

# *Interrupts*

# Objectives

- Introduce the x86 interrupt handling model
- Explain the functionality of the Interrupt Controller
- Explain how TOS handles interrupts

# Review: Segmentation

- All x86 memory references go through address translation.
- Used primarily for virtual memory
  - Details on VM later this semester
- Segmentation is always used, we can't avoid it!


# x86 Segmentation

- A *virtual address* is defined by selector:offset and is converted to a *linear address*
- Each selector value points to a segment and the offset value points to the offset within that segment
  - Selector: 16 bits
  - Offset: 32 bits

# Segmentation in TOS

- Segments in TOS are defined so that a physical address is identical to the virtual address.
- The segment table is constructed and loaded during the boot process; i.e., before calling `kernel_main()`
- `%CS` is loaded with `0x8` (GDT entry 1) and `%DS` and `%SS` are loaded with `0x10` (GDT entry 2)

# Inter-Segment Subroutines

- So far, when doing a `CALL` instruction, we only specified the 32-bit offset, but not the segment sector 
  - This is called an *Intra-Segment Jump* because the jump happens within the same segment
- An *Inter-Segment Jump* jumps to a different segment
- For an inter-segment procedure call, not only the return address (i.e., the offset) is pushed on the stack, but also `%CS`

# Reacting to External Events

- An OS frequently needs to react to external events:
  - User has pressed a key on the keyboard
  - User has moved the mouse
  - Network has received a new packet
  - Data has been read from the hard disk
- There are two possible ways to do this:
  - Polling
  - Interrupts

# Polling

- The OS periodically probes (or polls) the hardware
- This polling has to occur in order not to miss any events:

```
Some_app_code();  
Probe_hardware();  
Some_more_app_code();  
Probe_hardware();  
Some_more_app_code();  
Probe_hardware();
```

- **Advantages:**
  - Easy to understand
  - Easy to implement
  - No special support from the CPU needed
- **Disadvantage:**
  - Very, very messy code (because the hardware needs to be probed very frequently)



# Interrupts

- Interrupts are a special mechanism to react efficiently to external events.
- An interrupt essentially leads to an asynchronous subroutine call.
- When an interrupt occurs, the CPU “interrupts” the currently running program and calls a subroutine.
- This subroutine is called an *Interrupt Service Routine* (ISR) because it handles the interrupt.
- Note: the ISR is not a process!
- After the ISR has finished, it returns to the location from which it was called.
- The currently running process does not notice that it was interrupted!

# Calling the ISR



- Interrupt interrupts the currently running process
- For that reason, the ISR has to make sure it does not change the state of the CPU
- The ISR therefore has to save the context of the process that gets interrupted
- IRET is a special assembly instruction that exits an ISR (more on this later)

# Types of Interrupts

- There are two different types of interrupts
  - Normal interrupts: interrupts that are generated by external hardware. Normal interrupts can be masked (i.e., turned-off)
  - Non-maskable interrupts (NMI): NMI are generated for certain internal errors (e.g. division-by-zero). These interrupts can not be masked (i.e., turned-off)
- We will not write ISRs for NMIs in TOS because if everything is working, those should never happen.
- The only thing we will need to worry about are normal interrupts.

