## LAB3

这个实验开始设计到用户态的东西了。

注意，这个实验中的所有“环境”（env）相关的术语，就是和UNIX中的进程（proc）一一对应的，其实两者本质就是一个东西，但是因为提供的接口不同，所以在实验中给予区分。

这个实验接下来的很多实现，和UNIX的很多实验都有不少区别，虽然可以找到某种对应关系。

关于git commit -a的问题，这个和git add -A其实有区别。后者如果出现问题，会照样add成功的文件，但是前者出现问题就全部不add了。

### Part A

首先要阅读新增的头文件，inc/env.h，弄清楚相关的数据结构的含义。

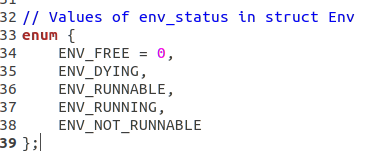
首先是结构体，env使用数组存放，数组的大小已经固定，所以整个系统中最多有NENV个进程运行（由宏可知为1024个）



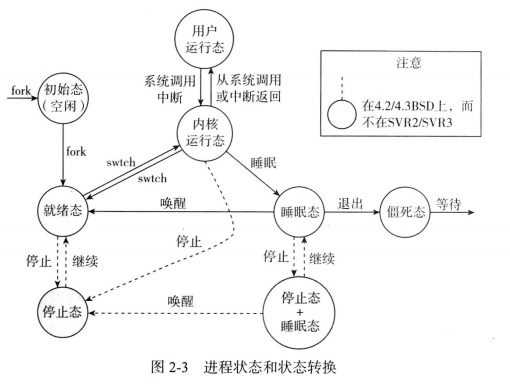
数组中有些环境结构没有被分配出去，它们都处于env\_free\_list上。

Curenv指针指向当前运行中的环境结构体变量。

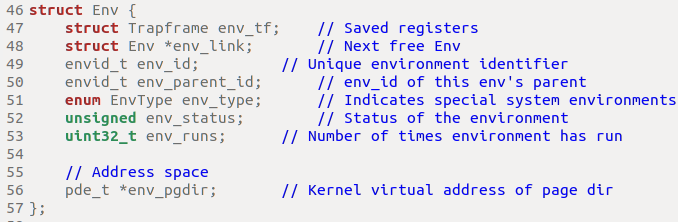
这个env数组，就对应于UNIX系统中的进程表，即proc结构数组（见《深入理解UNIX》2.3.4）



这几个枚举值表示了进程的状态，可以和书本上进行对照。



其中僵死态dying，就是等待父进程的wait回收返回值和其他信息，回收完后进程就彻底消失了。



这里env\_tf里存放了寄存器等硬件信息，相当于描述线程。由于这里只有一个结构体变量，所以实现的系统只支持一个进程对应单个线程。而env\_pgdir变量就对应虚拟地址空间的概念。

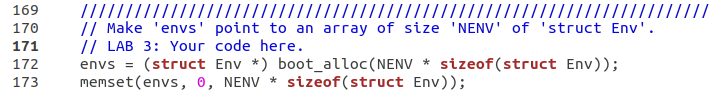
在UNIX中每个进程都有对应的内核栈（kernel stack），但是在JOS中只有一个全局的kernel stack（注意区别于内核区的栈，这个栈是用于处理用户请求的）。因为JOS中同一时刻只会有一个进程处于执行状态，因此不需要考虑内核的可重入问题。

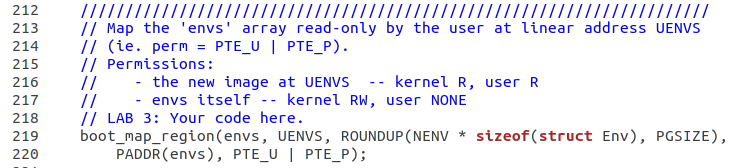
#### Exercise 1

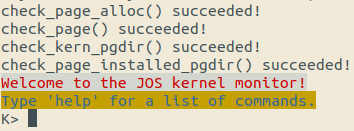
Modify mem\_init() in kern/pmap.c to allocate and map the envs array. This array consists of exactly NENV instances of the Env structure allocated much like how you allocated the pages array. Also like the pages array, the memory backing envs should also be mapped user read-only at UENVS (defined in inc/memlayout.h) so user processes can read from this array.

You should run your code and make sure check\_kern\_pgdir() succeed

基本就是仿照pages数组的分配方式来。







通过了测试！但上面的代码明显是错误的！甚至在之后都无法通过编译。



改成这样就行了。

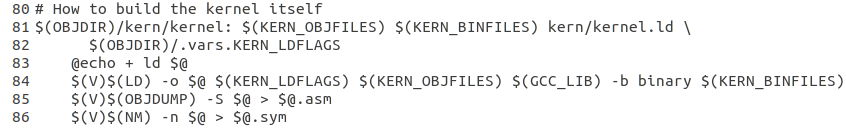
注意这里不需要初始化空闲进程链表，而是在env.c中进行初始化。

此外，这里使用了boot\_alloc，因为这段代码在page\_init函数调用之前，所以还没有配置好页式转换的相关信息，不能用page\_alloc。而后面在env.c中就得用page\_alloc了。此外，由于空闲链表的构建是从后往前构建的，所以一开始page\_alloc返回的基本都是在内核区的地址空间（内核区的堆区），只有这部分用完了，才会分配下面的。所以env.c中调用page\_alloc之后才能继续用page2kva等函数，因为前提就是一开始分配的地址都是在KERN\_BASE以上的。

#### Exercise 2

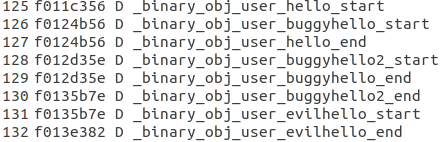
这个练习需要完成kern/env.c中的函数。

首先查看Makefrag文件



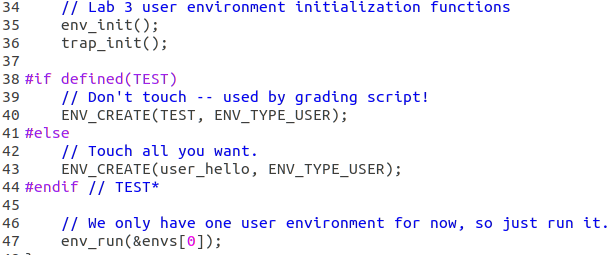
这里面用了-b binary选项来进行链接，对应的img文件在obj/user/directory中，是由GNUmakefile生成的。

在obj/kern/kernel.sym文件中产生了一系列与上述用户可执行镜像文件有关的符号

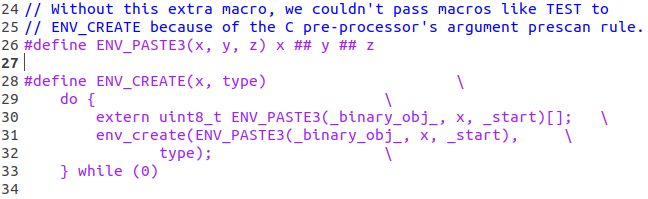


在i386\_init函数中，新增了一些调用，去掉了之前调用monitor，而是调用了ENV\_CREATE宏来建立环境，最后使用env\_run(&envs[0])进入第0个环境，在建立的环境中会进入monitor。

