

计算机系统基础  
Programming Assignment

## PA 4 异常、中断与I/O

### ——PA 4-1 异常和中断的响应

2020年12月24日 / 12月25日

南京大学《计算机系统基础》课程组

# 前情提要

- PA 1 - 实现了基本的运算单元
- PA 2 - 实现了各种指令和程序的装载，NEMU（几乎）等价于图灵机（除了没有无限长的纸带）
- PA 3 - 引入了保护机制，引入分页机制打破物理内存限制，多个进程互不干扰

能力上讲：能够同时执行多项复杂的科学计算任务，但是不能处理任何异常情况，也无法和外面的世界产生任何互动

# PA 4-1要解决的问题

- 异常和中断的响应（以做一道蛋炒饭为类比）
  - 正常的控制流
    - 热锅、添油、炒蛋、炒饭、加佐料……
    - 按照菜谱规定步骤顺序执行或跳转
  - 异常的控制流
    - 内部异常
      - 正要炒饭，发现饭没有煮熟！这饭没法炒了！
      - 和正在执行的指令有关的同步事件（执行到那一步才会出错）
    - 外部中断
      - 突然厨房外面一声大吼：“不要辣的！”
      - 和正在执行的指令无关的异步事件（不知何时到来）
  - 问题：CPU和操作系统如何配合响应内部和外部的异常？



# PA 4-1 异常和中断的响应

# 异常和中断

- 内部异常：在执行一条指令时，由处理器在其内部检测到的，与正在执行的指令相关的同步事件
  - 故障：缺页、非法操作码、除数为零.....
  - 陷阱：用户程序主动调用操作系统处理例程
  - 终止：执行指令时发生严重错误，如内存校验错误
- 外部中断：典型地由I/O设备触发，与当前正在执行的指令无关的异步事件

# 异常和中断的响应

宏观  
步骤  
1/2

在能够响应中断之前，  
操作系统先准备好IDT

- 操作系统先和机器打好招呼
  - 初始化中断描述符表（我们不模拟实模式的向量中断）



如果出现了n号异常，你  
就去执行内存地址x开始，  
我准备好的处理程序啊~

异常/中断号	处理程序入口地址
N1	XXXX 
N2	YYYY 
...	...
Nn	ZZZZ 

哦



# 异常和中断的响应

宏观  
步骤  
2/2

在操作系统准备好IDT之后，机器就能够响应中断了

- 响应过程

- CPU检测到异常或中断后，根据异常和中断号去查表得到处理程序的入口地址，处理完后返回原程序继续

1. 检测到异常或中断!



