

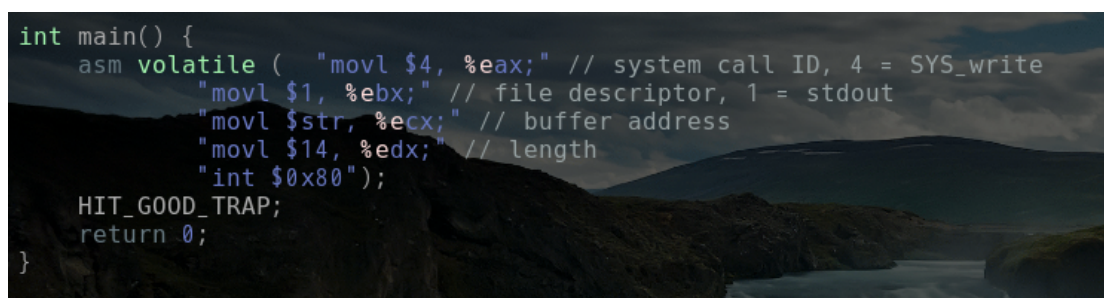
PA4 实验报告

—————161220070 李鑫烨

PA4-1

1. 详细描述从测试用例中的 `int $80` 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的转移过程），可通过文字或画图的方式来完成。

(1). `hello-inline` 的源码中将 `eax` 即系统调用号置为 4，即为 `SYS_write`，将 `ebx` 即 file descriptor 置为 1，即为 `stdout`，将字符串 `Hello World` 的地址及长度分别存入 `ecx`、`edx` 中，而后执行 `int 0x80` 系统调用指令。



```
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;
}
```

(2). CPU 读取到自陷指令 `int $0x80` 之后，读取到中断号 `intr_no` 为 `0x80`，以此作为参数调用函数 `raise_intr()`。由于 `int` 指令是通过自陷来实现系统调用而不是中断，因此保存的断点为下一条指令的地址，需要通过调用 `raise_sw_intr()` 使 `cpu.eip += 2` 之后间接调用 `raise_intr()`。

(3). 函数 `raise_intr()` 先后 push 了寄存器 `eflags`，`CS` 和当前 `eip` 的值，而后通过查询 `cpu.idtr` 得到中断表基址，并通过系统调用号 `intr_no` 得到对应的 IDT 项，读取 `offset` 与 `selector` 的值，`selector` 的值即为段选择符，将段寄存器 `CS` 的值设为 `selector` 并调用 `load_sreg()`，得到对应段描述符中的段偏移量 `offset`，

将 offset 与段偏移量相加跳转到 kernel 准备好的对应调用好 0x80 的处理程序，即 0xc00300a0 处所对应程序，push 了 irq 和 error_code 之后跳转到 asm_do_irq。

```
c00300a0 <vecsyz>:
c00300a0:  6a 00                push    $0x0
c00300a2:  68 80 00 00 00      push    $0x80
c00300a7:  eb 21              jmp     c00300ca <asm_do_irq>
```

(4). 在 asm_do_irq 汇编代码中，先 pusha 了所有通用寄存器的值，而后 push %esp 以 TrapFrame 指针的形式传递给 irq_handle 函数。

```
c00300ca <asm_do_irq>:
c00300ca:  60                pusha
c00300cb:  54                push    %esp
c00300cc:  e8 3a 02 00 00    call   c003030b <irq_handle>
c00300d1:  83 c4 04          add     $0x4,%esp
c00300d4:  61                popa
c00300d5:  83 c4 08          add     $0x8,%esp
c00300d8:  cf                iret
```

(5). 在 irq_handle 过程中，之前压栈存储在 TrapFrame 中 eax、ebx、ecx、edx 分别指出系统调用号，file descriptor，将要输出的字符串的地址与长度，并调用 do_syscall()函数，执行 sys_write()函数打印 Hello World 之后返回到 Irq_handle()函数。

