

Preemptive Scheduling and Programming Tips for FIFOs

CS552 – Operating Systems
08/11/2023
Anton Njavro

Slides by: Sasan Golchin

Agenda

- Last time:
 - Organization of task scheduler
 - Non-preemptive task switching
 - Setup of GDT
- Today:
 - Preemptive task switching
 - Setup of the system timer (PIT)
 - Interrupt handling (PIC)

Recap - Overview

Kernel Initialization

- **GDT** w/ at least kernel code and data descriptor
- (*) **IDT**: to handle hardware exceptions and IRQs
- (*) **PIC**: to deliver timer interrupts to the scheduler
- (*) **PIT**: to set preemption points
- Initialize a pool of (up to constant N) tasks
- Start the scheduler to launch the first task

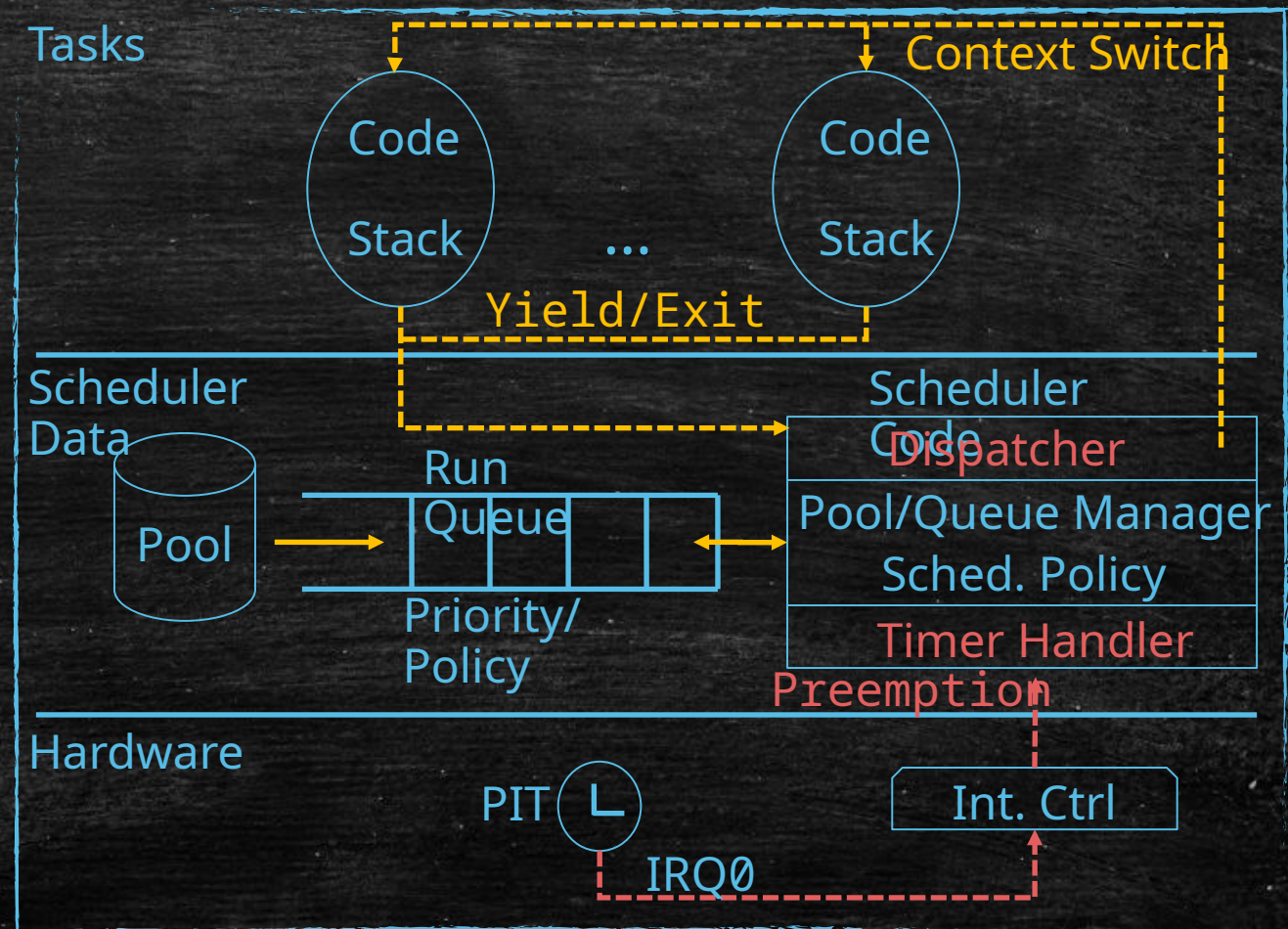
(*) Preemption support requirements

Scheduler Functionalities

- Scheduler's Public Interface
 - thread_create(func, stack)
 - thread_yield()
- Scheduler's Private Interface
 - current_thread()
 - find_next_thread()
 - switch_thread(from, to)
 - launch_thread(t)
 - exit_thread()
 - (*) preempt_thread()

Recap - Organization

- Functionalities
 - Add/Remove tasks
 - Find the next task to run
 - Handle state transitions
 - Context switching
- Main components
 - Task pool
 - Run Queue
 - Dispatcher



Preemption support

- Objective:
