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 and asynchronous
Synchronous

- Exceptions – react to an abnormal condition
  - E.g., page mapping (virtual address) is not present in the page table
    - Bring the swapped page back to memory (copy from disk)
    - Fix the page-table entry
  - Invoke a system call
    - Transition from user-level to kernel (more later)
  - Intel distinguishes 3 types: faults, traps, aborts

Asynchronous

- Interrupts – preempt normal execution
  - Notify that something has happened
    - New packet from the network, 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 path
- no change in privilege level
- e.g., we're already running in the kernel
Interrupt descriptor

Interrupt Gate

<table>
<thead>
<tr>
<th>31</th>
<th>16 15 14 13 12</th>
<th>8 7 5 4 0</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Offset 31..16</td>
<td>P D PL 0 D 1 1 0 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>31</th>
<th>16 15</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Segment Selector</td>
<td>Offset 15..0</td>
<td></td>
</tr>
</tbody>
</table>
Interrupt descriptor

Interrupt Gate

Segment Selector

Offset 31..16

Offset 15..0

F

DPL

0 D 1 1 0

0 0 0

4
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

Last stack frame

Kernel stack

EBP

Interrupt path

Kernel code

vector32

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

CS : #1
SS : #2
GDT: gdt
IDT: idt

EIP: <kernel>
ESP: <kernel>

CS : HANDLER ADDR

IDT

...
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 (disable further interrupts)
5. Begin execution of the handler
Stack Usage with No Privilege-Level Change

Interrupted Procedure’s and Handler’s Stack

<table>
<thead>
<tr>
<th>EFLAGS</th>
<th>CS</th>
<th>EIP</th>
<th>Error Code</th>
</tr>
</thead>
</table>

ESP Before Transfer to Handler
ESP After Transfer to Handler
Interrupt path

EBP →

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

Kernel stack

Interrupt Vector #

Timer: IRQ0 -> vector 32

IDT

... CS : HANDLER ADDR ...

... ...

Kernel code

vector32

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...
Detour:
What are those privilege levels?
Recap: Can a process overwrite kernel memory?
Privilege levels

- Each segment has a privilege level
  - DPL (descriptor privilege level)
  - 4 privilege levels ranging 0-3
Privilege levels

• Currently running code also has a privilege level
  • “Current privilege level” (CPL): 0-3
  • It is saved in the %cs register
WAT!
Segmented programming (not real)

```c
static int x = 1;
int y; // stack
if (x) {
    y = 1;
    printf ("Boo");
} else
    y = 0;
```

```c
ds:x = 1; // data
ss:y;      // stack
if (ds:x) {
    ss:y = 1;
    cs:printf(ds:"Boo");
} else
    ss:y = 0;
```
Actual switch

- Use long jump to change code segment

9153  ljmp  $(SEG_KCODE<<3), $start32

- Explicitly specify code segment, and address

- Segment is 0b1000 (0x8)
Privilege level transitions

- CPL can access only less privileged segments
  - E.g., 0 can access 1, 2, 3
- Some instructions are “privileged”
  - Can only be invoked at CPL = 0
  - Examples:
    - Load GDT
    - MOV <control register>
      - E.g. reload a page table by changing CR3
Real world

• Only two privilege levels are used in modern OSes:
  • OS kernel runs at 0
  • User code runs at 3
• This is called “flat” segment model
  • Segments for both 0 and 3 cover entire address space
• But then... how the kernel is protected?
  • Page tables
Page table: user bit

• Each entry (both Level 1 and Level 2) has a bit
  • If set, code at privilege level 3 can access
  • If not, only levels 0-2 can access
• Note, only 2 levels, not 4 like with segments
• All kernel code is mapped with the user bit clear
  • This protects user-level code from accessing the kernel
End of detour:
Back to handling interrupts
Thank you.