PA 4-1 实验报告

221220085 时昌军

一、实验目的

1. 掌握硬件如何对异常和中断进行识别并相应。
2. 掌握操作系统如何对异常和中断进行相应的处理。

二、实验过程

§4-1.3.1 通过自陷实现系统调用

1. 在`include/config.h`中定义宏`IA32_INTR`并`make clean`；
2. 在`nemu/include/cpu/reg.h`中定义`IDTR`结构体，并在`CPU_STATE`中添加`idtr`；
3. 实现包括`lidt`、`cli`、`sti`、`int`、`pusha`、`popa`、`iret`等指令；
4. 在`nemu/src/cpu/intr.c`中实现`raise_intr()`函数;
5. 执行`hello-inline`测试用例, 或执行`make test_pa-4-1`命令并看到屏幕输出

```
nemu trap output: Hello, world!
```

§4-1.3.2 响应时钟中断

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。

```
nemu: HIT GOOD TRAP at eip = 0x08049086
NEMU2 terminated
./nemu/nemu --autorun --testcase struct --kernel
NEMU load and execute img: ./kernel/kernel.img  elf: ./testcase/bin/struct
nemu trap output: [src/main.c,82,init_cond] {kernel} Hello, NEMU world!
nemu trap output: [src/elf/elf.c,29,loader] {kernel} ELF loading from ram disk.
nemu: HIT GOOD TRAP at eip = 0x080490ec
NEMU2 terminated
./nemu/nemu --autorun --testcase string --kernel
NEMU load and execute img: ./kernel/kernel.img  elf: ./testcase/bin/string
nemu trap output: [src/main.c,82,init_cond] {kernel} Hello, NEMU world!
nemu trap output: [src/elf/elf.c,29,loader] {kernel} ELF loading from ram disk.
nemu: HIT GOOD TRAP at eip = 0x08049150
NEMU2 terminated
./nemu/nemu --autorun --testcase hello-str  --kernel
NEMU load and execute img: ./kernel/kernel.img  elf: ./testcase/bin/hello-str
nemu trap output: [src/main.c,82,init_cond] {kernel} Hello, NEMU world!
nemu trap output: [src/elf/elf.c,29,loader] {kernel} ELF loading from ram disk.
nemu: HIT GOOD TRAP at eip = 0x080490d8
NEMU2 terminated
./nemu/nemu --autorun --testcase test-float --kernel
NEMU load and execute img: ./kernel/kernel.img  elf: ./testcase/bin/test-float
nemu trap output: [src/main.c,82,init_cond] {kernel} Hello, NEMU world!
nemu trap output: [src/elf/elf.c,29,loader] {kernel} ELF loading from ram disk.
nemu: HIT BAD TRAP at eip = 0x080490bd
NEMU2 terminated
make-[1]: Leaving directory '/home/pa221220085/pa_nju'
./nemu/nemu --autorun --testcase hello-inline --kernel
NEMU load and execute img: ./kernel/kernel.img  elf: ./testcase/bin/hello-inline
nemu trap output: [src/main.c,82,init_cond] {kernel} Hello, NEMU world!
nemu trap output: [src/elf/elf.c,29,loader] {kernel} ELF loading from ram disk.
nemu trap output: Hello, world!
nemu: HIT GOOD TRAP at eip = 0x08049023
NEMU2 terminated
pa221220085@icspa:~/pa_nju$ 
```

三、思考题

§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；**

答：执行 `int $0x80` 时，调用了 `int` 指令，通过解析操作码，获取中断号 `0x80`， 随后将其作为参数，调用 `raise_sw_intr()` 函数，该函数更新 `eip` 地 址后，便调用 `raise_intr()` 函数。

