PA4 实验报告

周宇航　161220182

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

```
#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;
}
```

（1）在 hello-inline 中，用通用寄存器来传递参数，eax 寄存器为 4，表明系统调用号为 4，ebx 寄存器为 1，表明标准输出，str = “Hello, world!\n"，赋给 ecx 用于输出，而 edx 为 14 表明输出的字符串长度为 14。然后通过指令 int $0x80 进行系统调用。

```
#include "cpu/intr.h"

make_instr_func(int_)
{
    OPERAND r;
    r.type = OPR_IMM;
    r.addr = cpu.eip + 1;
    r.data_size = 8;
    operand_read(&r);
    uint8_t intr_no = r.val;
    raise_sw_intr(intr_no);
    return 0;
}
```

（2）而 int 指令通过调用 raise_sw_intr(intr_no)函数开始运行，intr_no 为异常和中断类型号，此处 intr_no = 0x80。而在 raise_sw_intr 函数中，又调用 raise_intr(intr_no)函数来进行处理。

```
    uint8_t index = intr_no;
    // Push EFLAGS, CS, and EIP
    uint32_t val;
    cpu.esp -= data_size / 8;
    val = cpu.eflags.val;
    vaddr_write(cpu.esp, 0, data_size / 8, val);
    cpu.esp -= data_size / 8;
    val = cpu.cs.val;
    vaddr_write(cpu.esp, 0, data_size / 8, val);
    cpu.esp -= data_size / 8;
    val = cpu.eip;
    vaddr_write(cpu.esp, 0, data_size / 8, val);
    // Find the IDT entry using 'intr_no'
    GateDesc gatedesc;
    gatedesc.val[0] = laddr_read(cpu.idtr.base + 8 * index, 4);
    gatedesc.val[1] = laddr_read(cpu.idtr.base + 8 * index + 4, 4);
    // Clear IF if it is an interrupt
    if((gatedesc.type & 0xf) == 0xe)
    {
        cpu.eflags.IF = 0;
    }
    // Set EIP to the entry of the interrupt handler
    uint32_t offset = gatedesc.offset_15_0 + (gatedesc.offset_31_16 << 16);
    cpu.eip = offset;
    cpu.cs.val = gatedesc.selector;
#endif
```

（3）raise_intr(intr_no)函数中，将 eflags、cs、eip 等进行压栈。然后通过 intr_no 作为 idtr 表的序号搜索到 idtr 表项 gatedesc 读取出来，通过判断其类型确定是中断还是异常（在 pa4.1 中只须为异常，但在仙剑中为外部中断要进行判断），改变其 IF。将表项中 offset 处理后赋值给 eip(eip = 0xc00300a0，即为 vecsys 函数)，selector 赋值给 cs。

```
c003009e:   eb 2a                   jmp    c00300ca <asm_do_irq>

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

（4）vecsys 函数中调用 asm_do_irq。

```
c00300ca <asm_do_irq>:
c00300ca:   60                      pusha
c00300cb:   54                      push   %esp
c00300cc:   e8 ef 15 00 00          call   c00316c0 <irq_handle>
c00300d1:   83 c4 04                add    $0x4,%esp
c00300d4:   61                      popa
c00300d5:   83 c4 08                add    $0x8,%esp
c00300d8:   cf                      iret
```

```
    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);
```

（5）asm_do_irq 通过 pusha 将通用寄存器的值均压栈，然后调用 irq_handle 函数进行处理，在里面又因为 tf = 0x80，所以调用 do_syscall，

```
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）而后在 do_syscall 中因为系统调用号为 4，所以调用其中 sys_write(),

```
static void sys_write(TrapFrame *tf) {
    tf->eax = fs_write(tf->ebx, (void*)tf->ecx, tf->edx);
}
```

（7）而在 sys_write 中调用 fs_write()，通过此函数将 str 标准输出。

```
size_t fs_write(int fd, void *buf, size_t len) {
    assert(fd <= 2);
#ifdef HAS_DEVICE_SERIAL
    int i;
    extern void serial_printc(char);
    for(i = 0; i < len; i ++) {
        serial_printc( ((char *)buf)[i] );
    }
#else
    asm volatile(".byte 0x82" : "=a" (len) : "a"(4), "b"(fd), "c"(buf), "d"(len));
#endif

    return len;
}
```

（8）执行完之后，则依次返回 do_syscall、irq_handle、asm_do_irq 中，然后继续执行 popa 指令，将各寄存器出栈，

```
make_instr_func(iret)
{
    cpu.eip = vaddr_read(cpu.esp, 2, data_size / 8);
    cpu.esp += data_size / 8;
    cpu.cs.val = vaddr_read(cpu.esp, 2, data_size / 8);
    cpu.esp += data_size / 8;
    cpu.eflags.val = vaddr_read(cpu.esp, 2, data_size / 8);
    cpu.esp += data_size / 8;
    print_asm_0("iret", "", 1);
    return 0;
}
```

（9）通过 iret 指令将 eip、cs、eflags 等进行出栈。eip 为返回之前运行的下一条指令的地址，完成过后即开始运行到 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。

```
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;
```

push %esp 中的 esp 即 TrapFrame *tf，其值为 tf 中第一个成员 edi 的地址，以此来访问 tf 中的所有成员

如图中表所示：

……	
eflags	
cs	
eip	
0x0	(error_code)
0x80	(irq)
eax	
ecx	
edx	
ebx	
esp	0x80 地址
ebp	
esi	
edi	
esp	edi 地址
返回地址	现 esp