这里要注意！有两个env.h，一个是inc/env.h，另一个是kern/env.h，这两个是不一样的。上述宏定义在kern/env.h中。



在kern/env.h里面有下面的宏。

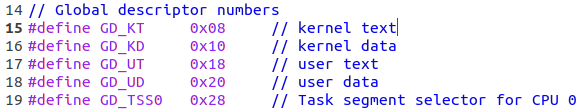


这里ENV\_PASTE3宏很精彩，它使用##来将多个符号直接拼接起来！

所有内置的二进制可执行代码对应的C语言源文件在user/目录中。

接下来可以开始完成kern/env.c了。

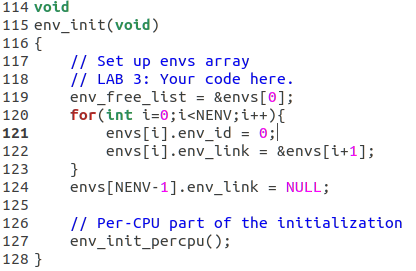
首先，全局描述符表的一些常量定义在memlayout.h中，虽然这个头文件没有被显示地包含。



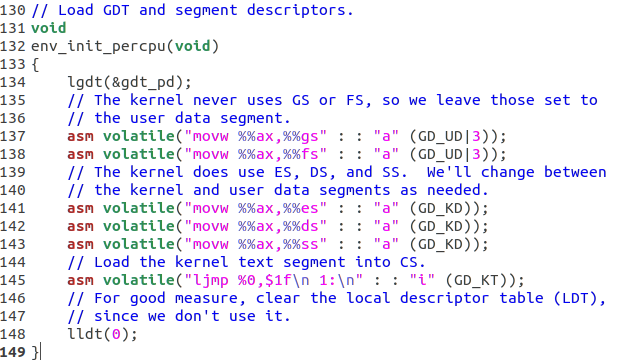
但是下面这个结构体应该就是存放在GDTR寄存器中的数据，第一个数据是16位的，表示GDT的长度，第二个数据是32位的，表示GDT的“线性地址”（也就是虚拟地址，因为这里所有的段基址都是0）。之后会用lgdt和sgdt来修改全局描述符表寄存器（GDTR）的内容。



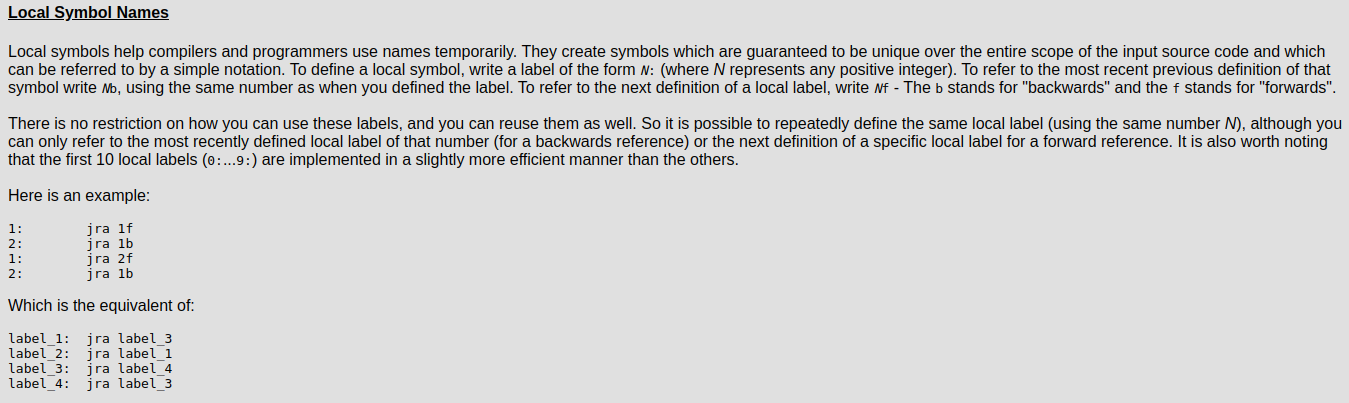
先完成env\_init函数，其实就是建立一个链表



Env\_init\_percpu函数用来配置全局描述符表，并设置段寄存器的值。



这里最后的ljmp比较难理解。它有两个参数，其中第一个表示段寄存器CS的值，第二个表示偏移。因此，这里先将GD\_KT这个立即数的值存入段寄存器CS中，然后进行跳转，但是后面的”$1f\n 1:\n”实在不知道啥意思。这里可以查看GNU assembler的文档。



在局部标号这一节说明了，使用整数来表示局部标号（其中0-9效率最高），然后使用f和b分别表示“之后”和“之前”，例如$1f就表示在这之后最近定义的数值为1的局部标号。

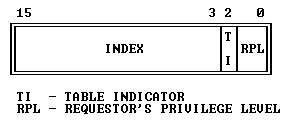
所以上面的内联汇编其实就是

Ljmp $GD\_KT, $1f

1:

Lldt(0);

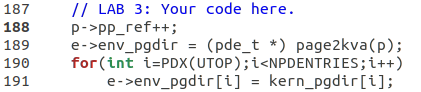
也就是这个ljmp跳转的目的地就是下一行lldt(0)语句对应的汇编语句的第一条指令的地址。相当于没有跳转，继续执行。但是，经过这个ljmp之后，段寄存器CS的值发生了变化。



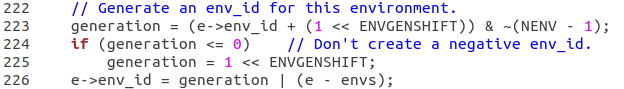
段选择子位宽16bit，如上图所示。这里低2位表示特权，第3位表示描述符表选择。TI=0表示选择GDT。

上面代码中，GD\_UD|3，表示特权为3，即用户权限。而全0表示最高权限，即内核权限。

接下来可以完成env\_setup\_vm函数了。一开始没搞懂这函数要干啥，其实就是新建一个一级页表，这个页表的大部分内容和kern\_pgdir暂时都是一致的（除了当前用户页表区不同）



