## LAB4

这个实验开始进入多核。

### Part A

这个实验要修改JOS，使其支持SMP（对称多处理器）模型。该模型中，有一个BSP（bootstrap processor）负责引导和加载操作系统本身，还有若干AP（application processor）能够被BSP激活来运行程序。硬件和BIOS决定BSP。

PIC全称Programmable Interrupt Controller，通常是指Intel 8259A双片级联构成的最多支持15个interrupts的中断控制系统。APIC全称Advanced Programmable Interrupt Controller，APIC是为了多核平台而设计的。它由两个部分组成IOAPIC和LAPIC，其中IOAPIC通常位于南桥中 用于处理桥上的设备所产生的各种中断，LAPIC则是每个CPU都会有一个。IOAPIC通过APICBUS(现在都是通过FSB/QPI)将中断信息分派给每颗CPU的LAPIC,CPU上的LAPIC能够智能的决定是否接受系统总线上传递过来的中断信息，而且它还可以处理Local端中断的pending、nesting、masking，以及IOAPIC于Local CPU的交互处理。

（<http://news.eeworld.com.cn/qrs/2015/0821/article_24256.html>）

一个处理器通过内存映射I/O（MMIO）来访问其LAPIC，物理地址为0xfe000000（距离4GB差32MB）。JOS将使用虚拟地址空间中的MMIOBASE这一块区域来做相应的映射。

#### Exercise 1

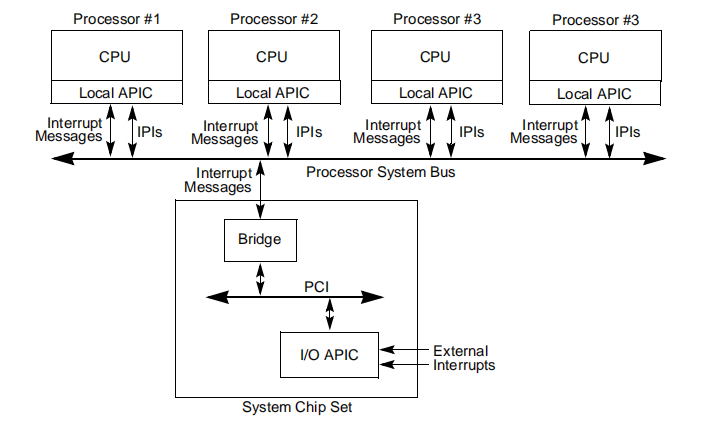
Implement mmio\_map\_region in kern/pmap.c. To see how this is used, look at the beginning of lapic\_init in kern/lapic.c. You'll have to do the next exercise, too, before the tests for mmio\_map\_region will run.

先阅读kern/lapic.c里的函数，然后完成kern/pmap.c新增的函数。

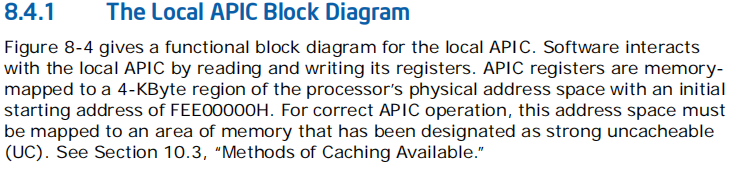


要求阅读Intel processor manual volume 3的第8章和附录C，以及vol. 3A的10.5

8.1节有个图不错，下图很好地展示了LAPIC和I/O APIC的关系以及各自的功能。

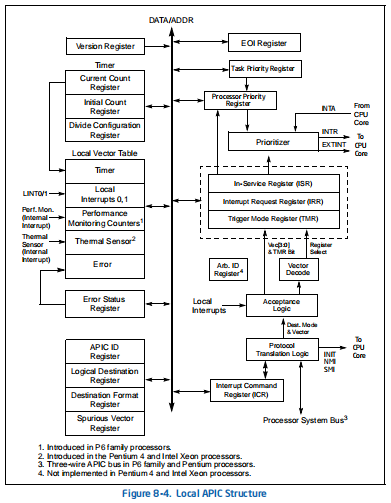


8.4节开始介绍LAPIC的架构。

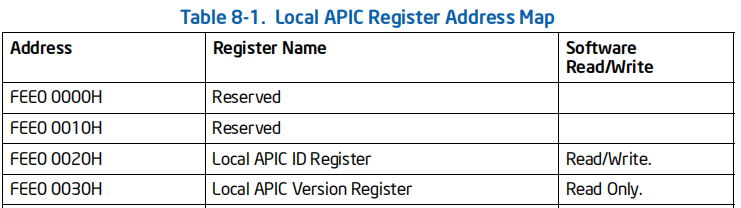


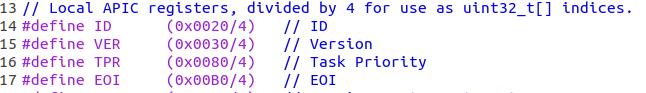
这里提到，LAPIC的寄存器是映射到0xfee00000开始的4K物理页上的，那这和实验中的32MB的区域有什么关系呢？（其实就是说最高的32MB是一个hole，LAPIC在这里，还有别的也在这里，但不重叠，LAPIC只占其中的4K）手册这里也提到，这块区域必须被映射成uncacheable的，否则可以想见会导致数据冲突。内存中的数据如果CPU不写的话，正常是不会变的，所以读数据的时候cache和内存是一致的。但这块区域的数据本质是映射到LAPIC内部寄存器，所以值不受CPU控制，是会变化的。

下图展示了LAPIC的内部寄存器，之后的表8-1会给出这些寄存器和物理地址的映射关系。



注意到映射关系是按16字节（128 bit）对齐的。



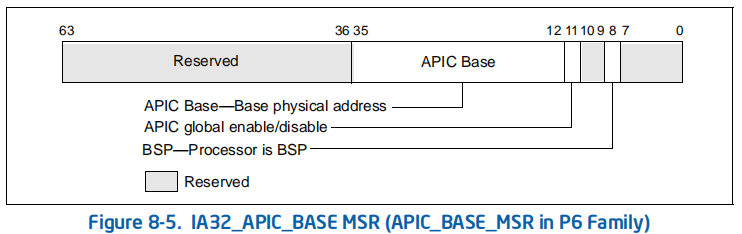


上面的表和实验里的宏是一一对应的！宏定义了在内存中MMIOBASE上的偏移，除以4之后就变成了索引（因为这些寄存器本身4B，但是地址映射按照16B对齐）

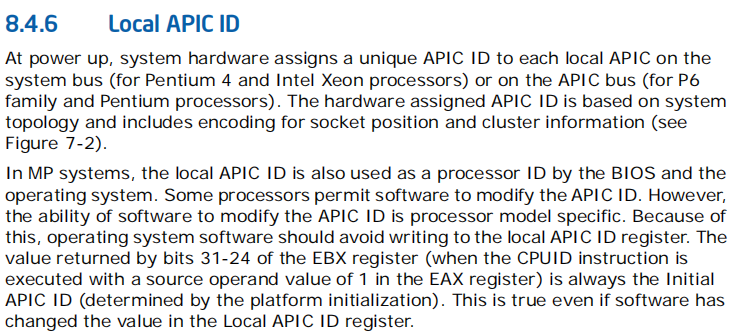
如何判断当前CPU是否有LAPIC呢？可以执行CPUID指令，原操作数EAX的值为1，则EDX的第9位表示是否有LAPIC。（ presence or absence）

