CS3210: Interrupts

Tim Andersen

Interrupt

- An interrupt informs the CPU that a service is needed
- Sources of interrupts
    - Internal faults: divide by zero, overflow
    - User software
    - Hardware
    - Reset

x86 Exceptions and Interrupts

- 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

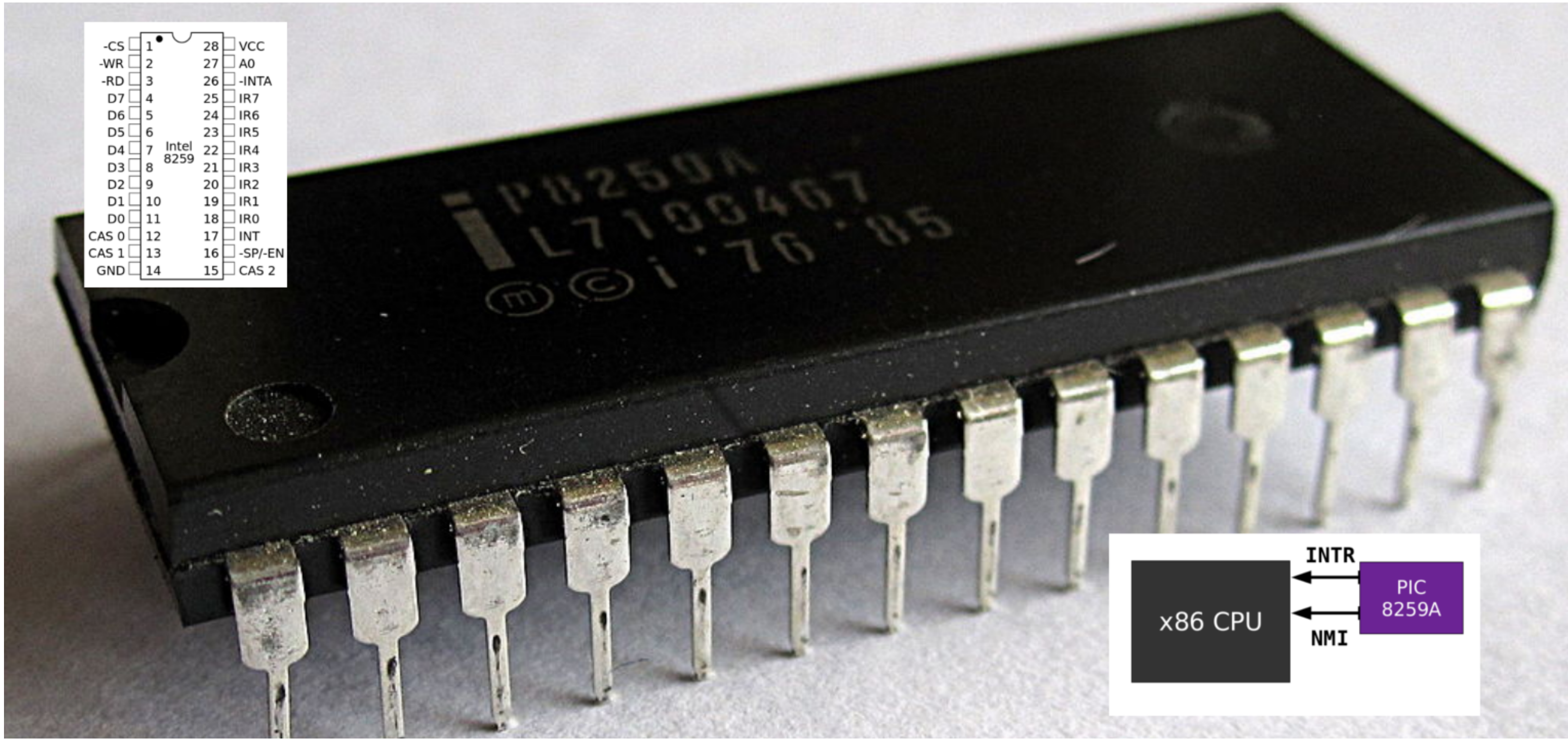
Hardware Interrupts

- **Non-Maskable Interrupts**
  - Never ignored, e.g., power failure, memory error
  - In x86, vector 2, prevents other interrupts from executing.
- **INTR Maskable**
  - Ignored when IF in `EFLAGS` is 0
  - Enabling/disabling: - `sti`: set interrupt - `cli`: clear interrupt
- INTA
  - Interrupt acknowledgement