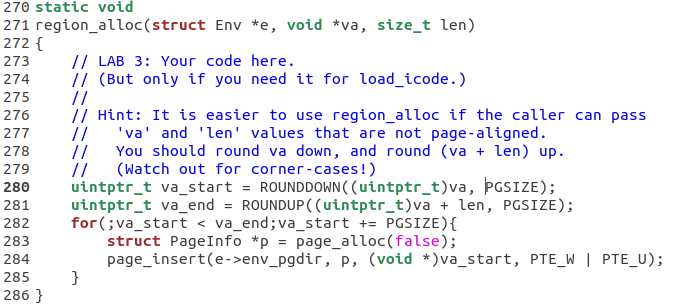
然后阅读env\_alloc的代码。



这一段代码用于生成进程ID，可以看到，ID的低10位永远等于e-envs，也就是这个结构体在envs数组中的偏移。而从ENVGENSHIFT（12）开始，更高的位可以被视为这个块第几次被分配出去。因此，即使前后两个进程分配到一个结构体，其ID也是不同的。例如，如果都分到第0个进程描述符，那么第一个进程的ID是0，第二个进程的ID是4K=4096

接下来的代码配置进程的中断帧，首先清空，然后配置帧中的段寄存器。

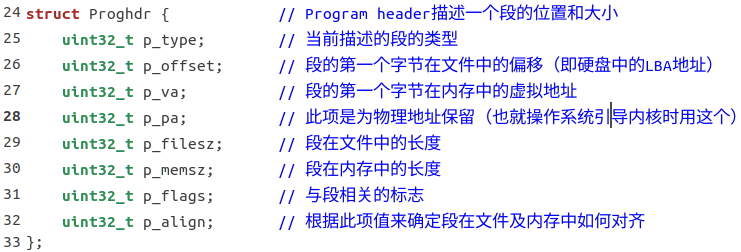
接下来完成region\_alloc函数。这里不需要增加pp\_ref，因为对其的修改会在page\_insert中完成。



接下来完成load\_icode函数，这个函数比较复杂，其功能是解析ELF文件，并将内容装载到新的进程的用户地址空间中。

可以参考boot/main.c中的代码来完成这个函数。

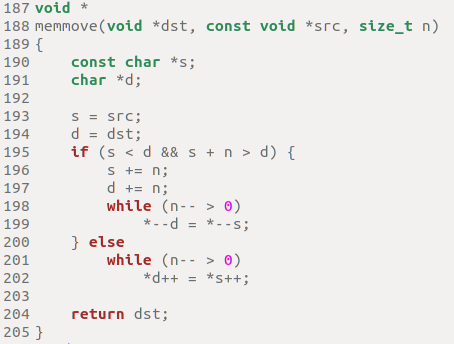
两个函数的区别在于，这里load\_icode函数是把已经在内存中的数据装载到用户的虚拟地址空间中，而不像boot那里是要从硬盘读取出来，然后装载到内存中。



要注意，程序头表里的p\_offset表示相对于文件开头的偏移，不一定就是相对于硬盘某个扇区的偏移。例如这个函数给出了binary就是一个char \*指针，也就指向了文件的开头，只不过这个文件已经存在于内存中了。

根据注释的提示，这里要先切换页目录，即切换成用户的虚拟地址空间。

将一个内存复制到另一块地方，可以用lib/string.c中提供的memcpy（调用了memmove）



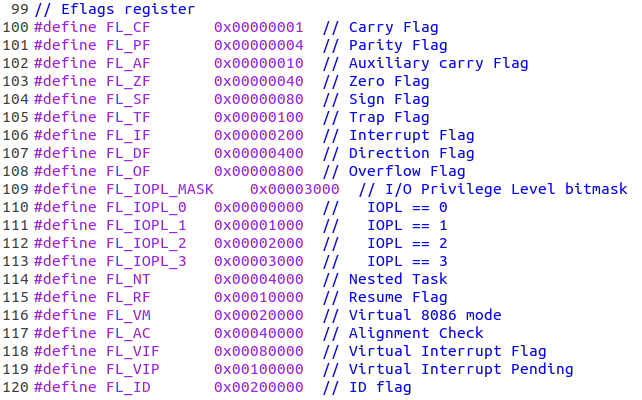
将各个段装载到虚拟地址空间之后，需要设置程序入口地址、堆栈和标志位等。这里不清楚标志位要怎么设置，全部清零？感觉也不太对头啊。



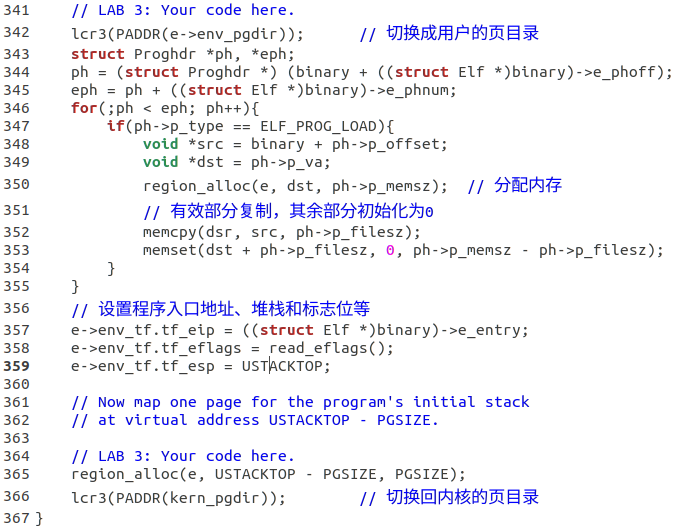
这里先看一下下面env\_pop\_tf函数，可以看到最后用iret返回。查看x86汇编手册可知，iret指令会将栈中依次弹出3个数，分别存入EIP、CS和EFLAGS中，而这就对应了Trapframe数据结构中的量。所以需要初始化这三个量。而esp似乎不需要？但还是设置一下好了。是否需要esp似乎是和执行指令时EFLAGS有关，但不太清楚，根据注释，当在用户态和内核态切换的时候，需要设置tf\_esp，所以应该是需要配置的。

我暂时就把eflags配置成和执行这个函数时的eflag相同吧，之后有问题再回来看咋回事。

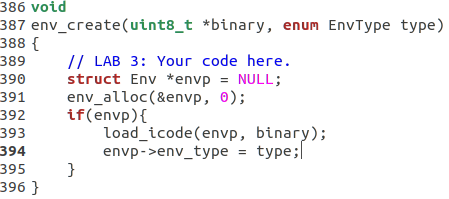
注意，在mmu.h中，定义了EFLAGS相关的宏。这里我觉得应该至少把I/O Privilege特权级设置成3（最低），因为这个环境肯定是用户进程。