 - Transfer the execution control to the scheduler after a set amount of time
 - Regardless of the task running on the CPU
 - Whether or not the current task is willing to yield
- Requirements:
 1. Program the CPU to respond to asynchronous events i.e. **IRQs**
 2. Program a timer (PIT in our case) to generate an IRQ at a set time
 3. Program the H/W to deliver the IRQ to the CPU
- Dependencies: PIT -> PIC -> IDT -> GDT

1 - Interrupt support in x86

- Software interrupts

- Exceptions: Runtime errors caught by the CPU e.g. DIV-0, Overflow, Page-Fault
 - Faults: Can be corrected e.g. DIV-0, Page-Fault
 - Traps: Due to controlled machine instructions e.g. Breakpoints
 - Abort: Unrecoverable Error e.g. Internal machine errors: Memory/Bus/Cache errors
 - There are 32 exceptions defined by x86. More info [here](#).
- User-defined:
 - Generated by the INT instruction : Defined by the OS or the firmware (e.g. BIOS)
 - They are maskable - Can be ignored by the CPU

- Hardware interrupts

- Interrupt Requests (IRQs): An external device requires CPU's attention
 - Examples: System Timer (IRQ-0: FIFOS), Keyboard Controller (IRQ-1: Primer)
 - 1 Interrupt line per core, thus, we need a multiplexing hardware e.g. PIC or IOAPIC
 - They are maskable - Can be ignored by the CPU

When does the CPU receive interrupts?

