iJOS LAB3

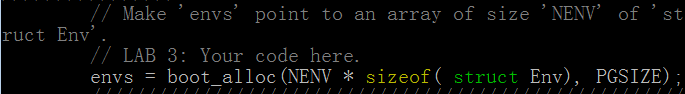
操作系统 实验3 实验记录

作者：卓达城

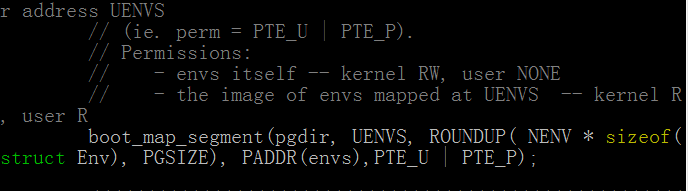
zhuodc@qq.com

首先为envs分配内存空间，然后映射到物理地址上。

具体实现如下：



kmap.c vm\_init() 分配空间



kmap.c vm\_init() 映射地址

这两个函数以后，内存的布局如下图所示。

线性地址：

0X00000000 KSTACKTOP-KSTKSIZE

4M

4M

4M

4M

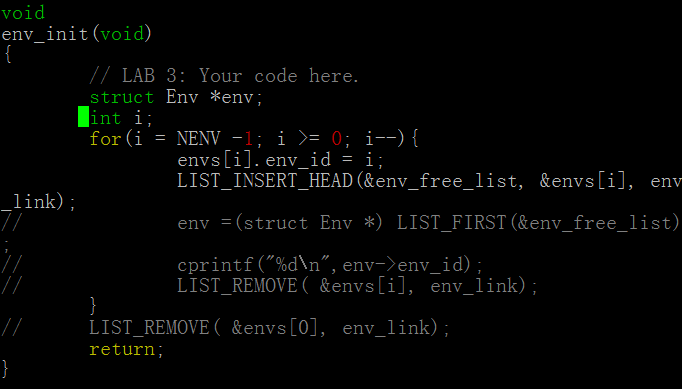
4M

UTOP UPAGES UVPT ULIM KSTACKTOP KERNBASE

UENVS

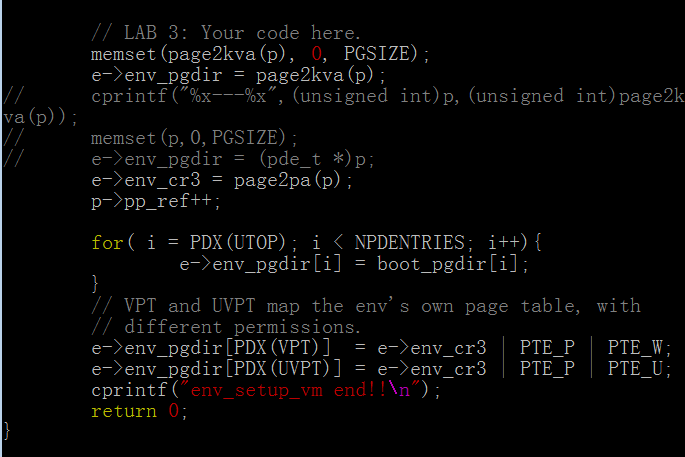
envs pages-..... bootstack-...... 0-256m

然后现在初始化环境空闲链表，我们想要修改env\_init函数，这里有一点要注意，应该把envs[0]放在链表的头部，方便下面调用。具体代码如下：



按照实验的要求，现在开始补充env\_setup\_vm函数：

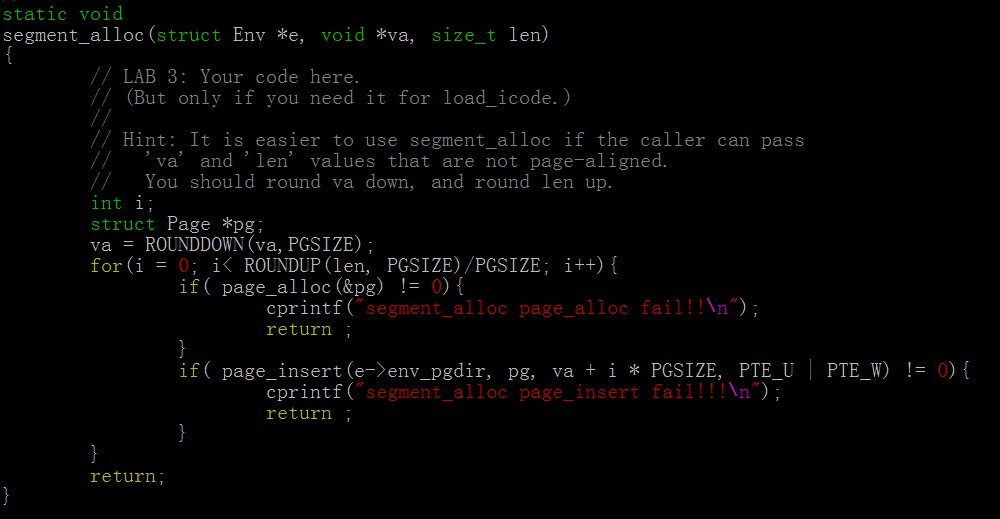
这个函数的主要作用是为新的环境设置页目录，请注意，这里每个新的环境都有一个属于自己的页目录，当设置好页目录之后，把内核的页目录（UTOP上面的线性地址）映射到新环境的页目录，以便新环境可以通过某些形式访问内核。