在 `raise_intr()` 函数中的 `intr_no` 依然是 `0x80`。随后，依次将 `eflags`， `CS` 和 `eip` 的值压栈，并从 `IDTR` 读出 `IDT` 的首地址，根据中断号 `0x80` 在 `IDT` 中索引得到一个门描述符，把门描述符的段选择符装载入 `CS` 寄存器，接着 调用 `load_sreg()` 函数加载 CS 的隐藏部分。根据段选择符中 `type` 的信息判断是中断还是陷阱。如果是中断便把 `IF` 清零。最后把 `offset` 赋 给 `eip`，`raise_intr()` 调用结束。

随后返回 `int` 指令，由于 `return 0`，此时的 `eip` 便是中断处理程序的入口地址。执行到这一步后，便是操作系统（kernel）的工作了。通过入口地址的信息，跳转到 `kernel/src/irq/do_irq.S` 的入口函数 `vecsys()`，执行 `pushl 0x80` 后，压入错误码和异常号，跳转到 `asm_do_irq` 中， 执行三个阶段：

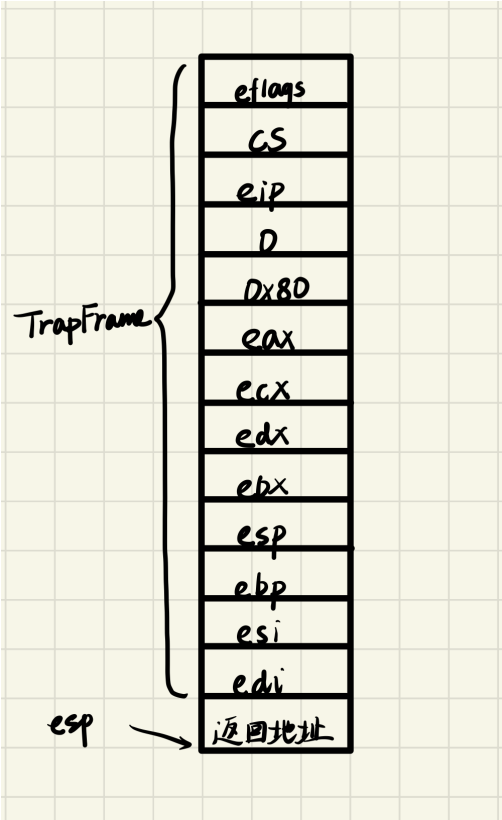
准备阶段：将所有寄存器的值压栈，保护程序运行的现场信息。和之前压入栈的 `eflags` 和 `eip` 构成 `TrapFrame` 结构。
处理阶段：调用 `irq_handle()` 函数，传递的参数是 `TrapFrame*` 类型的 `tf`。在 `irq_handle()` 中根据 `tf` 读出 `irq` 的中断类型，由于是 `0x80`， `kernel` 调用 `do_syscall()` 函数。在 `do_syscall()` 函数中，根据传入的参数 `tf->eax= 4`，调用 `sys_write()` 函数，该函数根据 `tf` 指针把 参数 `ebx，ecx，edx` 传入 `fs_write()` 函数调用，从而在屏幕输出 `Hello,world!`。

结束阶段:返回到`do_irq.S`。pop 所有寄存器的内容。恢复现场信息。通过`iret`恢复用户程序的程序状态，并修改`CS:EIP`，返回到断点处继续执行。之后变回出现`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。**

答：`push %esp`的作用是把执行完`pusha`后的`esp`压栈，而这个`esp`指向的是`TrapFrame`的首地址，因此这个步骤是在把`TrapFrame`的指针作为参数传给`irq_handl`。

系统栈的内容和`esp`位置：



§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。**

答：不同之处：处理时钟中断需要在一条指令执行完后查看中断引脚信号和`IRQ`请求号，判断是否有中断请求，而系统调用不需要。系统调用立即陷入内核态即可，而响应中断时还需要预先处理，如开中断，中断屏蔽字等。

相同之处：响应时钟中断和系统调用都会使用`int 0x80`指令陷入内核态执行，并保存断点，程序运行状态等信息。而两者都是通过调用号调用响应的程序进行处理，即处理过程都是通过kernel 内核态完成。