PIC: Programmable Interrupt Controller (8259A)

- Has 16 wires to devices (IRQ0-IRQ16)
- Can be programmed to map IRQ0-15 → vector number
- Vector number is signaled over INTR line
- In JOS/lab4
    - vector ← (IRQ# + OFFSET)

PIC Diagram


-CS 1
-WR 2
-RD 3
D7 4
D6 5
D5 6
D4 7
D3 8
D2 9
D1 10
D0 11
CAS 0 12
CAS 1 13
GND 14
28 VCC
27 A0
26 -INTA
25 IR7
24 IR6
23 IR5
22 IR4
21 IR3
20 IR2
19 IR1
18 IR0
17 INT
16 -SP/-EN
15 CAS 2
Intel 8259
P8259A
L7100467
i '76 '85
INTR
NMI
x86 CPU
PIC 8259A

"Software" interrupt: INT

- Intentionally interrupts
    - x86 provides the INT instruction
    - Invokes the interrupt handler for the vector (0-255)
    - JOS: `INT 0x30` for system calls
- Entering: `int N`
- Exiting: `iret`

The INT instruction

- The INT instruction has the following steps:
  - decide the vector number, in this case it's the 0x40 in int 0x40
  - fetch the interrupt descriptor for vector 0x40 from the IDT. The CPU finds it by taking the 0x40'th 8-byte entry starting at the physical address that the IDTR CPU register points to.
  - check that CPL <- DPL in the descriptor (but only if INT instruction).
  - save ESP and SS in a CPU-internal register (but only if target segment selector's PL < CPL).

The INT instruction (cont)

- Continued
  - load SS and ESP from TSS ("")
  - push user SS ("")
  - push user ESP ("")
  - push user EFLAGS
  - push user CS
  - push user EIP
  - clear some EFLAGS bits
  - set CS and EIP from IDT descriptor's segment selector and offset

Example: entering (usys.S)

- `vectorN` → `alltraps` → `trap()` → `syscall()`

```
#define SYSCALL(name)         \
  .globl name;                \
  name:                       \
    movl $SYS_ ## name, %eax; \
    int $T_SYSCALL;           \
    ret

SYSCALL(fork)
SYSCALL(exit)
...
```

Example: exiting (trapasm.S)

- `syscall()` → `trapret()` → `iret`

```
.globl trapret
trapret:
  popal
  popl %gs
  popl %fs
  popl %es
  popl %ds
  addl $0x8, %esp  # trapno and errcode
  iret
```

Interrupt Vector (vector.S)

- `int 0` → `vector0`

```
# handlers
vector0:
  pushl $0
  pushl $0
  jmp alltraps
...

# vector table
vectors:
  .long vector0
  .long vector1
...
```

Interrupt Vector

- Some vectors need to push 0 for their error code and others do not

```
vector0:
  pushl $0     ; error code
  pushl $0     ; #vector
  jmp alltraps

...
vector8:
  pushl $8     ; #vector
  jmp alltraps
...
```

Trap Handling DEMO

- `int 0x40` entered the kernel at vector64, generated by vectors.pl.

  `b vector64`
- What is the current CPL? How was it set?
  - Could the user abuse the INT instruction to exercise privilege or break the kernel?
- `x/6x $esp` in order to see what int put on the stack.
  - Compare to handout figure?
  - What stack is being used?
- `x/3i vector64`

Trap Return

- syscall() returns to trap(), and trap() returns to alltraps
- b trap.c:44 (instruction after call syscall).
    - print *tf
    - What is different and why?
    - si until popal.
    - x/19x $esp to see the trap frame again.
- single-step until iret, x/5x $esp, single-step
    - into user space. Print the registers and stack.

Fault Handling Traps

- What would happen if a user program divided by zero?
  - What if kernel code divided by zero?
- In Unix, traps often get translated into signals to the process.
  - Some traps, though, are (partially) handled internally by the kernel -- which ones?
- Some traps push an extra error code onto the stack (typically containing the segment descriptor that caused a fault).
  - But this error code isn't pushed by the INT instruction.

