LUNAIX

Operating System

从零开始

DEVELOPING YOUR OWN

自制操作系统

OPERATING SYSTEM FROM SCRATCH

O Brave New Kernel

Dear Princess Luna, my apology for the *Interruption*

OPERATING SYSTEM DEVELOPMENT TUTORIAL SERIES

EP 6-1

本节内容基于：
Intel® 64 and IA-32 Architectures Software Developer's Manual,
Vol. 3, Chapter 6 – *Interrupt and Exception Handling.*

中断 （ Interruption / Interrupts ）

CPU 很忙！可是有些事情却十万火急……

中断，标示着某些事件的发生，需要 CPU 去救场。

但 CPU 也可以选择无视……

中断

- 非屏蔽中断 (**N**on-**M**askable Interrupts)
- 可屏蔽中断 (**M**askable Interrupts)

真正严重的事情，
这基本上都是硬件错误
能够让 CPU **立刻**转移注意

中断 （ Interruption ）

怎么能让 CPU 忽略 (mask) 中断呢?

POPF, POPFD, or IRET.

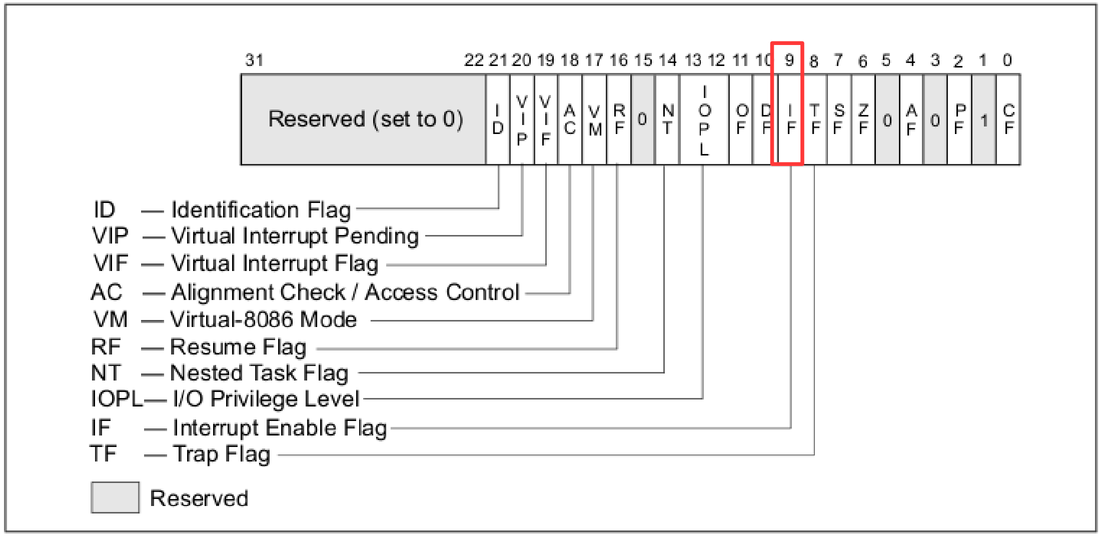


Figure 2-5. System Flags in the EFLAGS Register

IF **Interrupt enable (bit 9)** — Controls the response of the processor to maskable hardware interrupt requests (see also: Section 6.3.2, "Maskable Hardware Interrupts"). The flag is set to respond to maskable hardware interrupts; cleared to inhibit maskable hardware interrupts. The IF flag does not affect the generation of exceptions or nonmaskable interrupts (NMI interrupts). The CPL, IOPL, and the state of the VME flag in control register CR4 determine whether the IF flag can be modified by the CLI, STI, POPF, POPFD, and IRET.

