
- 256 entries (each 8-bytes) or descriptors; each corresponds to an interrupt vector
  - hardware interrupts mapped into vectors 0x20 to 0x2F
- All Linux interrupt handlers are activated by so called: *interrupt gates (a descriptor type)*
- Whenever an interrupt gate is hit, interrupts are disabled automatically by the processor



- Before the kernel enables interrupts it must initialize the *idtr* register to point to the IDT table *(set by the kernel using* `lidt` *instruction)*

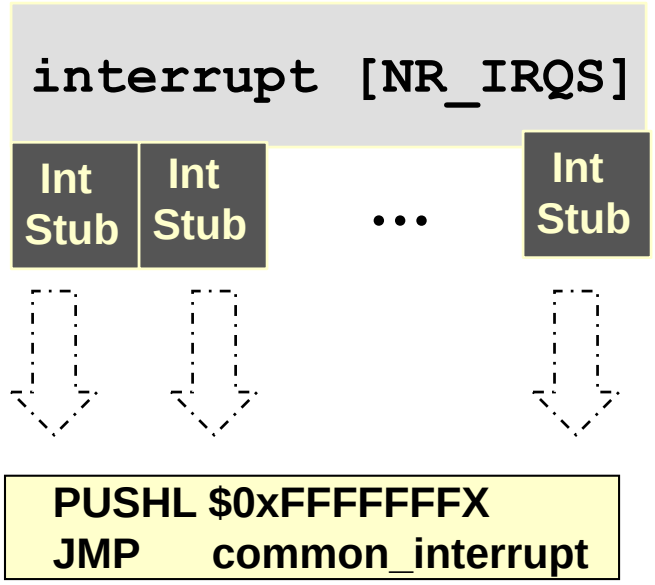
- **IDT Initialization**
- Kernel initialization (`setup_idt()`) fills all the 256 entries of IDT with the provisional (or null) handler

`irq_desc` Array



- Every interrupt vector has its own `irq_desc_t` descriptor
- Descriptors are grouped together in `irq_desc` array, a data structure supported by Linux
- When a device driver calls the `request_irq()` function a new structure to represent the handler is allocated and initialized

interrupt[NR_IRQS] Array



- Kernel maintains one global array of function pointers (`interrupt[NR_IRQS]`) in which it stores pointers to interrupt stubs (`NR_IRQS` is 16 if we use the PIC)

Initialization of Interrupt Gates

- During initialization `init_IRQ()` sets the status field of each IRQ descriptor to IRQ_DISABLED
- `init_IRQ()` updates the IDT by replacing the provisional interrupt gates with new ones

```
for (i = 0; i < NR_IRQS; i++)
     if (i+32 != 128)
          set_intr_gate(i+32, interrupt[i]);
```

- Interrupt gates are set to the addresses found in the `interrupt[NR_IRQS]` array