JOS Trap Frame

```c
struct Trapframe {
  struct PushRegs tf_regs;
  uint16_t tf_es;
  uint16_t tf_padding1;
  uint16_t tf_ds;
  uint16_t tf_padding2;
  uint32_t tf_trapno;
  /* below here defined by x86 hardware */
  uint32_t tf_err;
  uintptr_t tf_eip;
  uint16_t tf_cs;
  uint16_t tf_padding3;
  uint32_t tf_eflags;
  /* below here only when crossing rings, such as from user to kernel */
  uintptr_t tf_esp;
  uint16_t tf_ss;
  uint16_t tf_padding4;
} __attribute__((packed));
```

Real-mode

- For any INT n, n is multiplied by 4
  - In the address “4n” the offset address the handler is found
- Example:Intel has set aside INT 2 for the NMI interrupt
  - Whenever the NMI pin is activated, the CPU jumps to physical memory location 00008 to fetch the CS:IP of the interrupt service routine associated with the NMI.
- In protected mode, this scheme is replaced by the Interrupt Descriptor Table

Interrupt Descriptor Table

- **IDT**
  - Table of 256 8-byte entries (similar to 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

Interrupt Descriptor Table Diagram

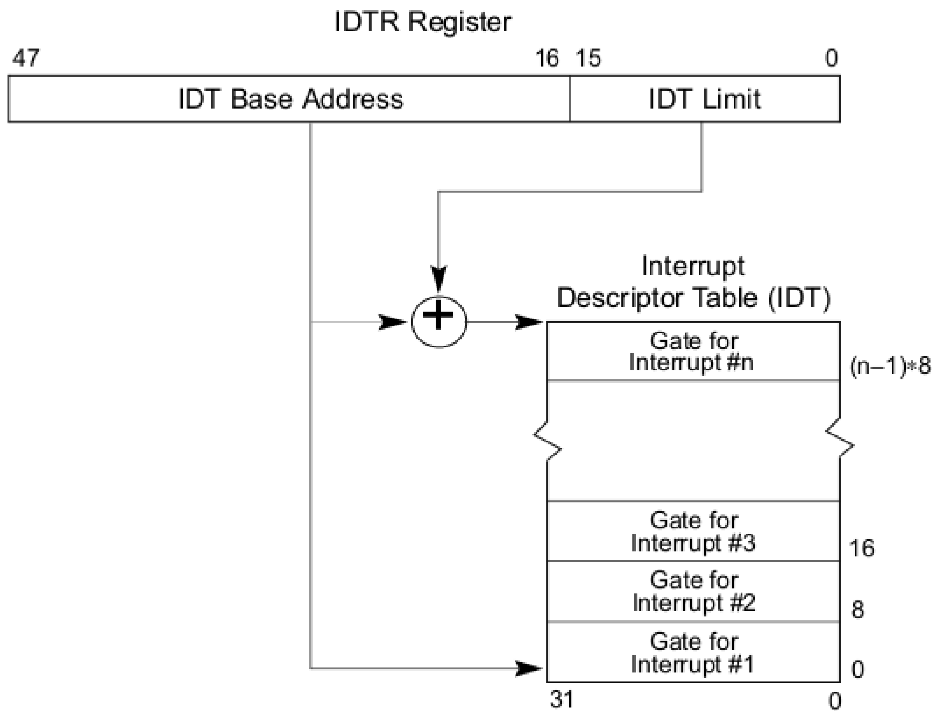


Figure 6-1. Relationship of the IDTR and IDT

Initializing IDT in xv6 (trap.c)

- `main()` → `tvinit()`

```
void tvinit(void)
{
  int i;
  for (i = 0; i < 256; i++)
    SETGATE(idt[i], 0, SEG_KCODE<<3, vectors[i], 0);

  // Q?
  SETGATE(idt[T_SYSCALL], 1, SEG_KCODE<<3, \
            vectors[T_SYSCALL], DPL_USER);
}
```

Initializing IDT in xv6 (trap.c)

