You have 12 identical-looking balls. One of these balls has a different weight from all the others. You also have a two-pan balance for comparing weights. Using the balance in the smallest number of times possible, determine which ball has the unique weight, and also determine whether it is heavier or lighter than the others. (Hint: 3 measurements)
Why do we need interrupts?

Remember:
hardware interface is designed to help OS
Why do we need interrupts?

- Fix an abnormal condition
  - Page not mapped in memory
- Notifications from external devices
  - Network packet received
- Preemptive scheduling
  - Timer interrupt
- Secure interface between OS and applications
  - System calls
Two types

Synchronous

- Exceptions – react to an abnormal condition
  - Map the swapped out page back to memory
  - Invoke a system call
  - Intel distinguishes 3 types: faults, traps, aborts

Asynchronous

- Interrupts – preempt normal execution
  - Notify that something has happened (new packet, disk I/O completed, timer tick, notification from another CPU)
Handling interrupts and exceptions

- Same procedure
  - Stop execution of the current program
  - Start execution of a handler
  - Processor accesses the handler through an entry in the Interrupt Descriptor Table (IDT)

- Each interrupt is defined by a number
  - E.g., 14 is pagefault, 3 debug
  - This number is an index into the interrupt table (IDT)
Interrupt descriptor

Interrupt Gate

Offset 31..16

Segment Selector

Offset 15..0

P

DPL

0 D 1 1 0 0 0 0
Interrupt descriptor

Interrupt Gate

31 16 15 14 13 12 8 7 5 4 0
Offset 31..16

Segment Selector
Offset 15..0
Interrupt handlers

- Just plain old code in the kernel
- The IDT stores a pointer to the right handler routine
Interrupt path

Interrupt Vector #

Timer: IRQ0 -> vector 32

Kernel stack

Last stack frame

EBP

Argument 1
Argument 2
Calling EIP ++
Old EBP
Local variables
Saved local values, e.g. push EAX, etc

CS : #1  EIP: <kernel>
SS : #2  ESP: <kernel>
GDT: gdt  TSS: tss
IDT: idt  CR3: pt

Kernel code

vector32
Processing of interrupt (same PL)

1. Push the current contents of the EFLAGS, CS, and EIP registers (in that order) on the stack
2. Push an error code (if appropriate) on the stack
3. Load the segment selector for the new code segment and the new instruction pointer (from the interrupt gate or trap gate) into the CS and EIP registers
4. If the call is through an interrupt gate, clear the IF flag in the EFLAGS register
5. Begin execution of the handler
Interrupted Procedure’s and Handler’s Stack

- EFLAGS
- CS
- EIP
- Error Code

ESP Before Transfer to Handler

ESP After Transfer to Handler

Stack Usage with No Privilege-Level Change
Interrupt path

- Argument 1
- Argument 2
- Calling EIP ++
- Old EBP
- Local variables
  - Saved local values, e.g. push EAX, etc
- EFLAGS
- CS
- EIP
- Error code

Interrupt Vector #

Timer: IRQ0 -> vector 32

CS : HANDLER ADDR
...
...

IDT
...
...

Kernel code

CS : #1
SS : #2
GDT: gdt
IDT: idt
EIP: <kernel>
ESP: <kernel>
TSS: tss
CR3: pt
Processing of interrupt (cross PL)

- Need to change privilege level...
Interrupt path

User stack of a process (can grow up to 2GBs)

Interrupt Vector #

Timer: IRQ0 -> vector 32

GDT
- NULL: 0x8
- KCODE: 0 - 4GB
- KDATA: 0 - 4GB
- K_CPU: 4 bytes
- CODE: 0 - 4GB
- DATA: 0 - 4GB
- TSS: sizeof(tss)

IDT
- CS : HANDLER ADDR
- ...

Page table
- Level 1
- 0 - 4MB
- 4 - 8MB
- ...
- Level 2
- 0 - 4K
- 4K - 8K
- ...
- (4MB - 4K) - 4MB

Kernel code

EBP →

Process

Argument 1
Argument 2
Calling EIP ++
Old EBP
Local variables
Saved local values, e.g. push EAX, etc

Last stack frame
Interrupt descriptor

Interrupt Gate

Offset 31..16

Segment Selector

Offset 15..0

P

DPL 0 D 1 1 0 0 0 0

4
Stack

- Can we continue on the same stack?
Stack

- But how hardware knows where it is?
Task State Segment

- Another magic control block
  - Pointed to by special task register (TR)
    - Selector
    - Actually stored in the GDT
  - Hardware-specified layout
- Lots of fields for rarely-used features
- Two features we care about in a modern OS:
  - Location of kernel stack (fields SS/ESP)
  - I/O Port privileges (more in a later lecture)
Processing of interrupt (cross PL)

