PA-4-1实验报告

§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，即相关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来完成;

   ①在`nemu`运行至`int $0x80`指令时, `nemu`将调用函数`raise_sw_intr()`并将`int`指令后的中段号`0x80`作为`raise_sw_intr()`的参数.相关过程如下所示:

```
make_instr_func(int_ib) {
    OPERAND rel;
    ......//read rel
    print_asm_1("int", "", 2, &rel);
    //GOTO TRAP-GATE-OR-INTERRUPT-GATE
    raise_sw_intr(rel.val);
    return 0;
}
```

   ②在`raise_sw_intr()`中将`nemu`的`cpu`的`eip`寄存器指向该`int`指令后2字节的位置, 即下一条指令的位置, 以在操作系统`kernel`处理完自陷调用后回到原来用户程序的下一条指令继续运行(只有用户自陷操作会这样). 然后调用`raise_intr()`函数, 中段号作为参数. 如下所示:

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

   ③`raise_intr()`函数中将完成现场保护(将`cpu`的`EFLAGS`, `CS:EIP`压入栈中), 然后根据中段类型设置`IF`位(这里由于`int $0x80`是自陷类型的中段, 允许嵌套, 故`IF`为为1), 再查找`IDT`表找到`0x80`对应中段的处理程序的门描述符, 根据门描述符中的段选择符和逻辑地址设置`cpu`的`CS:EIP`的值, 之后`nemu`将跳转至对应处理程序的入口并执行之.

   ④由`kernel/src/irq/idt.c`中对`0x80`自陷的设置:

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

   `vecsys`为③中所说的`int $0x80`对应的处理程序的入口. `kernel/src/irq/do_irq.S`中对`vecsys`的实现如下:

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

   `asm_do_irq`内容如下:

```
asm_do_irq:
    pushal

    pushl %esp      # parameter for irq_handle
    call irq_handle

    addl $4, %esp
    popal
    addl $8, %esp
    iret
```

`vecsys`和`asm_do_irq`中的`pushl $0;  pushl $0x80;  pushal;`指令相当于像栈中压入一个`TrapFrame`结构体变量. 之后的`pushl %esp`指令是向栈中压入`irq_handle()`函数的参数`(TrapFrame) *tf`. 之后调用中断处理函数`irq_handle()`.

⑤在函数`irq_handle()`中, 根据传入`TrapFrame`结构体对应的中段号, 将执行`do_syscall(tf)`.

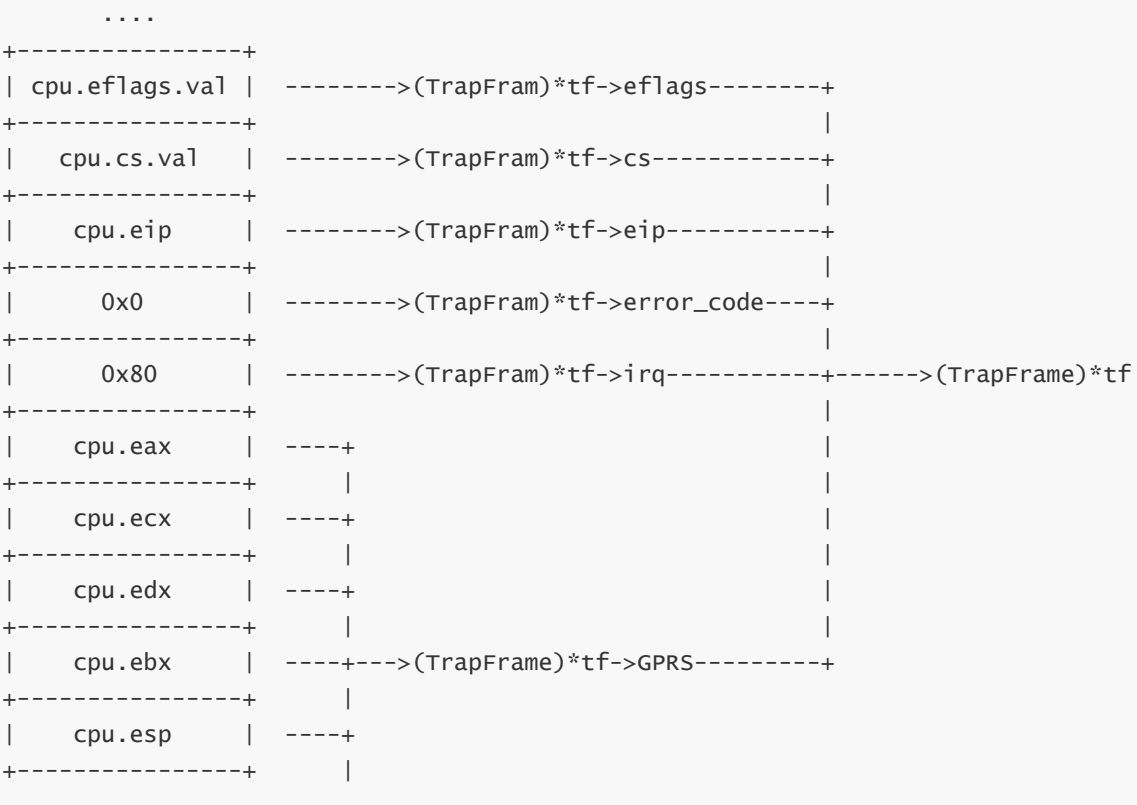
⑥`kernel/src/syscall/do_syscall.c`中根据`tf`的`eax`值执行相应的操作. 这里`eax`的值在`hello-inline.c`中已给出，是`4`， 对应`SYS_write`，故将执行`sys_write(tf)`. 最终将输出`tf->ecx`中存放的字符串`Hello, world!\n`, 长度为`tf->edx`中的`14`.