# Normal Interrupts

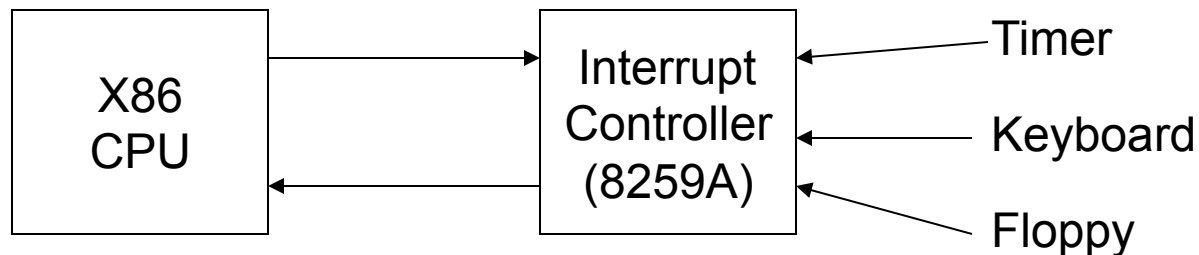
- Normal interrupts can be masked, i.e., they can be turned off
- When normal interrupts are turned off, they still happen, but they are simply ignored by the CPU
- Whether interrupts are masked or not is determined by the IF (Interrupt Flag) of the EFLAGS register
  - `IF == 0`: interrupts are masked
  - `IF == 1`: interrupts are not masked
- IF can be set to 0 with the assembly instruction `cli` (clear interrupts)
- IF can be set to 1 with the assembly instruction `sti` (set interrupts)
- When TOS is booted, interrupts are masked (i.e., the boot loader executes a `cli`)
- Before doing a `sti`, ISRs have to be initialized properly

# Raising an Interrupt

- Here is the sequence of events:
  - External hardware sends signal to the interrupt controller
  - Interrupt controller raises the appropriate interrupt with the x86
  - After the x86 has finished the current instruction, the following things happen:
    - » EFLAGS are pushed onto the stack
    - » %CS is pushed onto the stack (as a 32-bit value)
    - » %EIP is pushed onto the stack
    - » CLI (disable interrupts)
    - » Do an inter-segment jump to the entry point of the ISR (defined in the IDT, see later slide)
- Once the ISR is entered, the ISR has to save all x86 registers onto the stack in order to save the context of the program it was interrupting

# Hardware

- What does an interrupt look on the hardware side?



- A special interrupt controller receives a signal from an external hardware (e.g. floppy)
- The Interrupt Controller then raises an interrupt with the x86
- If  $IF == 1$  in the EFLAGS register an appropriate ISR is executed.
- The x86 supports 256 different interrupts
- The 8259A maps signals from external hardware to one of those 256 interrupts

# Interrupt Controller

- Every PC has an Interrupt Controller (8259A)
- Its purpose is to mediate between external hardware and the x86 CPU
- When the PC is turned on, the 8259A maps external hardware to certain interrupts. E.g., the timer is mapped to interrupt 8
- BIG PROBLEM: with newer x86 CPUs, the first 16 interrupts (0-15) are NMIs which have a specific meaning (e.g. interrupt 8 is a Double Fault)
- How can the x86 then distinguish between a double fault and a timer interrupt? Answer: It can't!!
- Solution: we have to re-program the 8259A to map interrupts for external hardware to other interrupt numbers
- Function `re_program_interrupt_controller()` in `~/tos/kernel/intr.c` is doing this
- This function is given to you (see next slide), but you have to call it from `init_interrupts()`

```

void re_program_interrupt_controller()
{
    // Send initialization sequence to 8259A-1
    asm ("movb $0x11,%al;outb %al,$0x20;call delay");
    // Send initialization sequence to 8259A-2
    asm ("movb $0x11,%al;outb %al,$0xA0;call delay");
    // IRQ base for 8259A-1 is 0x60
    asm ("movb $0x60,%al;outb %al,$0x21;call delay");
    // IRQ base for 8259A-2 is 0x68
    asm ("movb $0x68,%al;outb %al,$0xA1;call delay");
    // 8259A-1 is the master
    asm ("movb $0x04,%al;outb %al,$0x21;call delay");
    // 8259A-2 is the slave
    asm ("movb $0x02,%al;outb %al,$0xA1;call delay");
    // 8086 mode for 8259A-1
    asm ("movb $0x01,%al;outb %al,$0x21;call delay");
    // 8086 mode for 8259A-2
    asm ("movb $0x01,%al;outb %al,$0xA1;call delay");
    // Don't mask IRQ for 8259A-1
    asm ("movb $0x00,%al;outb %al,$0x21;call delay");
    // Don't mask IRQ for 8259A-2
    asm ("movb $0x00,%al;outb %al,$0xA1;call delay");
}