从 eflags 到 edi 部分为 TrapFrame 对应内容。

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。

响应时钟中断的过程如下：

（1） 从 timer.c 中 timer_intr 函数中发出中断信号，调用 i8259_raise_intr 函数，同时对 intr_no 进行处理，加上 IRQ_BASE = 32，并且将 cpu.intr 置 1。而在 cpu.c 中的 exec 函数中，每次执行都要调用 do_intr 函数。而此时如果 cpu.intr 和 cpu.eflags.IF 均为 1，则调用 raise_intr(intr_no)函数开始处理。

（2） raise_intr(intr_no)函数中，将 eflags、cs、eip 等进行压栈。然后通过 intr_no 作为 idtr 表的序号搜索到 idtr 表项 gatedesc 读取出来，通过判断其类型确定是中断还是异常（在 pa4.1 中只须为异常，但在仙剑中为外部中断要进行判断），改变

其 IF。将表项中 offset 处理后赋值给 eip(此时 eip = 0xc00300a9，即为 irq0 函数)，selector 赋值给 cs。

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

(3) 在 irq0 函数中将 0x3e8 入栈，然后调用 asm_do_irq。

```
  } 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 pan
gured 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;
      }
```

(4) asm_do_irq 通过 pusha 将通用寄存器的值均压栈，然后调用 irq_handle 函数进行处理，在里面又因为 tf = 0x3e8 即大于 1000，所以 irq_id = 0，触发 panic，然后执行 popa 和 iret，过程与上面基本相同。

所以综合来说，之前第一步是通过 int 指令进行陷阱进行系统调用，返回地址为原 eip +2，而现在是通过时钟发出中断信号，用模拟中断引脚来系统调用，返回地址为 eip。而之前的中断处理程序为 vecsys，irq = 0x80，现在是 irq0 来处理，irq = 0x3e8，但都是调用 asm_do_irq 来处理。但是在处理过程中，前者调用 do_syscall，后者直接处理 panic。

4-2.3.3 完成键盘的模拟

1、注册监听键盘事件是怎么完成的?

(1) 在 echo 测试用例中，通过调用 add_irq_handler 函数来处理注册键盘事件。

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

(2) add_irq_handler 函数中又通过 int $0x80 指令完成系统调用，按照上面系统调用的步骤，然后调用 do_syscall().

```
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);
    }
```

(3) 而 tf->eax = 0，所以调用 add_irq_handle()，里面 tf->ebx = irq = 1，tf-ecx = key_event_handle.

```
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;
}
```

(4) add_irq_handle 中，将 func = keyboard_event_handler 插入到监听事件的链表中去。

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) 对于键盘事件被 NEMU_SDL_Thread 线程捕获，将捕获并检测到的键盘码 SDL_DOWN 和 SDL_UP 作为 keyboard_down 和 keyboard_up 等函数。

(2) 而在键盘 keyboard_down 和 keyboard_up 缓存扫描码, 然后通过 i8259_raise_intr 函数用中断请求方式将按键抬起事件用中断请求号 1 发送给 CPU。

(3) 在 CPU 收到中断请求号后，调用 kernel 的中断响应程序，检测到键盘事件后即响应，调用注册响应函数来调用相应的函数。

(4) 注册响应函数 keyboard_event_handler，通过 in 指令从键盘数据端口读取扫描码，将其转换为 ASCII 码，调用 printc 函数。

(5) printc 函数调用 writec 函数

(6) writec()函数位于 echo.c 文件中，writec()通过 int $0x80 指令实现系统调用 do_syscall()，具体过程如上面的系统调用相似。

(7) 在 do_syscall()中, 由 writec()各参数可知, tf->eax = SYS_write(用于调用 sys_write 函数)，tf->ebx = fd = 1（调用标准输出），tf->ecx 为字符 str，tf->edx = 1（字符长度）。

(8) sys_write()函数中调用 fs_write()函数。

(9) 在 fs_write()函数中又继续调用 serial_printc 函数进行串口打印。

(10) serial_printc()函数中又调用 out_byte()函数进行系统输出。

(11) out_byte()函数中又主要是通过 out 指令来实现。

（12） 在 out 指令中我们调用 pio_write()函数，而在 pio_write()函数中调用 write_io_port()函数在 io_port 数组进行数据写入，并以 port 为索引去查找 pio_handler_table 数组并进行调用相应的处理函数 handler_serial()，然后在控制台上将数据打印出来。

如图：out 指令的实现

```
make_instr_func(out_v)
{
    uint16_t port = cpu.gpr[2]._16;
    if(data_size == 32)
    {
        pio_write(port, 4, cpu.eax);
    }
    else
    {
        pio_write(port, 2, cpu.gpr[0]._16);
    }
    return 1;
}

make_instr_func(out_b)
{
    uint16_t port = cpu.gpr[2]._16;
    uint32_t al = cpu.gpr[0]._8[0];
    pio_write(port, 1, al);
    return 1;
}
```

PA 所遇到的问题：

1、call_near_indirect 指令中 eip 变化错误，导致在模拟硬盘部分出现缺页。

2、Cache 跨页问题，在解决完模拟硬盘的指令问题后发现，开启 cache 后，在 matrix mul 测试用例中，hit badtrap，所以 cache 出现了问题，在 cache 中进行处理跨页之后，即解决问题。

```
if(addr + len >= 64)
    memcpy(&ret, hw_mem + paddr, len);
else
{
```

3、raise_intr()函数中，由于没有判断中断描述符是否是中断(TYPE ==14)，就直接把 cpu.eflags.IF 置 0 时，所以导致在 fs_read 函数 ide_read 函数无法结束，一直停留在 wait_intr 函数中。

4、仙剑出现马赛克，刚开始发现是 test 指令没有对所有操作数进行符号拓展，所以导致最终出现马赛克，但是只是修改 test 还不够，在运行到李大娘出门之后，就出现缺页，然后将所有宏定义的函数均进行符号拓展处理，缺页问题得以解决。