4.1

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：当触发int $0x80指令之后，进入Int函数，随后进入raise\_intr函数

在raise\_intr()函数中，将标志寄存器，段寄存器 和 eip 压栈，之后读取 IDTR 中存放的 IDT 首地址，根据传入的intr\_no获取偏移量，从而获得中断处理程序的首地址。

跳转过程的函数的注册由kernel实现。

在kernel的do\_irq.S文件中，都定义了一些中断处理函数。注册这些中断处理函数之后，就可以在int的时候调用这些函数。（提前注册好了，只不过现在才用）

在kernel的irq\_handle.c 文件中，有一个irq\_handle函数，这个函数的参数就是指向push的寄存器的地址。在do\_irq.S中，都有一条push %esp的语句，这个就是参数，指向了我们pusha之后的所有寄存器的地址。

随后有一个链表的结构，代表着一些函数（使用了函数指针）。

使用add\_irq\_handle函数，将一些处理函数加入到handles数组里。

随后一个接一个的调用处理函数（**如果非0x80**）

**如果是0x80**，就是调用了do\_syscall函数，根据eax的内容判断要执行什么功能。最后执行完毕返回。

1. 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。

|  |
| --- |
| eax |
| ecx |
| edx |
| ebx |
| esp |
| ebp |
| esi |
| edi |
| (puah esp之后)旧esp(指向edi) |

Push %esp的作用就是把8个通用寄存器的地址当作参数，传递给了iqr\_handle函数。（pusha之后，这8个寄存器地址连续，edi在最后）

TrapFrame对应的就是这段栈的内容（由一个旧esp当作参数指向的）

4.1.3

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。

不同之处：响应时钟中断是通过检查cpu.intr中断引脚来实现的，当中断引脚被置为 1 时系统进入中断处理程，属于外部中断。之前的是用int指令实现，属于软中断。外部中断，保存的eip的当前指令的地址；int陷阱指令的中断，保存的eip的下一条指令的地址。

相同之处：都是通过中断号确定中断服务程序，后续处理和int指令一样

4.2

1. 注册监听键盘事件是怎么完成的？

使用了make\_pio\_handle宏进行的注册。首先定义了关于键盘的端口消息处理函数(使用的是端口io而非内存io) handler\_keyboard\_data。

我们本身有端口数组 uint8\_t io\_port[IO\_PORT\_SPACE];

然后定义了端口->函数映射关系的数组pio\_handler\_table

然后把它添加进于所有端口处理函数的数组 pio\_handler\_table里，并指定了端口号（KEYBOARD\_DATA\_PORT，也是一个宏）。

然后当键盘消息进入的时候，就会通过这个端口，调用对应的函数（就是在pio\_write和pio\_read函数）

1. 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

按下一个键，然后键盘监听函数得到扫描码(和数电实验的类似)，然后进入echo函数，进行读取并转化，解析成ASCII码，最后对写的串口进行写入，并进行pio\_write(对写的端口进行pio\_write)，就可以显示在控制台上了。

此处有个问题，就是如果不做控制台输出，开启了keyboard会怎么样。。答案是不会进入echo(皮一下)