PA4-1.3.1 中的问题回答：

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：当指令执行到 int 时，会首先获取其后紧跟的中断/异常号。获取到该编号后，将这个编号作为参数，传递给框架代码中提供的函数 raise_sw_intr(uint8_t Num) 中 。

Raise_sw_intr（）函数首先会将 eip 的值修改为下一条指令的 eip，紧接着再将接受到的参数——也就是中断/异常的编号再传递给在其内部调用的函数，raise_intr();

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

Raise_intr()函数会开始正式处理中断，从前期准备：保存三个寄存器的值；获取中断处理程序的地址：从 cpu 的 idtr 寄存器中加上中断\异常号获得该地址；最后修改 eip 到这个地址，正式执行中断\异常处理程序。

就以 int 0x80 为例， 转入处理程序时，首先根据查表进入这一片段：

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

注意这里也将中断和异常号压栈。转入的新函数如下：

```
c00300ca <asm_do_irq>:
c00300ca:   60                      pusha
c00300cb:   54                      push   %esp
c00300cc:   e8 5e 0b 00 00          call   c0030c2f <irq_handle>
c00300d1:   83 c4 04                add    $0x4,%esp
c00300d4:   61                      popa
c00300d5:   83 c4 08                add    $0x8,%esp
c00300d8:   cf                      iret
```

这里 push %esp 就是将其作为参数传递给 irq_handle 函数，正确执行 irq_handle 函数后，执行 iret 之后就会出现“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：如上一问所说，push %esp 实际就是在向 irq_handle 传递参数，即参数 tf；

在 call 之前的栈如下：

中断时的 eflags
中断时的 cs
中断位置处下一条指令的 eip
$0（从汇编可知）
中断/异常号（在上一问中就是 80）
原 eax 的值
原 ecx 的值

原 edx 的值
原 ebx 的值
原 esp 的值（该位置栈的地址）
原 ebp 的值
原 esi 的值
原 edi 的值
当前 esp-4 （当前 esp 也指向这里）

上面画出的栈中除了最底一层，都是 TrapFrame 的内容。

PA4-1.3.2 的问题回答

2、详细描述 NEMU 和 kernel 相应始终中断的过程和之前的系统调用的过程的不同之处在哪里？相同的地方又在哪里？通过文字或画图的方式叙述。

答：首先，二者性质不同，一个是内部异常，一个是外部中断。所以其中断/异常号自然是不一样的。并且二者启用的中断/异常处理程序不同。但是二者在遭遇异常/中断至调用处理程序之间的步骤是一样的：首先二者都会保存当前信息，eip，eflags，cs， 各个通用寄存器的值等等信息，然后再查表转入执行处理程序，查表的方式也是相同的。

PA4-2.3.3 问题回答

1、注册监听键盘事件是怎么完成的？

答:先看下图：

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

这里首先通过中断的方式来进行键盘输入的判断，进入keyboard_event_handler（）函数，该函数的第一个参数为 1，恰好对应键盘的中断号为 1。

转入执行 keyboard_event_handler（）

```
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.
        printc(c);
    }

}
```

顾名思义，in_byte（）函数就是通过指定端口获得数据返回，这里传递的参数就恰好是键盘的端口 0x60，通过这个端口就可以将按下的键传递给变量

key_pressed（具替按下了那个键，传递了什么扫描码由之前 guide 中提到的 SDL 线程来捕捉）。

在获得对应的扫描码后，还不能直接将其打印在屏幕上，需要将其转换为 ASCII 码才可以通过屏幕输出。具替的转换规则就由函数 translate_key（）来实现完成。

```
static int letter_code[] = {
    30, 48, 46, 32, 18, 33, 34, 35, 23, 36,
    37, 38, 50, 49, 24, 25, 16, 19, 31, 20,
    22, 47, 17, 45, 21, 44
};

char translate_key(int scan_code) {
    int i;
    for (i = 0; i < 26; i ++) {
        if (letter_code[i] == scan_code) {
            return i+0x41;
        }
    }
    return 0;
}
```

（注意由于 letter_code[]数组提供的扫描码只有字母的扫描码，所以在 echo 执行时，按其他键并不能输出打印。）

在转换完成后，就可以实现了在屏幕上打印出按下的字母的大写。

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。

答：系统通过中断，转入中断处理程序的方式来显示键盘上输入的字符。具体的执行流程与之前 PA4-1 中叙述的类似，不再赘述。