然后是segment\_alloc()函数

这个函数的主要作用是从地址va开始分配len字节的空间

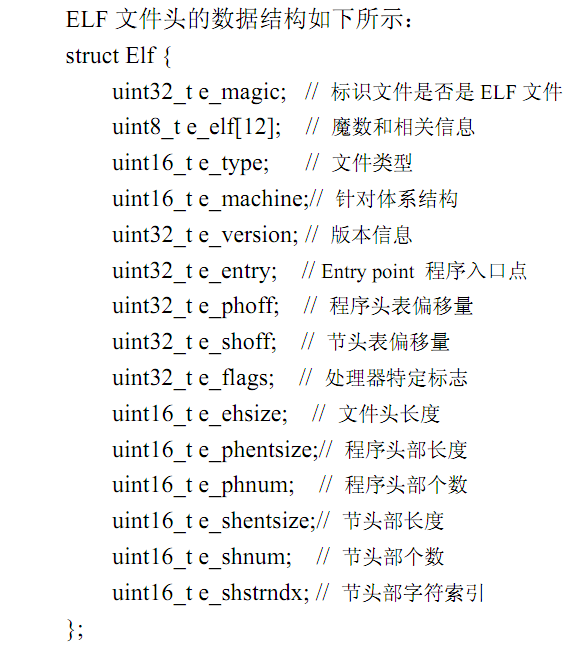
具体实现代码如下：

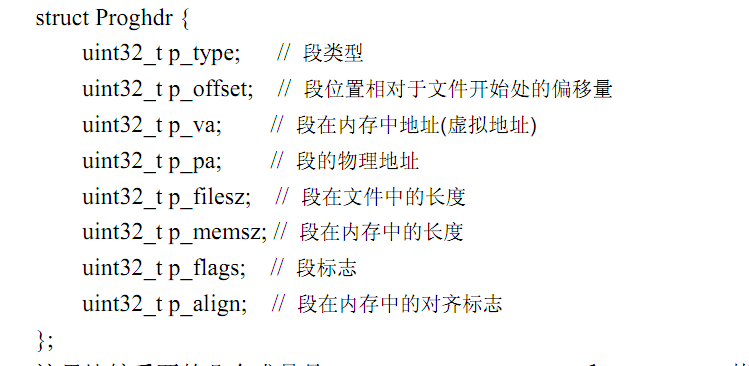


load\_icode()函数

这个函数是第一部分最难的一个函数，它的主要任务是把程序加载到新环境的虚拟地址上，参数提供了要加载的程序的首地址。由于要在新环境（进程）中加载所以必须把cr3换成新环境的cr3，加载完之后回到原来的cr3。

然后要明白函数里面循环部分的意义必须明白elf的结构，每个应用程序都有自己的elf结构，包括内核，所有应用程序都为操作系统提供elf结构，然后操作系统根据这个结构把程序加载到合适的地方，elf的结构如下图：