开启和关闭LAPIC需要使用IA32\_APIC\_BASE MSR，这里有了一个新的概念即MSR（模式特定寄存器），在附录B中介绍了一系列MSR。RDMSR和WRMSR指令用于读写MSR寄存器。

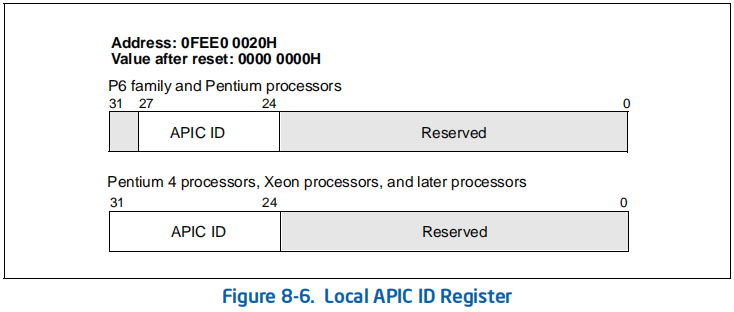
CPUID指令能够获取当前CPU支持的MSR集合，这些在Intel Pentium处理器开始引入。



可以通过修改这个MSR来重新建立与物理地址的映射关系。



开机时硬件给每一个LAPIC赋予一个ID号，可以用过CPUID指令将ID号存放在EBX寄存器的31-24位。



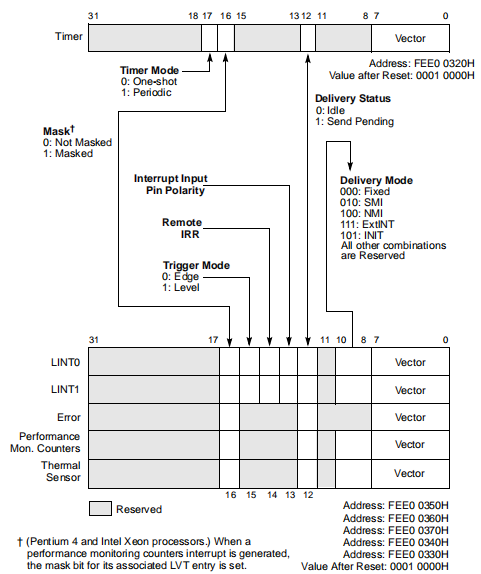
8.5节开始讲如何处理中断。

首先要理解的一个概念是LVT，

需要说明的是LINT0和LINT1这两个寄存器，它们对应到Local APIC模块的INTR和NMI引脚，外部的中断会引发这两个寄存器的中断发起。

图中的Vector就表示中断向量号（0-255），其中16-255是有效的。

所以，这里LAPIC主要是接受外部中断，然后按照INT0和INT1设置的vector号，把中断信息传入内部的处理器核，处理器核就会接收到一个外部中断，且中断号就是上面配置的vector。



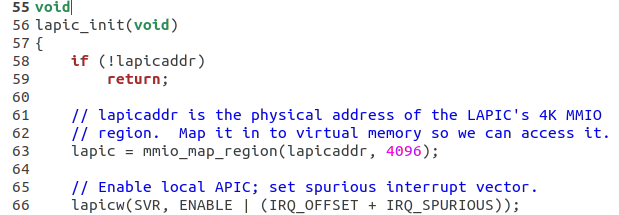
Mpconfig.c里面有很多和硬件相关的代码，那得深入理解这部分体系结构的接口才能看懂了。

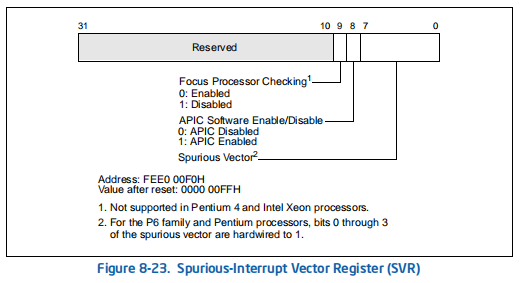
先把lapic.c看懂吧。

这开始两行就不简单，首先是调用要求我编写的mmio\_map\_region函数，把lapicaddr这个物理地址开始的4K区域，映射到MMIOBASE里面。

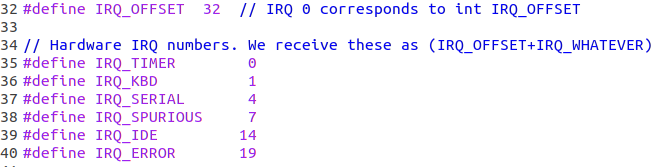
然后lapicw函数对LAPIC寄存器进行写操作。SVR寄存器用于开启和关闭LAPIC，这三个宏的含义可以对照后面的手册上的图来看。

#define ENABLE 0x00000100 // Unit Enable



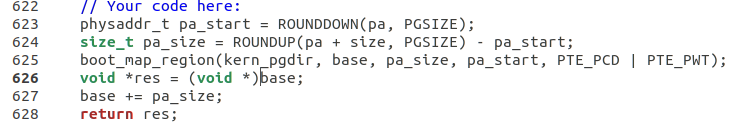


相应的宏定义在inc/trap.h中，结合这几个信息来看，就是这个语句把SVR寄存器的APIC位置1，开启LAPIC功能，然后伪中断向量置为32+19，当然最后有没有配置和这个伪中断相应的中断服务程序现在还不清楚。



说实话，这个lapic.c里面的内容比较难，一时很难看懂。但完成这个exercise 1已经足够了。

注意返回类型是void \*



应该就是调用boot\_map\_regioin来实现静态映射。

关于这个地方的物理地址映射其实有个注意点，LAPIC是映射到0xfee00000这个物理地址，但显然模拟器只开辟了256MB的物理内存空间，那会有问题吗？其实这里要知道，物理地址本身也并不完全是实在的，因为CPU已经配置好了LAPIC，所以访问这个物理地址的时候，不会区访存而是会访问LAPIC寄存器！但如果访问0xf0000000这么高的“物理”地址的话，由于这里没有映射到什么设备上，所以就会报错了。当然，在操作系统的分页式内存管理下，不出问题的话不会访问超出256MB的物理内存地址，因为page\_alloc不会分配那样的空闲块。

#### Exercise 2

BSP在环形其他AP之前要先获取所有其它处理器的信息，这些在kern/mpconfig.c中的mp\_init函数实现了，它是通过读取BIOS区的内存中的MP配置表来获取这些信息的。

我稍微看了下，反正这是真的难，要看懂恐怕要花很多的时间。

这个exercise 2主要看kern/init.c和kern/mpentry.S。

Boot\_aps函数驱动AP的引导程序，AP也是先运行在实模式下。Boot\_aps函数把AP的引导程序mpentry.S的代码装载到0x7000（MPENTRY\_PADDR），并设置AP从这里开始执行。

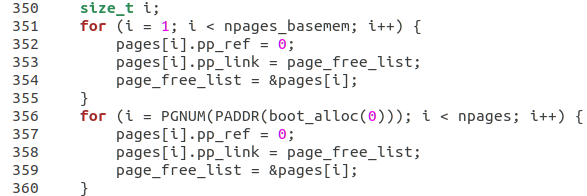
Boot\_aps一个接一个激活AP，先给第一个AP发STARTUP的IPI，同时还告诉它CS：IP。等待第一个AP启动完毕后，struct CpuInfo的cpu\_status字段会有CPU\_STARTED的标志位。此时boot\_aps继续激活下一个AP。