# 异常和中断的响应

宏观  
步骤  
2/2

在操作系统准备好IDT之后，机器就能够响应中断了

- 响应过程

- CPU检测到异常或中断后，根据异常和中断号去查表得到处理程序的入口地址，处理完后返回原程序继续





# 异常和中断的响应

宏观  
步骤  
2/2

在操作系统准备好IDT之后，机器就能够响应中断了

- 响应过程

- CPU检测到异常或中断后，根据异常和中断号去查表得到处理程序的入口地址，处理完后返回原程序继续



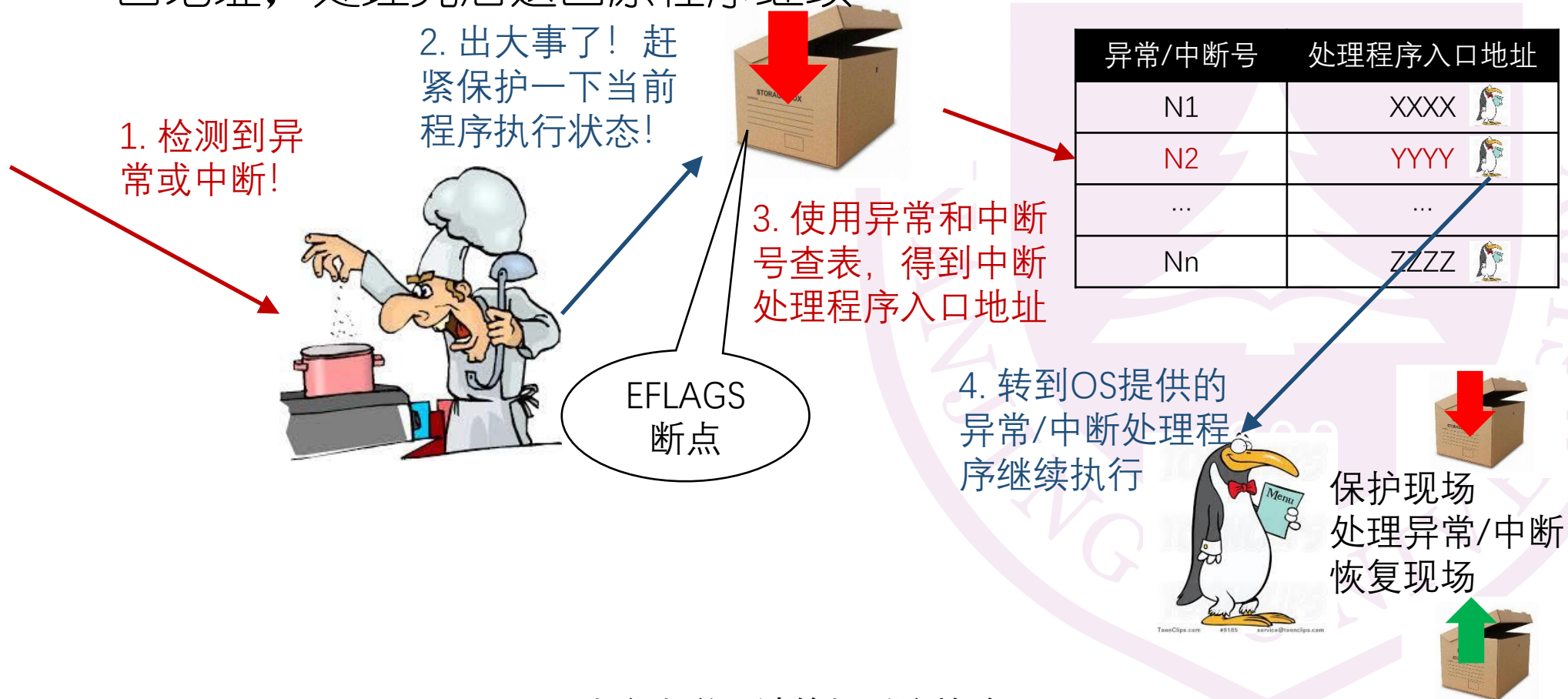
# 异常和中断的响应

宏观  
步骤  
2/2

在操作系统准备好IDT之后，机器就能够响应中断了

## • 响应过程

- CPU检测到异常或中断后，根据异常和中断号去查表得到处理程序的入口地址，处理完后返回原程序继续



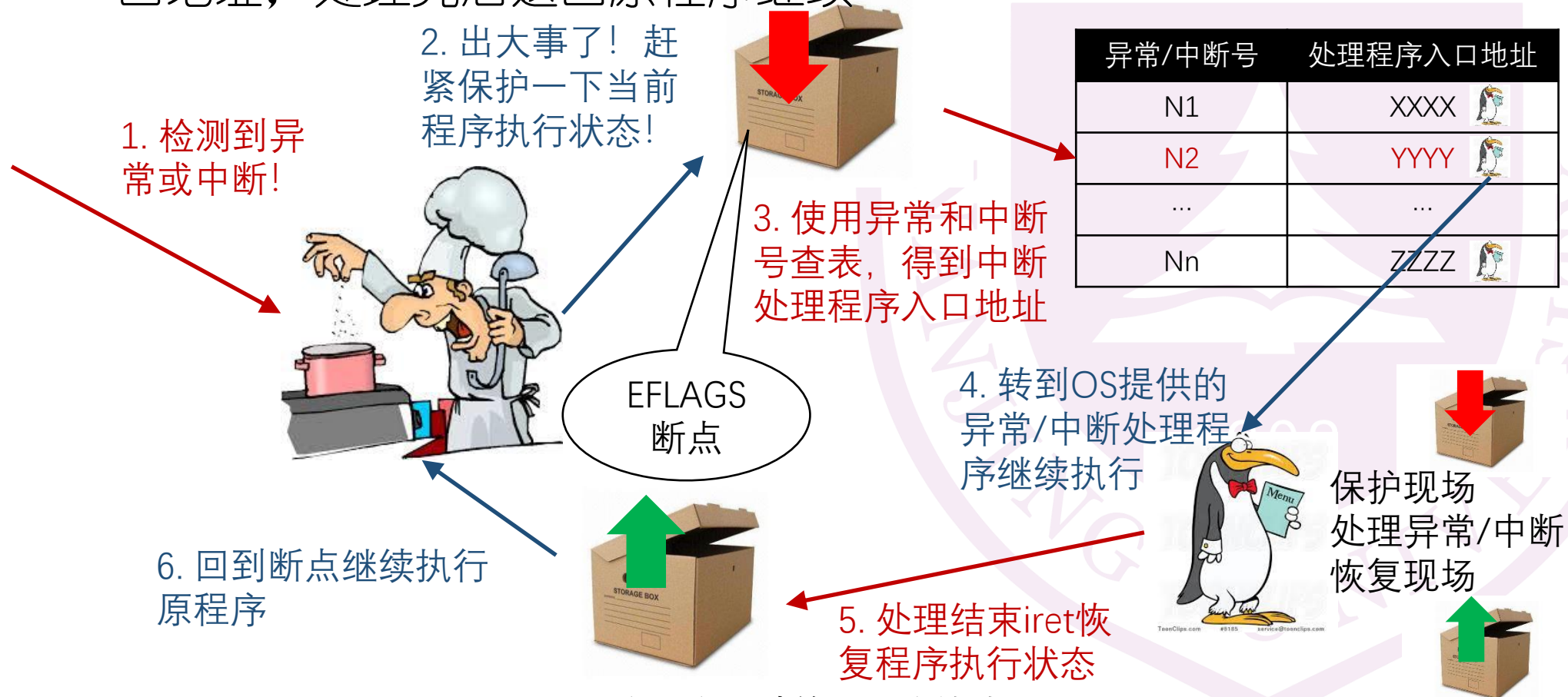
# 异常和中断的响应

宏观  
步骤  
2/2

在操作系统准备好IDT之后，机器就能够响应中断了

## • 响应过程

- CPU检测到异常或中断后，根据异常和中断号去查表得到处理程序的入口地址，处理完后返回原程序继续



# 异常和中断的响应

- 响应过程各个步骤 – 检测异常或中断的到来

1. 检测到异常或中断!