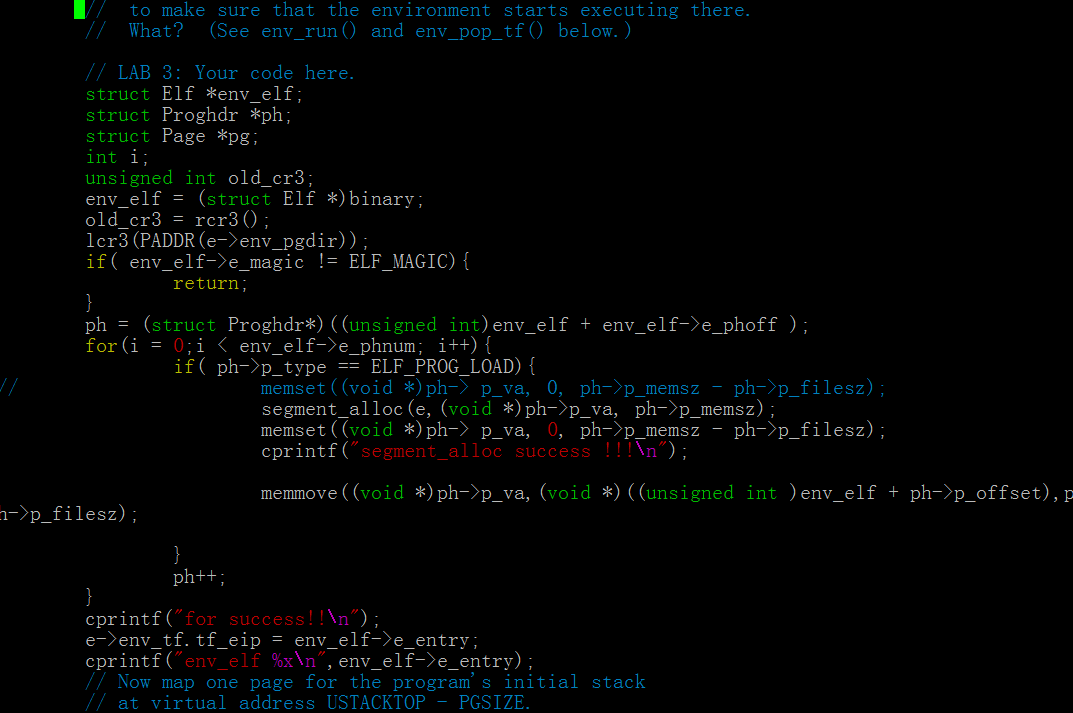


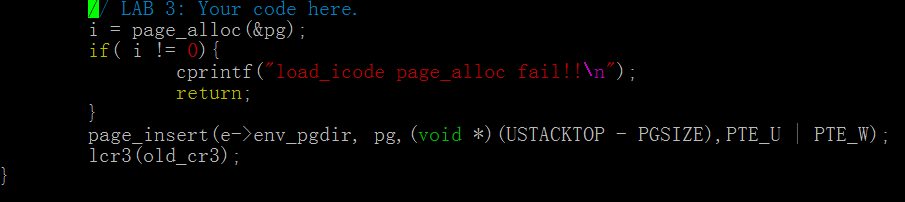


程序头表把程序分成好几个段，然后段的信息放在proghdr中，通过proghdr就可以把程序加载到指定的虚拟内存地址上。

具体信息可参照 第三章的书稿 《系统的启动和初始化》

具体实现代码如下：





方框部分注意了，这样写比较安全，这个上课听说是bss段有可能不全部为零，所以有了这样的写法，具体原因我暂时还不是很清楚。

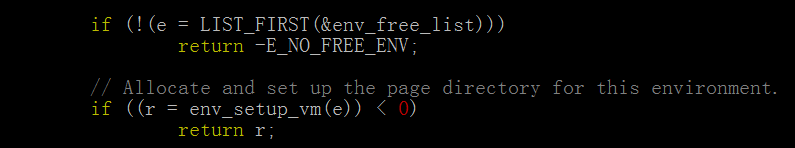
如果把方框里的前两句调换位置，这个程序就有问题了。+

写完这个函数之后，第一部分就差不多了。

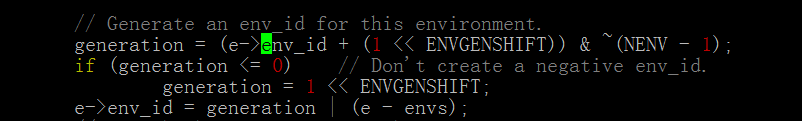
然后进入env\_create函数

这个函数里面调用了env\_alloc这个函数我们不用改，但是最好理解。

现在先解释这个函数：



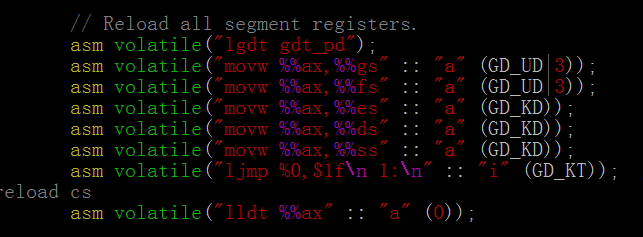
从空闲链表拿出一个新的env，然后初始化。



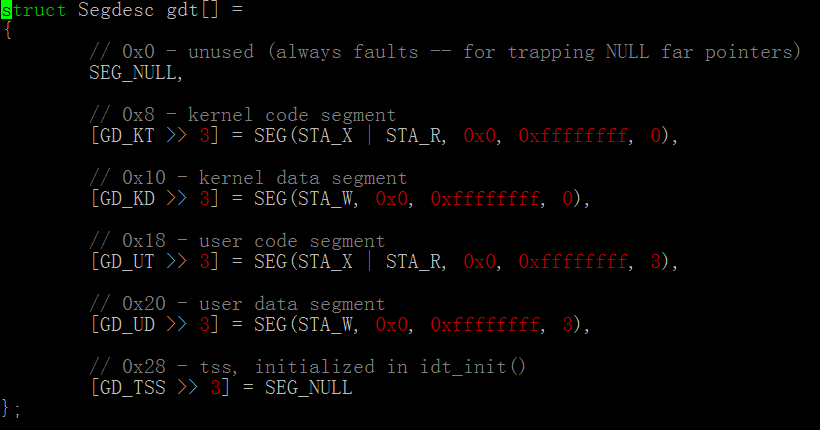