通过两个指令来开关

`cli`

CLear Interrupt Flag

`sti`

SeT Interrupt Flag

一点冷知识：何为 Maskable？

为什么非要叫 Maskable，而不是 Ignorable？

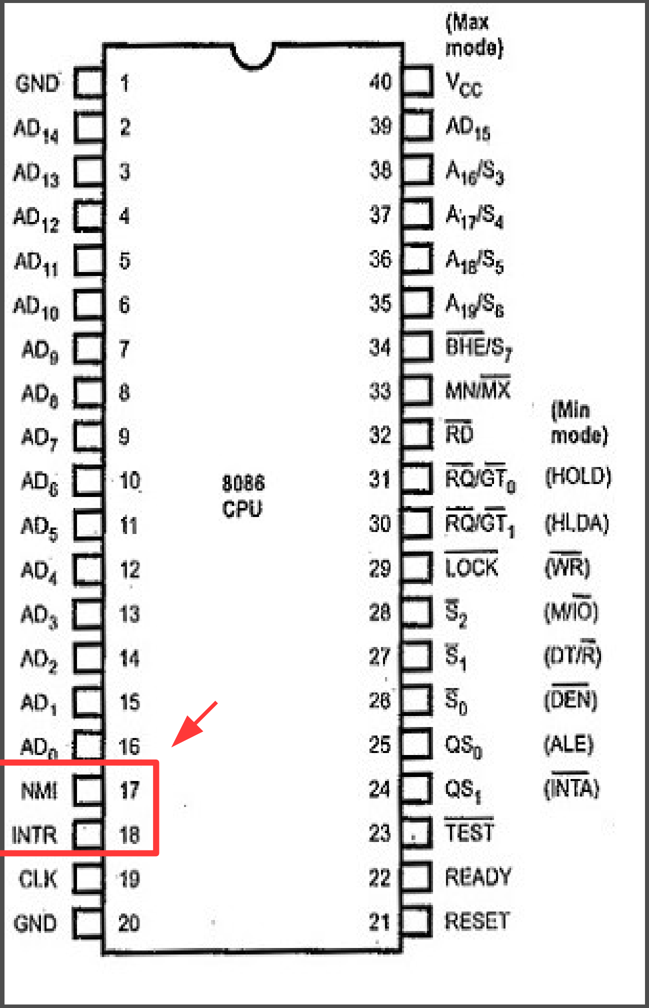
历史原因！（估计现在也一样）

INTR 引脚：若有中断产生，该引脚 ve+

但是 CPU 看到的是： INTR & IF

IF 相当于是一个掩码，将 INTR *mask*（掩）掉

NMI 引脚：若有非屏蔽中断产生，该引脚 ve+。这一引脚由 CPU 直接检测，没有中间人！



那些可以被称之为“中断”？

任何事情！

e.g.，致命错误，键盘 / 鼠标输入， U 盘插入，甚至是某些系统调用……

CPU 可支持 256 个中断

32 个系统级中断……

剩余的 224 个中断可由用户定义

Table 6-1. Protected-Mode Exceptions and Interrupts

Vector	Mnemonic	Description	Type	Error Code	Source
0	#DE	Divide Error	Fault	No	DIV and IDIV instructions.
1	#DB	Debug Exception	Fault/ Trap	No	Instruction, data, and I/O breakpoints; single-step; and others.
2	—	NMI Interrupt	Interrupt	No	Nonmaskable external interrupt.
3	#BP	Breakpoint	Trap	No	INT3 instruction.
4	#OF	Overflow	Trap	No	INTO instruction.
5	#BR	BOUND Range Exceeded	Fault	No	BOUND instruction.
6	#UD	Invalid Opcode (Undefined Opcode)	Fault	No	UD instruction or reserved opcode.
7	#NM	Device Not Available (No Math Coprocessor)	Fault	No	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Abort	Yes (zero)	Any instruction that can generate an exception, an NMI, or an INTR.
9		Coprocessor Segment Overrun (reserved)	Fault	No	Floating-point instruction.[1]
10	#TS	Invalid TSS	Fault	Yes	Task switch or TSS access.
11	#NP	Segment Not Present	Fault	Yes	Loading segment registers or accessing system segments.
12	#SS	Stack-Segment Fault	Fault	Yes	Stack operations and SS register loads.
13	#GP	General Protection	Fault	Yes	Any memory reference and other protection checks.
14	#PF	Page Fault	Fault	Yes	Any memory reference.