- 内部异常（陷阱 - 系统调用，其它情况不模拟）

int指令触发

- 外部中断（典型由I/O设备触发）

CPU中断引脚

每执行完一条指令查看一次

# NEMU中实现中断响应

- 在NEMU中实现对中断的响应
  - 在`nemu/src/cpu/cpu.c`中`do_intr()`函数

```
void exec(uint32_t n)
{
    while (n > 0 && nemu_state == NEMU_RUN)
    {
        if(!is_nemu_hlt)
        {
            instr_len = exec_inst();
            cpu.eip += instr_len;
            n--;
        }
        #if defined(HAS_DEVICE_TIMER) || defined(HAS_DEVICE_VGA) || defined(HAS_DEVICE_KEYBOARD) || defined(HAS_DEVICE_AUDIO)
            do_devices();
        #endif
        #ifdef IA32_INTR
            do_intr(); // check for interrupt
        #endif
    }
}
```

# 异常和中断的响应

- 响应过程各个步骤 – 检测异常或中断的到来 并获得

1. 检测到异常或中断!



## 异常和中断号

- 内部异常（陷阱 - 系统调用，其它情况不模拟）

int 0x80

- 外部中断（典型由I/O设备触发）

中断控制器  
(i8259) 提供



# NEMU中实现中断响应

- 在NEMU中实现对中断的响应
  - 在nemu/src/cpu/cpu.c中do\_intr()函数

```
void exec(uint32_t n)
{
    while (n > 0 && nemu_state == NEMU_RUN)
    {
        if(!is_nemu_hlt)
        {
            instr_len = exec_inst();
            cpu.eip += instr_len;
            n--;
        }
        #if defined(HAS_DEVICE_TIMER) ...
            do_devices();
        #endif
        #ifdef IA32_INTR
            do_intr();
        #endif
    }
}
```

```
#ifdef IA32_INTR
void do_intr()
{
    if (cpu.intr && cpu.eflags.IF)
    {
        // get interrupt number
        uint8_t intr_no = i8259_query_intr_no();
        assert(intr_no != I8259_NO_INTR);
        // tell the PIC interrupt info received
        i8259_ack_intr();
        // raise interrupt
        raise_intr(intr_no);
    }
}
#endif
```

# 异常和中断的响应

- 响应过程各个步骤 - 第一阶段保护程序状态

2. 出大事了！赶紧保护一下当前程序执行状态！



EFLAGS  
断点

第一阶段保护（硬件完成）：  
依次将EFLAGS, CS, EIP寄存器的值压栈



# 异常和中断的响应

- 响应过程各个步骤 - 根据异常或中断号查询IDT

## 异常和中断号

- 内部异常（陷阱 - 系统调用，其它情况不模拟）

int 0x80

- 外部中断（典型由I/O设备触发）

中断控制器  
(i8259) 提供

3. 使用异常和中断号查表，得到中断处理程序入口地址

异常/中断号	处理程序入口地址
N1	XXXX 
N2	YYYY 
...	...
Nn	ZZZZ 

# NEMU中实现中断响应

- 在NEMU中实现对中断的响应
  - 在`nemu/src/cpu/cpu.c`中`do_intr()`函数

```
void exec(uint32_t n)
{
    while (n > 0 && nemu_state == NEMU_RUN)
    {
        if(!is_nemu_hlt)
        {
            instr_len = exec_inst();
            cpu.eip += instr_len;
            n--;
        }
        #if defined(HAS_DEVICE_TIMER) ...
            do_devices();
        #endif
        #ifdef IA32_INTR
            do_intr();
        #endif
    }
}
```

```
#ifdef IA32_INTR
void do_intr()
{
    if (cpu.intr && cpu.eflags.IF)
    {
        // get interrupt number
        uint8_t intr_no = i8259_query_intr_no();
        assert(intr_no != I8259_NO_INTR);
        // tell the PIC interrupt info received
        i8259_ack_intr();
        // raise interrupt
        raise_intr(intr_no);
    }
}
#endif
```

```
void raise_intr(uint8_t intr_no) {  
#ifdef IA32_INTR  
    - printf("Please implement raise_intr()");  
    - assert(0);  
    + // Trigger an exception/interrupt with 'intr_no'  
    + // 'intr_no' is the index to the IDT  
  
    + // Push EFLAGS, CS, and EIP  
    + // Find the IDT entry using 'intr_no'  
    + // Clear IF if it is an interrupt  
    + // Set CS:EIP to the entry of the interrupt handler, need to reload CS with load_sreg()  
#endif  
}
```

# 异常和中断的响应

- 响应过程各个步骤 – 使用异常或中断号查中断描述符表

中断描述符表 (IDT)

门描述符的数组

OFFSET 31..16	P   DPL   0 1 1 1   0 0 0   (NOT USED)	4
SELECTOR	OFFSET 15..0	0
OFFSET 31..16	P   DPL   0 1 1 1   0 0 0   (NOT USED)	4
SELECTOR	OFFSET 15..0	0
OFFSET 31..16	P   DPL   0 1 1 1   0 0 0   (NOT USED)	4
SELECTOR	OFFSET 15..0	0

异常/中断号	处理程序入口地址
N1	XXXX 
N2	YYYY 
...	...
Nn	ZZZZ 

3. 使用异常和中断号查表，得到中断处理程序入口地址

首地址存储在idtr寄存器中，由lidt指令负责装入

# 异常和中断的响应

- 响应过程各个步骤 - 中断描述符表

中断描述符表 (IDT)

异常/中断号	处理程序入口地址
N1	XXXX 
N2	YYYY 
...	...
Nn	ZZZZ 

3. 使用异常和中断号查表, 得到中断处理程序入口地址

64位门描述符(Gate Descriptor)