这个练习需要先仔细阅读boot\_aps、mp\_main这两个init.c里面的函数，以及kern/mpentry.S。

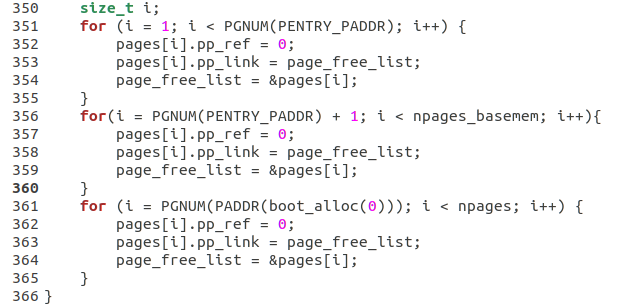
之后，要修改pmap.c中的page\_init函数，来避免把MPENTRY\_PADDR加入空闲链表。

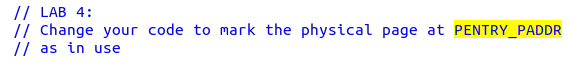


Memlayout.h中定义了这个宏。



要去掉的块应该术语basemem这个区域，修改第一个for循环。





注意虽然这里注释写的是PENTRY\_PADDR，但实际的宏是MPENTRY\_PADDR。

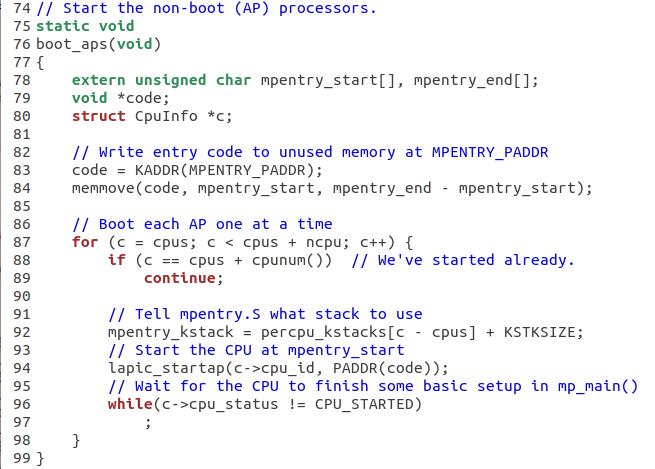


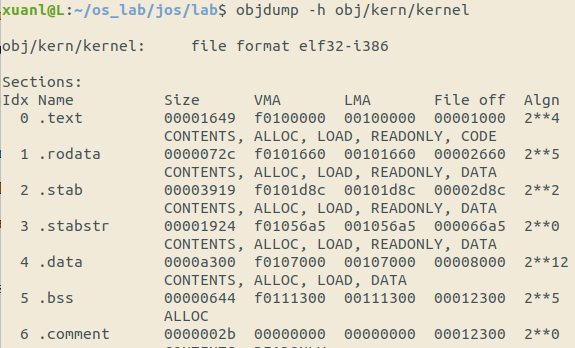
这样就能通过前3个check了。

接下来阅读代码。

首先是进入boot\_aps，然后使用papic\_startap来激活AP并执行apentry.S里的代码。

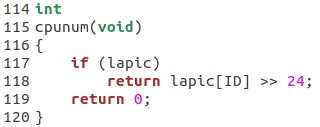
这里要注意，为什么可以直接用memmove把代码移动到内核的一块虚拟地址而不会产生冲突。

回忆lab1中链接程序输出的各个段的虚拟地址可知，所有的段都被装载于0x00100000以上，其虚拟地址都在0xf0100000以上。而MPENTRY\_PADDR对应的虚拟地址为0xf0007000开始的区域，属于内核不使用的空隙，因此可以直接把代码复制到这一部分而不用担心产生冲突。此外，物理地址0x000A0000一下的部分属于low memory，可以任意使用，因此直接复制代码没有问题（再往上是vga映射区）

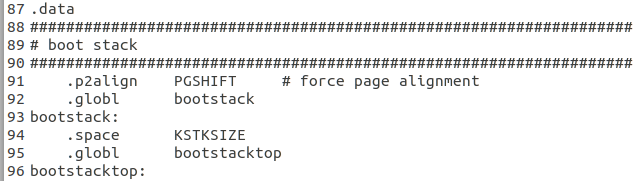


然后这里面的cpu数组，在cpu.h里面有，是在mpconfig.c里面进行初始化的。

之后有时间最好把mpconfig.c好好看一遍。



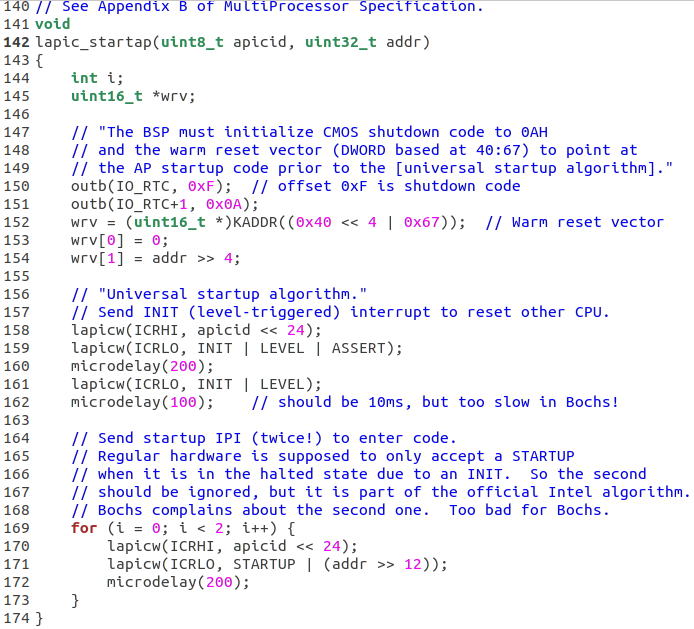
这里面的cpunum函数，就是获取编号为ID的LAPIC寄存器的值，并取其高8位作为当前CPU的编号。这在上面的截图里面有，是奔腾4及之后的处理器架构才是高8位，再之前是24-27。



此外，在这个实验之前，BSP使用的内核栈直接就是CPU0虚拟地址的那部分，但实际上BSP未必就是CPU0。所以这里需要遍历所有的CPU，当cpuname相等时说明就是当前CPU的编号。BSP的内核栈的物理地址就是bootstack，这是内核装载好之后就有了的。估计在这个实验中，BSP可能不是CPU0。

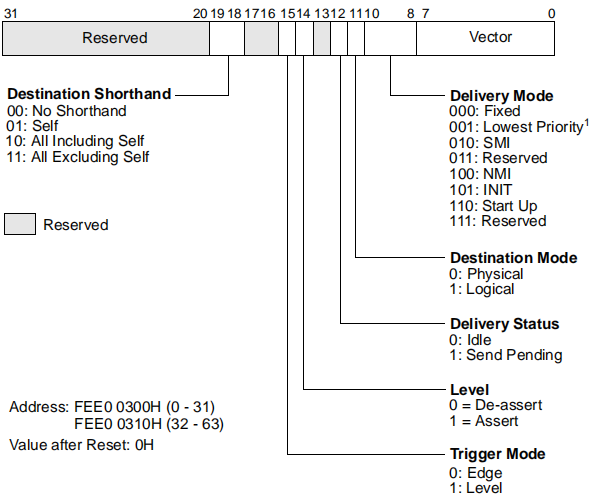
最后如何激活AP呢，是调用lapic\_startap函数来实现的。这个函数就真的难了，也属于不太能看懂的函数。