CPU 的救场技巧

CPU 遇到中断，他怎么知道如何救场，用什么救？

他不知道，但我们知道。

ISR (Interrupt Service Routines)
中断服务过程

IDT (Interrupt Descriptor Table)
中断描述表

我们为每个中断写一个处理函数，编成一张表，然后告诉 CPU：“您瞧我的吧。多咱的遇到中断，您就查我这个表，照着里头的步骤去做。准保稳妥！”

IDT & ISR

ISR：没什么特别的，就是某个由 CPU 直接调用的函数（比如， C 函数）

IDT：和 GDT 差不多结构，一个描述符数组，每个描述符包含了指向 ISR 的指针以及一些附加信息。

描述符的索引代表中断向量号

Vector	Mnemonic	Description
0	#DE	Divide Error
1	#DB	Debug Exception
2	—	NMI Interrupt
3	#BP	Breakpoint
4	#OF	Overflow
5	#BR	BOUND Range Exceeded
6	#UD	Invalid Opcode (Undefined Opcode)
7	#NM	Device Not Available (No Math Coprocessor)
8	#DF	Double Fault
9		Coprocessor Segment Overrun (reserved)
10	#TS	Invalid TSS
11	#NP	Segment Not Present
12	#SS	Stack-Segment Fault
13	#GP	General Protection
14	#PF	Page Fault