- 中断门(Interrupt Gate)
- 陷阱门(Trap Gate)
- ~~任务门(Task Gate)~~

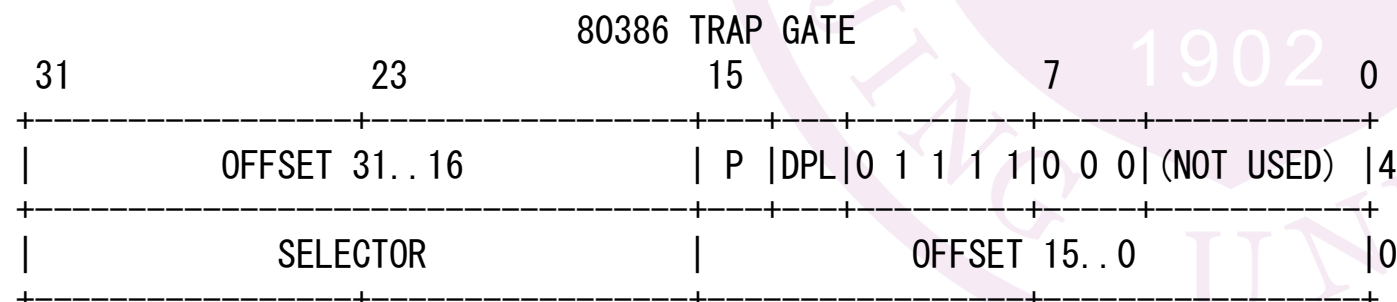
门描述符的数组

OFFSET 31..16	P   DPL   0 1 1 1   0 0 0   (NOT USED)	4
SELECTOR	OFFSET 15..0	0

OFFSET 31..16	P   DPL   0 1 1 1   0 0 0   (NOT USED)	4
SELECTOR	OFFSET 15..0	0

OFFSET 31..16	P   DPL   0 1 1 1   0 0 0   (NOT USED)	4
SELECTOR	OFFSET 15..0	0

首地址存储在idtr寄存器中, 由lidt指令负责装入



# 异常和中断的响应

- 响应过程各个步骤 - 中断描述符表

中断描述符表 (IDT)