先完成了load\_icode函数



接下来的env\_create就不那么复杂了。

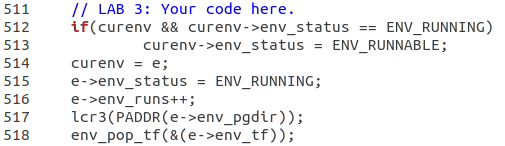


然后可以阅读一下env\_free，虽然这不是要求的。

其实主要工作就是两个，显示销毁所有的页表和页，也就是每个pte指向的页都销毁，然后销毁所有pde指向的页（二级页表本身），然后销毁页目录（一级页表本身）。

第二个工作就是释放环境结构，即将环境加入env\_free\_list中，并修改状态为空闲。

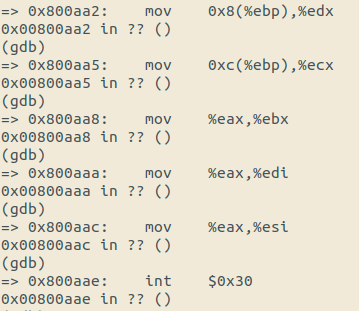
最后完成env\_run函数。



来了来了，Triple fault！！！

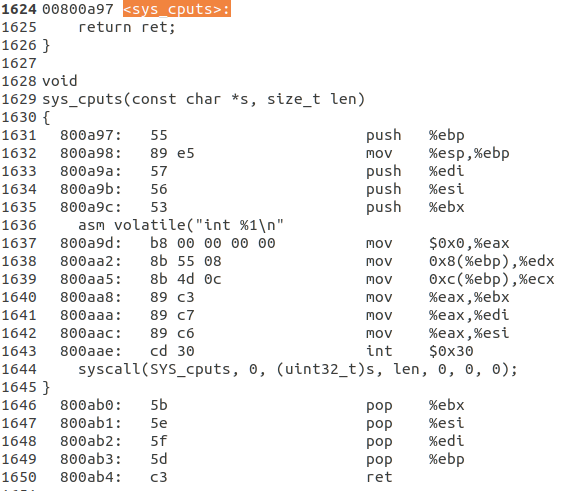
使用gdb调试发现，iret成功跳转了，但是执行的指令都是错误的。有几种可能，要么是地址有问题，要么是装载有问题，还有其它一些可能。

但是我检查了半天，发现都没有问题，甚至在iret之后单步调试了很久都没事，最后在int 30这个地方出现问题。



再回去看官网，发现确实会有这样的问题，所以我其实没错。

其实本来可以对照obj/user/hello.asm来看是否正确，可以在sys\_cputs里面找到上面对应的反汇编指令。



When the CPU discovers that it is not set up to handle this system call interrupt, it will generate a general protection exception, find that it can't handle that, generate a double fault exception, find that it can't handle that either, and finally give up with what's known as a "triple fault". Usually, you would then see the CPU reset and the system reboot. While this is important for legacy applications (see [this blog post](http://blogs.msdn.com/larryosterman/archive/2005/02/08/369243.aspx) for an explanation of why), it's a pain for kernel development, so with the 6.828 patched QEMU you'll instead see a register dump and a "Triple fault." message.

最后，官网的上面这一段给出了Triple fault的解释，其实就是出现无法处理的异常，而这个异常本身也无法处理，这也是个异常，最后彻底放弃，共三个异常。

#### Exercise 3

接下来就要开始中断了，这里先阅读80386手册的第九章，了解中断相关的知识。

中断向量：一个0~255之间的整数，是一个索引值

中断向量表：interrupt descriptor table (IDT)，每一项都包含段选择子和偏移，即CS和EIP。

任务状态段：The Task State Segment，用于描述ESP0和SS0，即处理中断时内核使用的栈

同步异常的编号为0-31，超过31的要么是软中断，使用int指令产生，要么是外部中断。

有些中断会把error code错误代码也压入内核栈，例如缺页异常，这需要根据80386手册上来看。不同的中断，其错误代码的含义也不同。

如果当前正处于内核态（CS寄存器低2位全0），则中断被成为嵌套中断，此时不需要切换栈，也不用保存旧的SS和ESP。

如果内核栈空间不足以保存旧的信息，且当前正处于内核态，则会陷入无法恢复的死机状态，只能重启解决问题。

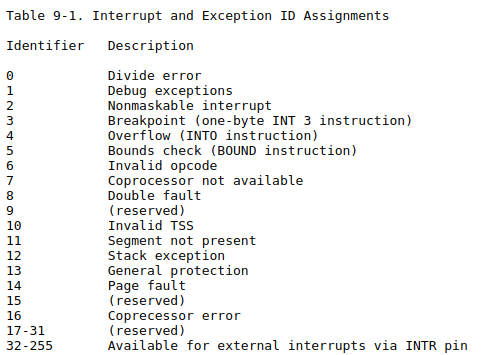
中断分为可屏蔽中断（maskable interrupt，INTR引脚）和不可屏蔽中断（Nonmaskable interrupts，NMI引脚）

异常一般指内部出现的中断，可分为fault（出错）、trap（陷入）、abort（中止）三种。

Fault一般是在指令执行之前出现的异常，典型的是缺页异常（page fault）

Trap一般是指令执行刚执行完出现的异常，例如软中断int

Abort则在指令执行的任何位置都可能出现，一般是严重的硬件错误、系统的严重错误等。



中断一般在指令的边缘（boundary）处理，使用rep作为前缀的字符串处理指令除外（没次循环都处理）。

对于NMI来说，当一个NMI处理程序正在执行时，忽略在此期间的所有其它NMI中断信号。

IF允许中断位控制可屏蔽中断的响应。IF=0则屏蔽INTR。可以用CLI和STI来设置。只有当CPL<=IOPL时才能设置，否则会触发保护异常。除此之外，PUSHF、POPF、IRET和中断门会改变IF。

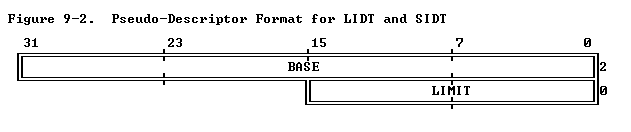
（注意，CPL表示CS的RPL，即低2位的值，表示当前执行代码段的特权级）

EFLAGS中的RF位控制Debug fault



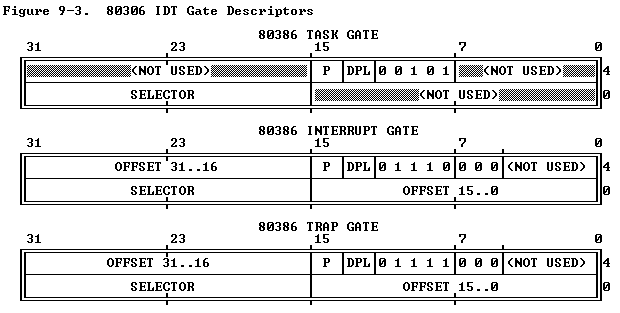
为了避免中断对上面两条指令造成影响（导致SS和ESP不匹配），在修改SS之后的空隙不处理中断。可以勇敢LSS指令来替代上述指令，避免这个问题。