中断描述符的三种形态

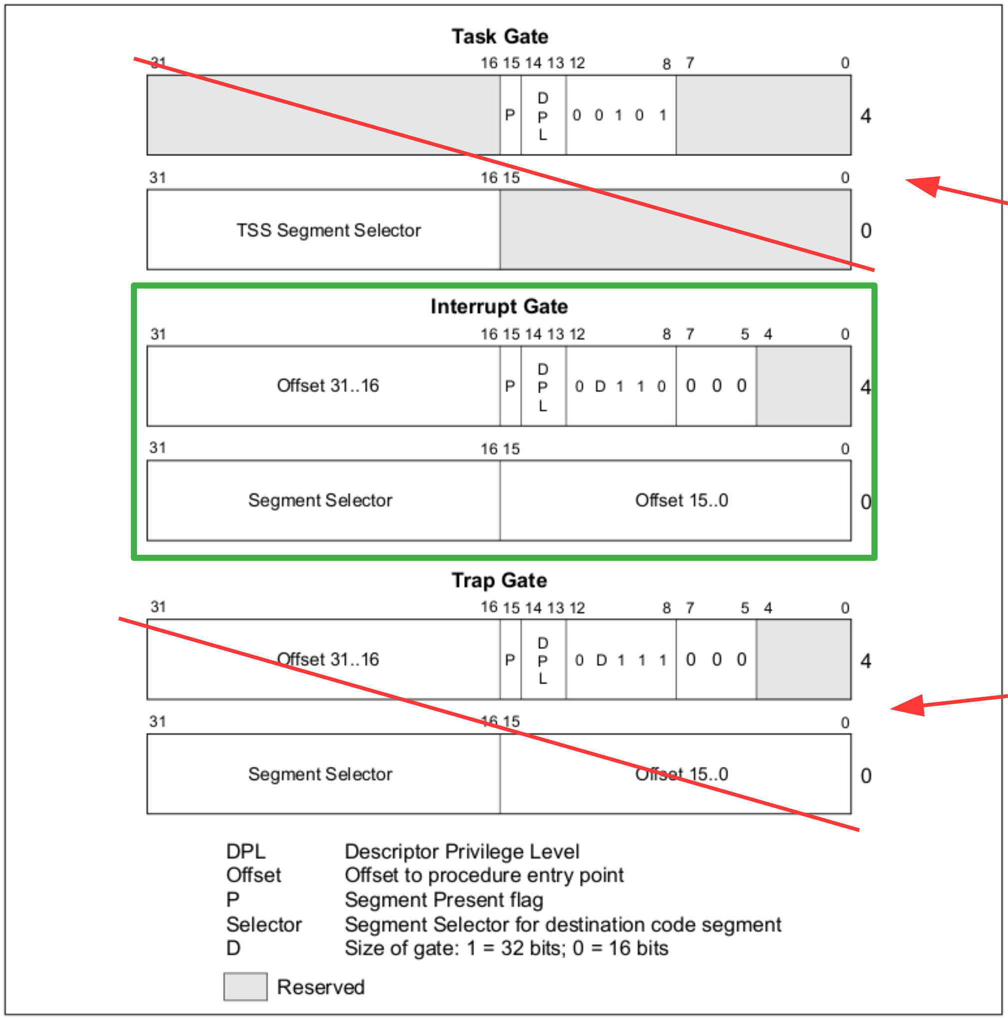


Figure 6-2. IDT Gate Descriptors

由 bits 8-10 决定究竟是哪一种。

暂时不用

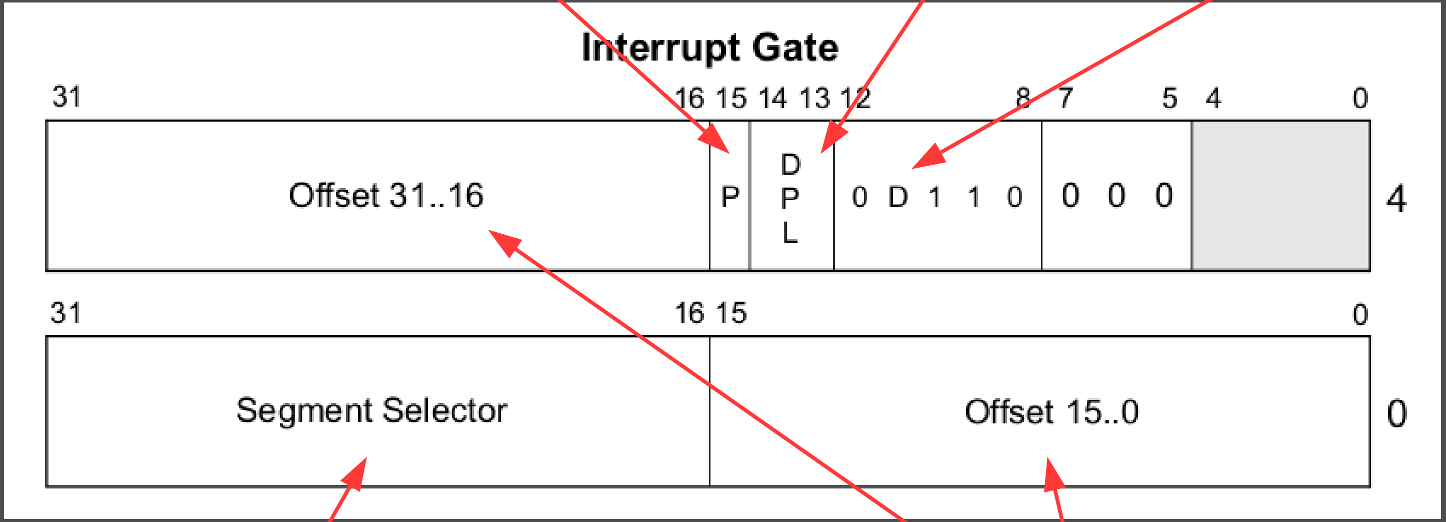
暂时不用

中断描述符之 Interrupt Gate

段是否在内存中

ISR 的所在 Ring

ISR 的位数
（D=1， 32bits）



ISR 所在的段的段描述子

ISR 所的段内偏移