异常/中断号	处理程序入口地址
N1	XXXX 
N2	YYYY 
...	...
Nn	ZZZZ 

3. 使用异常和中断号查表, 得到中断处理程序入口地址

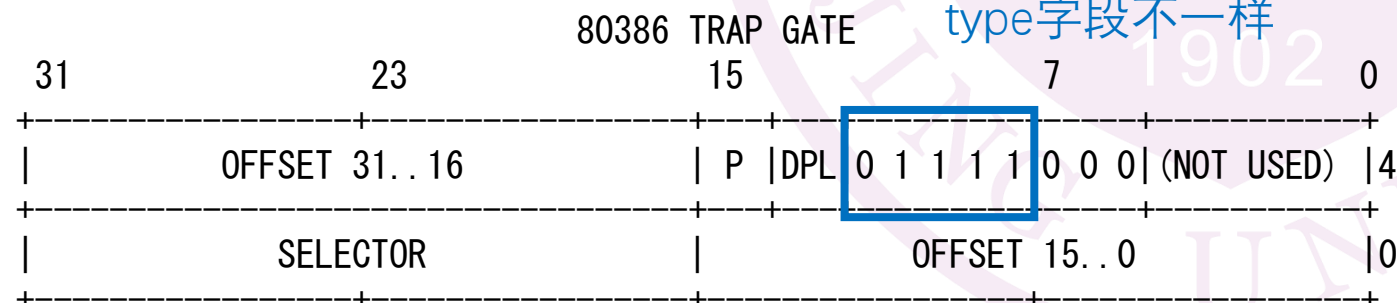
中断门和陷阱门就是type字段不一样

## 64位门描述符(Gate Descriptor)

- 中断门(Interrupt Gate)
- 陷阱门(Trap Gate)
- ~~任务门(Task Gate)~~

门描述符的数组

OFFSET 31..16	P   DPL   0 1 1 1   0 0 0   (NOT USED)	4
SELECTOR	OFFSET 15..0	16



首地址存储在idtr寄存器中, 由lidt指令负责装入

# 异常和中断的响应

- 响应过程各个步骤

## 64位门描述符(Gate Descriptor)

- 中断门(Interrupt Gate)
- 陷阱门(Trap Gate)
- ~~任务门(Task Gate)~~

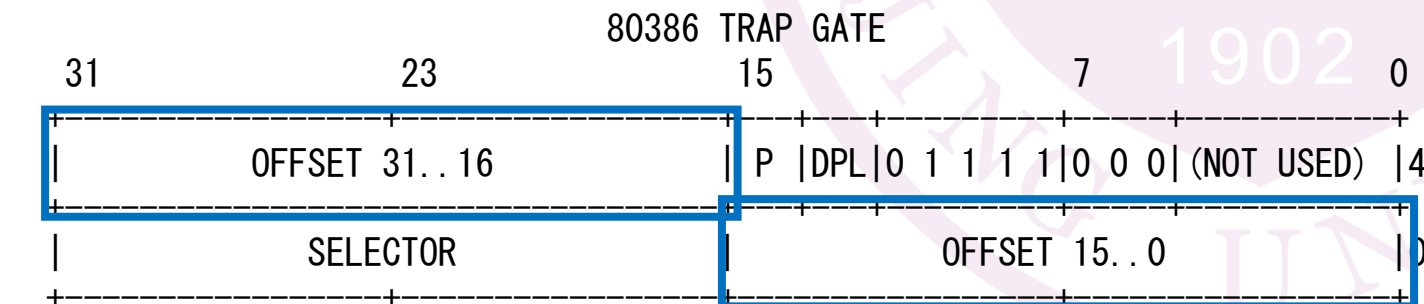
门描述符的数组

OFFSET 31..16	P   DPL   0 1 1 1   0 0 0   (NOT USED)	4
SELECTOR	OFFSET 15..0	10

OFFSET 31..16	P   DPL   0 1 1 1   0 0 0   (NOT USED)	4
SELECTOR	OFFSET 15..0	10

OFFSET 31..16	P   DPL   0 1 1 1   0 0 0   (NOT USED)	4
SELECTOR	OFFSET 15..0	10

首地址存储在idtr寄存器中，由lidt指令负责装入



处理程序入口地址对应的段内偏移量

中断描述符表 (IDT)

异常/中断号	处理程序入口地址
N1	XXXX 
N2	YYYY 
...	...
Nn	ZZZZ 

3. 使用异常和中断号查表，得到中断处理程序入口地址

跳转逻辑地址=段选择符+段内偏移量

# 异常和中断的响应

- 响应过程各个步骤 – 跳转到处理程序执行

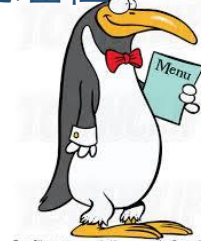
## 跳转前决定是否允许中断嵌套？

- 当处理外部中断时，清除EFLAGS寄存器中的IF位，实现关中断，不允许嵌套
- 当处理内部异常时，不清除EFLAGS寄存器中的IF位，不关闭中断，允许嵌套

中断描述符表 (IDT)