中断向量表的地址存放在IDTR寄存器中，可以通过LIDT和SIDT来配置，供6个字节。



中断向量表的每一项占8个字节，所以将索引值（中断类型号）乘8，再加上IDT\_BASE，就得到了中断向量表项的地址。

IDT的每个表项可能是三种类型之一：任务门（task gate）、中断门（interrupt gate）、陷阱门（trap gate）。



可以看到，中断和自陷都是有16位的段选择子和32位的中断向量（即中断服务程序的入口地址）。

中断和自陷都会影响TF（清0），避免单步调试影响调试程序自身。

但是，中断还会影响IF（清0，使得屏蔽其它中断），而自陷不会影响IF。

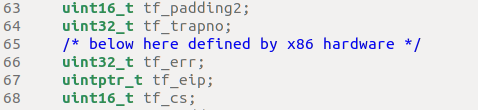
在IRET时会通过栈中保存的EFLAGS来恢复上述修改的内容。

#### Exercise 4

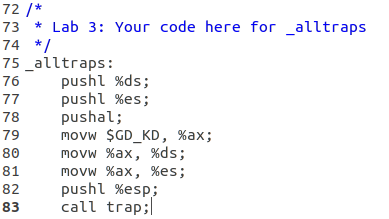
首先完成trapentry.S

为了给出每一个中断处理函数的地址，需要知道每一种中断是否有error code，然后调用相应的宏来生成对应的函数。

在中断程序执行之前，处理器就已经完成了一系列工作，也就是把一系列寄存器入栈，但是没有完全。具体入了多少呢？



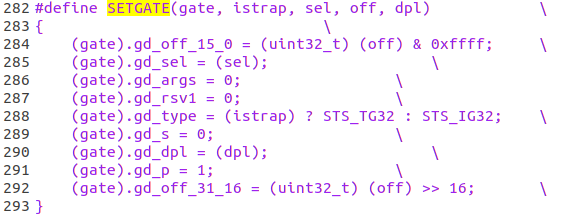
这里可以看注释，就知道tf\_err以及下面的所有量都是硬件负责入栈的，而上面的变量则需要软件来设置。这就是\_alltraps的作用。



最后call trap，所以接下来就要继续完成trap.c中的相关函数。其中，trap函数可以通过trapframe结构中的tr\_trapno来识别中断号，从而执行不同的中断函数。这里要好好理解！

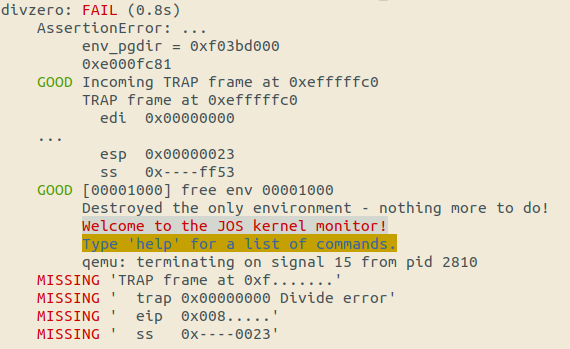
首先完成trap\_init函数，这里需要用到在inc/mmu.h中定义的SETGATE宏。

这里sel不知道怎么获取，应该就是GD\_KT？

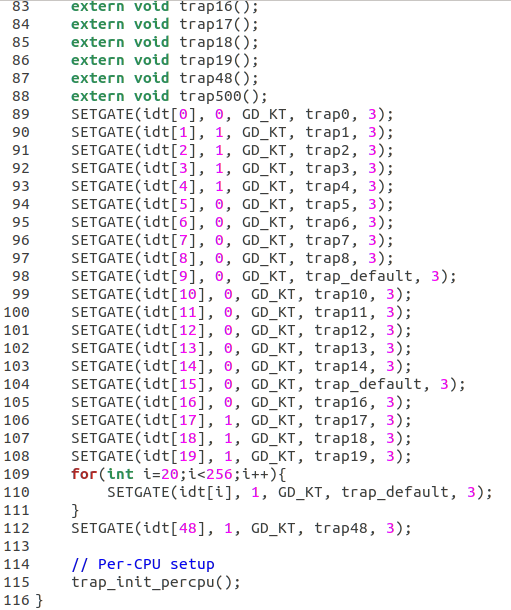


最后用了暴力写法，也不知道行不行。

完成这个之后，暂时就可以了。剩余部分在Part B中继续。



但是无法通过测试。

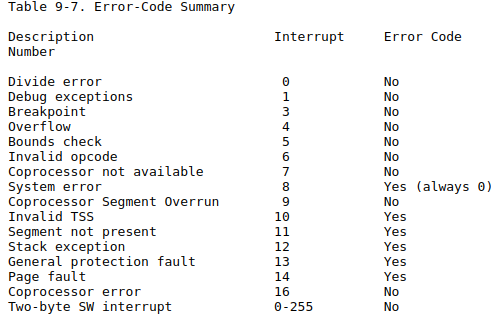


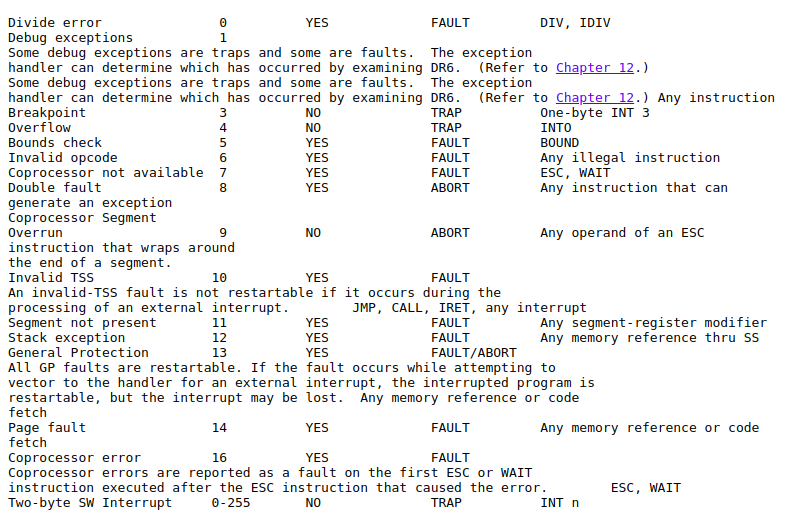
检查发现，无论什么中断，最后都只显示double fault。

非常麻烦，使用gdb调试的话，如果单步执行idiv %ecx这条指令，则最后是正常divide zero，但如果直接continue跳过，则最后会是Double Fault。

而且不同地方开始continue，最终显示的esp也不同。正常直接运行的话，esp显示为0xeebfe000，而eip为0x00800020，也就是一进入用户环境就报错了。

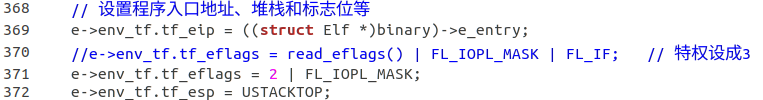
那应该不是中断的问题，可能是前面切换环境就有问题？





经过很长时间的摸索，发现是eflags配置出错，不能设置IF位。

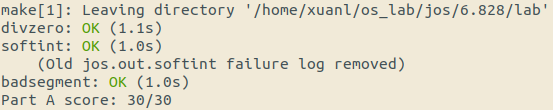
可能设置了IF位之后，一进入用户态就会收到中断。



但还是有部分错位问题，导致没有拿到30分，即软中断还有问题。

根据说明，int 14最后应该显示为General protection（int 13），也就是特权级不够。用户态不能使用int 14来自陷入缺页中断，否则会死循环（缺页中断返回后重新执行这个指令）。

问题在于特权级设置有问题，本来所有的中断门特权级都是3，但实际上很多特权级应该设置成0，例如int 14的特权级。



这样就通过了Part A的所有测试。注意在monitor.c中不能用彩色打印，否则无法通过softint的测试。

#### Question 1

1. What is the purpose of having an individual handler function for each exception/interrupt? (i.e., if all exceptions/interrupts were delivered to the same handler, what feature that exists in the current implementation could not be provided?)