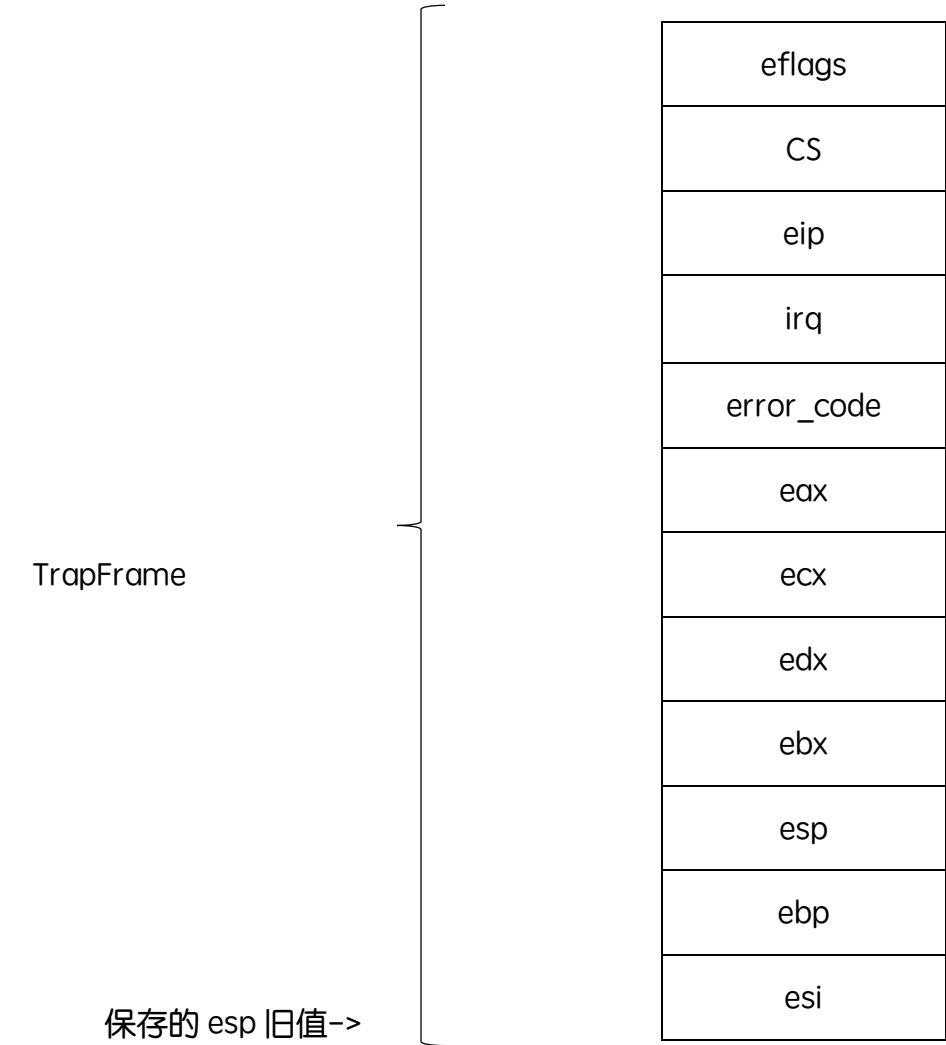
```
void do_syscall(TrapFrame *tf) {
    switch(tf->eax) {
        case 0:
            cli();
            add_irq_handle(tf->ebx, (void*)tf->ecx);
            sti();
            break;
        case SYS_brk: sys_brk(tf); break;
        case SYS_open: sys_open(tf); break;
        case SYS_read: sys_read(tf); break;
        case SYS_write: sys_write(tf); break;
        case SYS_lseek: sys_lseek(tf); break;
        case SYS_close: sys_close(tf); break;
        default: panic("Unhandled system call: id = %d", tf->eax);
    }
}
```

(6). 执行完 irq_handle 函数之后返回到 asm_do_irq 汇编代码中，addl \$4, %esp 之后使%esp 指向之前 pusha 压栈的所有通用寄存器并 popa，然后

addl \$8, %esp, 使%esp 越过被压栈的的 irq 及 error_code 之后, 实行 iret, 将先前 raise_intr()中 push 的 eflags, CS 及 eip 的值 pop 出来, 使 eip 指向最初保存的断点地址, 由内核态转为用户态, 开始执行 int \$80 的下一条地址, 完成系统调用过程, 最终 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 do_irq.S 中 push %esp 指令的作用在于将先前压栈的 TrapFrame 的地址作为参数传递给 irq_handle 函数。



esp ->

edi
保存的 esp 旧值

3. 详细描述 NEMU 和 kernel 响应时钟中断的过程和先前的系统调用不同之处在哪里？相同的地方又在哪里？可通过文字或者画图的方式来完成。

响应时钟中断的过程如下：

(1). time_intr()函数发出时钟中断，将 cpu.intr 置为 1。

(2). cpu 每执行完一条指令后，都会检查是否有时间中断，若 cpu.intr 即 cpu.eflags.IF 均为 1，则直接调用 raise_intr()函数。

(3). 在 raise_intr()函数中，与系统调用相似的，将 eflags, CS, eip 压栈，查询 cpu.idtr 得到中断表基址，并通过系统调用号 intr_no 得到对应的 IDT 项，读取 selector 并加载 CS 寄存器后，将 eip 设为 kernel 准备好的处理时钟中断的程序地址，即 0xc00300a9, push 了 irq 和 error_code 之后跳转到 asm_do_irq。

```
c00300a9 <irq0>:
c00300a9:  6a 00                push    $0x0
c00300ab:  68 e8 03 00 00      push    $0x3e8
c00300b0:  eb 18                jmp     c00300ca <asm_do_irq>
```

(4). 与系统调用相似的，跳转到 asm_do_irq 之后压栈好所需内容之后以 TrapFrame 的地址为参数调用 irq_handle 函数，由于 irq = 0x3e8 = 1000，减去 1000 之后等于 0，触发 panic 中断。注释掉 panic 之后，继续执行返回至 asm_do_irq，执行收到时钟中断时将要执行的那一条指令。

```

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
        .");

        struct IRQ_t *f = handles[irq_id];

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

