Interrupt and Exception Handling on the x86
Every Exception/Interrupt type is assigned a number: its vector

When an interrupt occurs, the vector determines what code is invoked to handle the interrupt.

JOS example: vector 14 → page fault handler
vector 32 → clock handler → scheduler

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Divide Error</td>
</tr>
<tr>
<td>2</td>
<td>Non-Maskable Interrupt</td>
</tr>
<tr>
<td>3</td>
<td>Breakpoint Exception</td>
</tr>
<tr>
<td>6</td>
<td>Invalid Opcode</td>
</tr>
<tr>
<td>11</td>
<td>Segment Not Present</td>
</tr>
<tr>
<td>12</td>
<td>Stack-Segment Fault</td>
</tr>
<tr>
<td>13</td>
<td>General Protection Fault</td>
</tr>
<tr>
<td>14</td>
<td>Page Fault</td>
</tr>
<tr>
<td>18</td>
<td>Machine Check</td>
</tr>
<tr>
<td>32-255</td>
<td>User Defined Interrupts</td>
</tr>
</tbody>
</table>
Hardware Interrupt Types:
- **Non-Maskable Interrupt**
  - Never ignored
- **INTR Maskable**
  - Ignored when IF is 0

**PIC: Programmable Interrupt Controller (8259A)**
- Has 16 wires to devices (IRQ0 – IRQ15)
- Can be programmed to map IRQ0-15 → vector number
- Vector number is signaled over INTR line.
- In JOS/lab4:
  
  \[
  \text{vector } \leftarrow (\text{IRQ}# + \text{OFFSET})
  \]
Sources: Software-generated Interrupts

Programmed Interrupts
- x86 provides INT instruction.
- Invokes the interrupt handler for vector N (0-255)
- JOS: we use 'INT 0x30' for system calls

Software Exceptions
- Processor detects an error condition while executing an instruction.
- Ex: `divl %eax, %eax`
  - Divide by zero if EAX = 0
- Ex: `movl %ebx, (%eax)`
  - Page fault or seg violation if EAX is un-mapped virtual address.
- Ex: `jmp $BAD_JMP`
  - General Protection Fault (jmp'd out of CS)
Maskable Hardware Interrupts
- Clearing the IF flag inhibits processing hardware interrupts delivered on the INTR line.
- Use the STI (set interrupt enable flag) and CLI (clear interrupt enable flag) instructions.
- IF affected by: interrupt/task gates, POPF, and IRET.

Non-Maskable Interrupt
- Invoked by NMI line from PIC.
- Always Handled immediately.
- Handler for interrupt vector 2 invoked.
- No other interrupts can execute until NMI is done.
IDT:
- Table of 256 8-byte entries (similar to the GDT).
- In JOS: Each specifies a protected entry-point into the kernel.
- 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.
IDT Entries

Selector  Segment Selector for dest. code segment
Offset    Offset to procedure entry point
P         Segment Present Flag
DPL       Descriptor Privilege Level
D         Size of gate: 1 = 32 bits; 0 = 16 bits
[bit 40]  0 = interrupt gate; 1 = trap gate
- JOS approach tries to minimize segmentation usage
  - so ignore segmentation issues with interrupts

**Priority Level Switch**
- CPL is low two bits of CS (11=kernel, 00=user)
- Loading new CS for handler can change CPL.
- JOS interrupt handlers run with kernel CPL.

**Addressing Switch**
- No address space switch when handler invoked.
- Paging is not changed.
- However in: Kernel VA regions now accessible

**Stack Switch (User » Kernel)**
- stack switched to a kernel stack before handler is invoked.
- Specialized Segment for hardware supported multi-tasking
  *(we don't use this x86 feature)*

- TSS Resides in memory
  - TSS descriptor goes into GDT
    *(size and linear address of the TSS)*
  - \texttt{ltr(GD\_TSS)} loads descriptor

- **In JOS's TSS:**
  - SS0:ESP0 kernel stack used by interrupt handlers.
  - All other TSS fields ignored
Exception Entry Mechanism

Kernel » Kernel

(New State)
- SS    unchanged
- ESP   (new frame pushed)
- CS:EIP (from IDT)

User » Kernel

(New State)
- SS:ESP  TSS ss0:esp0
- CS:EIP  (from IDT)
- EFLAGS: interrupt gates: clear IF
struct Trapframe {
    ...  
    u_int tf_trapno;  
    /* below here defined by x86 hardware */  
    u_int tf_err;  
    u_int tf_eip;  
    u_short tf_cs;  
    u_int : 0;  
    u_int tf_eflags;  
    /* below only when crossing rings(e.g. user to kernel) */  
    u_int tf.esp;  
    u_short tf.ss;  
    u_int : 0;  
};
**Exception Return Mechanism**

**iret**: interrupt return instruction
  (top of stack should point to old EIP)

**Where do we return?**

- **Hardware Interrupts**
  old CS:EIP points past last completed instruction.

- **Traps** (INT 30, ...)
  old CS:EIP points past instruction causing exception

- **Faults** (page fault, GPF, ...)
  old CS:EIP points to instruction causing exception

- **Aborts** (hardware errors, bad system table vals...)
  uncertain CS:EIP, serious problems, CPU confused
Example: Page Fault Exceptions

Why?

x86 Page Translation Mechanism encountered an error translating a linear address into a physical address.

Error Code

special error code format:

CR2 register

Linear Address that generated the exception.

Saved CS:EIP

Point to the instruction that generated the exception