1. Save ESP and SS in a CPU-internal register
2. Load SS and ESP from TSS
3. Push user SS, user ESP, user EFLAGS, user CS, user EIP onto new stack (kernel stack)
4. Set CS and EIP from IDT descriptor's segment selector and offset
5. If the call is through an interrupt gate clear some EFLAGS bits
6. Begin execution of a handler
Stack Usage with Privilege-Level Change

Interrupted Procedure’s Stack

ESP Before Transfer to Handler

Handler’s Stack

SS
ESP
EFLAGS
CS
EIP
Error Code

ESP After Transfer to Handler
Complete interrupt path

User stack of a process (can grow up to 2GBs)

Interrupt Vector #

Timer: IRQ0 -> vector 32

Kernel Stack of a process (4K)

User state (saved by hardware)

ESP

Argument 1
Argument 2
Calling EIP ++
Old EBP
Local variables
Saved local values, e.g. push EAX, etc

EBP

Process

Last stack frame

Code, data, heap

ESP

SS
ESP
EFLAGS
CS
CS

CS : #1
SS : #2
EIP: <kernel>
GDT: gdt
TSS: tss
IDT: idt
CR3: pt

GDT
NULL: 0x0
KCODE: 0 - 4GB
KDATA: 0 - 4GB
K_CPU: 4 bytes
CODE: 0 - 4GB
DATA: 0 - 4GB
TSS: sizeof(tss)

IDT
CS : HANDLER ADDR

TSS
...
550:
ESP0:
...

Page table
Level 1
0 - 4MB
4 - 8MB
...
2GB - 2GB + 4MB

Level 2
0 - 4K
4K - 8K
...
(4MB-4K) - 4MB

Kernel code

vector32
Return from an interrupt

- Starts with IRET
  1. Restore the CS and EIP registers to their values prior to the interrupt or exception
  2. Restore EFLAGS
  3. Restore SS and ESP to their values prior to interrupt
     - This results in a stack switch
  4. Resume execution of interrupted procedure
interrupt gate disables interrupts
  - clears the IF flag in EFLAGS register
trap gate doesn't
  - IF flag is unchanged
x86 interrupt table

Device IRQs

Reserved for the CPU

Software Configurable
Interrupts

- Each type of interrupt is assigned an index from 0—255.
  - 0—31 are for processor interrupts fixed by Intel
  - E.g., 14 is always for page faults
- 32—255 are software configured
  - 32—47 are often for device interrupts (IRQs)
  - Most device’s IRQ line can be configured
  - Look up APICs for more info (Ch 4 of Bovet and Cesati)
  - 0x80 issues system call in Linux (more on this later)
Sources

- **Interrupts**
  - External
    - Through CPU pins connected to APIC
  - Software generated with INT n instruction

- **Exceptions**
  - Processor generated, when CPU detects an error in the program
    - Fault, trap, abort
  - Software generated with INTO, INT 3, BOUND
Software interrupts

- The INT n instruction allows software to raise an interrupt
  - 0x80 is just a Linux convention
  - You could change it to use 0x81!
- There are a lot of spare indexes

- OS sets ring level required to raise an interrupt
  - Generally, user programs can’t issue an int 14 (page fault manually)
  - An unauthorized int instruction causes a general protection fault
  - Interrupt 13
Disabling interrupts

• Delivery of maskable interrupts can be disabled with IF (interrupt flag) in EFLAGS register