这么做的目的就是为不同的中断设置不同的中断号，也就是设置TrapFrame结构体中的tf\_trapno。之后的trap\_dispatch函数就是根据这个中断号来分配不同的中断处理函数的。

如果所有的中断都被分配到同一个处理函数，那这个处理函数是没有办法识别不同的中断类型的，因为把中断号入栈不是硬件完成的，而是软件完成的（即TRAPHANDLER宏完成的）

1. Did you have to do anything to make the user/softint program behave correctly? The grade script expects it to produce a general protection fault (trap 13), but softint's code says int $14. Why should this produce interrupt vector 13? What happens if the kernel actually allows softint's int $14 instruction to invoke the kernel's page fault handler (which is interrupt vector 14)?

一开始就是没注意这个点，导致只有20分。软中断不应当陷入trap 14，因为缺页中断返回之后，会重新执行触发中断的指令，这样int 14就会反复执行，陷入死循环。因此，不应当给用户态陷入trap 14的权限，因此要将14号中断的描述符权限级别设置成0，即最高权限，这样只有硬件检测到缺页异常时才会进入trap 14，而用户无法自行进入14号中断。

#### Challenge 1

Challenge! You probably have a lot of very similar code right now, between the lists of TRAPHANDLER in trapentry.S and their installations in trap.c. Clean this up. Change the macros in trapentry.S to automatically generate a table for trap.c to use. Note that you can switch between laying down code and data in the assembler by using the directives .text and .data.

这个应该就是在trapentry.S里面新增两个宏，同样也是用于处理是否有error code两种情况。宏里面既有.text也有.data，这里主要是.data部分比较麻烦，因为没法直接调用SETGATE宏了，但可以仿照其思路。

### Part B

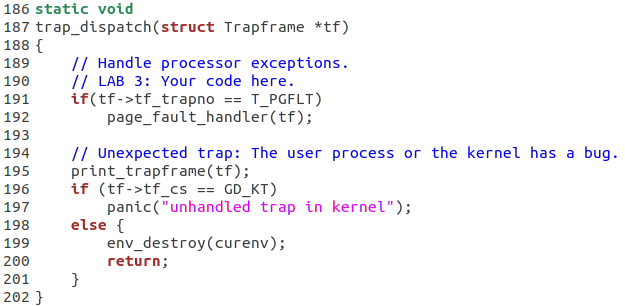
导致缺页的虚拟地址会存放在寄存器cr2中。

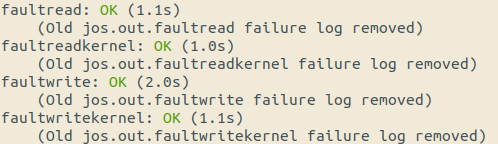
注意，可以用make run-x 或 make run-x-nox来选择执行哪一个用户进程。

#### Exercise 5

Modify trap\_dispatch() to dispatch page fault exceptions to page\_fault\_handler(). You should now be able to get make grade to succeed on the faultread, faultreadkernel, faultwrite, and faultwritekernel tests. If any of them don't work, figure out why and fix them. Remember that you can boot JOS into a particular user program using make run-x or make run-x-nox.

非常简单，就两行

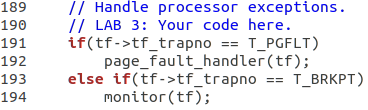




#### Exercise 6

Modify trap\_dispatch() to make breakpoint exceptions invoke the kernel monitor. You should now be able to get make grade to succeed on the breakpoint test.

也很简单，同样只需要2行



#### Quention 2

1. The break point test case will either generate a break point exception or a general protection fault depending on how you initialized the break point entry in the IDT (i.e., your call to SETGATE from trap\_init). Why? How do you need to set it up in order to get the breakpoint exception to work as specified above and what incorrect setup would cause it to trigger a general protection fault?

这个还是老问题了，在调用SETGATE设置权限的时候，对于断点中断int 3的权限级别应该设置成3，这使得用户态可以通过调用int 3来自陷进入断点中断程序。如果设置成了0，则会导致general protection fault，因为用户态调用int 3时会检查CPL和DPL，发现CPL=3，而DPL=0，因此无权进入中断门。

1. What do you think is the point of these mechanisms, particularly in light of what the user/softint test program does?

这个估计还是问的为什么要设置DPL权限？因为有些异常不应该由用户态主动触发，例如缺页中断，否则可能造成很多BUG，例如int 14会导致死循环。对于这样的异常，其DPL应当设置成0，保证用户态无法陷入。

而对于系统调用，和设置断点等，用户态需要这些中断请求，因此要将其DPL设置成3，保证用户有足够的权限进入中断门。

#### Challenge 2

Modify the JOS kernel monitor so that you can 'continue' execution from the current location (e.g., after the int3, if the kernel monitor was invoked via the breakpoint exception), and so that you can single-step one instruction at a time. You will need to understand certain bits of the EFLAGS register in order to implement single-stepping.

Optional: If you're feeling really adventurous, find some x86 disassembler source code - e.g., by ripping it out of QEMU, or out of GNU binutils, or just write it yourself - and extend the JOS kernel monitor to be able to disassemble and display instructions as you are stepping through them. Combined with the symbol table loading from lab 1, this is the stuff of which real kernel debuggers are made.