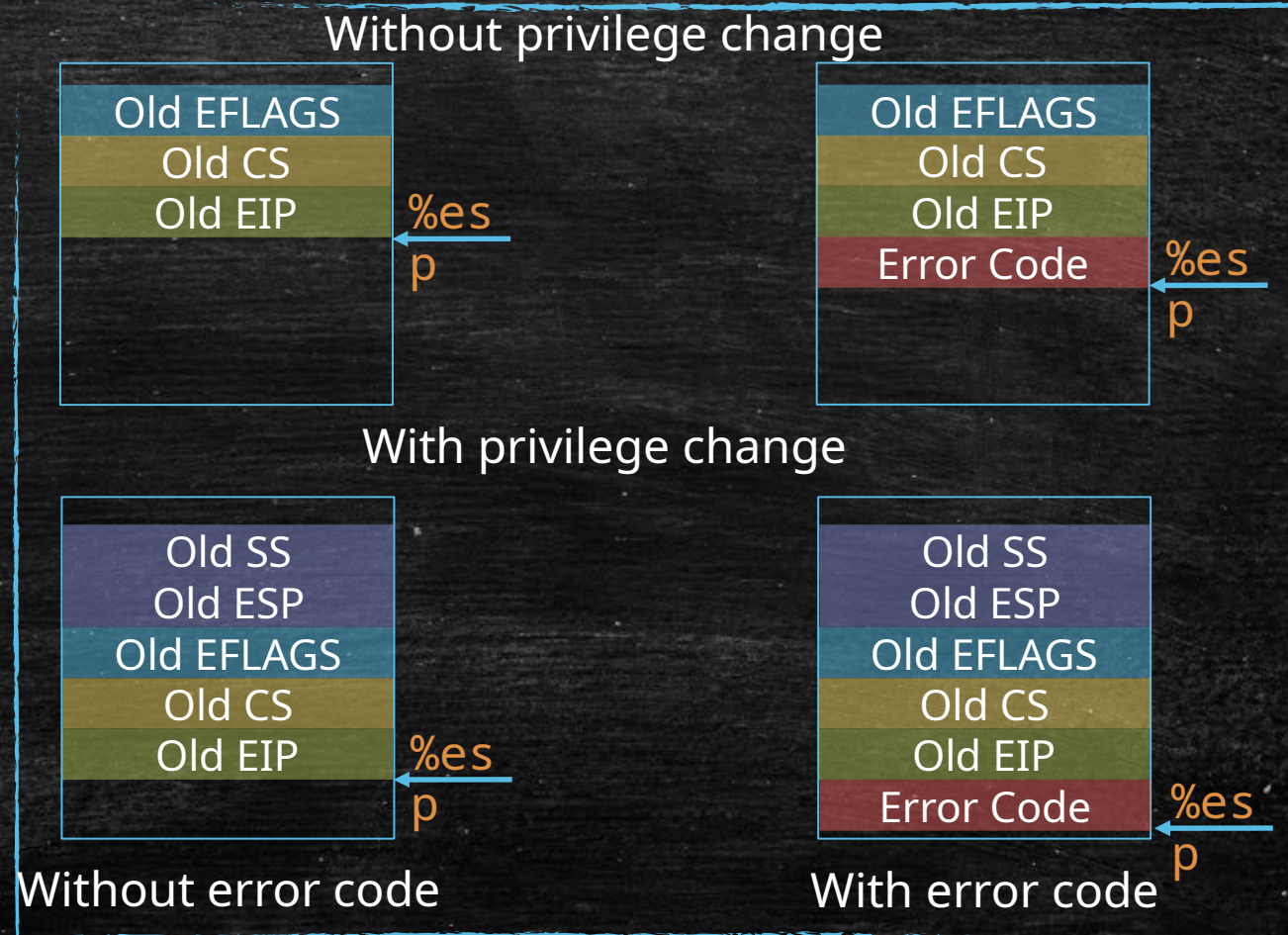
- The CPU receives an Interrupt (IRQ or INT) if:
 - There is a pending IRQ signal or the INT instruction is issued
 - Interrupts are enabled in the CPU i.e. EFLAGS.IF (bit 9 of the [flags register](#)) is set
 - Otherwise the CPU will ignore the interrupt (aka masking)
 - Modify EFLAGS.IF by the **cli** or **sti** instructions or popf
 - There's a valid **Interrupt Descriptor** entry corresponding to the interrupt number:
 - An interrupt descriptor tells the CPU what to do (Where to jump in the code)
 - Interrupt descriptor are defined by
 - The Interrupt Vector Table ([IVT](#)) in Real-mode (remember the BIOS calls?)
 - The Interrupt Descriptor Table ([IDT](#)) in Protected-mode

What does the CPU do?

Before executing the next instruction, the CPU will check if there's an interrupt. If so:

- Retrieves the Interrupt/Exception number
- Pushes the following information on the stack
- Disables the interrupts
- Jumps to the code location specified in the corresponding interrupt descriptor

Otherwise, carries on with the execution of the next instruction



Interrupt Service Routines

- A function that CPU calls upon reception of an interrupt
- The content pushed on top the stack depends on
 - Type of the interrupt/exception (whether there's an error code or not)
 - The privilege (ring of protection) of the current running code vs. the ISR
 - So, the calling convention is different from that of a C function
 - Different Prologue and Epilogue
- The ISR returns using `iret` instead of `ret`
 - Pops everything pushed to the stack as a result of an interrupt except the error code.
 - Can re-enable the interrupts by popping the old flags register
 - If your ISR is handling an exception with an error-code you should pop it yourself before issuing `iret`
 - No other register is saved. So, what if you call a C function from an ISR?
- Can be written in C or assembly. Look [here!](#)

