PA_4-1 实验报告

计算机科学与技术系 张桓 191220156

§ 1.通过自陷实现系统调用

1. 详细描述从测试用例中的 int $0x80 开始一直到 HIT_GOOD_TRAP 为止的详细的系统行为(完整描述控制的转移过程，即相应函数的调用和关键参数传递过程)，可以通过文字或画图的方式完成。

答：以 hello-inline 测试用例为例，其代码如下：

```
#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;
}
```

其中以 movl 开头的四条内联汇编指令准备 int $0x80 系统调用所需要的参数，依次保存操作系统调用号、输出设备、字符串首地址、长度到相应的通用寄存器中，然后执行 int $0x80 陷入内核。

int $0x80 执行本质上也是查询 IDT，在查询之前先将 EFLAGS,CS,EIP 压入内核栈保存起来，这是通过调用 raise_sw_intr 函数并进一步调用 raise_intr 函数实现的，参数为 intr_no 即中断号 0x80。然后在 kernel/src/irq/idt.c 中找到对应的第 0x80 号门描述符代码如下:

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

可以得知 0X80 号门的处理程序的入口地址在 vecsys 处，置 CS:EIP 为这个地址，从而开始执行该地址处的代码。在 kernel/src/irq/do_irq.S 中找到其代码如下:

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

将 error_code 和 irq 号压入内核栈中，再跳转到 asm_do_irq 处，代码如下:

```
asm_do_irq:
    pushal
    pushl %esp      # ???
    call irq_handle
    addl $4, %esp
    popal
    addl $8, %esp
    iret
```

先 pusha 将所有通用寄存器的值(GPRS)压栈，再压入 ESP，然后调用 irq_handle 中断/异常处理程序。找到 irq_handle 函数类型如下:

```
void irq_handle(TrapFrame *tf)
{
    int irq = tf->irq;
    if (irq < 0)
    {
        panic("Unhandled exception!");
    }
    else if (irq == 0x80)
    {
        do_syscall(tf);
    }
    else if (irq < 1000)
    {
        panic("Unexpected exception #%d at eip = %x", irq, tf->eip);
    }
    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.");
        struct IRQ_t *f = handles[irq_id];
        while (f != NULL)
        { /* call handlers one by one */
            f->routine();
            f = f->next;
        }
    }
}
```

可以看到它又调用了 do_syscall 函数，其参数也是 tf，在 kernel/src/syscall/do_syscall.c 中找到代码如下：

```
void do_syscall(TrapFrame *tf)
{
    switch (tf->eax)
    {
    case 0:
        cli();
        add_irq_handle(tf->ebx, (void *)tf->ecx);
        sti();
        break;
    /***/
    case SYS_write:
        sys_write(tf);
        break;
    /***/
    default:
        panic("Unhandled system call: id = %d", tf->eax);
    }
}
```

可以看到 tf->eax 就是中断对应的系统调用号(这里为 4)，找到对应的 case 的 sys_xxx 处理函数(这里为 sys_write)，再调用 fs_write 函数完成输出。

这之后的 popa 和 iret 指令完成处理处理过程并且返回。返回到 hello-inline 测试用例中将 CS:EIP 修改为断点处，即 HIT_GOOD_TRAP。

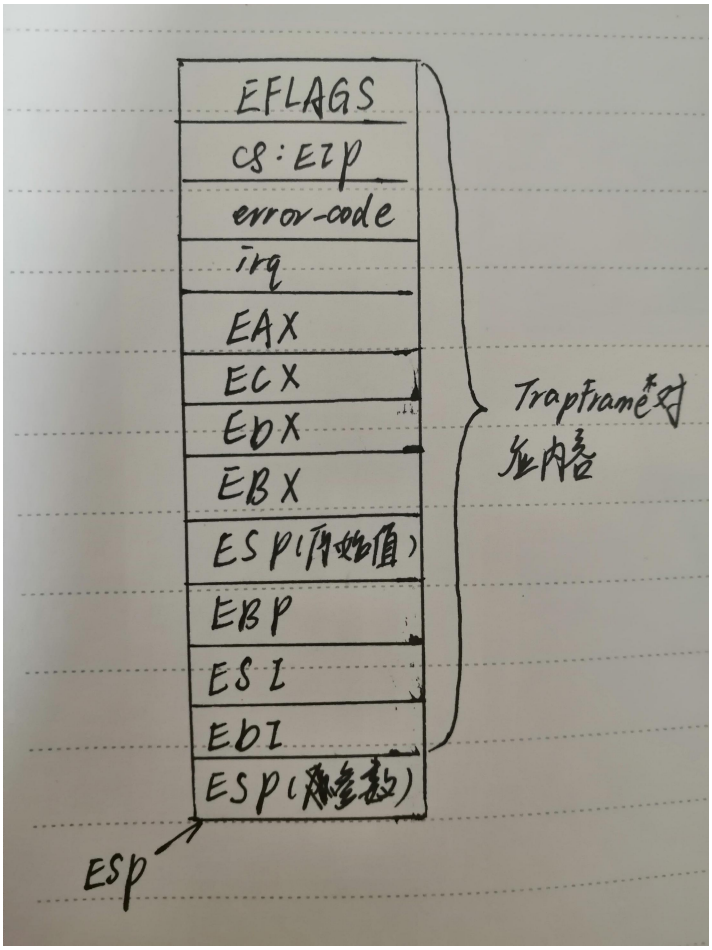
2. 在描述过程中，回答 kernel/src/irq/do_irq.S 中的 push %esp 起什么作用，画出在 call irq_handle 之前，系统栈内容和 ESP 的位置，指出 TrapFrame 对应系统栈的哪一段内容。

答：根据 irq_handle 函数类型可以看到它有一个 TrapFrame 指针类型的参数，因此 push %esp 的作用就是将 ESP 作为指针为 irq_handle 函数准备参数。在 kernel/include/x86/memory.h 中找到 TrapFrame 结构的定义如下:

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

因此参数 tf 指向栈，栈内的内容即中断/异常处理过程中压入内核栈的东西(即 eflags,CS:EIP 等等)，从而完成向处理函数传参的过程。

在 call irq_handle 之前，系统栈内容和 ESP 的位置，以及 TrapFrame 对应内容如下图：


EFLAGS
CS:EIP
error-code
irq
EAX
ECX
EDX
EBX
ESP(初始值)
EBP
ESI
EDI
ESP(参数)
ESP
TrapFrame对
象内容

§ 2.响应时钟中断

1. 详细描述 NEMU 和 Kernel 响应时钟中断的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完成。

答：系统调用使用 int 指令，这是通过 raise_intr()函数实现的。

相同之处：时钟中断处理过程和系统调用过程都是在 push 现场之后，在 irq_handle()函数中再判断是时钟中断还是系统调用，相关代码如下：

```
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.");
        struct IRQ_t *f = handles[irq_id];
        while (f != NULL)
        { /* call handlers one by one */
            f->routine();
            f = f->next;
        }
    }
```

本质上都是 kernel 处理中断。

不同之处：

时钟中断会触发 panic，即上表中注释部分。如果是，就会执行 panic，代码如下：

```
#define panic(format, ...)                                      \
    do                                                          \
    {                                                           \
        Log("\33[1;31msystem panic: " format, ##__VA_ARGS__); \
        HIT_BAD_TRAP;                                           \
    } while (0)
```

强制触发 HIT_BAD_TRAP，这也是需要删除掉的地方。

而系统调用则会创建一个链表数组，一个接一个地调用目标处理函数。