相当于要写一个简易的调试器，能够单步执行。如果学有余力，还可以加上反汇编的功能。虽然挺难的，但还是挺有趣的。

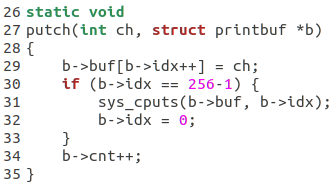
#### Exercise 7

系统调用是int 0x30，调用的系统服务编号存放在%eax中，参数依次存放在%edx，%ecx，%ebx，%edi和%esi中。返回值存放在%eax中。

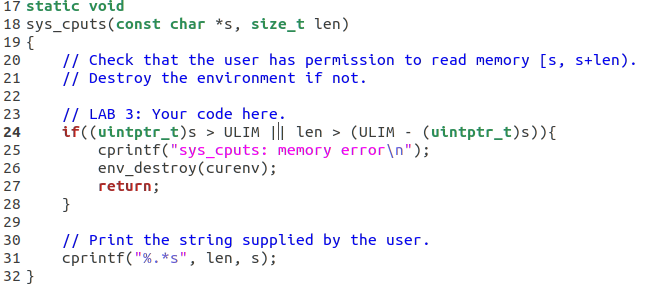
首先阅读lib/syscall.c来弄清楚syscall函数的运作方式。注意，还有一个kern/syscall.c，后者是需要我们来实现的，而前者已经写好了。

这个练习又是一个比较复杂的练习，工作量比较大。

现在回过头来可以发现，在lib/printf.c和kern/printf.c里面的putch实现是不同的，在lib/printf.c里面提供的是给用户调用的putch，其最终还是调用sys\_cputs函数，从而会最终执行int 0x30。而kern/printf.c则直接调用console.c中实现的cputchar，这个区别要弄清楚。



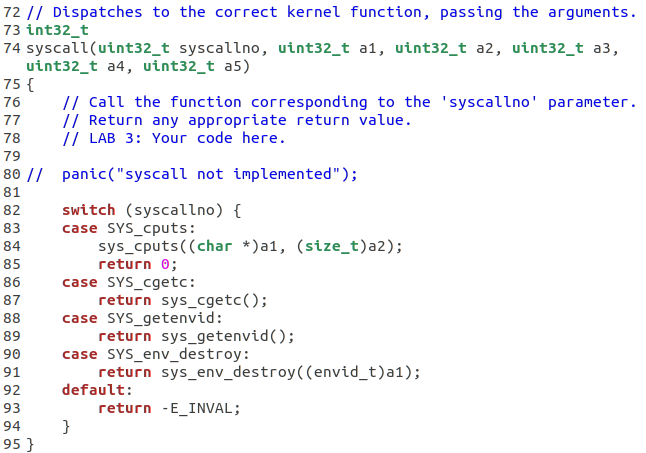
完成sys\_cputs函数，主要工作是检查权限。应该只要不访问ULIM以上的内容，都可以，那直接判断就行了。



上面的s>ULIM感觉应该改成s>=ULIM，因为这个地址本身应该不属于用户访问的空间。

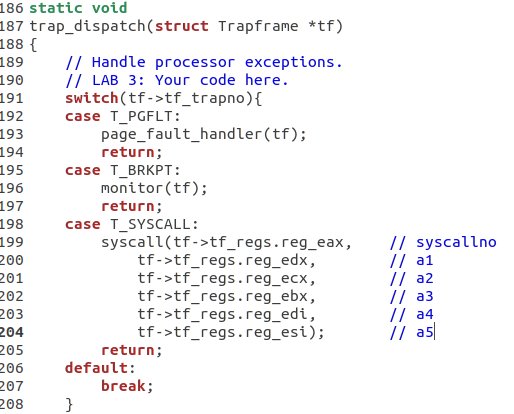
而且这里感觉可以用之后完成的user\_mem\_check来检查？不清楚啊。

然后完成syscall函数，这个函数是调用了int 0x30之后，由trap\_dispatch调用的函数。其工作就是根据调用号，调用不同的系统功能调用函数。



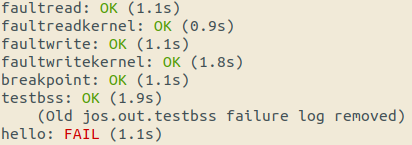
最后是修改trap\_dispatch函数，这里需要当心。

但还是通不过测试，发现少了几行输出。



上面这里应该中间都是return，之前都是break，导致总是会destroy env。

这样就可以通过testbss测试了。



#### Challenge 3

Implement system calls using the sysenter and sysexit instructions instead of using int 0x30 and iret.

X86提供的两个更快的指令，但是得先学习一下。

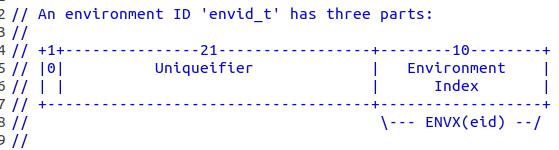
然后这个challenge和之后的实验是有冲突的，所以可以等后面的完成之后再回来考虑这个challenge。

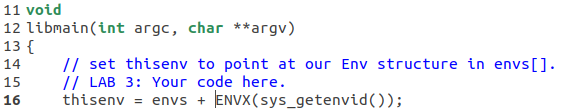
#### Exercise 8

Add the required code to the user library, then boot your kernel. You should see user/hello print "hello, world" and then print "i am environment 00001000". user/hello then attempts to "exit" by calling sys\_env\_destroy() (see lib/libmain.c and lib/exit.c). Since the kernel currently only supports one user environment, it should report that it has destroyed the only environment and then drop into the kernel monitor. You should be able to get make grade to succeed on the hello test.

这里只要理解了inc/env.h里面的envid\_t的规定，就知道怎么获取thisenv了。

使用ENVX作用于eid，就可以得到偏移，然后加上envs，就得到指针了。

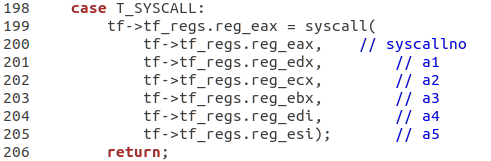




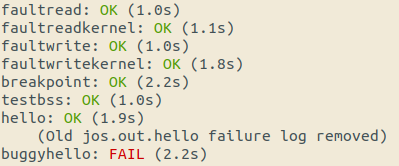
但是出现了问题，明明是00001000，但输出确实i am env 00000000。

原因就在于，syscall调用没有正确设置返回值！

应该修改成如下形式。



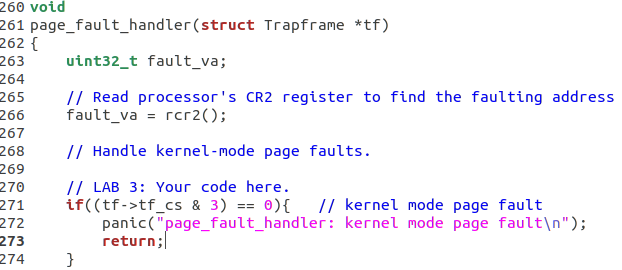
这下终于通过了



#### Exercise 9

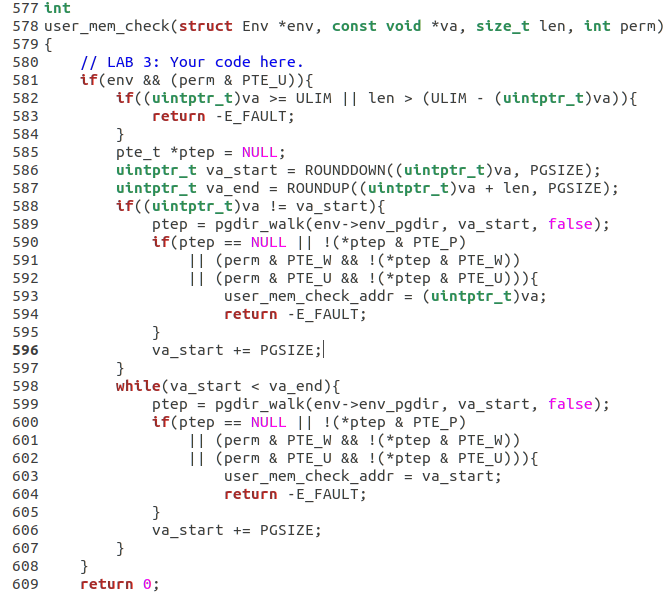
内核处理用户传入的指针参数时要额外当心。一方面，对指针的解引用可能导致缺页异常，但这个缺页本质上是由用户导致的。（如果内核自身出现缺页，那就说明操作系统故障，因为内核的所有页已经完成了静态映射，常驻内存，不可能出现缺页）