```



# Interrupt Controller

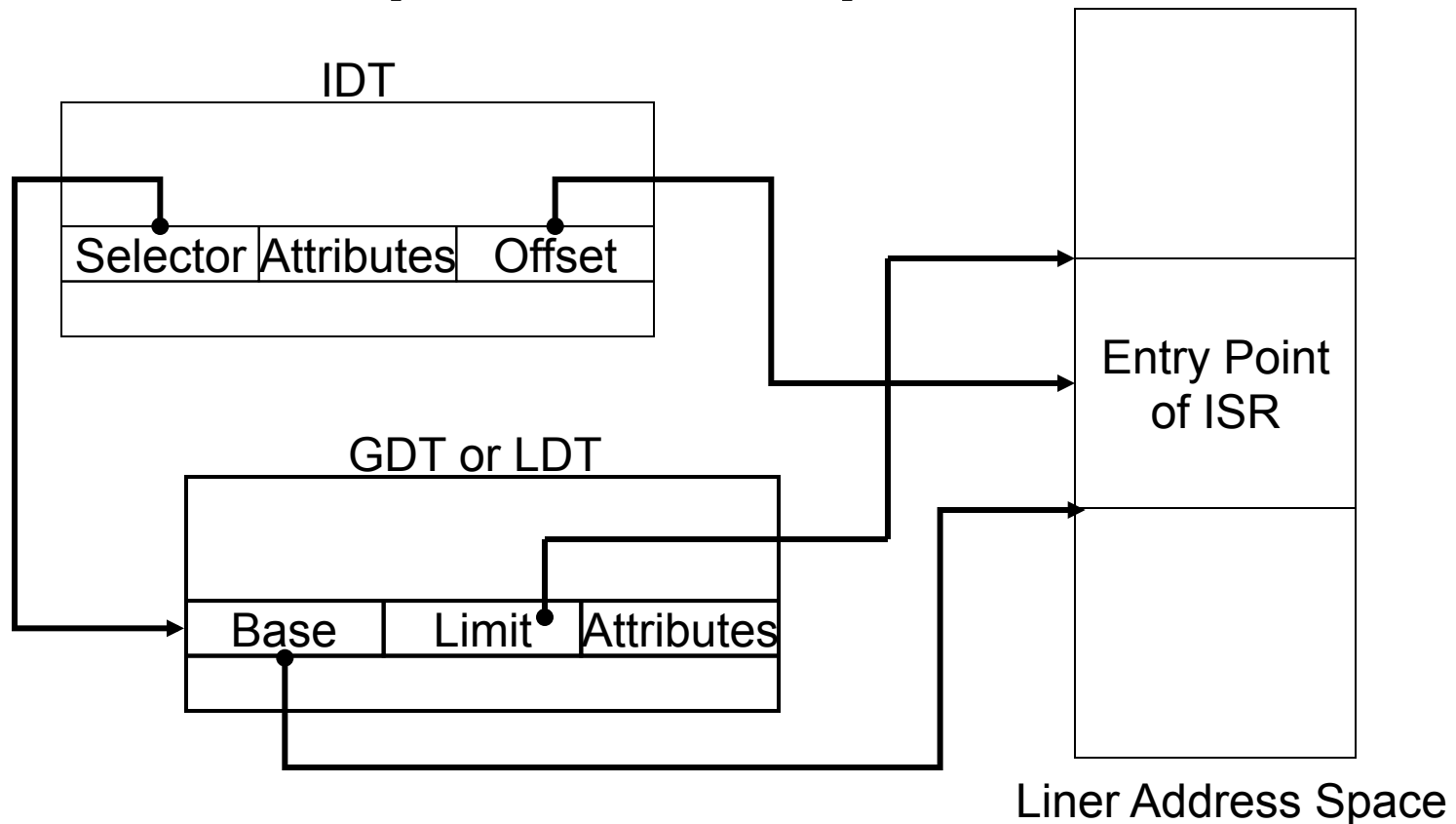
- In TOS we will only make use of three interrupts
  - Timer: used for all timing related issues
  - COM1: used for communicating with the train
  - Keyboard: used whenever the user types a key on the keyboard
- After `re_program_interrupt_controller()` is called, the timer is mapped to interrupt `0x60`, COM1 is mapped to interrupt `0x64` and the keyboard is mapped to interrupt `0x61`
- There are three defines for this in `~/tos/include/kernel.h`:

```
#define TIMER_IRQ 0x60
#define COM1_IRQ 0x64
#define KEYB_IRQ 0x61
```

