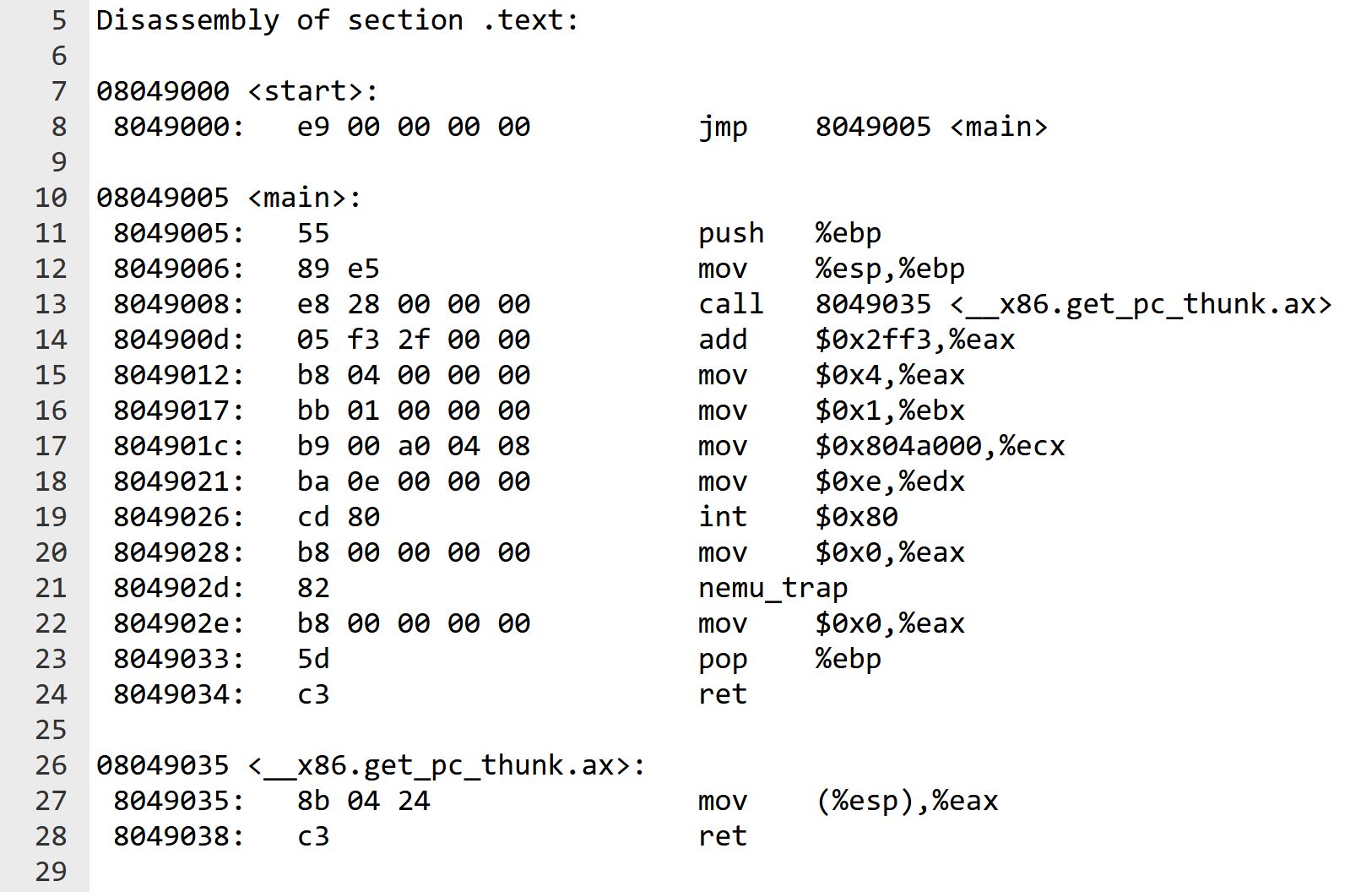
§4-1.3.1 通过自陷实现系统调用

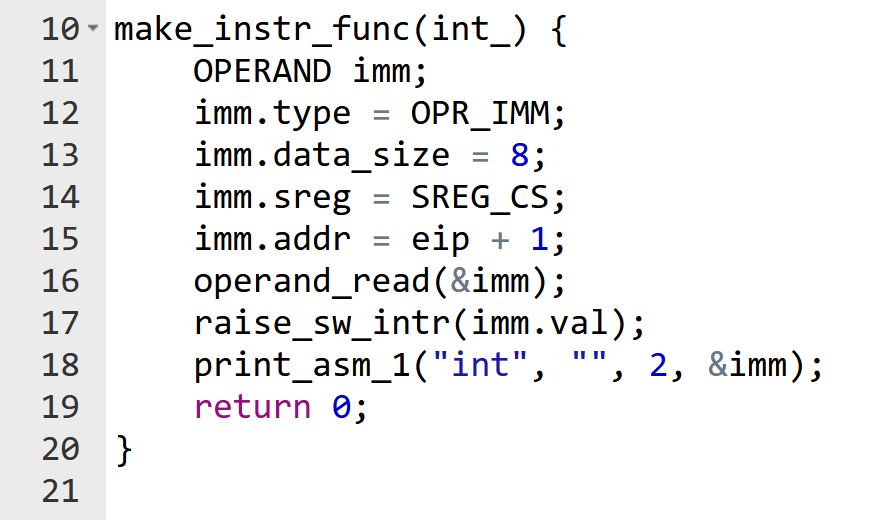
1. 详细描述从测试用例中int $0x80开始到HIT\_GOOD\_TRAP为止的详细系统行为（完整描述控制的转移过程，即相关函数的调用和关键参数传递过程），通过文字或画图的方式；

