§4-1.3.1 通过自陷实现系统调用

在实验报告中还需要回答如下问题：  
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；

解答：从int $0x80开始，系统调用opecode.c中的into函数，找到对应的中断号0x80，将中断号当作参数传入raise\_sw\_int()函数中，在这个函数中eip+2后继续调用raise\_intr(uint8\_t intr\_no)参数是中断号。在这里先push eflags、CS、eip现场信息，通过idtr根据参数intr\_no找到对应的门描述符，根据门描述符更新CS和新的eip的值，结束这个函数后，自动进入了内核态的代码段，按照新的eip继续执行。

然后进入do\_irq.S的代码，pusha保存现场信息，push esp保存TrapFrame，传入参数继续调用irq\_handle进行处理。调用do\_syscall（）函数系统调用，根据传入的信息输出hello world

接着完成do\_irq.S的代码，返回原本代码，执行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。

解答：push %esp起到传参的作用。

在 call irq\_handle 之前，系统栈的内容如下：

|  |
| --- |
| … |
| Eflags.val |
| Cs.val |
| cpu.eip |
| 0 |
| 0x80 |
| Eax |
| Ecx |
| Edx |
| Ebp |
| Esi |
| Edi |
| （Esp指向） |

跳转到 asm\_do\_irq 时. 在 asm\_do\_irq 中, 代码pushal将会把用户进程的通用寄存器保存到堆栈上, 这些寄存器的内容连同之前保存的错误码, irq , 以及硬件保存的EFLAGS, CS, EIP形成了TrapFrame的内容, 它记录了用户进程陷入内核时的状态,。

§4-1.3.2 响应时钟中断

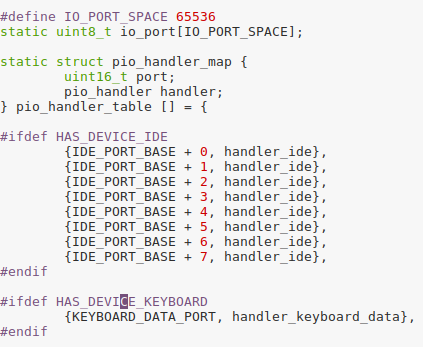
在实验报告中，需要回答：  
1. 详细描述 NEMU 和 Kernel 响应时钟中断的过程和先前的系统调用过程不同之处在哪里？相同的地方又在哪里？可以通过文字或画图的方式来完成。

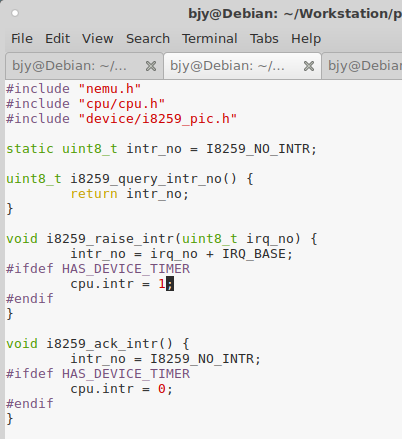
解答：不同之处在于系统调用时传入的intr\_no是由指令直接提供，而NEMU和Kerne需要中断引脚好+32。相同之处在于他们都是通过软中断，raise\_intr()函数来实现对时钟中断控制

4.2 在实验报告中， 结合代码详细描述：

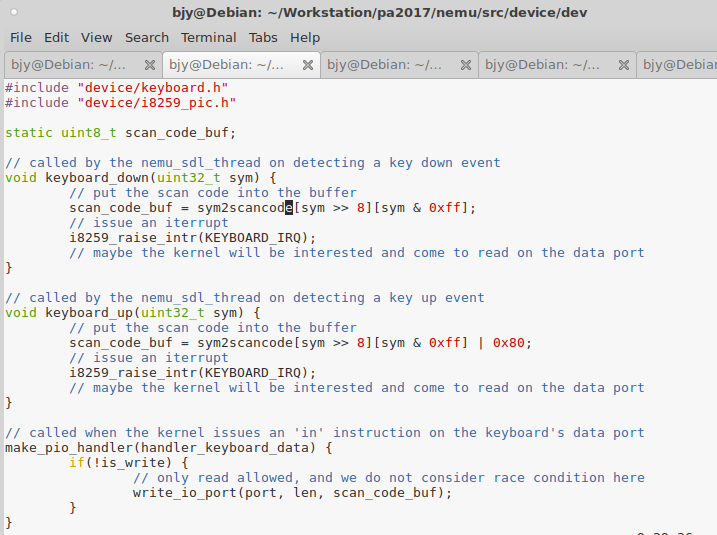
1.注册监听键盘事件是怎么完成的？

解答：注册相关的端口在pio\_handle\_table[]中，

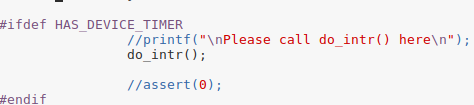




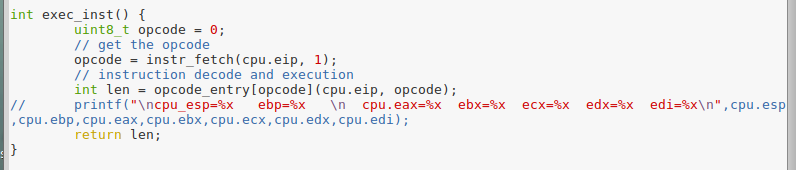
i8042初始化时会注册 0x60 处的端口作为数据寄存器,

每当用户敲下/释放按键时, 将会把键盘扫描码放入数据寄存器, 然后发起键盘中断, CPU收到中断后, 可以通过端口I/O访问数据寄存器, 获得键盘扫描码  
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。

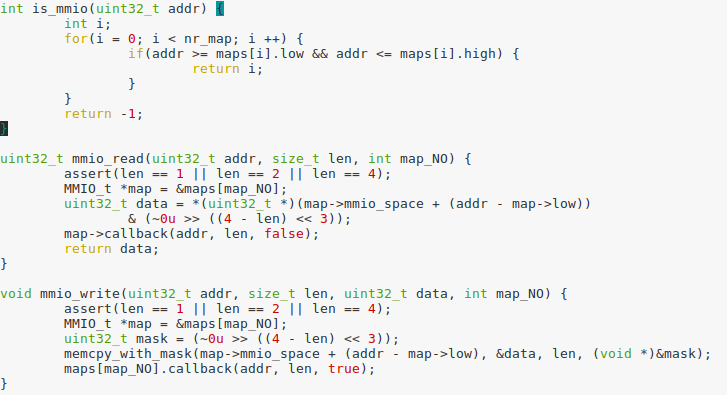
解答：



CPU exec()函数中 while 循环体的末尾，即在结束当前指令的执行后，调用do\_intr()函数，查看 EFLAGS 的 IF 位和 INTR 引脚，若是开中断状态且有中断到来，则向PIC 查询中断号。利用中断号调用在上一小节中实现的 raise\_intr()函数， 注意传参时使用的 irq\_no 是通过PIC 增加了 IRQ\_BASE 后得到的值。



然后系统继续上面一题的过程，对irq\_no进行判断，然后获取键盘内存的信息



调用相关函数获取信息后执行输出：