这个函数里设计到一些硬件相关的设置，实在是难顶。总之这个函数能够让apicid编号的CPU从物理地址addr处开始执行。

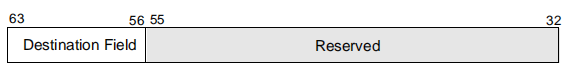


上面的代码中，一开始一段是配置CMOS，与硬件和模拟器相关。

下面是对ICR寄存器的操作，ICR寄存器有64位，其中低32位是这里的ICRLO。

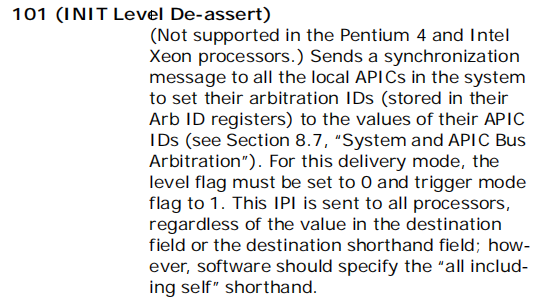


高32位则是ICRHI

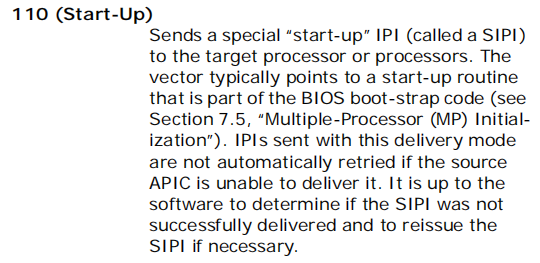


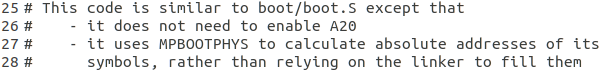
所以上面这段代码，一开始对目标CPU发送INIT信号，而对所有CPU再发送INIT Level De-assert信号。这两个信号的区别就是Level和assert的设置。

INIT使得目标CPU初始化，而INIT Level De-assert信号则让其余CPU设置仲裁ID。暂时不太清楚为什么要设置仲裁ID。可能在8.7节的APIC总线仲裁能找到答案。



最后是向目标CPU发送Start-Up信号。在手册上可以找到，这个信号的vector字段指向bootstrap code的start-up例程。根据实验给出的代码来看，就是指出这个CPU是从哪个物理块开始执行（0x7000正好是第7块开始，而右移12位后的值为7，就是块号）。详细信息得参考手册的7.5节，关于多处理器初始化。

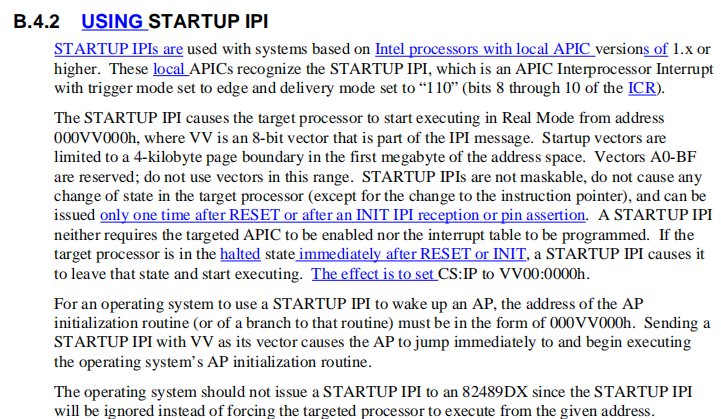


然后，在另一个参考资料，Multi-processor Specification中提到，STARTUP IPI将使得目标处理器从地址0x000VV000开始执行，其中VV是IPI中的8位vector，至此就完全说明了上面这段代码的含义。

注释中给出了mpentry.S和boot.S的不同之处，主要有上面这两点。

首先不用开启A20，我认为是A20开启与否并不取决于单个CPU，而是BIOS的工作，也就是说BSP开启A20之后，这根地址线就对所有CPU来说都开启了，而不会出现每个CPU都有各自的A20的情况。但是，因为当前AP运行于实模式，所以开启A20它也依然无法访问1MB以上的内存。

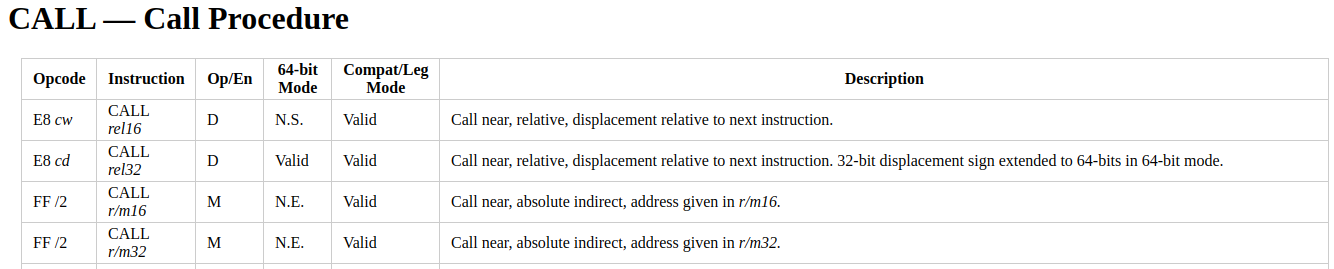
使用MPBOOTPHYS是因为当前代码的运行地址不是连接器装载时的地址，而是被OS移动过了的，所以不能直接用连接器提供的符号表来给出地址，而只能用相对地址计算偏移。



最后mpentry.S有一个小练习



问为什么要用寄存器寻址来call，其实这个问题我在LAB1的时候在看kern/entry.S的时候就思考过了，当时在Part3里的jmp也使用了寄存器间接寻址。



Call指令表的一部分如上图所示。可见，如果使用立即数寻址的话，对应的机器指令使用相对寻址方式。当前EIP在1MB一下的低地址区，而要跳转到的地方是KERN\_BASE以上的高地址区，相差了3GB以上，而32位数相对寻址只能跳转±2GB，因此无法实现

由表可知，使用r/m32是absolute indirect，也就是用间接寻址方式获取绝对地址。所以，这里要么使用寄存器间接寻址，要么使用内存间接寻址（即call \*$addr之类的）。显然寄存器间接寻址快很多，所以这里的call指令使用了寄存器间接寻址。

#### Question 1

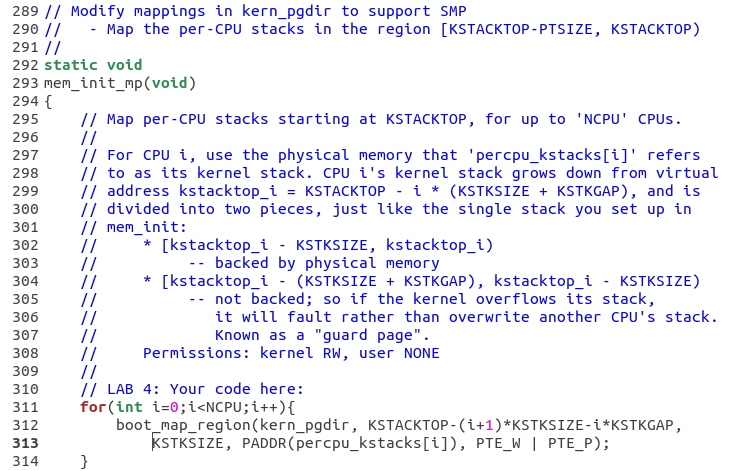
Compare kern/mpentry.S side by side with boot/boot.S. Bearing in mind that kern/mpentry.S is compiled and linked to run above KERNBASE just like everything else in the kernel, what is the purpose of macro MPBOOTPHYS? Why is it necessary in kern/mpentry.S but not in boot/boot.S? In other words, what could go wrong if it were omitted in kern/mpentry.S?