异常/中断号	处理程序入口地址
N1	XXXX 
N2	YYYY 
...	...
Nn	ZZZZ 

4. 转到OS提供的  
异常/中断处理程  
序继续执行



保护现场  
处理异常/中断  
恢复现场

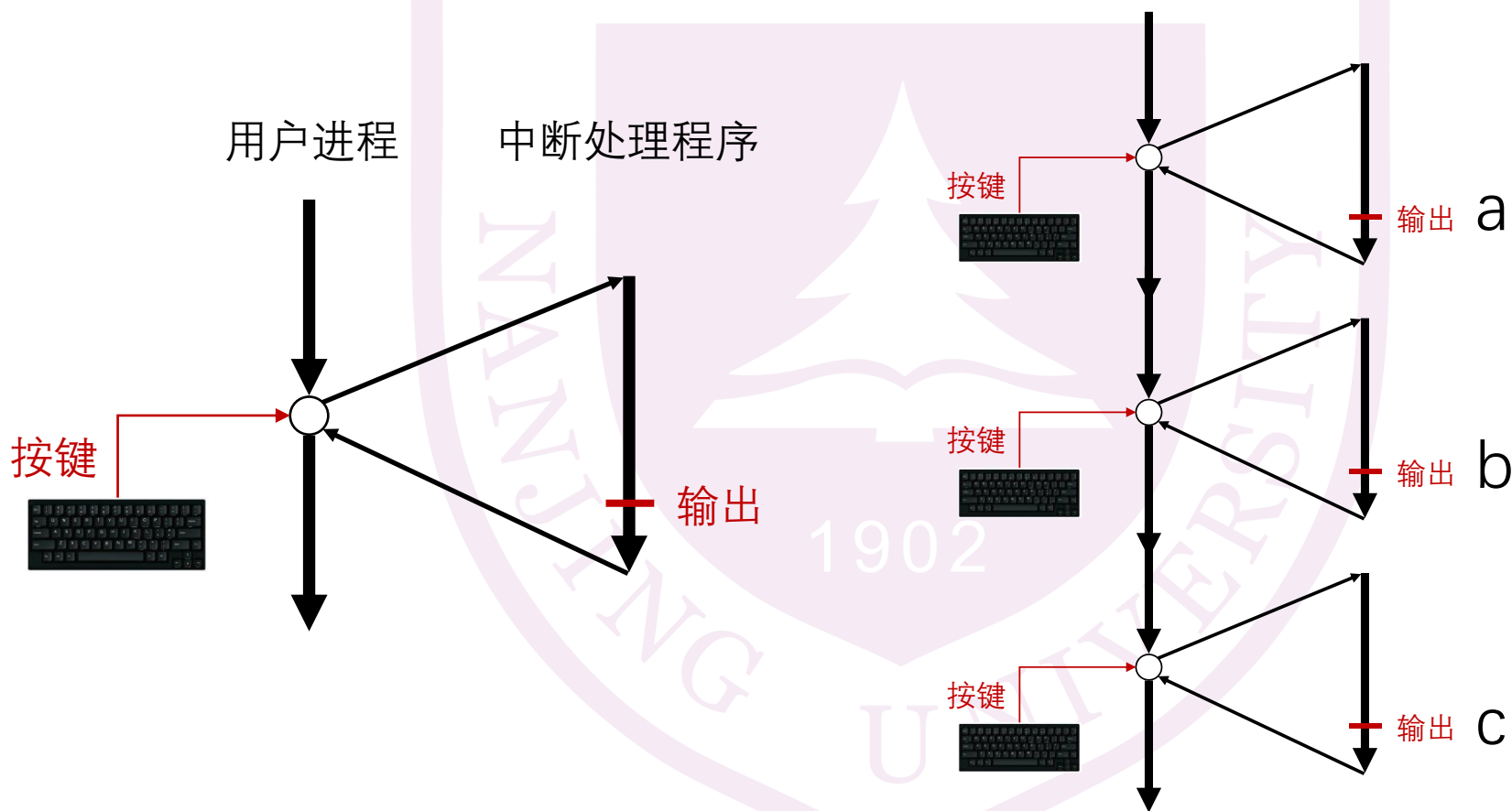




# 小例子

- NEMU约定：当处理外部中断时，清除EFLAGS寄存器中的IF位，实现关中断，不允许嵌套（也不考虑中断的优先级）

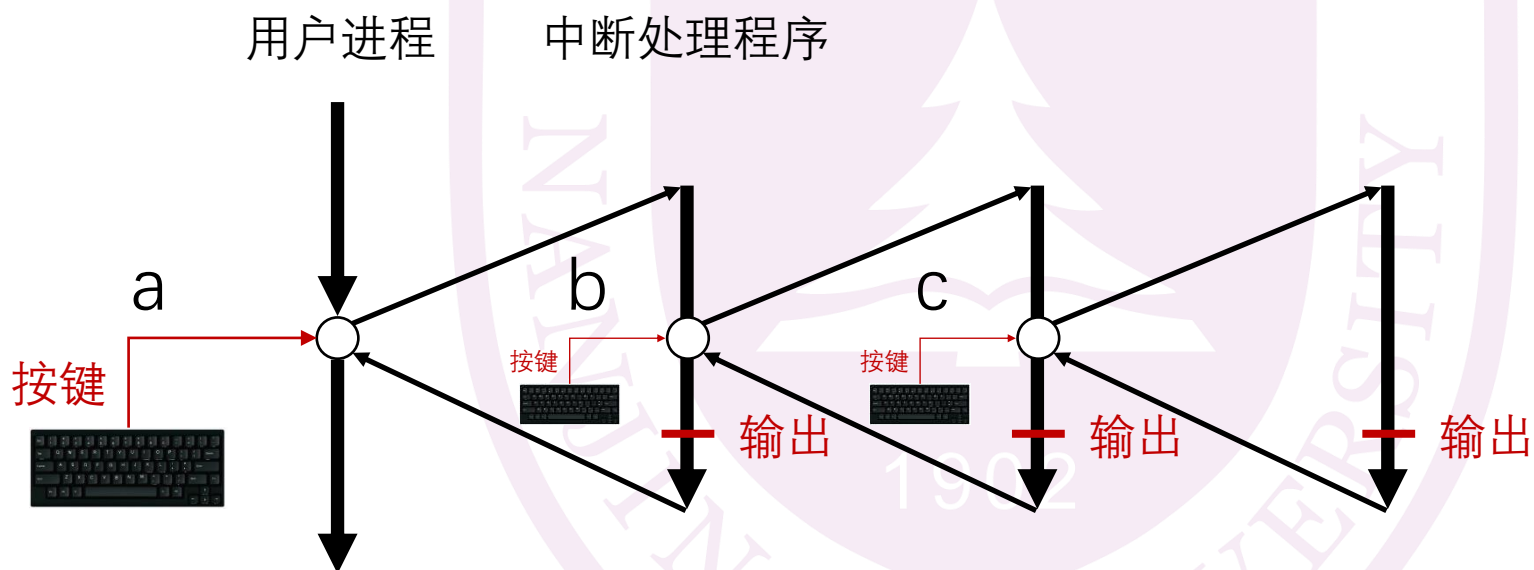
键盘按键： a -> b -> c  
期待输出： a -> b -> c



# 小例子

- NEMU约定：当处理外部中断时，清除EFLAGS寄存器中的IF位，实现关中断，不允许嵌套（也不考虑中断的优先级）

键盘按键： a -> b -> c  
期待输出： a -> b -> c



# 异常和中断的响应

- 响应过程各个步骤 – 跳转到处理程序执行

第二阶段保护程序状态（处理程序软件完成）：

pusha - 将各通用寄存器的值压栈

中断描述符表（IDT）

异常/中断号	处理程序入口地址
N1	XXXX 
N2	YYYY 
...	...
Nn	ZZZZ 

4. 转到OS提供的异常/中断处理程序继续执行



保护现场  
处理异常/中断  
恢复现场



# 异常和中断的响应

- 响应过程各个步骤 – 跳转到处理程序执行

中断描述符表 (IDT)

异常/中断号	处理程序入口地址
N1	XXXX 
N2	YYYY 
...	...
Nn	ZZZZ 

## 处理异常/中断:

- 使用TrapFrame传递参数 (若有)
  - 先把参数按约定放在各通用寄存器中
  - pusha保护现场
  - push %esp ??? 对应代码理解
- 完成对异常/中断的处理

4. 转到OS提供的  
异常/中断处理程  
序继续执行



保护现场  
处理异常/中断  
恢复现场



# 异常和中断的响应

- 响应过程各个步骤
  - 跳转到处理程序执行

恢复现场:

popa - 将各通用寄存器的值弹栈

- 修改TrapFrame中的内容
  - popa的时候将TrapFrame中内容弹栈到各通用寄存器，可以实现返回值功能

中断描述符表 (IDT)