⑦至⑥完成了对`int $0x80`的自陷处理, 接下来将恢复现场. 首先将一路返回至`asm_do_irq`执行`call irq_handle`之后的语句, 包括回收栈空间, pop之前pusha时压入的数据, 执行`iret`恢复现场. 现场恢复后`nemu`将回到`hello-inline`测试文件中`int $0x80`的下一指令继续运行(`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，系统栈的内容和`esp`的位置，指出`TrapFrame`对应系统栈的哪一段内容。

在题1.中的④已经提到了`kernel/src/irq/do_irq.S`中的`push %esp`起到给`irq_handle()`函数提供参数的作用.

执行`call irq_handle`之前，系统栈的内容和`esp`的位置的示意图如下:(一格对应4Byte)



```
|    cpu.esi     |  ----+
+----------------+      |
|    cpu.edi     |  ----+
+----------------+
|      esp       |  -------->&(TrapFram)*tf
+----------------+
|                |  <--------cpu.esp
+----------------+
       ....
```

§4-1.3.2 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来完成。

   ①不同之处：时钟中断属于外部中断, 需要 `cpu` 在每执行完一条指令后根据模拟的 `INTR` 中断引脚传入的中断请求信号和 `IF` 位状态进行中断处理. 如下所示(`nemu/src/cpu/cpu.c`):

   ```
   if (cpu.intr && cpu.eflags.IF)
       {
           is_nemu_hlt = false;
           // get interrupt number
           uint8_t intr_no = i8259_query_intr_no(); // get interrupt number
           assert(intr_no != I8259_NO_INTR);
           i8259_ack_intr();   // tell the PIC interrupt info received
           raise_intr(intr_no); // raise intrrupt to turn into kernel handler
       }
   ```

   通过模拟的 `i8259` 芯片获取中断号后调用 `raise_intr()` 执行中断处理. 而 `int` 指令的中断号由指令给出故不需要与 `i8259` 进行交互. 另外, 这里并不像 `int $0x80` 指令需要调用 `raise_sw_intr()` 来使 `eip` 指向下一条指令, 因为当前 `eip` 指向的指令尚未执行.

   在 `raise_intr()` 中, 由于时钟中断是外部中断, `IF` 位需要置0, 即关中断. 在处理该中断过程中 `cpu` 不再受理其他中断请求. `int $0x80` 由于是自陷, 不需要关中断.

   在 `irq_handle()` 中对时钟中断的操作由 `irq >= 1000` 的 `if` 分支实现. 这里使用了链表 `IRQ_t` 来存储未处理的中断号并依次根据中断号执行对应的处理程序. 如下所示:

   ```
   struct IRQ_t *f = handles[irq_id];

           while (f != NULL)
           { /* call handlers one by one */
               f->routine();
               f = f->next;
           }
   ```

   ②相同之处: `raise_intr()` 中的现场保护; 在进入 `asm_do_irq()` 后直到调用对应中断处理程序的部分; 执行完处理程序的恢复现场部分.