Hint: recall the differences between the link address and the load address that we have discussed in Lab 1.

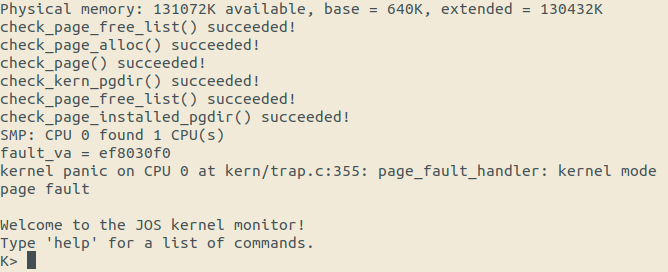
在boot/boot.S中，使用的是#define RELOC(x) ((x) - KERNBASE)这个宏，那能否用这个宏来取代这里的MPBOOTPHYS呢？答案是否定的。原因就在于，boot.S的load addr就是实际的物理地址，即0x7c00。但是，对于mpentry.S来说，其load addr不是实际的物理地址。为什么呢？是因为其实际的地址是被OS重新使用memmove函数移动到0x7000这个地方的，而不是链接的时候就装载到这个地方的。因此，不能直接使用连接器提供的符号表来获取地址，而只能使用偏移来获取地址。

#### Exercise 3

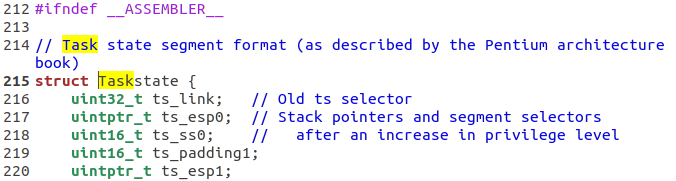
这个练习要求把每个CPU的内核栈映射到二维数组percpu\_kstacks上。



这样就可以通过check\_kern\_pgdir的测试了。



#### Exercise 4



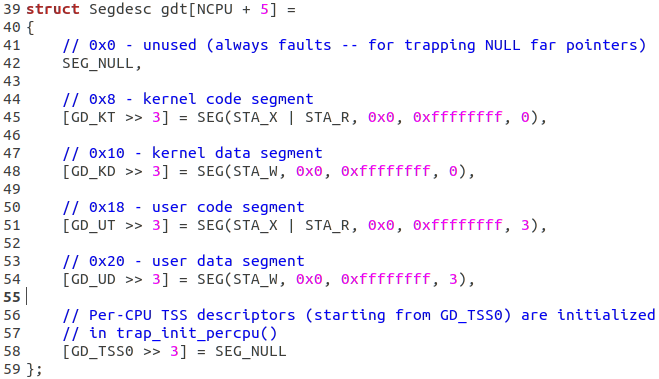
在mmu.h里面定义了Taskstate数据结构，非常长，上面仅是一小部分。

然后又忘了GD\_TSS0定义在哪里了，使用命令

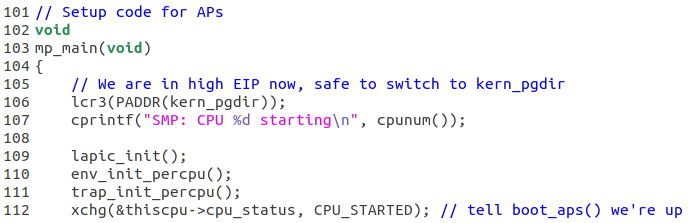
find . -name '\*.h' | xargs grep "GD\_TSS0" 找到这个宏定义在memlayout.h中



同样，也可以找到gdt数组的定义是在env.c中。



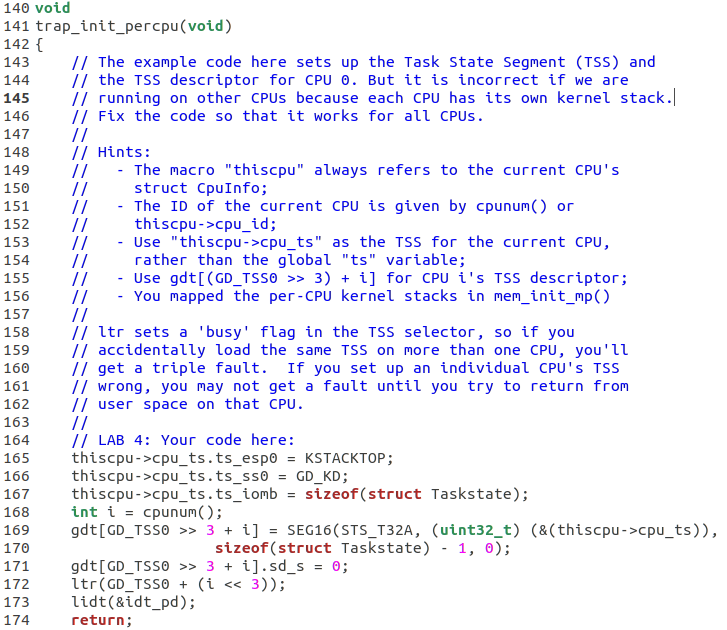
所以，GD\_TSS0就是CPU0的TSS描述符相对于gdt数组开头的字节偏移，又因为每个段描述符占8字节，所以其索引就是GD\_TSS0 >> 3。



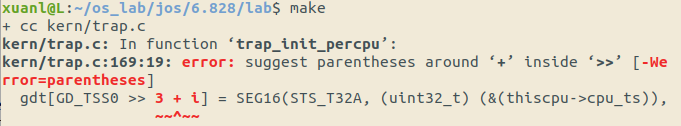
这里要注意回去看mp\_main，可以发现所有的CPU都共享同一个页目录，但其实每个CPU内部都有一个cr3寄存器，只不过它们的值都是一样的。

这里每个CPU都会调用trap\_init\_percpu()，而CPU0本身在之前调用了trap\_init，在trap\_init最后又会调用trap\_init\_percup，所以这个函数应该根据执行它的CPU的编号不同，作出不同的选择。

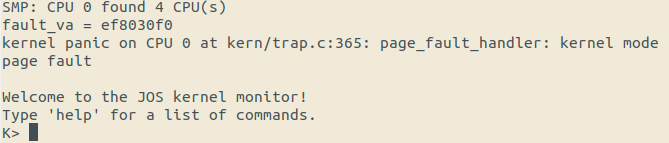


Thiscpu宏本质还是调用了cpunum来获取cpu的信息，所以肯定比cpunum()来的慢。

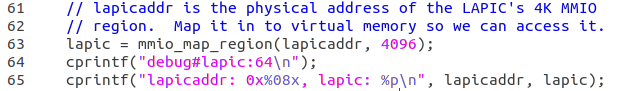
当然这里暂时不关注性能了，否则的话应该提前把cpunum()计算好，然后不调用thiscpu。