这几句是用来生成env\_id的，低10位势根据e - envs定的，NENV等于1024，刚好是10位，这样就确保每一个env\_id都不重复。如果前面把env\_id初始化成零的话，那么(1)env就会根据e - envs来定，具体就是0-1024，(2)而且在init.c里面还有一个就是env\_run(&env[0]),这就是上面的env\_init为什么要把第零项放第一的原因，其实更重要的是后一个原因，但是第一个原因也是有的（个人认为）。



这几句是设置好段寄存器，并把他们的特权级设置为3，至于gdt表在上面时候设置，请看下面（i386\_vm\_init)：



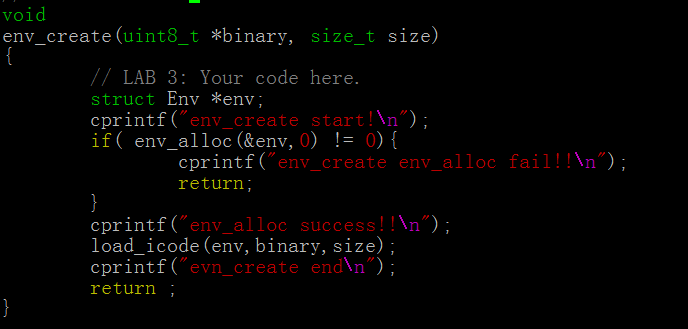
gdt表项在：



这里的段基地址都是一样，至于JOS是怎样使用段保护的，我暂时也不是很清楚。到后面应该会明白。

然后回到env\_create函数：

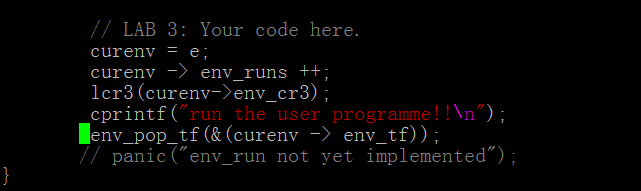
它把应用程序放到指定的虚拟内存上去。然后就完了。具体代码如下：



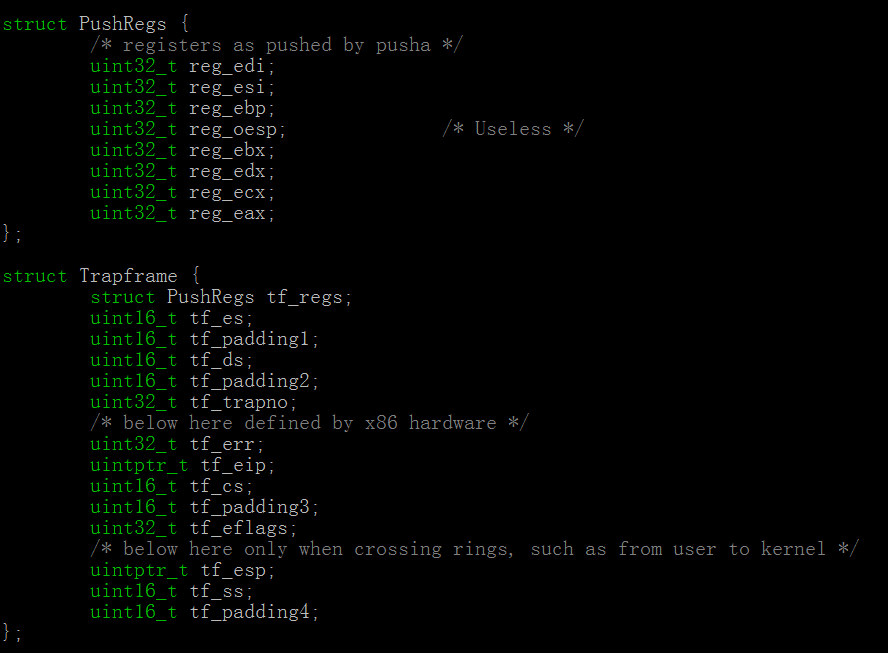
我们再来看这个宏

我们一定想知道user\_hello到底在哪里。这个也是暂时不知道，我这几个文件中都找不到它的信息。补充信息：后来我在kernel.sym中找到了user\_hello的信息。