# Software

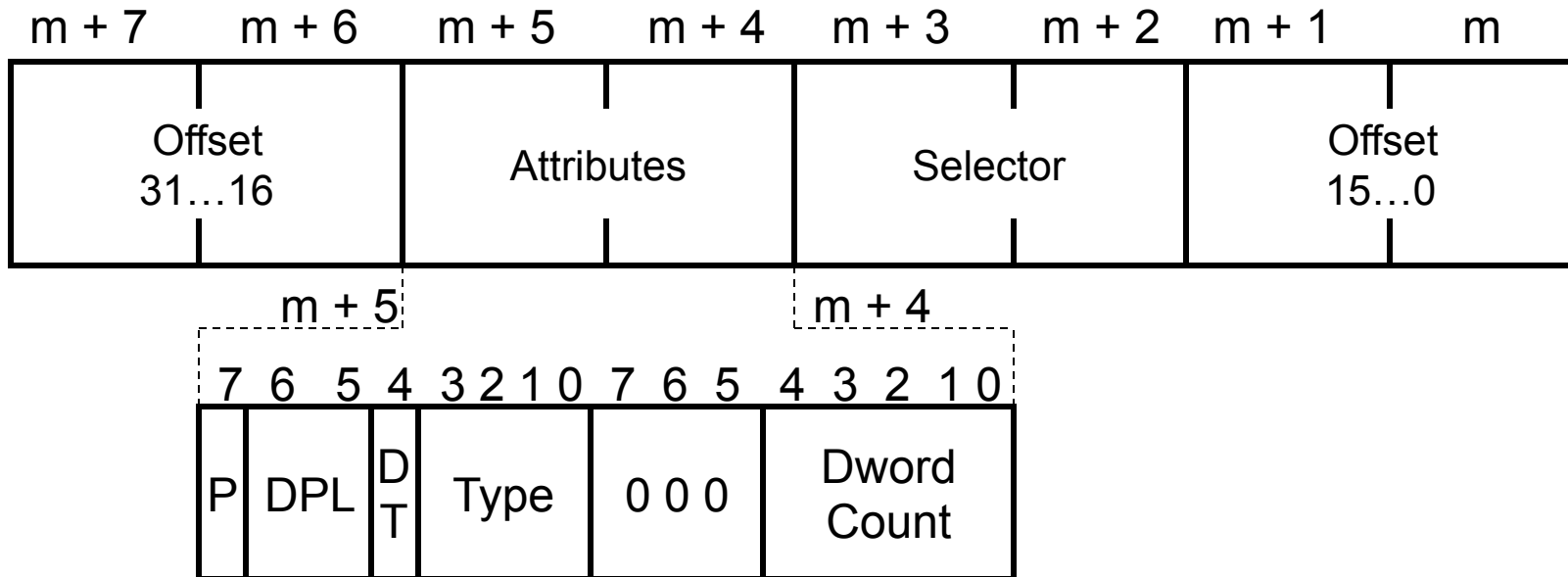
- What does an interrupt look like in software?
- First of all, interrupts are only handled between two assembly instructions. I.e. interrupt handling is deferred until the current instruction has finished executing
- As long as interrupts are enabled ( $IF == 1$ ), the currently running program can be interrupted at any time
- An interrupt basically causes an inter-segment subroutine call to the ISR
- Since every interrupt can be handled by its own ISR, the x86 needs to know the entry point of the ISR
- This is done via the Interrupt Descriptor Table (IDT)