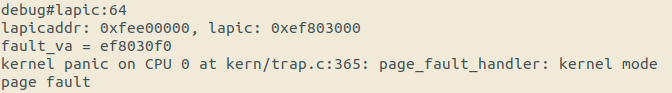


编译时遇到一个warning，这里确实有问题，没加括号，而+的优先级是高于移位符的！所以必须加括号。本来这不是语法错误，但却被检查出来了，可见gcc还是很高级的。



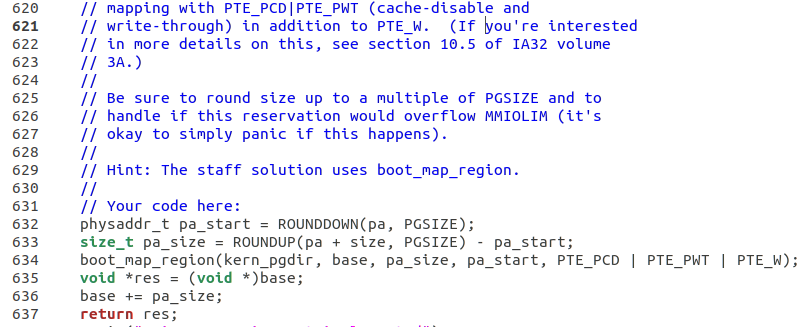
但还是跑步起来，CPU0遇到了缺页中断！访问0xef8030f0的时候出现缺页中断。





根据调试信息来看，是mmio\_map\_region的返回值错了，应该返回0xef80000才对，但却返回的是0xef803000。

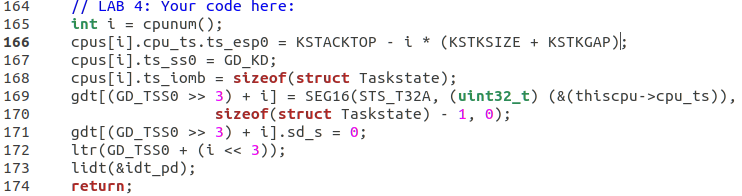
但好像这是因为调用了check函数，所以不从0xef80000开始，那么还是映射的问题？



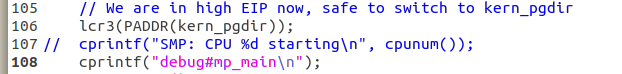
结果发现是英语的问题，in addition to PTE\_W是说除了PTE\_W还需要PTE\_PCD和PTE\_PWT，然后我没加PTE\_W，所以没有写权限导致缺页异常。

但接下来又遇到了老朋友，Triple fault！

显然是代码的第一行就不对，每个CPU的内核栈的虚拟地址显然是不同的。

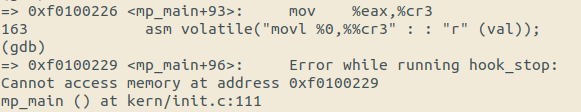


发现调用cpunum()就会出问题。



调换一下位置，其实是lcr3没有成功执行，为什么装载不了kern\_pgdir呢？

用gdb检查，也发现修改了cr3之后

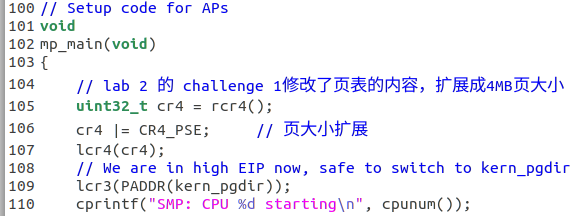


最后终于过了，因为回去仔细检查了pmap发现在实验2里配置的扩展页表与这里的代码有充足。

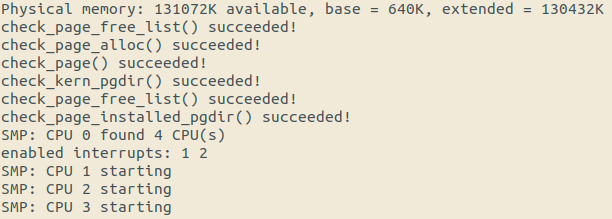


其实实验指导已经提示地很清楚了

修改mp\_main代码，对每个AP都要修改cr4寄存器的值。



结果终于正确了。

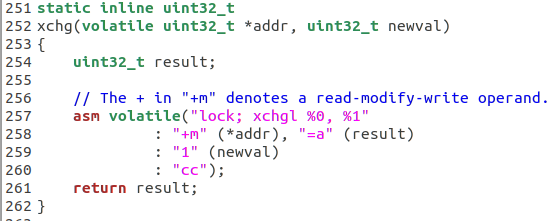


#### Exercise 5

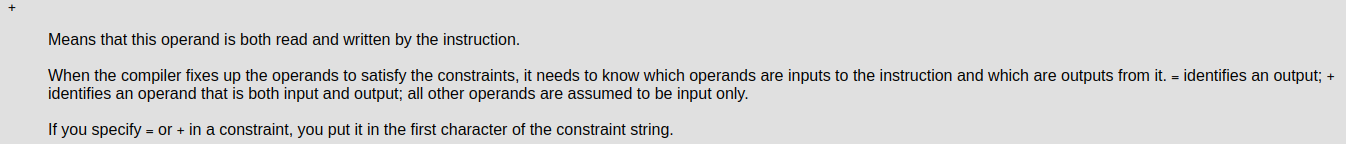
这里可以看一下，内核加锁很简单地使用了xchg指令，



这里对应的内联汇编又有些看不懂了。



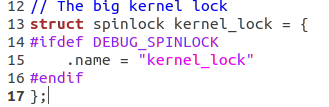
查阅资料后知道，lock指令用于宣告原子操作。为什么要宣告呢？因为即使加锁和解锁都只需要一条指令就能完成，但是在多处理器环境下，在一条指令执行过程中，内存的数据依然可能被其它处理器读写。因此，需要使用lock指令宣告某个CPU锁定一块内存地址。



而+的含义，可以参考TIGCC的C扩展文档中的内联汇编部分（[http://tigcc.ticalc.org/doc/gnuexts.html#SEC94\_asm](http://tigcc.ticalc.org/doc/gnuexts.html" \l "SEC94_asm)）

这个练习需要在i386\_init、mp\_main、trap以及env\_run函数中加锁或解锁。

首先是i386\_init。



所有的CPU共享上面的全局变量kernel\_lock

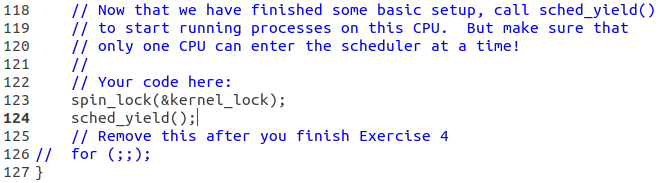
注意传递的参数应当是其地址！

I386\_init()

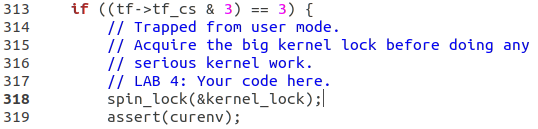


注意是kernel\_lock不是kern\_lock。

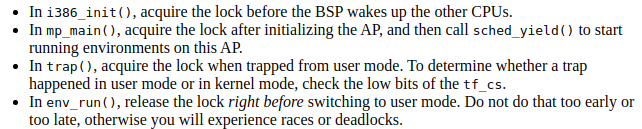
Mp\_main()



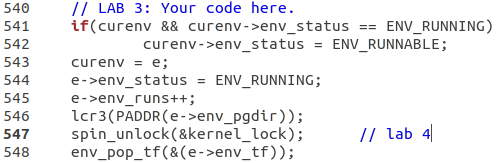
Trap()



Env\_run()



官网上提到要注意不要造成竞争和死锁，但这里全局的临界资源只有kernel\_lock一个，感觉不会造成死锁？



#### Question 2

It seems that using the big kernel lock guarantees that only one CPU can run the kernel code at a time. Why do we still need separate kernel stacks for each CPU? Describe a scenario in which using a shared kernel stack will go wrong, even with the protection of the big kernel lock.