然后就是env\_run，这个函数很简单，但是它调用的函数env\_pop\_tf比较有意思，先看看env\_run的具体实现，很简单：

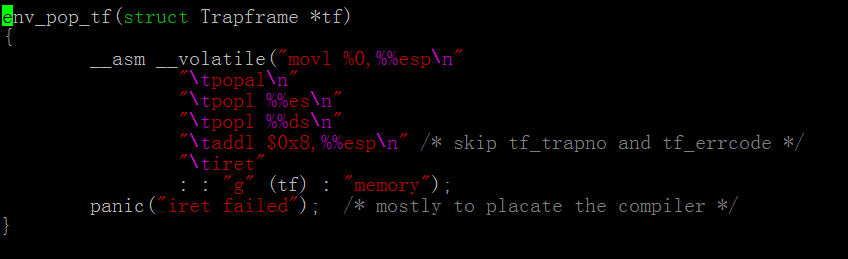


现在进入env\_pop\_tf函数，在看代码之前要先看看Trapframe的结构：



我们可以看到Trapframe的前面部分按照一定的顺序，就是pushal的顺把32为的寄存器排好，然后后面就是段寄存器，tf\_padding1是用来占位用的，因为段寄存器包含的隐藏信息是不能被程序员访问的。

然后我们看下面的代码：



先把esp设置成tf

然后我们看内存的结构

0x00 0xffffffff

tf\_err

cs

eip

trapno

ds

es

regs

esp

我们走到esp是读出数据，在esp += x的，然后iret是popl eip 然后popl cs的，这样我们就明白了，这个函数是要cpu切换到新的环境中，还有就是cpu调用iret是会切换到用户态的，这个是cpu的硬件规定的，但是貌似在进入新环境之前没有保存内核的运行状态，至于为什么？我现在还不知道，不过我相信以后就会真相大白的。也许是根本就不用保存内核当前的运行状态。

现在进入PART2

事先声明

要完美的完成PART2必须使用gcc 3.4.3或者以下的版本。这里包括G++ 等也有要求。

第一步：

先理解中断的原理：

由于中断，异常什么什么的，谁都说不清，这里只能说是个人观点，不一定正确。

第一：中断时有软件或者硬件发出的。处理完中断之后，程序能继续运行。

例如：

程序一要在屏幕输出一个字符，那么它就必须使用中断，修改0xB8000处内存的值，才能实现显示功能，显示完毕之后我们回到我们的应用程序。这是软件中断的例子。硬件中断也是一样，例如键盘中断（一般键盘中断跟其它中断的处理有区别，为了更好的理解，可以想象成程序在等待键盘输入，输入完后，程序继续执行）。

但是为什么我们程序需要用中断，而不直接去改呢？原因很简单，因为0xb8000一般由操作系统管理，用户程序不能执行。通过使用中断，操作系统可以调节和管理不同程序的显示，例如两个程序都要在同一个地方显示，那操作系统就会处理这一问题。但是如果不通过中断，两个程序就会冲突了。这里先说明一下，我们现在看到的图形界面不是在0xb8000处，而是在0xa0000处。

第二：异常，异常出现之后，程序一般停在发生异常的第一，除非经过特殊处理，否则程序不能继续运行。有很多高级语言都可以写异常处理，例如java。这里不探究。我们可以简单的认为，异常出现后程序停留在原来的位置，不执行。

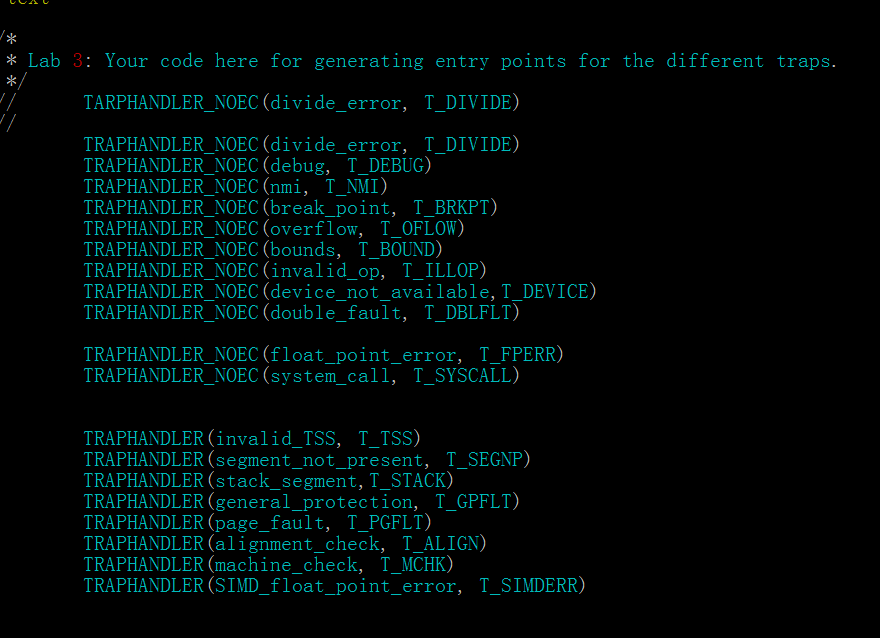
第三：中断和异常都可以理解成调用一个函数，只不过是中断调用完后运行函数的下一句代码，异常就听到触发异常的代码里头。