- `main()` → `idtinit()`

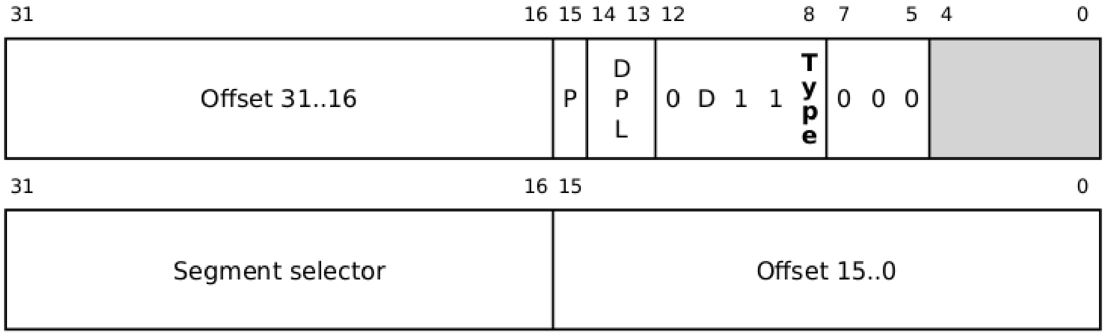
```
void
idtinit(void)
{
  lidt(idt, sizeof(idt));
}
```

Interrupt Descriptor Entry

- Offset is a 32-bit value split into two parts pointing to the destination IP or EIP
- Segment selector points to the destination CS in the kernel
- Present flag indicates that this is a valid entry
- Descriptor Privilege Level indicates the minimum privilege level of the caller to prevent users from calling hardware interrupts directly
- Size of gate can be 32 bits or 16 bits
- Gate can be interrupt (`int` instruction) or trap gate

Interrupt Descriptor Entry

Interrupt/trap gate



Type	0=Interrupt gate 1=Trap gate	P	Present
Selector	Destination CS	DPL	Descriptor privilege level (CPL required to invoke gate)
Offset	Destination IP or EIP	D	Size of gate (0=16-bits, 1=32-bits)

Interrupt Descriptor Table

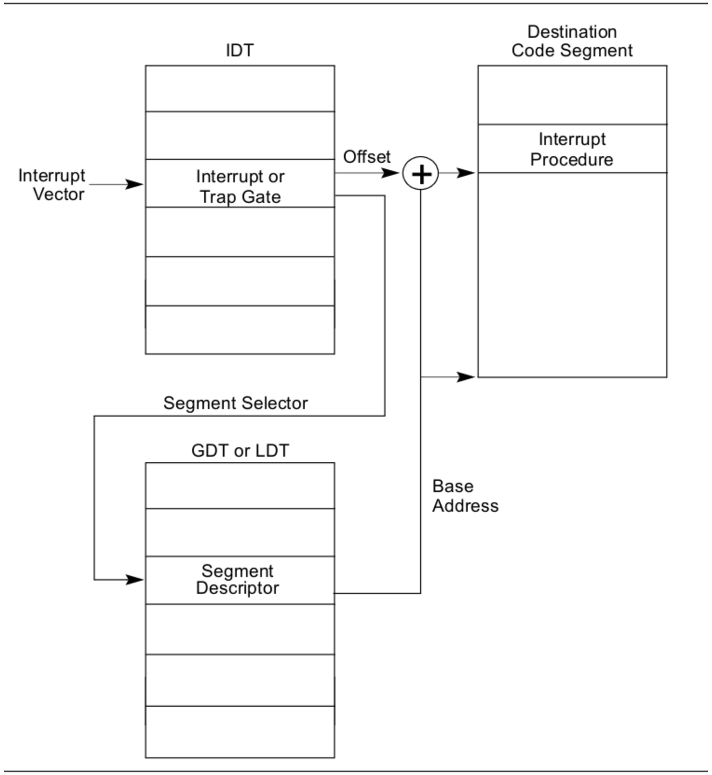


Figure 6-3. Interrupt Procedure Call

Predefined Interrupt Vectors

- `0`: Divide Error
- `1`: Debug Exception
- `2`: Non-Maskable Interrupt
- `3`: Breakpoint Exception (e.g., `int3`)
- `4`: Invalid Opcode
- `13`: General Protection Fault
- `14`: Page Fault
- `18`: Machine Check (abort)
- `32-255`: User Defined Interrupts

Software Exceptions

- Processor detects an error condition while executing
- E.g., divl %eax, %eax
  - Divide by zero if eax = 0
- E.g., movl %ebx, (%eax)
  - Page fault or seg violation if eax is unmapped
- E.g., jmp $BAD_JMP
  - General Protection Fault (jmpd out of CS)