CPU 通过查表 IDT 寻找到 ISR

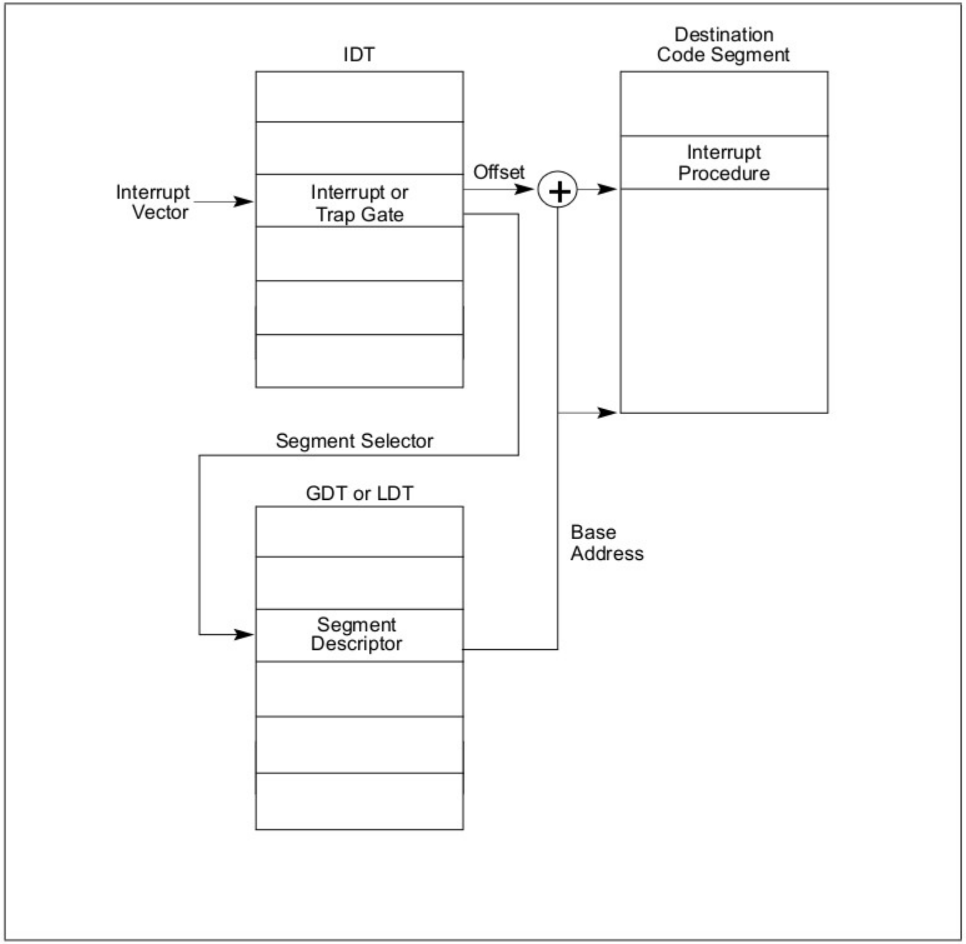


Figure 6-3. Interrupt Procedure Call

安装 IDT

指令： `lidt`

和 `lgdt` 一样，需要从内存读取 IDTR 的值

Let’s Code

关于 ISR 的参数

ISR 有参数！由 CPU 传入。

根据 cdecl，参数顺序是：

1. ErrorCode
2. EIP
3. CS
4. EFLAGS

ErrorCode != Vector No.

