§4-1.3.1 通过自陷实现系统调用

1. 详细描述从测试用例中 int $0x80 开始到 HIT_GOOD_TRAP 为止的详细系统行为（完整描述控制的转移过程，即相关函数的调用和关键参数传递过程），通过文字或画图的方式；

首先查看 hello-inline 的反汇编代码：

```
Disassembly of section .text:

08049000 <start>:
 8049000:   e9 00 00 00 00          jmp    8049005 <main>

08049005 <main>:
 8049005:   55                      push   %ebp
 8049006:   89 e5                   mov    %esp,%ebp
 8049008:   e8 28 00 00 00          call   8049035 <__x86.get_pc_thunk.ax>
 804900d:   05 f3 2f 00 00          add    $0x2ff3,%eax
 8049012:   b8 04 00 00 00          mov    $0x4,%eax
 8049017:   bb 01 00 00 00          mov    $0x1,%ebx
 804901c:   b9 00 a0 04 08          mov    $0x804a000,%ecx
 8049021:   ba 0e 00 00 00          mov    $0xe,%edx
 8049026:   cd 80                   int    $0x80
 8049028:   b8 00 00 00 00          mov    $0x0,%eax
 804902d:   82                      nemu_trap
 804902e:   b8 00 00 00 00          mov    $0x0,%eax
 8049033:   5d                      pop    %ebp
 8049034:   c3                      ret

08049035 <__x86.get_pc_thunk.ax>:
 8049035:   8b 04 24                mov    (%esp),%eax
 8049038:   c3                      ret

```

注意到 int 指令的参数为立即数 0x80

接着查看（好吧，编写）int 指令代码：

```
make_instr_func(int_) {
    OPERAND imm;
    imm.type = OPR_IMM;
    imm.data_size = 8;
    imm.sreg = SREG_CS;
    imm.addr = eip + 1;
    operand_read(&imm);
    raise_sw_intr(imm.val);
    print_asm_1("int", "", 2, &imm);
    return 0;
}

```

根据手册，int 指令获得对应的中断号后

将中断号作为参数传给 raise_sw_intr 函数

接着查看 raise_sw_intr 代码：

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

```

可见 raise_sw_intr 函数将 eip 加 2（转移至下一条指令的开始）

然后以传进来的中断号为参数，再调用 raise_intr 函数

```
void raise_intr(uint8_t intr_no) {
    // 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 EIP to the entry of the interrupt handler
}
```

根据文档的说明，容易看出 raise_intr 需要完成的工作是：

1）保护现场（把当前的 eflags 寄存器，cs 寄存器和 eip 压栈）；

2）根据传入的中断号计算对应处理程序的线性地址；

3）检查是否需要设置关中断；

4）对 eip 赋值来进行跳转。

接着查看 init_idt 函数代码找到中断表的初始化：

```
82      /* the system call 0x80 */
83      set_trap(idt + 0x80, SEG_KERNEL_CODE << 3, (uint32_t)vecsys, DPL_USER);
```

找到 set_trap 函数，发现对应的是中断门的设置：

```
23  /* Setup a trap gate for cpu exception. */
24  static void set_trap(GateDesc *ptr, uint32_t selector, uint32_t offset, uint32_t dpl)
25  {
26      ptr->offset_15_0 = offset & 0xFFFF;
27      ptr->segment = selector;
28      ptr->pad0 = 0;
29      ptr->type = TRAP_GATE_32;
30      ptr->system = 0;
31      ptr->privilege_level = dpl;
32      ptr->present = 1;
33      ptr->offset_31_16 = (offset >> 16) & 0xFFFF;
34  }
```

在 do_irq.S 中可以找到 set_trap 作为 offset 传入的函数 vecsys：

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

发现它的作用是先后将 0 和 0x80 压栈，然后跳转到 asm_do_irq

```
33  asm_do_irq:
34      pushal
35
36      pushl %esp      # ???
37      call irq_handle
38
39      addl $4, %esp
40      popal
41      addl $8, %esp
42      iret
```

可见 asm_do_irq 完成的工作是：

1）调用 pusha 将所有通用寄存器压栈（保护现场）；

2）使用 push %esp（利用栈帧准备调用参数）；

3）调用 irq_handle 函数；

4）复原栈帧和调用 popa 恢复现场，iret 跳出。

irq_handle 函数参数为指针变量 TrapFrame，即最后压栈的 esp。

irq_handle 对参数进行判断并处理中断。结束后跳转返回原程序。

2. 在描述过程中，回答 kernel/src/irq/do_irq.S 中的 push %esp 起到了什么作用，画出在 call irq_handle 前系统栈的内容和 esp 的位置，指出 TrapFrame 对应系统栈的哪一段内容。

由 1 中分析可见，push %esp 的作用就是传参数，调用 pusha 后，esp 就是当前栈顶指针的地址，把这个地址压栈就是把 pusha 准备的数据的地址作为指针传参，函数 irq_handle 的参数是 TrapFrame 类型的指针，找到 TrapFrame 的定义如下：

```
107 typedef struct TrapFrame
108 {
109     uint32_t edi, esi, ebp, xxx, ebx, edx, ecx, eax; // GPRs
110     int32_t irq;                                     // #irq
111     uint32_t error_code;                             // error code
112     uint32_t eip, cs, eflags;                        // execution state saved by hardware
113 } TrapFrame;
114
```

可见 TrapFrame 包含 eflags，cs 和 eip 信息，即 raise_intr 中准备的三个数据，error_code 和 irq 分别对应 vecsys 中压栈的 0 和 0x80，通用寄存器对应 asm_do_irq 中 pusha 的调用，再次证实 push %esp 就是传参给 irq_handle。call irq_hadle 之前系统栈的内容如下图：

<table>
<tr><td>……</td><td>……</td></tr>
<tr><td>EFLAGS</td><td rowspan="13">TrapFrame</td></tr>
<tr><td>CS</td></tr>
<tr><td>EIP</td></tr>
<tr><td>0</td></tr>
<tr><td>0x80</td></tr>
<tr><td>EAX</td></tr>
<tr><td>ECX</td></tr>
<tr><td>EDX</td></tr>
<tr><td>EBX</td></tr>
<tr><td>XXX</td></tr>
<tr><td>EBP</td></tr>
<tr><td>ESI</td></tr>
<tr><td>EDI</td></tr>
<tr><td>ESP 旧址</td><td><-- ESP</td></tr>
</table>

§4-1.3.2 响应时钟中断

1. 详细描述 NEMU 和 Kernel 响应时钟中断过程和先前的系统调用过程不同之处在哪里？相同的地方又在哪里？可以通过文字或画图的方式来完成。

响应时钟中断和系统调用都保护现场、准备参数和调用 irq_handle，结束后都返回下一条指令继续执行。通过观察和分析代码可见前者调用 irq_handle 进入的是这一分支：

```
else if (irq >= 1000)
{
    int irq_id = irq - 1000;
    assert(irq_id < NR_HARD_INTR);
    //if (irq_id == 0)
        //panic("You have hit a timer interrupt, remove this panic after you've figured out how the control flow gets here.");
```

区别在于响应时钟中断是由系统事件引起的，与现行指令无关，是异步执行的，而系统调用是由执行的现行指令引起的，是同步执行的。此外响应中断是直接读取中断向量表（此处添加了 panic 来表示），系统调用则通过调用函数 do_syscall 间接跳转。