这个问题是说既然内核加了锁，使得所有CPU中只能有一个处于内核态，那么为什么要区分不用的内核栈。

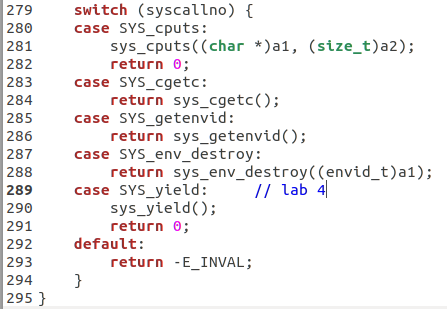
其实就是在执行spin\_lock之前，可能就已经有多个CPU处于内核态了。如果共用一个内核栈，会导致函数调用出现问题。

最有可能发生的或许就是spin\_unlock函数本身出现问题。在调用完成spin\_unlock函数之后，该CPU需要通过栈中的数据来返回到原函数。而因为执行了unlock，所以其它CPU也会进入内核态开始执行代码。此时如果共用内核栈的话，就会导致后来的CPU破坏了之前栈中的数据，导致spin\_unlock无法正常返回。

当然，肯定还有其它很多情况会导致多处理器之间的冲突。总之不能共享内核栈。

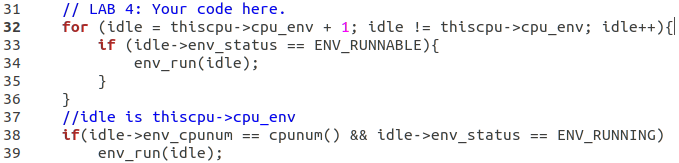
#### Exercise 6

完成sched\_yield函数，并完善syscall函数。

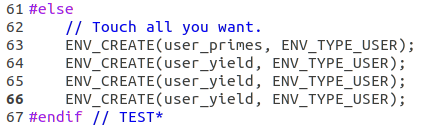


在syscall中新增SYS\_yield项。

然后完成sched\_yield函数



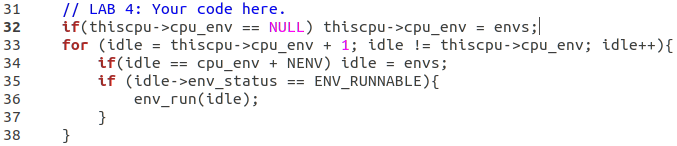
然后在init.c中新建3个yield环境



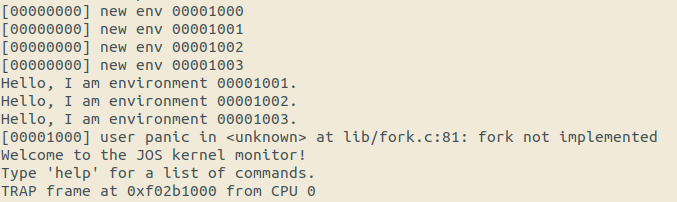


运行之后还是CPU 0出现缺页异常。

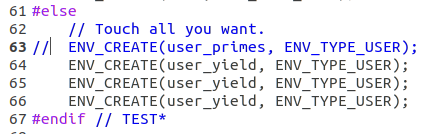
检查之后，暂时先找到两个问题，一是如果这个cpu本来就没有运行什么环境，那就从envs开头遍历。另外，如果idle越界的话，而再从头开始。

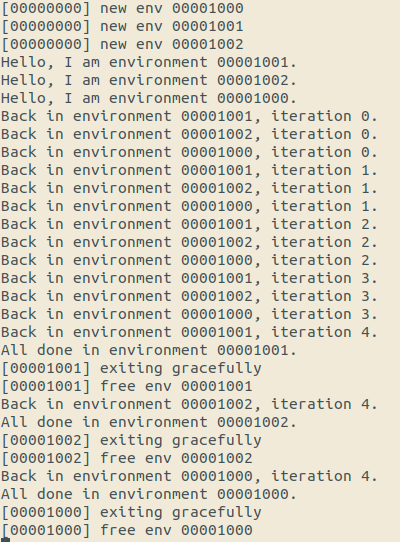


改了之后能正常输出了，然后遇到了fork没有实现的panic，不知道对不对。



之后把user\_prime注释掉，结果就和官网上将的类似的





最后一直卡住，因为上面的代码还有问题。首先，如果thiscpu->cpu\_env是NULL的话，那不能初始化为envs，而应该envs - 1，因为刚进入循环的时候是要先+1的。

其次，这种情况如果所有的环境都退出了，就会陷入死循环。所以，要额外判断本来是NULL的情况。

但修改之后，虽然能进入monitor，但并没有出现异常的情况。检查发现，只有CPU0进入过halt函数。那么其它CPU此时在干啥呢？

应该是死锁！因为检查sched\_halt的代码可以看到unlock\_kernel函数，这个调用在调用monitor之后，那CPU0进入monitor之后如果没有unlock的话，其它CPU肯定都被锁住了。

顺带一提，其实前面的加锁和解锁可以直接用lock\_kernel和unlock\_kernel这两个封装好的函数来实现。这里考虑到了模拟器的性能，在unlock\_kernel里面故意加入了时延，使得不同CPU能够交替。

这么说来感觉没问题。但还是会陷入死循环，上面的循环里面还是不对，应该是envs - 1。

最后检查发现，确实其它的CPU都进入halt状态了，并没有出现异常。暂时不管官网所说的异常，先继续做下面的实验。

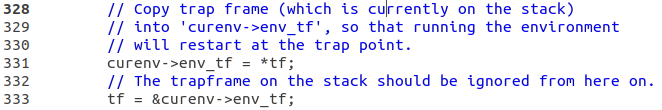
#### Question 2

1. In your implementation of env\_run() you should have called lcr3(). Before and after the call to lcr3(), your code makes references (at least it should) to the variable e, the argument to env\_run. Upon loading the %cr3 register, the addressing context used by the MMU is instantly changed. But a virtual address (namely e) has meaning relative to a given address context--the address context specifies the physical address to which the virtual address maps. Why can the pointer e be dereferenced both before and after the addressing switch?

这里的环境指针e，指向的是数组envs中的某一个元素的地址。而数组envs本身是在内核区的，即KERN\_BASE以上的部分。这一部分无论使用内核的kern\_pgdir，还是使用用户的user\_pgdir，都有对内核部分的映射。因此，切换页目录对于指针e的解引用是没有影响的。

1. Whenever the kernel switches from one environment to another, it must ensure the old environment's registers are saved so they can be restored properly later. Why? Where does this happen?