首先查看hello-inline的反汇编代码：



注意到int指令的参数为立即数0x80

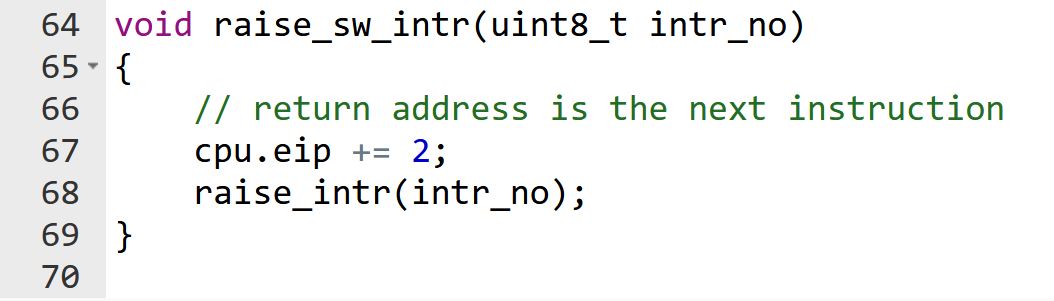
接着查看（好吧，编写）int指令代码：



根据手册，int指令获得对应的中断号后

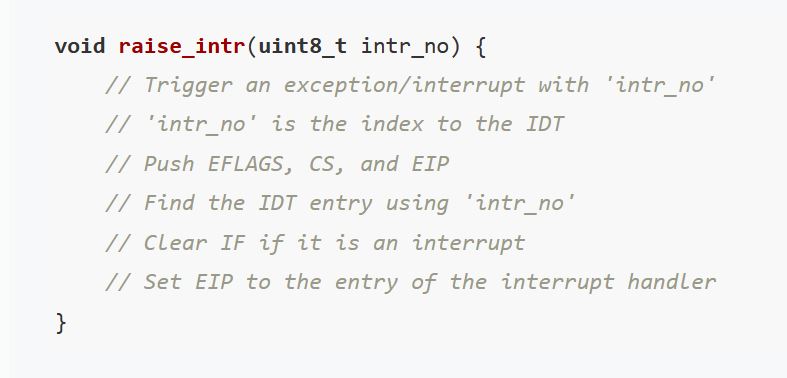
将中断号作为参数传给raise\_sw\_intr函数

接着查看raise\_sw\_intr代码：



可见raise\_sw\_intr函数将eip加2（转移至下一条指令的开始）

然后以传进来的中断号为参数，再调用raise\_intr函数



根据文档的说明，容易看出raise\_intr需要完成的工作是：

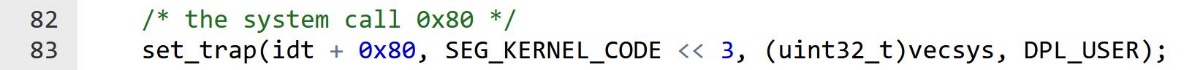
1）保护现场（把当前的eflags寄存器，cs寄存器和eip压栈）；

2）根据传入的中断号计算对应处理程序的线性地址；

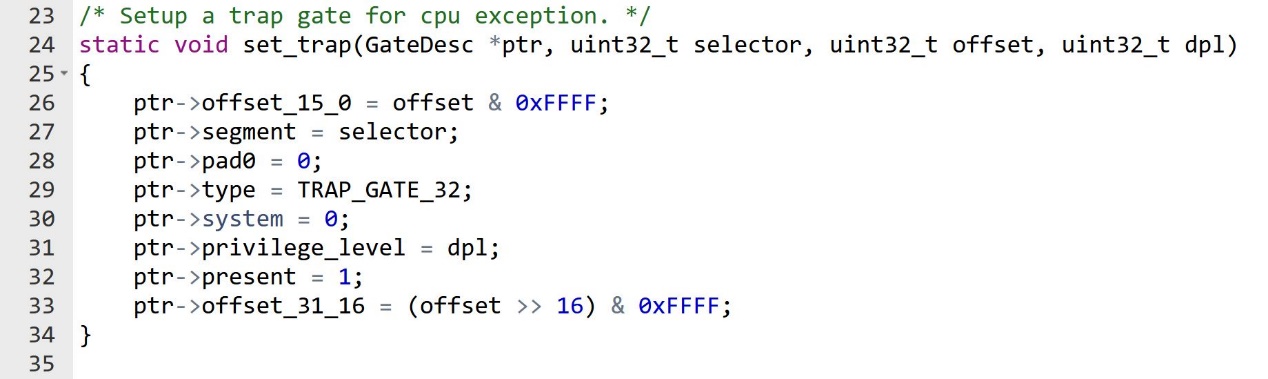
3）检查是否需要设置关中断；

4）对eip赋值来进行跳转。

接着查看init\_idt函数代码找到中断表的初始化：



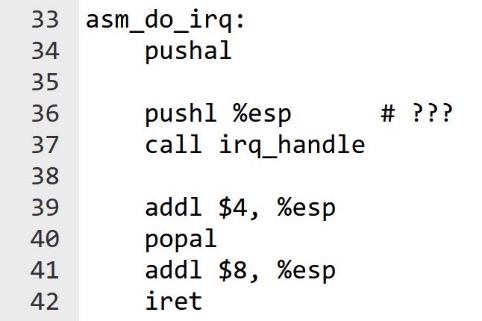
找到set\_trap函数，发现对应的是中断门的设置：



在do\_irq.S中可以找到set\_trap作为offset传入的函数vecsys：



发现它的作用是先后将0和0x80压栈，然后跳转到asm\_do\_irq



可见asm\_do\_irq完成的工作是：

1）调用pusha将所有通用寄存器压栈（保护现场）；

2）使用push %esp（利用栈帧准备调用参数）；

3）调用irq\_handle函数；

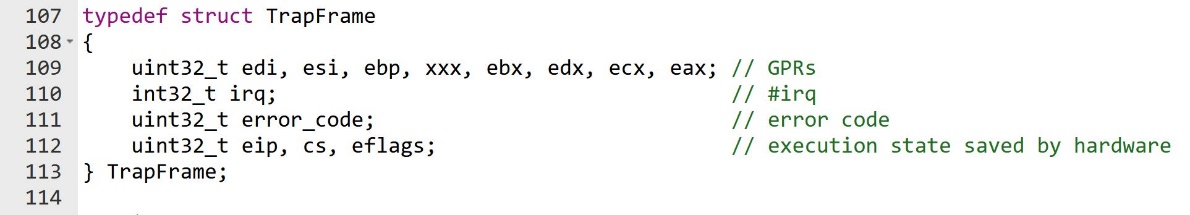
4）复原栈帧和调用popa恢复现场，iret跳出。

irq\_handle函数参数为指针变量TrapFrame，即最后压栈的esp。

irq\_handle对参数进行判断并处理中断。结束后跳转返回原程序。