```

与系统调用的相同之处在于调用 `raise_intr()` 之后的查询中断表，跳转至 kernel 准备好的异常处理程序然后跳转至 `asm_do_irq`，然后调用 `irq_handle()` 函数，这些过程是相同的。

与系统调用的不同之处在于

- (1). 系统调用所保存的断点是 `eip + 2`，处理完系统调用之后从下条指令开始执行，而时钟中断所保存的系统断点是 `eip`，CPU 处理完时钟中断之后继续执行指令。
- (2). 所对应的异常处理程序与 `irq` 自然也不同。

PA4-2

1. 注册监听键盘事件是怎样完成的？

- (1). 注册监听键盘事件是应用程序即 `echo.c` 运行时，调用 `add_irq_handler` 来注册对键盘事件的监听。


```
int main() {
    // register for keyboard events
    add_irq_handler(1, keyboard_event_handler);
    while(1) asm volatile("hlt");
    return 0;
}
```

(2). 在 add_irq_handler 函数调用过程中，执行汇编代码 int 0x80，其中传入参数 TrapFrame 中 eax 的值为 0，ecx 即为对键盘事件进行监听的函数地址。

```
// register a handle of interrupt request in the Kernel
void add_irq_handler(int irq, void *handler) {
    // refer to kernel/src/syscall/do_syscall.c to understand what has happened
    asm volatile("int $0x80" : : "a"(0), "b"(irq), "c"(handler));
}
```

(3). 在处理该系统调用的过程中，相似的跳转到 kernel 准备好的异常处理程序然后跳转到 asm_do_irq，并先后调用 irq_handle()及 do_syscall()函数。而 do_syscall()中对该系统调用进行处理时，eax = 0，调用 kernel 中的 add_irq_handle 函数。

```
void do_syscall(TrapFrame *tf) {
    switch(tf->eax) {
        case 0:
            cli();
            add_irq_handle(tf->ebx, (void*)tf->ecx);
            sti();
            break;
        case SYS_brk: sys_brk(tf); break;
        case SYS_open: sys_open(tf); break;
        case SYS_read: sys_read(tf); break;
        case SYS_write: sys_write(tf); break;
        case SYS_lseek: sys_lseek(tf); break;
        case SYS_close: sys_close(tf); break;
        default: panic("Unhandled system call: id = %d", tf->eax);
    }
}
```

在 add_irq_handle 函数中将对应的中断号与处理函数的地址插入 handles 链表中，完成对监听键盘事件的监听。

```

void
add_irq_handle(int irq, void (*func)(void) ) {
    assert(irq < NR_HARD_INTR);
    assert(handle_count <= NR_IRQ_HANDLE);

    struct IRQ_t *ptr;
    ptr = &handle_pool[handle_count++]; /* get a free handler */
    ptr->routine = func;
    ptr->next = handles[irq]; /* insert into the linked list */
    handles[irq] = ptr;
}

```

2. 从键盘按下一个键到控制台输出对应的字符，系统的执行过程是什么？如果涉及与之前报告重复的内容，简单引用之前的内容即可。

(1). 首先由 NEMU_SDL_Thread 线程捕获关于键盘的按下和抬起两个事件，并将检测的键盘码传递给 keyboard_down()和 keyboard_up()函数。

(2). Keyboard_down()和 Keyboard_up()函数记录键盘扫描码之后，调用 i8259_raise_intr()发出中断请求。

```

// called by the nemu_sdl_thread on detecting a key down event
void keyboard_down(uint32_t sym) {
    // put the scan code into the buffer
    scan_code_buf = sym2scancode[sym >> 8][sym & 0xff];
    // issue an interrupt
    i8259_raise_intr(KEYBOARD_IRQ);
    // maybe the kernel will be interested and come to read on the data port
}

// called by the nemu_sdl_thread on detecting a key up event
void keyboard_up(uint32_t sym) {
    // put the scan code into the buffer
    scan_code_buf = sym2scancode[sym >> 8][sym & 0xff] | 0x80;
    // issue an interrupt
    i8259_raise_intr(KEYBOARD_IRQ);
    // maybe the kernel will be interested and come to read on the data port
}

```

(3). CPU 处理该中断过程中查找是否有对键盘事件的响应程序，而我们已经注册了键盘按下或抬起的响应程序 keyboard_event_handler()，因此调用 keyboard_event_handler()来处理键盘事件。

(4). keyboard_event_handler()从对应输入端口读入扫描码之后，将其转化为 ASCII 码并调用 writec()打印到显示屏上。至于如何将字符打印到显示屏上在之

前问题已有说明。

```
// the keyboard event handler, called when an keyboard interrupt is fired
void keyboard_event_handler() {

    uint8_t key_pressed = in_byte(0x60);

    // translate scan code to ASCII
    char c = translate_key(key_pressed);
    if(c > 0) {
        // can you now fully understand Fig. 8.3 on pg. 317 of the text book?
        printf(c);
    }
}
```