如果大家对call指令比较熟悉的话就可以理解以下比喻（实模式），中断就是一个call，异常也是一个call，不同点在于他们压入的ip不同。中断是下一条指令，异常是当前指令。

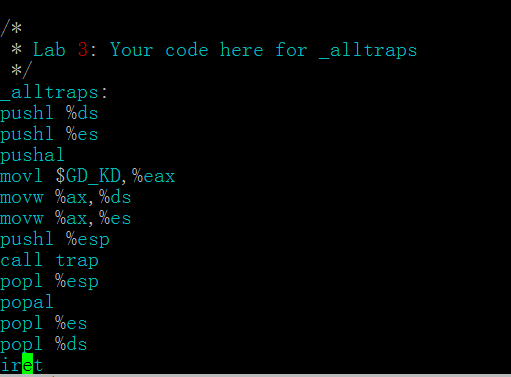
第四：中断时怎么发生的，先要有一个程序或者硬件引发一个中断，然后cpu根据中断号在IDT搜索，可以这样理解，IDT里面放的都是中断函数的符号，cpu会进行call，当然跟我们实际的call不一样，它的压栈方式不同。

第五：以上纯粹个人理解，不知道是否全对。

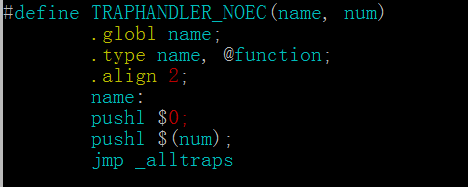
先完成函数定义：



完成\_alltraps



这里我们来研究研究这个宏：

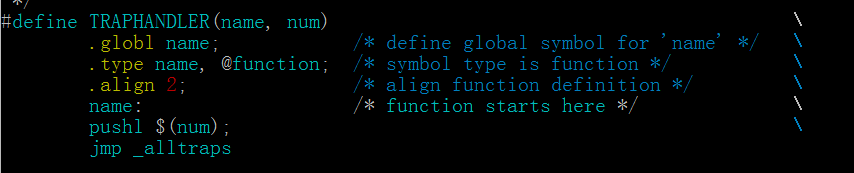


这个宏是什么意思呢？

.globl name是定义一个函数名为全局都可以引用的符号（symbol我只能这样翻译了）。

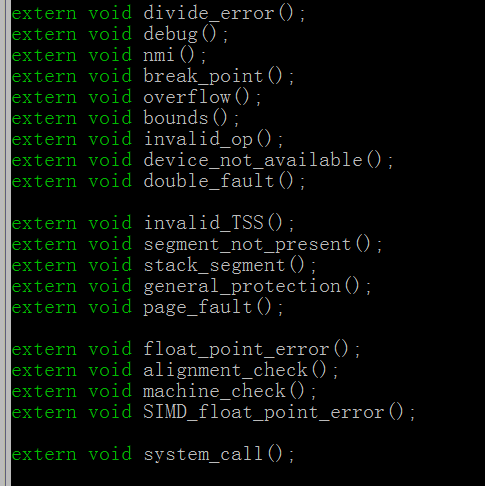
然后name:表明函数从这里开始，用过汇编就知道，我们可以call name的，这里的道理是一样的。