2. 在描述过程中，回答kernel/src/irq/do\_irq.S中的push %esp起到了什么作用，画出在call irq\_handle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。

由1中分析可见，push %esp的作用就是传参数，调用pusha后，esp就是当前栈顶指针的地址，把这个地址压栈就是把pusha准备的数据的地址作为指针传参，函数irq\_handle的参数是TrapFrame类型的指针，找到TrapFrame的定义如下：



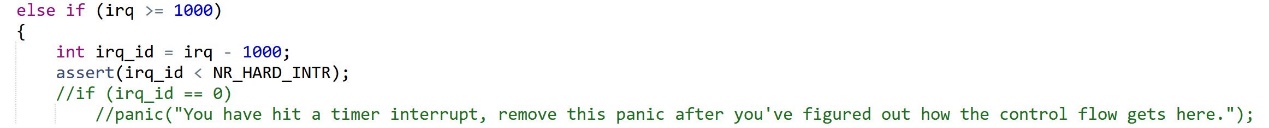
可见TrapFrame包含eflags，cs和eip信息，即raise\_intr中准备的三个数据，error\_code和irq分别对应vecsys中压栈的0和0x80，通用寄存器对应asm\_do\_irq中pusha的调用，再次证实push %esp就是传参给irq\_handle。call irq\_hadle之前系统栈的内容如下图：

|  |  |
| --- | --- |
| …… | …… |
| EFLAGS | TrapFrame |
| CS |
| EIP |
| 0 |
| 0x80 |
| EAX |
| ECX |
| EDX |
| EBX |
| XXX |
| EBP |
| ESI |
| EDI |
| ESP旧址 | <-- ESP |

§4-1.3.2 响应时钟中断

1. 详细描述NEMU和Kernel响应时钟中断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。

响应时钟中断和系统调用都保护现场、准备参数和调用irq\_handle，结束后都返回下一条指令继续执行。通过观察和分析代码可见前者调用irq\_handle进入的是这一分支：



区别在于响应时钟中断是由系统事件引起的，与现行指令无关，是异步执行的，而系统调用是由执行的现行指令引起的，是同步执行的。此外响应中断是直接读取中断向量表（此处添加了panic来表示），系统调用则通过调用函数do\_syscall间接跳转。