注意： ErrorCode 是存在与否取决于当前产生的中断。

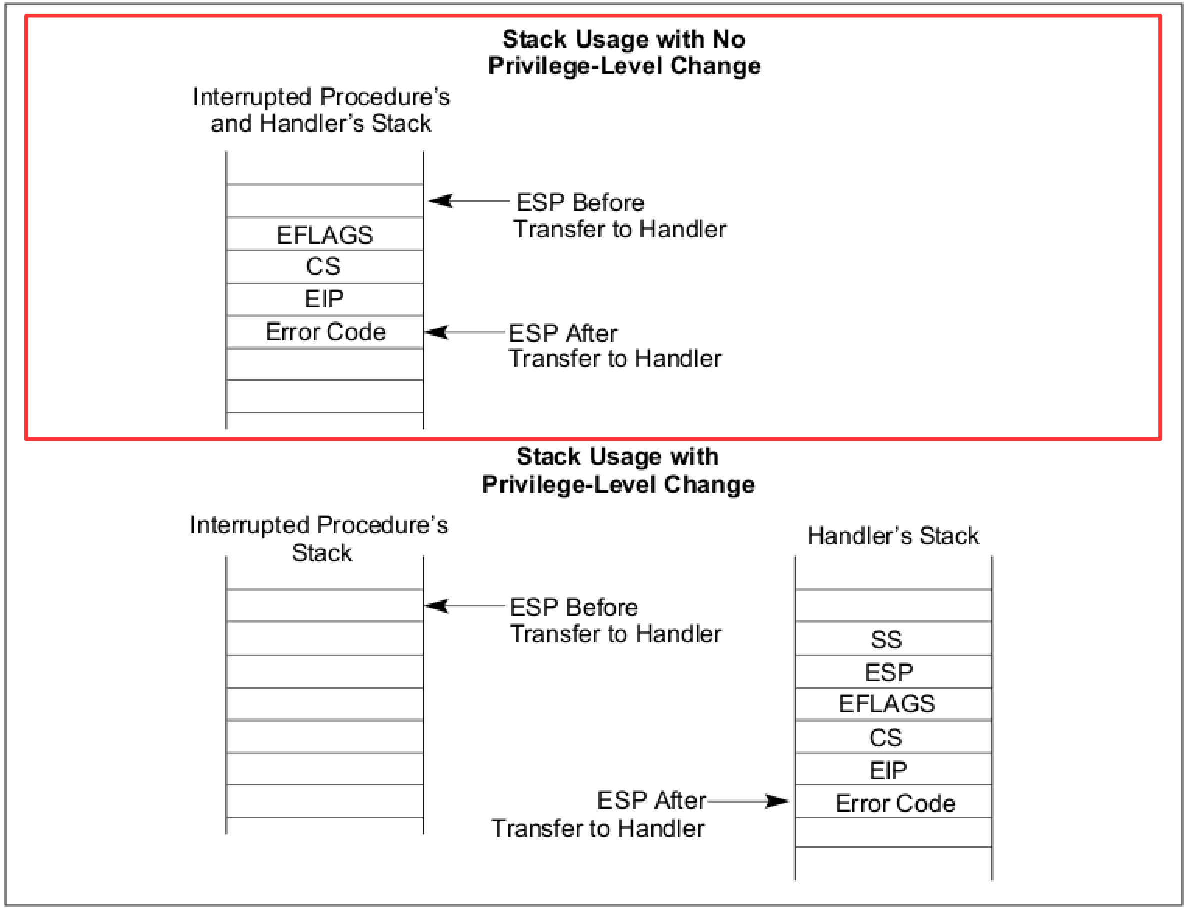


Figure 6-4. Stack Usage on Transfers to Interrupt and Exception-Handling Routines

ISR 不是 C 函数

To return from an exception- or interrupt-handler procedure, the handler must use the IRET (or IRETD) instruction. The IRET instruction is similar to the RET instruction except that it restores the saved flags into the EFLAGS register. The IOPL field of the EFLAGS register is restored only if the CPL is 0. The IF flag is changed only if the CPL is less than or equal to the IOPL. See Chapter 3, "Instruction Set Reference, A-L," of the *Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A*, for a description of the complete operation performed by the IRET instruction.

```c
int main() {
        int a = 21;
        return *(&a + 4);
}
```

i686-elf-gcc -S

```
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    $21, -4(%ebp)
        movl    12(%ebp), %eax
        leave
        ret
```

ISR 需要以 IRET 结尾!

IRET vs. RET ?

```
IF OperandSize = 32
    THEN
        EIP := Pop();
        CS := Pop(); (* 32-bit pop, high-order 16 bits discarded *)
        tempEFLAGS := Pop();
```

IRET

```
IF OperandSize = 32
    THEN
        IF top 4 bytes of stack not within stack limits
            THEN #SS(0); FI;
        EIP := Pop();
```

RET

需要栈顶元素依次为 IP 与 CS
同时修改 CS， IP
恢复 EFLAGS

需要栈顶为 IP
只修改 IP！
CPU 无法恢复断点发生时的状态！

iret 假定 error code 不存在！

Hate on Intel +1 :-(

耍耍小聪明 - 内联汇编注入 iret

```c
void my_isr(int myargs) {
        // Tell CPU what to do with this interrupt ...
        int k = myargs *42;

        __asm__("iret"); // inject iret!
}
```

```
i686-elf-gcc -O0 -S example.c \
 -fomit-frame-pointer
```

```
my_isr:
        subl    $16, %esp
        movl    20(%esp), %eax
        imull   $42, %eax, %eax
        movl    %eax, 12(%esp)

        iret
        nop
        addl    $16, %esp
        ret
```

不可取

分配栈空间给局部变量

假设这里是我们的中断处理代码。

iret 返回！
可是……我们的栈里又是什么呢？

未释放的局部变量空间
内存泄露！

凡事都有例外……

```c
void my_isr(int myargs) {
        // Tell CPU what to do with this interrupt ...
        int k = myargs *42;

        __asm__(
                "leave\n"
                "iret"
        ); // inject iret!
}
```

i686-elf-gcc -O0 -S example.c

注意：我们拿掉了 -fomit-frame-pointer

```
my_isr:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    8(%ebp), %eax
        imull   $42, %eax, %eax
        movl    %eax, -4(%ebp)

        leave
        iret

        nop
        leave
        ret
```

