PA4-1 实验报告

191220163 计算机科学与技术系 张木子苗

§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）cpu 执行到 int $0x80 指令时，将中断号 0x80 传递给函数 raise_sw_intr，函数 raise_sw_intr 将 %eip 加 2 跳过int指令之后，转到函数 raise_intr，中断号不变

   （2）函数 raise_intr 通过访问寄存器 idtr 获得中断描述符表首地址，再以中断号 0x80 为下标访问中断描述符表得到门描述符，保存 eflags 和断点（cs，eip）后将 %eip 指向门描述符对应的地址。然后系统就会执行异常所对应的异常处理程序。

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

   （3）系统调用对应的传参函数是 vecsys 函数。vecsys 函数将硬件错误码 0 和中断号 0x80 压栈，转到asm_do_irq，asm_do_irq 将先用 pusha 保存通用寄存器，再将由硬件保存的 eip, cs, eflags 和 push 的 error_code, irq 以及 pusha 保存的通用寄存器组成结构体 TrapFrame，并把结构体首地址 （即%esp）传递给函数 irq_handle，irq_handle 根据中断号0x80决定进行系统调用，再根据 TrapFrame 访问系统调用的参数，完成输出字符串操作。

   ```
   asm_do_irq:
       pushal

       pushl %esp      # ???
       call irq_handle

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

   （4）之后，通过 popa 和 iret 指令恢复栈帧和保存的现场，然后就返回之前的用户程序的下一条指令，继续 执行接下来的指令，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，系统栈的内容和`esp`的位置，指出`TrapFrame`对应系统栈的哪一段内容。

   push %esp 指令的作用是将 `TrapFrame` 结构体首地址 （%esp）传递给函数 irq_handle，`TrapFrame` 结构体如下：

   ```
   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;
   ```

执行该指令前的栈帧如下表格：

地址	所存内容	保存者
%esp + 0x30	eflags	硬件保存
%esp + 0x2c	cs	硬件保存
%esp + 0x28	%eip	硬件保存
%esp + 0x24	error_code（硬件出错码）	软件保存
%esp + 0x20	irq（中断号）	软件保存
%esp + 0x1c	%eax	pusha保存
%esp + 0x18	%ecx	pusha保存
%esp + 0x14	%edx	pusha保存
%esp + 0x10	%ebx	pusha保存
%esp + 0xc	xxx	pusha保存
%esp + 0x8	%ebp	pusha保存
%esp + 0x4	%esi	pusha保存
%esp	%edi	pusha保存

§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。

   相同之处：都通过 raise_intr 函数把 EIP 指向 kernel 代码，在 kernel 代码中处理中断。都根据中断处理号push 相关参数，转 asm_do_irq 保存现场和传递参数后，再转 irq_handle 处理。

   不同之处：

   （1）处理的时机不同：系统调用是通过 int 指令进行，而响应中断是在每条指令执行之后利用 do_intr() 函数查询中断引脚并检测是否为开中断状态，然后再进行相应的中断处理。

   （2）在 irq_handle 中所走的分支不同。系统调用走的是中断处理号为 0x80 的分支，转 do_syscall 处理

   ```
   else if (irq == 0x80)
   {
       do_syscall(tf);
   }
   ```

   而中断处理走的是中断处理号大于1000的分支（需要减去1000得到实际参数）。在这里我们根据中断类型，从handles中得到该中断的多个处理函数形成的链表的首地址，在kernel中一一执行。

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

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