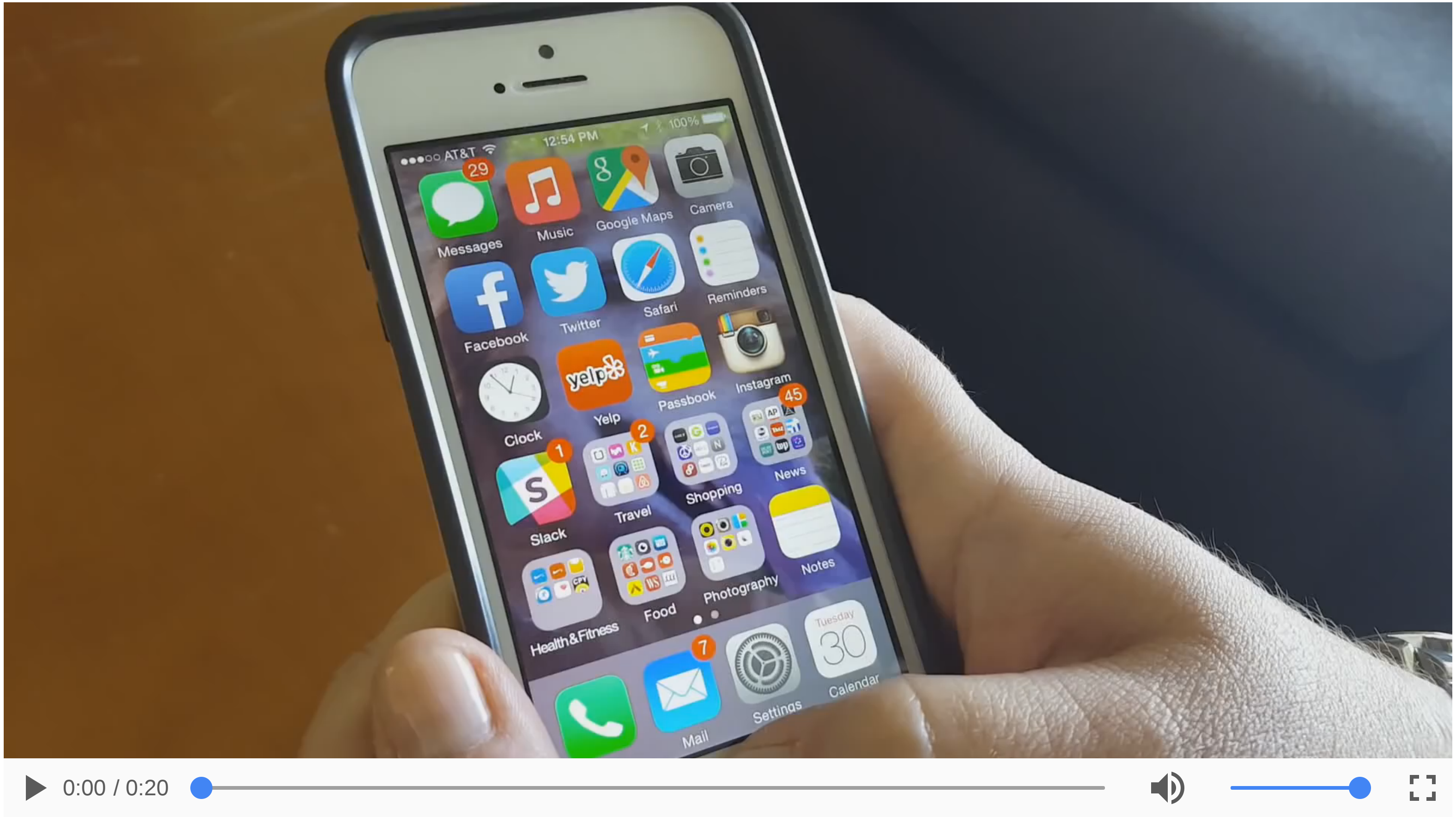
Example: Divide Error

```
int main(int argc, char **argv)
{
  int x, y, z;
  if (argc < 3)
    exit();

  x = atoi(argv[1]);
  y = atoi(argv[2]);

  // Q?
  z  = x / y;
  printf(1, "%d / %d = %d\n", x, y, z);
  exit();
}
```

Example: 0/0 = ?


AT&T
12:54 PM
100%
Messages
Music
Google Maps
Camera
Facebook
Twitter
Safari
Reminders
Clock
Yelp
Passbook
Instagram
Slack
Travel
Shopping
News
Health&Fitness
Food
Photography
Notes
Mail
Settings
Tuesday
30
Calendar
0:00 / 0:20

Let's implement, 0/0 = 0!

- Q: plan?

```
$ div 0 0
0 / 0 = 0
$
```