• Exceptions
  • Non-maskable interrupts (see next slide)
  • INT n – cannot be masked as it is synchronous
<table>
<thead>
<tr>
<th>Vector No.</th>
<th>Mnemonic</th>
<th>Description</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#DE</td>
<td>Divide Error</td>
<td>DIV and IDIV instructions.</td>
</tr>
<tr>
<td>1</td>
<td>#DB</td>
<td>Debug</td>
<td>Any code or data reference.</td>
</tr>
<tr>
<td>2</td>
<td>#NMI</td>
<td>NMI Interrupt</td>
<td>Non-maskable external interrupt.</td>
</tr>
<tr>
<td>3</td>
<td>#BP</td>
<td>Breakpoint</td>
<td>INT 3 instruction.</td>
</tr>
<tr>
<td>4</td>
<td>#OF</td>
<td>Overflow</td>
<td>INTO instruction.</td>
</tr>
<tr>
<td>5</td>
<td>#BR</td>
<td>BOUND Range Exceeded</td>
<td>BOUND instruction.</td>
</tr>
<tr>
<td>6</td>
<td>#UD</td>
<td>Invalid Opcode (UnDefined Opcode)</td>
<td>UD2 instruction or reserved opcode.</td>
</tr>
<tr>
<td>7</td>
<td>#NM</td>
<td>Device Not Available (No Math Coprocessor)</td>
<td>Floating-point or WAIT/FWAIT instruction.</td>
</tr>
<tr>
<td>8</td>
<td>#DF</td>
<td>Double Fault</td>
<td>Any instruction that can generate an exception, an NMI, or an INTR.</td>
</tr>
<tr>
<td>9</td>
<td>#MF</td>
<td>CoProcessor Segment Overrun (reserved)</td>
<td>Floating-point instruction.</td>
</tr>
<tr>
<td>10</td>
<td>#TS</td>
<td>Invalid TSS</td>
<td>Task switch or TSS access.</td>
</tr>
<tr>
<td>11</td>
<td>#NP</td>
<td>Segment Not Present</td>
<td>Loading segment registers or accessing system segments.</td>
</tr>
<tr>
<td>12</td>
<td>#SS</td>
<td>Stack Segment Fault</td>
<td>Stack operations and SS register loads.</td>
</tr>
<tr>
<td>13</td>
<td>#GP</td>
<td>General Protection</td>
<td>Any memory reference and other protection checks.</td>
</tr>
<tr>
<td>14</td>
<td>#PF</td>
<td>Page Fault</td>
<td>Any memory reference.</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>#MF</td>
<td>Floating-Point Error (Math Fault)</td>
<td>Floating-point or WAIT/FWAIT instruction.</td>
</tr>
<tr>
<td>17</td>
<td>#AC</td>
<td>Alignment Check</td>
<td>Any data reference in memory.</td>
</tr>
<tr>
<td>18</td>
<td>#MC</td>
<td>Machine Check</td>
<td>Error codes (if any) and source are model dependent.</td>
</tr>
<tr>
<td>19</td>
<td>#XM</td>
<td>SIMD Floating-Point Exception</td>
<td>SIMD Floating-Point Instruction.</td>
</tr>
<tr>
<td>20-31</td>
<td></td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>32-255</td>
<td></td>
<td>Maskable Interrupts</td>
<td>External interrupt from INTR pin or INT n instruction.</td>
</tr>
<tr>
<td>Vector No.</td>
<td>Mnemonic</td>
<td>Description</td>
<td>Source</td>
</tr>
<tr>
<td>-----------</td>
<td>----------</td>
<td>-------------</td>
<td>--------</td>
</tr>
<tr>
<td>0</td>
<td>#DE</td>
<td>Divide Error</td>
<td>DIV and IDIV instructions.</td>
</tr>
<tr>
<td>1</td>
<td>#DB</td>
<td>Debug</td>
<td>Any code or data reference.</td>
</tr>
<tr>
<td>2</td>
<td>#NMI</td>
<td>NMI Interrupt</td>
<td>Non-maskable external interrupt.</td>
</tr>
<tr>
<td>3</td>
<td>#BP</td>
<td>Breakpoint</td>
<td>INT 3 instruction.</td>
</tr>
<tr>
<td>4</td>
<td>#OF</td>
<td>Overflow</td>
<td>INTO instruction.</td>
</tr>
<tr>
<td>5</td>
<td>#BR</td>
<td>BOUND Range Exceeded</td>
<td>BOUND instruction.</td>
</tr>
<tr>
<td>6</td>
<td>#UD</td>
<td>Invalid Opcode (UnDefined Opcode)</td>
<td>UD2 instruction or reserved opcode.¹</td>
</tr>
<tr>
<td>7</td>
<td>#NM</td>
<td>Device Not Available (No Math Coprocessor)</td>
<td>Floating-point or WAIT/FWAIT instruction.</td>
</tr>
<tr>
<td>8</td>
<td>#DF</td>
<td>Double Fault</td>
<td>Any instruction that can generate an exception, an NMI, or an INTR.</td>
</tr>
<tr>
<td>9</td>
<td>#MF</td>
<td>CoProcessor Segment Overrun (reserved)</td>
<td>Floating-point instruction.²</td>
</tr>
<tr>
<td>10</td>
<td>#TS</td>
<td>Invalid TSS</td>
<td>Task switch or TSS access.</td>
</tr>
<tr>
<td>11</td>
<td>#NP</td>
<td>Segment Not Present</td>
<td>Loading segment registers or accessing system segments.</td>
</tr>
<tr>
<td>12</td>
<td>#SS</td>
<td>Stack Segment Fault</td>
<td>Stack operations and SS register loads.</td>
</tr>
<tr>
<td>13</td>
<td>#GP</td>
<td>General Protection</td>
<td>Any memory reference and other protection checks.</td>
</tr>
<tr>
<td>14</td>
<td>#PF</td>
<td>Page Fault</td>
<td>Any memory reference.</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td>Reserved</td>
<td>Any memory reference.</td>
</tr>
<tr>
<td>16</td>
<td>#MF</td>
<td>Floating-Point Error (Math Fault)</td>
<td>Floating-point or WAIT/FWAIT instruction.</td>
</tr>
<tr>
<td>17</td>
<td>#AC</td>
<td>Alignment Check</td>
<td>Any data reference in memory.³</td>
</tr>
<tr>
<td>18</td>
<td>#MC</td>
<td>Machine Check</td>
<td>Error codes (if any) and source are model dependent.⁴</td>
</tr>
<tr>
<td>19</td>
<td>#XM</td>
<td>SIMD Floating-Point Exception</td>
<td>SIMD Floating-Point Instruction⁵</td>
</tr>
<tr>
<td>20-31</td>
<td></td>
<td>Reserved</td>
<td>Any memory reference.</td>
</tr>
<tr>
<td>32-255</td>
<td></td>
<td>Maskable Interrupts</td>
<td>External interrupt from INTR pin or INT n instruction.</td>
</tr>
</tbody>
</table>
Nonmaskable interrupts (NMI)