不可取

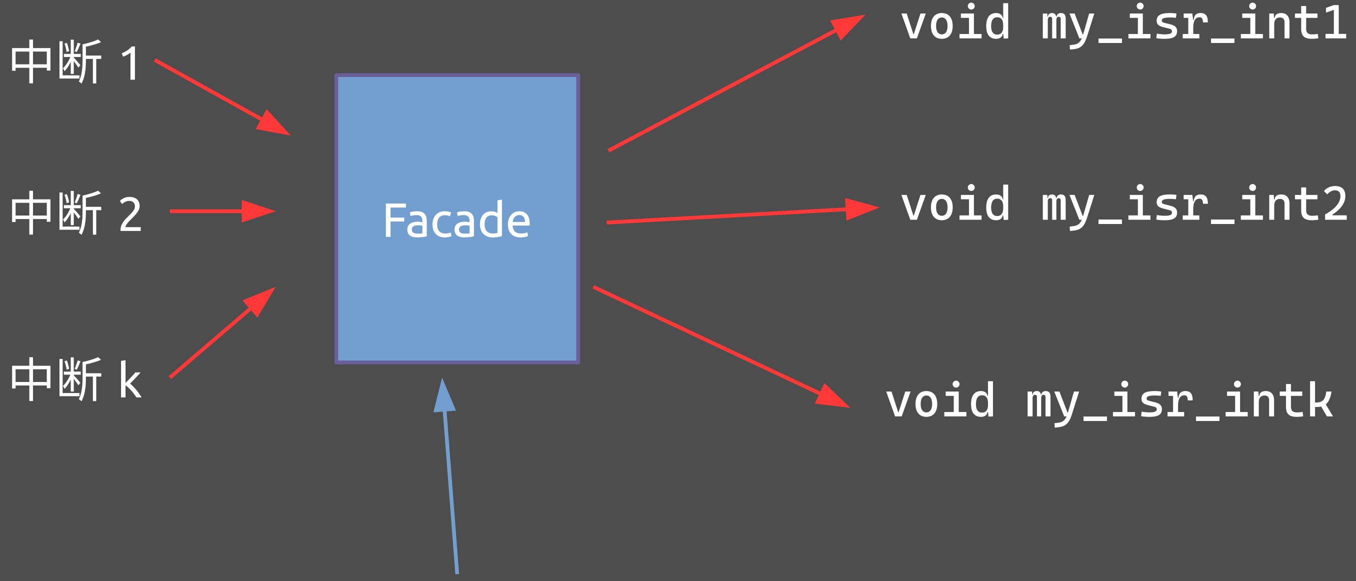
```
IF StackAddressSize = 32
    THEN
        ESP := EBP;
    ELSE IF StackAddressSize = 64
        THEN RSP := RBP; FI;
    ELSE IF StackAddressSize = 16
        THEN SP := BP; FI;
FI;

IF OperandSize = 32
    THEN EBP := Pop();
    ELSE IF OperandSize = 64
        THEN RBP := Pop(); FI;
    ELSE IF OperandSize = 16
        THEN BP := Pop(); FI;
FI;
```

LEAVE 是关键

释放了局部变量空间
恢复了 %ebp
iret 可以正确返回！
但假设没有 error code 的存在

设计模式来救场 - 门面模式（Facade）



处理 ErrorCode 的 corner case

分发到对应的 isr

正确 iret 调用

Let’s Code

为什么我们的中断时好时坏?

```
return 1 / 0;
```

错误信息没有显示

```
__asm__("int $0");
```

错误信息正常显示

中断也有类型之分

Fault

指令出错了，但救一下还可以试试。

Trap

指令出错了，没法救，只能跳过。

Abort

爆炸！

异常
Exception $\subset$ 中断
Interrupt

ISR - Trap 与 Fault 的区别

	Trap Gate	Interrupt Gate
行为	被 CPU 调用	被 CPU 调用
EIP	指向下一个指令	不变
EFLAGS	IF 不变 (sti)	IF = 0 (cli)

变更发生在原先 EFLAGS 入栈后

使用 int 产生的中断

int 指令一律产生 Trap 类型的中断（也叫做软件中断）

不管你指明的中断号的实际类型

`int a = 1 /0;` 是一个真正的中断（硬件中断），类型遵循其实际定义

Table 6-1. Protected-Mode Exceptions and Interrupts

Vector	Mnemonic	Description	Type	Error Code	Source
0	#DE	Divide Error	Fault	No	DIV and IDIV instructions.
1	#DB	Debug Exception	Fault/ Trap	No	Instruction, data, and I/O breakpoints; single-step; and others.

iret 返回，并尝试重新执行错误指令，产生错误，调用 ISR，清屏打印错误信息， iret 返回……→ 死循环！

Well, it's a tough topic...

下期预告：分页与虚拟内存

时间： 2.27 ± 7 days