这个文件的作用就是声明所有中断函数，然后发生中断以后都跳入\_alltraps函数，这个函数的作用是按照Trapframe结构压栈。

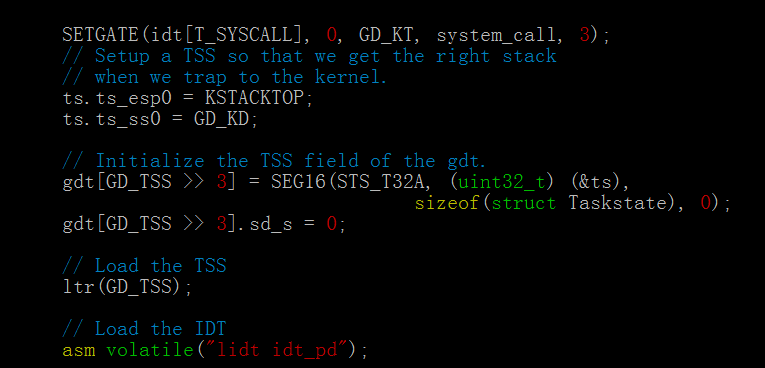
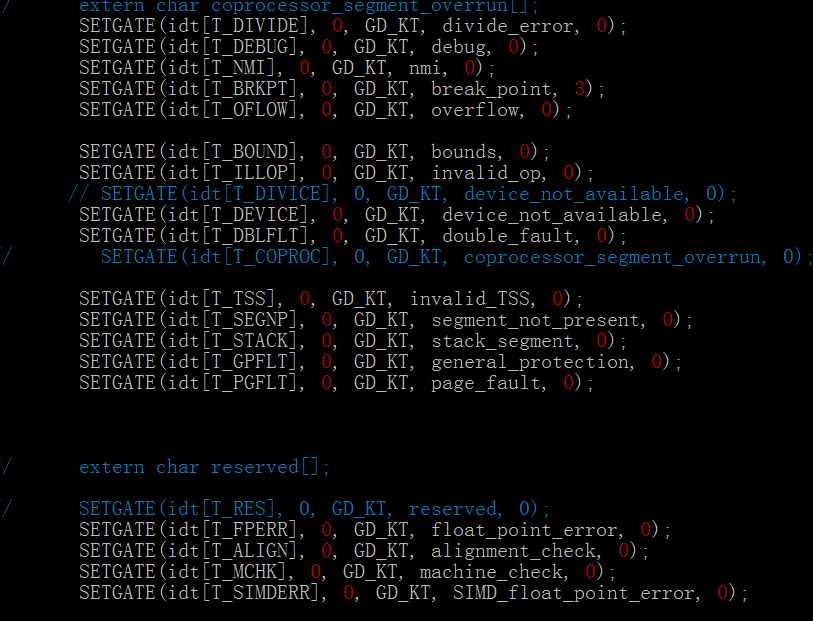


对于这个宏，cpu会自动把errorcode压栈，但是为什么可以，cpu根据什么定义压栈的errorcode呢？后面应该会有发现。后来发现cpu有些中断号是定死的，例如缺页中断，中断号必须是14，往往这些中断会自动压入errorcode。