另一方面，指针可能指向内核区的地址，如果随意解引用则会泄露内核数据，甚至对内核造成破坏。

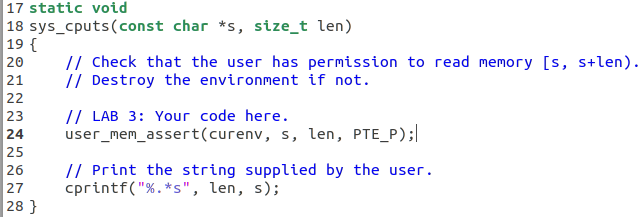


上述代码保证缺页异常是由用户造成的。

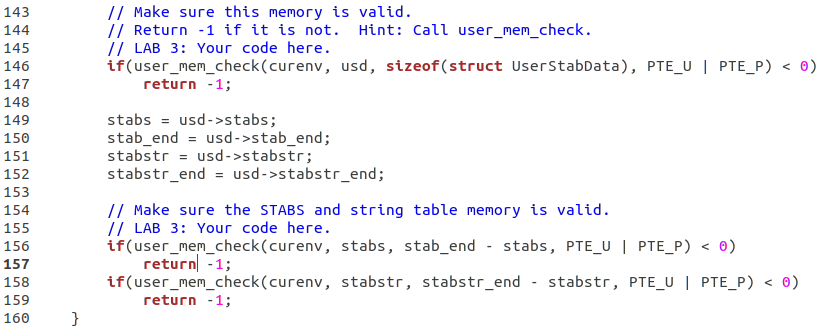
接下来回到kern/pmap.c中，阅读user\_mem\_assert并完成user\_mem\_check。



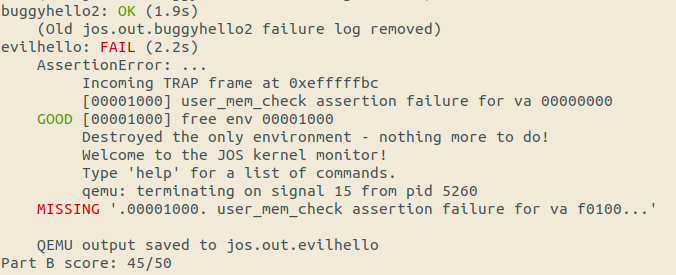
然后之前syscall.c里的还得改，其实之前不需要写的。



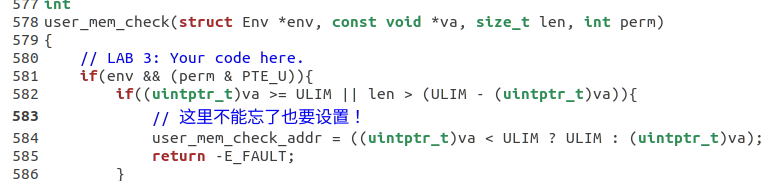
最后还要修改kdebug.c里面的内容，保证符号表是有效的。



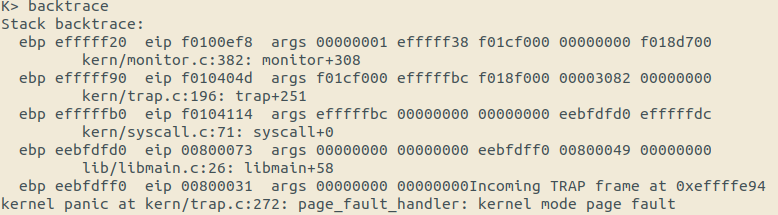
但最后的evilhello还是不对，这里给出的错误地址有问题。



其实就是有个地方错误退出前没有设置user\_mem\_check\_addr的值。这个地方要注意，如果va < ULIM，说明第一个无效地址是ULIM，否则第一个无效地址是va。

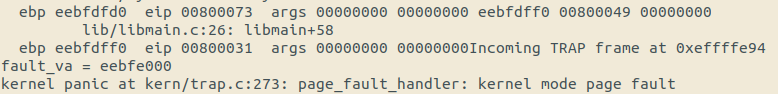


执行backtrace的结果

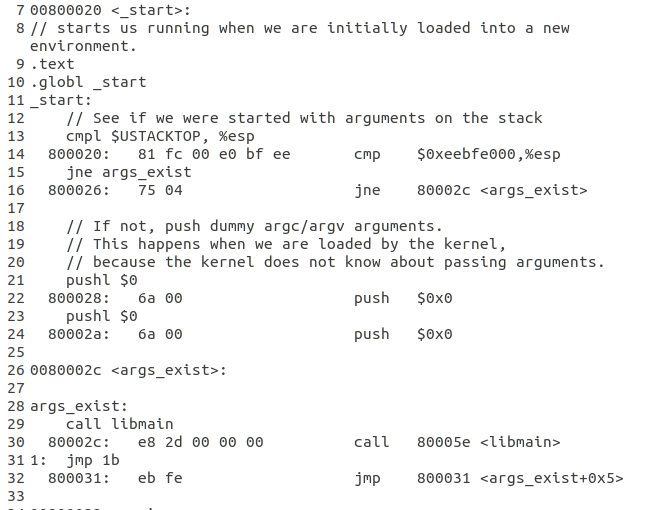


这里最后确实出现了缺页中断，而且是内核自身的缺页中断。

添加了一个调试信息，就是内核缺页前打印导致缺页的虚拟地址



再仔细检查可以发现，缺页的地址为0xeebfe000，也就是用户栈的栈底！此时用户栈已经空了，因为在libmain之前没有函数了。



检查obj中的反汇编代码可知，在env\_pop\_tf函数执行最后一条汇编指令iret之后，就跳转到了上面\_start的地方开始执行，这时候栈是空的。先后push两次，push $0x0，作为内核调用的参数。然后再执行了call libmain，之后才建立了libmain的栈帧。因此在libmain之前没有栈帧，且栈中只有两个值为0的双字。

因此，上面的backtrace在libmain之前能够获取其地址，并且之后还打印了两个0，然后陷入内核缺页中断。因为此时用户栈已经空，继续向上访问已经越界了，上面的一块区域没有被映射到物理地址空间。

终于通过了所有测试！