Example of an ISR in assembly

```
my_isr0:
    // retrieve the error code (if any)
    ..
    // pass some parameters to C if needed
    ..
    // call the handler code in C
    call my_handler_in_c
    // return from the ISR
    iret
```

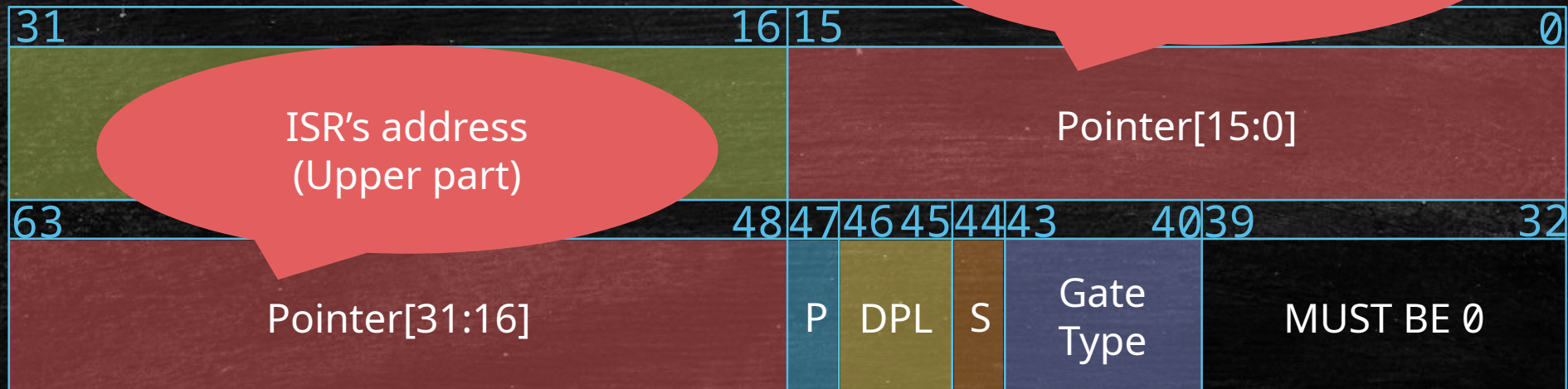
Setting up an IDT

- Pretty much like GDT:
 - Create a table with some entries following the specific [format](#) required by x86
 - Tell the CPU where to find the table using a special instruction: `lidt`
- Format of each entry in IDT:



Setting up an IDT

- Pretty much like GDT:
 - Create a table with some entries following the specific [format](#) required by x86
 - Tell the CPU where to find the table using a special instruction: `lidt`
- Format of each entry in IDT:

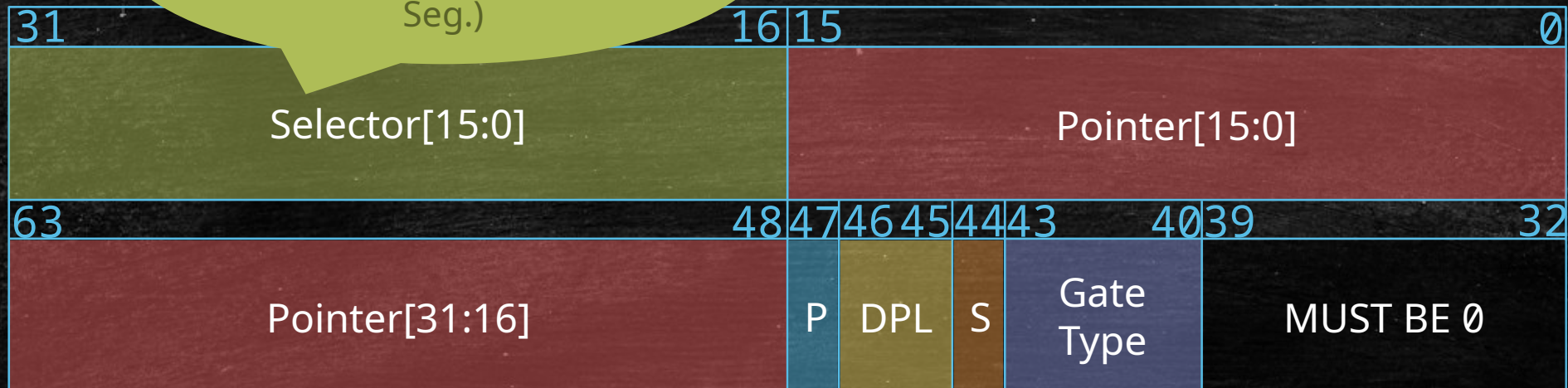


Setting up an IDT

- Pretty much like GDT:
 - Create a table with some entries following the specific [format](#) required by x86
 - Tell the CPU where to find the table using a special instruction: `lidt`

- Format of an IDT entry:

Offset of the entry in GDT
(Has to be a ring 0 Code Seg.)



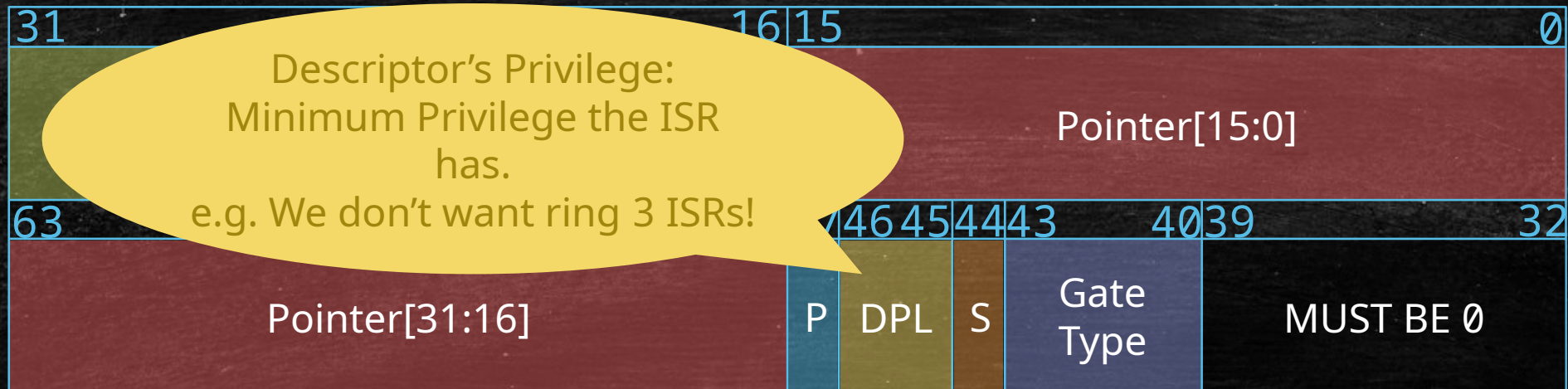
Setting up an IDT

- Pretty much like GDT:
 - Create a table with some entries following the specific [format](#) required by x86
 - Tell the CPU where to find the table using a special instruction: `lidt`
- Format of each entry in IDT:



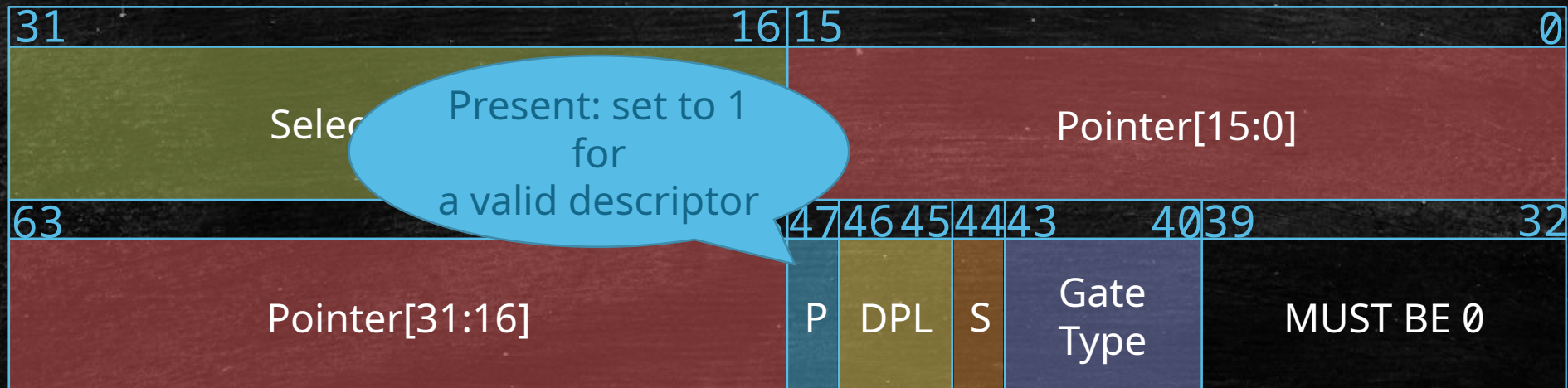
Setting up an IDT

- Pretty much like GDT:
 - Create a table with some entries following the specific [format](#) required by x86
 - Tell the CPU where to find the table using a special instruction: `lidt`
- Format of each entry in IDT:



Setting up an IDT

- Pretty much like GDT:
 - Create a table with some entries following the specific [format](#) required by x86
 - Tell the CPU where to find the table using a special instruction: `lidt`
- Format of each entry in IDT:



Example of an IDT setup code in asm

```
# 1) Allocation of our IDT and ISR pointers
# An empty IDT w/ 49 8-byte entries
.globl idt_base
idt_base:
    .fill 49,8,0

# Definition of IDT pointer to use w/
lidt
.globl idt_pointer
idt_pointer:
    # size - 1
    .short idt_pointer - idt_base - 1
    # base addr. of our IDT
    .long idt_base

# List of pointer to my ISRs
idt_vectors:
    # ISRs 0 to 31 handle CPU exceptions
    .long excep_div_by_zero
    ...

# ISRs 32 to 47 handle IRQ0 to IRQ15
    .long irq0_handler

# 2) Code to populate our IDT
mov $49, %ecx
lea idt_base, %edi
lea idt_vectors, %ebx
1:
# put the first byte of the entry in %eax
# and the second byte in %edx
...
movl %eax, (%edi)
movl %edx, 4(%edi)
addl $8, %edi
addl $4, %ebx
dec %ecx
jne 1b

# Load the IDT
lidt idt_pointer
```

How to test our IDT quickly?

- Cause an exception:
 - E.g. Divide-By-Zero
 - Define an ISR that prints something on the screen
 - After initializing IDT, perform a division by zero and see if the ISR kicks in
- INT instruction:
 - Using inline assembly, issue an INT instruction in your code after IDT is setup
 - Make the corresponding ISR print something on the screen