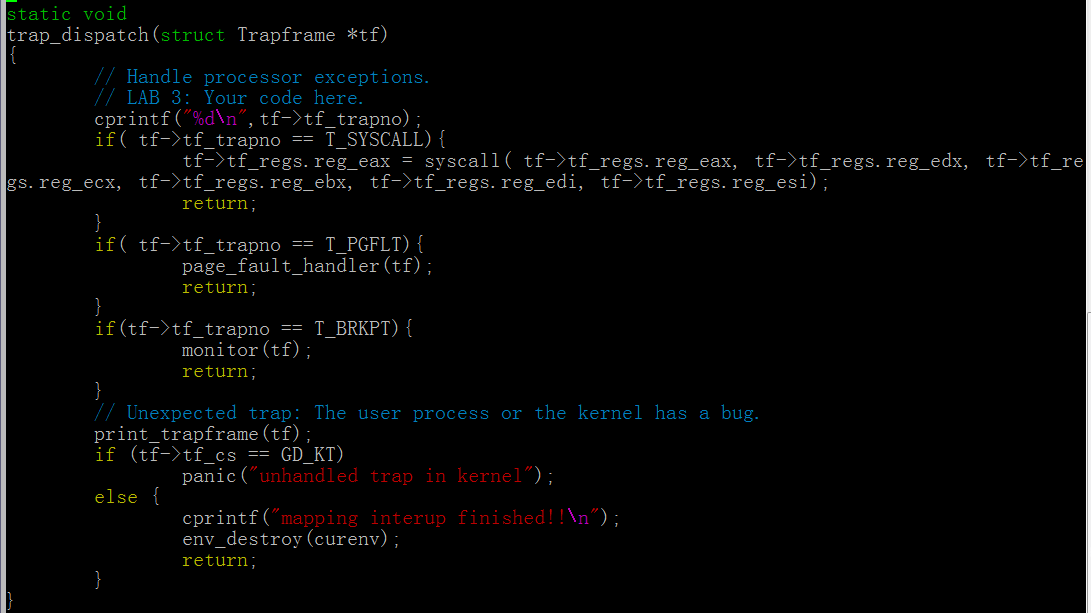
完成中断映射：



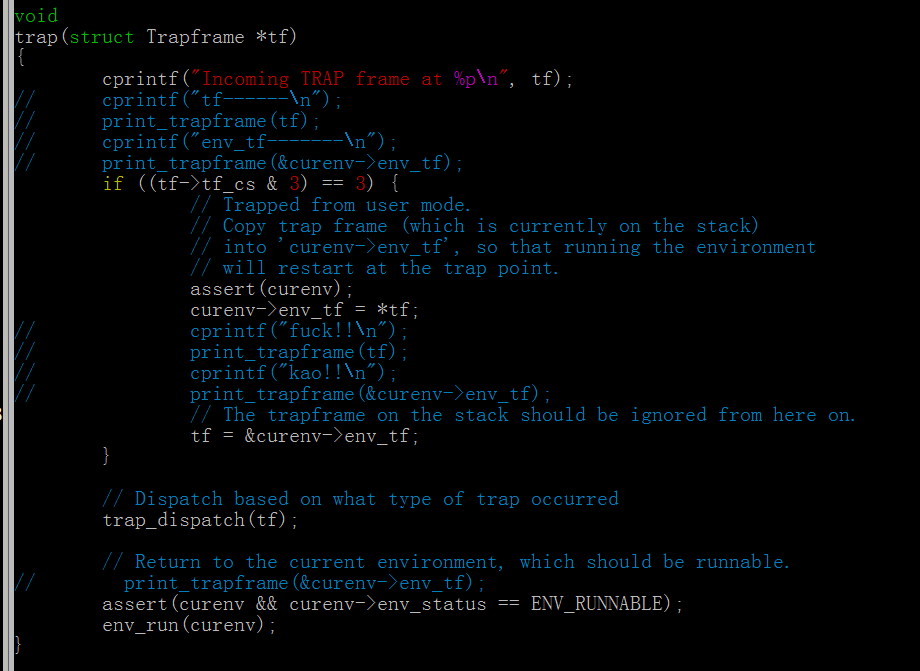
用extern声明一下刚才在trapency.S中声明的函数



中断号映射完之后，我们就可以改trapdispatch函数



现在我们来看看这个函数，trap函数



先检查中断的类型是不是user中断

然后分配中断

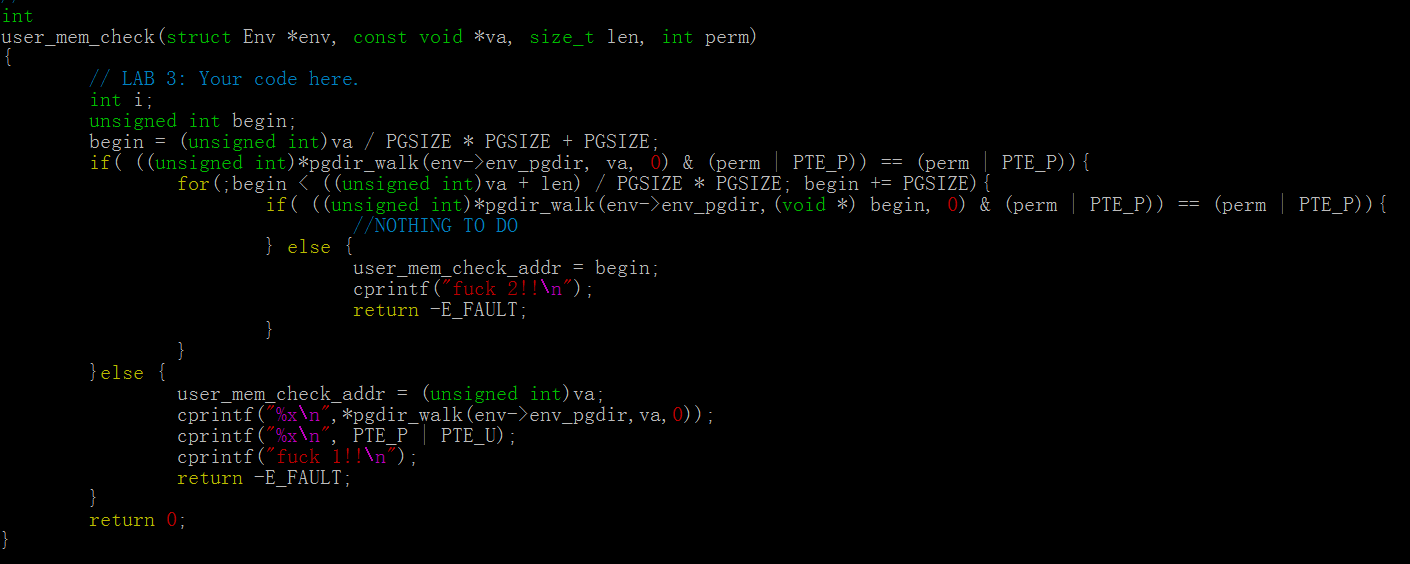
然后恢复程序当前的运行状态。

因为这里已经把当前的状态付给了curenv环境了。所以可以恢复。

现在特别声明一下：如果用的GCC在4.3.3以上，则以上代码在0地址不会出现缺页中断，必须得换到4.3.3或者以下，建议用Ubuntu9.04操作系统，其它版本可能不行，例如10.04。我自己的就不行。

做到这里我们可以修改init.c里面的CREATE\_ENV宏的参数，来运行各种应用程序验证效果。

完成user\_mem\_check函数：



这个函数不复杂，就是检查指定内存可不可以用而已。

我写的代码有点难看，敬请原谅。

修改CREATE\_ENV宏的参数，运行指定测试程序。

到此为止，实验完成。