异常/中断号	处理程序入口地址
N1	XXXX 
N2	YYYY 
...	...
Nn	ZZZZ 

4. 转到OS提供的异常/中断处理程序继续执行



保护现场  
处理异常/中断  
恢复现场



# 异常和中断的响应

- 响应过程各个步骤
  - `iret`恢复原程序断点



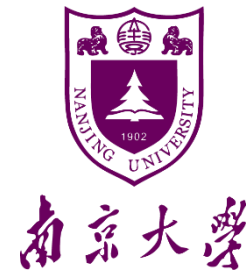
6. 回到断点继续执行  
原程序



5. 处理结束iret恢  
复程序执行状态

`iret`

- 按照第一阶段保护程序状态对应的次序正确弹栈
- 将eip设置为原程序需要恢复执行的位置（断点）



# NEMU中的任务

# NEMU中实现异常（系统调用）响应

- 第一步：更改配置和在nemu中添加必要的硬件/指令
  - `include/config.h`
    - 定义宏 `#define IA32_INTR`
    - `make clean`
  - 在`nemu/include/cpu/reg.h`中定义 `IDTR` 结构体
    - 参照手册
  - 并在 `CPU_STATE`中添加`idtr`寄存器 and 中断引脚（框架代码已经提供）
  - 实现包括`lidt`、`cli`、`sti`、`int`、`pusha`、`popa`、`iret`等指令
    - 部分指令要先写完第三步才能做



# NEMU中实现异常（系统调用）响应

- 第二步：理解Kernel初始化中断描述符表的内容
  - 在init\_cond()中调用了init\_idt()
  - init\_idt()位于kernel/src/irq/idt.c

```
void init_cond()
{
#ifdef IA32_INTR
    /* Reset the GDT, since the old GDT in start.S cannot be used in the future. */
    init_segment();

    /* Set the IDT by setting up interrupt and exception handlers.
     * Note that system call is the only exception implemented in NEMU.
     */
    init_idt();

    /* Enable interrupts. */
    sti();
#endif
}
```

# NEMU中实现异常（系统调用）响应

- 第二步：理解Kernel初始化中断描述符表的内容
  - 在init\_cond()中调用了init\_idt()
  - init\_idt()位于kernel/src/irq/idt.c

```
void init_cond()
{
#ifdef IA32_INTR
    /* Reset the GDT, since the old GDT in s
    init_segment();

    /* Set the IDT by setting up interrupt a
    * Note that system call is the only exc
    */
    init_idt();

    /* Enable interrupts. */
    sti();
#endif
}
```

```
/* Each entry of the IDT is either an interrupt gate, or a trap gate */
static GateDesc idt[NR_IRQ];

/* Setup a interrupt gate for interrupt handler. */
static void set_intr(GateDesc *ptr, uint32_t selector, uint32_t offset, uint32_t dpl) {...}

void init_idt() {
    ...
    set_trap(idt + 0, SEG_KERNEL_CODE << 3, (uint32_t)vec0, DPL_KERNEL);
    ...
    /* the system call 0x80 */
    set_trap(idt + 0x80, SEG_KERNEL_CODE << 3, (uint32_t)vecsys, DPL_USER);

    set_intr(idt+32 + 0, SEG_KERNEL_CODE << 3, (uint32_t)irq0, DPL_KERNEL);
    set_intr(idt+32 + 1, SEG_KERNEL_CODE << 3, (uint32_t)irq1, DPL_KERNEL);
    set_intr(idt+32 + 14, SEG_KERNEL_CODE << 3, (uint32_t)irq14, DPL_KERNEL);

    /* the ``idt`` is its virtual address */
    write_idtr(idt, sizeof(idt));
    sti();
}
```

# NEMU中实现中断响应

- 第三步：在NEMU中实现对中断的响应
  - 在nemu/src/cpu/cpu.c中do\_intr()函数

```
void exec(uint32_t n)
{
    while (n > 0 && nemu_state == NEMU_RUN)
    {
        instr_len = exec_inst();
        cpu.eip += instr_len;
        n--;
    }

#ifdef IA32_INTR
    // check for interrupt
    do_intr();
#endif
}
```

```
#ifdef IA32_INTR
void do_intr()
{
    if (cpu.intr && cpu.eflags.IF)
    {
        // get interrupt number
        uint8_t intr_no = i8259_query_intr_no();
        assert(intr_no != I8259_NO_INTR);
        // tell the PIC interrupt info received
        i8259_ack_intr();
        // raise interrupt
        raise_intr(intr_no);
    }
}
#endif
```

# NEMU中实现异常（系统调用）响应

- 第三步：在NEMU中实现对异常的响应
  - 在nemu/src/cpu/intr.c中实现raise\_intr()函数

```
void raise_intr(uint8_t intr_no) {  
#ifdef IA32_INTR  
    - printf("Please implement raise_intr()");  
    - assert(0);  
    + // Trigger an exception/interrupt with 'intr_no'  
    + // 'intr_no' is the index to the IDT  
  
    + // Push EFLAGS, CS, and EIP  
    + // Find the IDT entry using 'intr_no'  
    + // Clear IF if it is an interrupt  
    + // Set CS:EIP to the entry of the interrupt handler  
    + // need to reload CS with load_sreg()  
#endif  
}
```

int 指令调用这个接口， why?



```
void raise_sw_intr(uint8_t intr_no) {  
    // return address is the  
    // next instruction  
    cpu.eip += 2;  
    raise_intr(intr_no);  
}
```