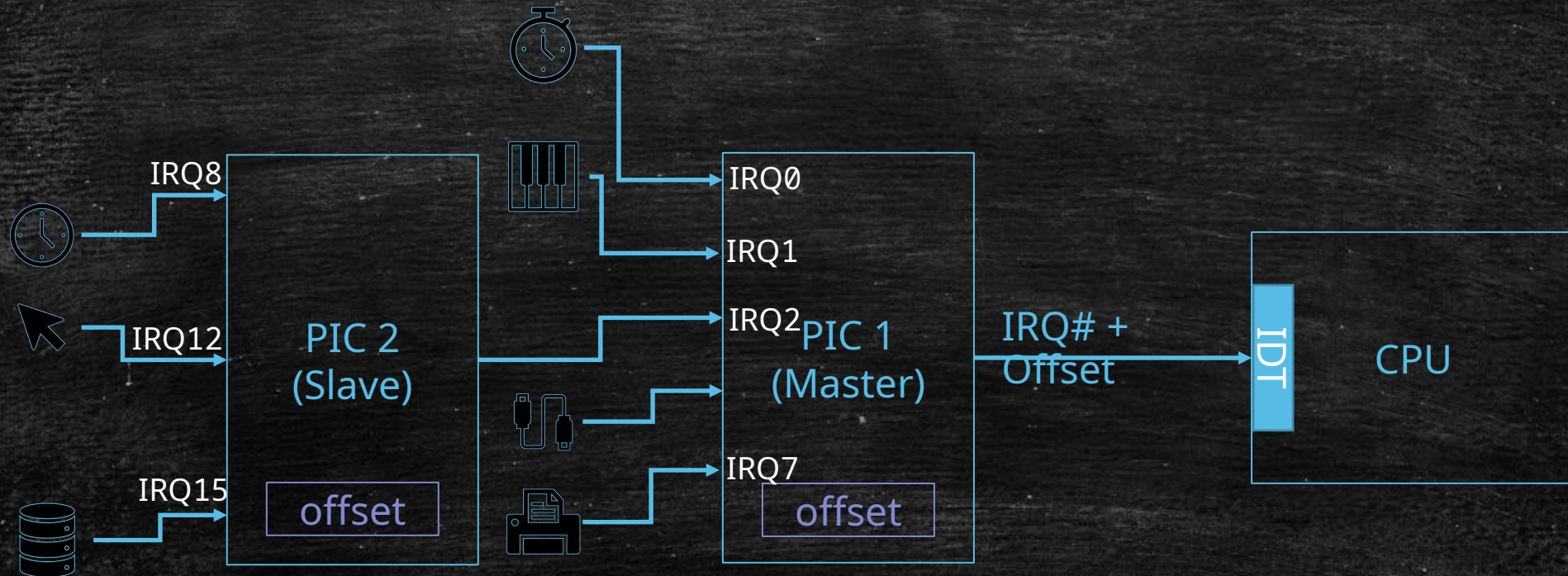
2 - Programmable Interval Timer (PIT)

- A peripheral device that can be programmed using I/O ports
 - Remember the `inb` and `outb` instructions we used in the Primer?
 - Base frequency of about 1.19MHz that can be decreased by a single prescaler (a circuit that performs integer division on a clock frequency)
- It is an interval timer with 3 channels
 - While the prescaler affects all channels, each channel has its own frequency divider. So, they can run with different speeds.
 - Channel 0 is usually used as the system timer and generates the "IRQ-0" upon some event that depends on the mode it's set up to operate in (e.g. when its counter reaches zero in mode 2 - rate generator)
 - Channel 1 and 2 are not really used anymore.
- Tutorial on how to set it up: [here](#)
- Question? How to make PIT generate an interrupt every 10ms?

3 - Programmable Interrupt Ctrl (PIC)

- A peripheral port-based I/O device that delivers IRQs to the CPU
- There are 2 PIC chipsets in the system (master and slave)
 - Each can handle IRQs from 8 devices
 - The second PIC (slave) is connected to IRQ2 (third input) of the master to provide support for an addition 8 devices (15 devices in total)
 - List of IRQs going to each of the PICs: [here](#).
 - Can enable the master only or both
 - A tutorial on how to program PIC: [here](#).

3 - Programmable Interrupt Ctrl (PIC)



Additional tips about PIC/PIT

- Disable the interrupts before both PIC and PIT are setup
 - Why?
- Map IRQs to interrupt numbers beyond 31
 - The interrupt number that CPU uses to look up IDT = IRQ number + Offset
 - Why? Remember the first 32 interrupt numbers are reserved for CPU [exceptions](#)
 - How? Look [here](#).
- Send an End-Of-Interrupt command to
 - The master PIC if the IRQ number is from 0 to 7
 - The slave PIC and then the master PIC if the IRQ number is from 8 to 15
 - Why?
 - How? Look [here](#).