# Interrupt Descriptor Table



For every one of the 256 interrupts, the IDT defines Selector, Attributes and Offset

# Details of one IDT entry



- In TOS, the attributes are initialized as follows:
  - P = 1
  - DPL = 0
  - DT = 0
  - Type = 0xE (x86 Interrupt Gate)
  - Dword Count = 0
  - Selector = 8 (this is the segment selector for the code segment in TOS)

# C Definition for IDT Entry

- Defined in `~/tos/  
include/kernel.h`
- Makes use of bitfields in C
- `sizeof (IDT) == 8`
- TOS needs to declare an array with 256 elements of struct IDT

```
typedef struct
{
    unsigned short offset_0_15;
    unsigned short selector;
    unsigned short dword_count : 5;
    unsigned short unused      : 3;
    unsigned short type        : 4;
    unsigned short dt          : 1;
    unsigned short dpl         : 2;
    unsigned short p           : 1;
    unsigned short offset_16_31;
} IDT;
```

# Building the IDT

- Just like the GDT, the IDT is defined in main memory
- The IDT is an array with 256 elements of struct IDT
- The x86 needs to know where the IDT is stored in memory
- Just like with the GDT, there is a special x86 register that tells the CPU where the IDT is located
- TOS provides the function `load_idt()` that is doing this
- This function makes use of the `lidt` instruction
- This function needs to be called from `init_interrupts()`

# Notes on the ISR

- The ISR is a regular C-function that should not have input or return parameters
- The ISR can not have local variables
- All registers have to be pushed onto the stack as the very first thing
- Before popping the registers off the stack, the ISR needs to reset the interrupt controller via:

```
movb    $0x20,%al
outb    %al,$0x20
```

- The ISR needs to be exited via the assembly instruction IRET (interrupt return)
- IRET pops off `%EIP`, `%CS` and the EFLAGS (this will return to the location where the interrupt interrupted the currently running program)

# Template of for an ISR

```
void dummy_isr ()
{
    asm ("isr:");
    asm ("push %eax; push %ecx; push %edx");
    asm ("push %ebx; push %ebp; push %esi; push %edi");

    /* react to the interrupt */

    asm ("movb $0x20,%al");
    asm ("outb %al,$0x20");
    asm ("pop %edi; pop %esi; pop %ebp; pop %ebx");
    asm ("pop %edx; pop %ecx; pop %eax");
    asm ("iret");
}
```



# Initializing the IDT

- All 256 entries of the IDT need to be initialized in `init_interrupts()`
- This initialization happens via `init_idt_entry()` that initializes one IDT entry
- By default, all 256 IDT entries point to some default ISR:
  - For interrupts 0 to 15 (i.e., NMIs) it points to an ISR that prints an error message and then enters an endless loop (i.e., this ISR never returns and thereby stopping the system)
  - For interrupts 16 to 255 it points to an ISR that does nothing (basically it does only what was shown earlier for a template of ISR)
- Later we will augment the initialization process once we have written the ISR, for the timer and COM1

# Interrupt handling in TOS

- `void init_idt_entry (int intr_no, void (*isr) (void))`  
Initialize the IDT entry for interrupt number `intr_no`. The only other argument is a function pointer to the ISR.
- `void init_interrupts()`  
Initialize the interrupt subsystem of TOS the way explained on an earlier slide. When the initialization is completed, it sets the global variable `interrupts_initialized` to true. As the last instruction, `init_interrupts()` enables the interrupts by executing the assembly instruction `sti`.



# Assignment 6

- Implement the functions located in `kernel/intr.c`:
  - `init_idt_entry()`
  - `init_interrupts()`
  - (interrupt handlers as described before)
- Test case:
  - `test_isr_1`
- Note: For the test case it is beneficial to see the behavior of the reference implementation by typing:
  - `make run_ref`