**NEMU的重要简化：**  
**都是ring0不考虑权限**

# NEMU中实现异常（系统调用）响应

- 第四步：理解Kernel对于0x80号系统调用的响应方式

```
#include "trap.h"

const char str[] = "Hello, world!\n";

int main()
{
    asm volatile( "movl $4, %eax;" // system call ID, 4 = SYS_write
                  "movl $1, %ebx;" // file descriptor, 1 = stdout
                  "movl $str, %ecx;" // buffer address
                  "movl $14, %edx;" // length
                  "int $0x80");

    HIT_GOOD_TRAP;
    return 0;
}
```

[testcase/src/hello-inline.c](http://testcase/src/hello-inline.c)

# NEMU中实现异常（系统调用）响应

- 第四步：理解Kernel对于0x80号系统调用的响应方式

```
#include "trap.h"

const char str[] = "Hello, world!\n";

int main()
{
    asm volatile( "movl $4, %eax;" //
                  "movl $1, %ebx;" //
                  "movl $str, %ecx;" // buffer address
                  "movl $14, %edx;" // length
                  "int $0x80");

    HIT_GOOD_TRAP;
    return 0;
}
```

```
void raise_sw_intr(uint8_t intr_no) {
    // return address is the
    // next instruction
    cpu.eip += 2;
    raise_intr(intr_no);
}
```

[testcase/src/hello-inline.c](http://testcase/src/hello-inline.c)

# NEMU中实现异常（系统调用）响应

- 第四步：理解Kernel对于0x80号系统调用的响应方式

```
void init_idt() {  
    ...  
    /* the system call 0x80 */  
    set_trap(idt + 0x80, SEG_KERNEL_CODE << 3, (uint32_t)vecsys, DPL_USER);  
    ...  
}
```

[kernel/src/irq/idt.c](#)

```
.globl vecsys; vecsys: pushl $0; pushl $0x80; jmp asm_do_irq  
  
.globl asm_do_irq  
.extern irq_handle  
  
asm_do_irq:  
    pushal  
  
    pushl %esp          # ???  
    call irq_handle  
  
    addl $4, %esp  
    popal  
    addl $8, %esp  
    iret
```

[kernel/src/irq/do\\_irq.S](#)

# NEMU中实现异常（系统调用）响应

- 第四步：理解Kernel对于0x80号系统调用的响应方式

```
void irq_handle(TrapFrame *tf) { pushl %esp # ???
    int irq = tf->irq; //结合kernel/src/irq/do_irq.S, 理解tf怎么传进来的

    ...

    else if (irq == 0x80) {
        do_syscall(tf); //tf又当作参数传给了do_syscall, tf里面有什么?
    }
    ...
}
```

[kernel/src/irq/irq\\_handle.c](#)

```
static void sys_write(TrapFrame *tf) {
    // 设置 tf -> eax 是想干吗?
    tf->eax = fs_write(tf->ebx, (void*)tf->ecx, tf->edx);
}

void do_syscall(TrapFrame *tf) {
    switch(tf->eax) { //tf -> eax哪儿来的, 最早是谁设置的?
        ...
        case SYS_write: sys_write(tf); break; // SYS_write等都是预定义好的常量
        ...
    }
}
```

[kernel/src/syscall/do\\_syscall.c](#)



# NEMU中实现异常（系统调用）响应

- 第四步：理解Kernel对于0x80号系统调用的响应方式

```
void init_idt() {  
    ...  
    /* the system call 0x80 */  
    set_trap(idt + 0x80, SEG_KERNEL_CODE << 3, (uint32_t)vecsys, DPL_USER);  
    ...  
}
```

[kernel/src/irq/idt.c](#)

```
.globl vecsys; vecsys: pushl $0; pushl $0x80; jmp asm_do_irq
```

```
.globl asm_do_irq  
.extern irq_handle
```

```
asm_do_irq:
```

```
    pushal
```

```
    pushl %esp          # ???
```

```
    call irq_handle
```

```
    addl $4, %esp
```

```
    popal
```

```
    addl $8, %esp
```

```
    iret
```

响应完后恢复现场，iret，  
对于0x80系统调用，当初  
保存的返回地址是多少？

[kernel/src/irq/do\\_irq.S](#)

# NEMU中实现异常（系统调用）响应

PA 4-1 小任务一：完成上述编码和理解工作，并且：

执行`hello-inline`测试用例并看到屏幕输出

`nemu trap output:` Hello, world!

# NEMU中实现中断响应

## PA 4-1 小任务二：开启时钟中断并理解响应过程

- 以时钟中断为例

1. 在`include/config.h`中定义宏`HAS_DEVICE_TIMER`并`make clean`;
2. 在`nemu/include/cpu/reg.h`的`CPU_STATE`中添加`uint8_t intr`成员，模拟中断引脚；
3. 在`nemu/src/cpu/cpu.c`的`init_cpu()`中初始化`cpu.intr = 0`;
4. 在`nemu/src/cpu/cpu.c`的`exec()`函数`while`循环体，每次执行完一条指令后调用`do_intr()`函数查看并处理中断事件；
5. 执行`make test_pa-4-1`;
6. 触发Kernel中的`panic`，找到该`panic`并移除

完整跟踪一下从`nemu/src/device/dev/timer.c`中发出  
timer interrupt到触发panic的整个过程，就可以全懂

timer interrupt是由`sdl.c`中的线程定时调用发出的



PA 3-3截止：2020年1月10日24时

PA 4-1到此结束

祝大家学习快乐，身心健康！

欢迎大家踊跃参加问卷调查