- Delivered even if IF is clear, e.g. interrupts disabled
  - CPU blocks subsequent NMI interrupts until IRET
- Sources
  - External hardware asserts the NMI pin
  - Processor receives a message on the system bus, or the APIC serial bus with NMI delivery mode
- Delivered via vector #2
<table>
<thead>
<tr>
<th>Vector No.</th>
<th>Mnemonic</th>
<th>Description</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>#DE</td>
<td>Divide Error</td>
<td>DIV and IDIV instructions.</td>
</tr>
<tr>
<td>1</td>
<td>#DB</td>
<td>Debug</td>
<td>Any code or data reference.</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>NMI Interrupt</td>
<td>Non-maskable external interrupt.</td>
</tr>
<tr>
<td>3</td>
<td>#BP</td>
<td>Breakpoint</td>
<td>INT 3 instruction.</td>
</tr>
<tr>
<td>4</td>
<td>#OF</td>
<td>Overflow</td>
<td>INTO instruction.</td>
</tr>
<tr>
<td>5</td>
<td>#BR</td>
<td>BOUND Range Exceeded</td>
<td>BOUND instruction.</td>
</tr>
<tr>
<td>6</td>
<td>#UD</td>
<td>Invalid Opcode (UnDefined Opcode)</td>
<td>UD2 instruction or reserved opcode.(^1)</td>
</tr>
<tr>
<td>7</td>
<td>#NM</td>
<td>Device Not Available (No Math Coprocessor)</td>
<td>Floating-point or WAIT/FWAIT instruction.</td>
</tr>
<tr>
<td>8</td>
<td>#DF</td>
<td>Double Fault</td>
<td>Any instruction that can generate an exception, an NMI, or an INTR.</td>
</tr>
<tr>
<td>9</td>
<td>#MF</td>
<td>CoProcessor Segment Overrun (reserved)</td>
<td>Floating-point instruction.(^2)</td>
</tr>
<tr>
<td>10</td>
<td>#TS</td>
<td>Invalid TSS</td>
<td>Task switch or TSS access.</td>
</tr>
<tr>
<td>11</td>
<td>#NP</td>
<td>Segment Not Present</td>
<td>Loading segment registers or accessing system segments.</td>
</tr>
<tr>
<td>12</td>
<td>#SS</td>
<td>Stack Segment Fault</td>
<td>Stack operations and SS register loads.</td>
</tr>
<tr>
<td>13</td>
<td>#GP</td>
<td>General Protection</td>
<td>Any memory reference and other protection checks.</td>
</tr>
<tr>
<td>14</td>
<td>#PF</td>
<td>Page Fault</td>
<td>Any memory reference.</td>
</tr>
<tr>
<td>15</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>#MF</td>
<td>Floating-Point Error (Math Fault)</td>
<td>Floating-point or WAIT/FWAIT instruction.</td>
</tr>
<tr>
<td>17</td>
<td>#AC</td>
<td>Alignment Check</td>
<td>Any data reference in memory.(^3)</td>
</tr>
<tr>
<td>18</td>
<td>#MC</td>
<td>Machine Check</td>
<td>Error codes (if any) and source are model dependent.(^4)</td>
</tr>
<tr>
<td>19</td>
<td>#XM</td>
<td>SIMD Floating-Point Exception</td>
<td>SIMD Floating-Point Instruction(^5)</td>
</tr>
<tr>
<td>20-31</td>
<td>Reserved</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32-255</td>
<td>Maskable Interrupts</td>
<td>External interrupt from INTR pin or INT (n) instruction.</td>
<td></td>
</tr>
</tbody>
</table>

\(^1\) See Section 2.7.2 of the instruction set architecture documentation.

\(^2\) Reserved for floating-point instructions.

\(^3\) Reserved for data references in memory.

\(^4\) Reserved for machine checks.

\(^5\) Reserved for SIMD floating-point exceptions.
Thank you.