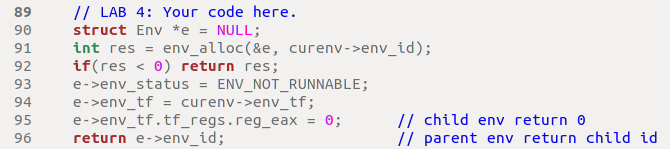
这个问题是个好问题。这是在trap（）函数中完成的，把trap frame结构保存到curenv的env\_tf结构中，而curenv指针指向的就是引发中断的环境在数组envs中的位置。所以这样就把中断帧存入了对应环境的数据结构中。



#### Exercise 7

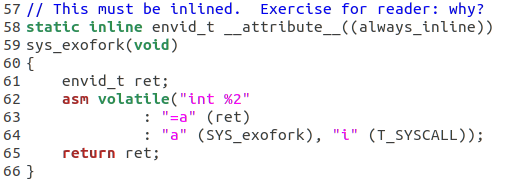
Part A的最后一个Exercise了，主要完成kern/syscall.c里面的与fork实现相关的函数。

首先完成sys\_exofork



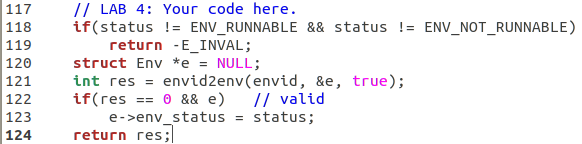
这里注意子进程的寄存器都是复制父进程的，除了eax寄存器的值应当置为0，因为这会作为子进程的fork()函数的返回值，因为子进程不会通过这次中断激活，而只会在之后调度的时候运行。而父进程是通过这次中断回去的，所以只需要用返回值就行了。

另外，在lib/syscall.c中是没有这个函数给用户调用的，而是直接内联在inc/lib.h中

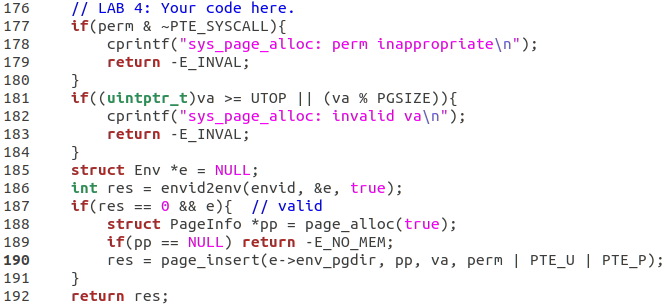


这里为什么一定要是内联的，还不太清楚，先继续往下做。

接下来完成sys\_env\_set\_status，这里我在思考可否出现envid = 0，然后e返回NULL的情况，也就是curenv是NULL.应该是不会的，因为能调用这个函数应该是用户态执行中断自陷进入的，所以curenv不应该是NULL。



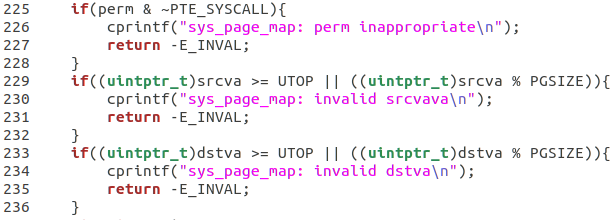
然后是sys\_page\_alloc

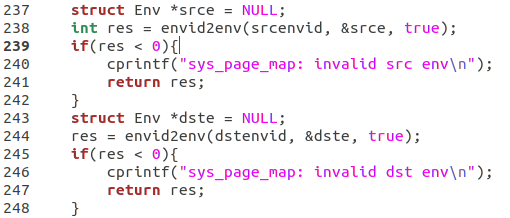


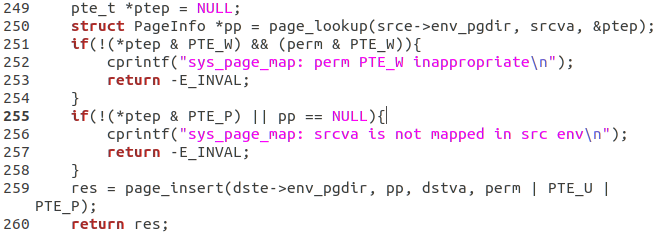
然后是sys\_page\_map，要注意不能把写权限加到一个只读的页上。

这里注释里经常出现一个单词，wrapper，这就是封装的意思。

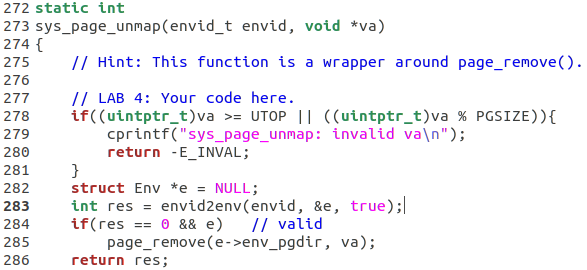
然后是sys\_page\_map，大部分代码都用来检查参数了。



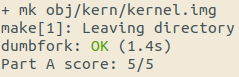




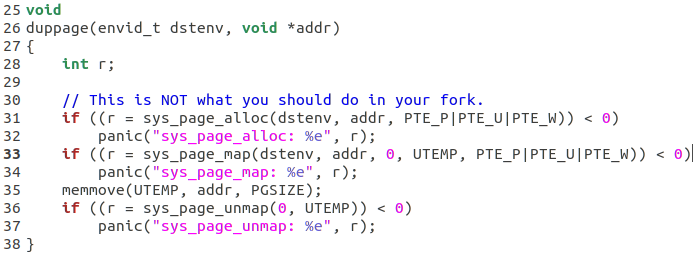
最后是sys\_page\_unmap函数



这样就通过了Part A，虽然只有5分。。。

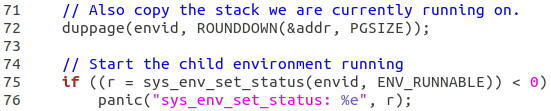


完成了这里之后，再仔细看一下dumbfork，可以看到这里实现代码复制的方法，是用duppage。



首先给子进程在对应地址处分配新的物理块，然后把父进程的UTEMP区域映射到新的物理块，接着将父进程的这一页的数据复制过去，最后再取消映射关系。这样就实现了将父进程的代码段和数据段复制给子进程的效果了。（当然，实际的操作系统似乎是不复制的，而是增加引用数）

最后，在复制完堆栈之后，将子进程设置成ENV\_RUNNABLE。这里直接把局部变量的地址作为栈地址，应该是考虑到这个程序所占用的栈空间不会超过PGSIZE，或者是栈帧有某种对齐方式，但这里应该是前者。

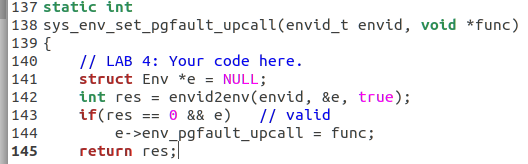


这里就解释了为什么sys\_exofork函数必须是内联的。因为复制栈数据在sys\_exofork调用之后。也就是说，在复制栈的时候，父进程已经从sys\_exofork返回了。这时候复制栈的话，会导致子进程的栈数据错误，无法从sys\_exofork函数返回！而内联之后，子进程的当前栈帧就是dumbfork函数，和父进程相同，因此能正常返回。

### Part B

#### Exercise 8

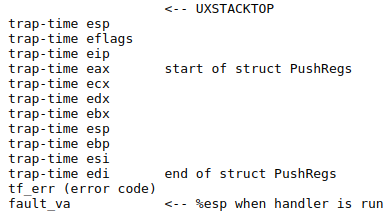
接下来是sys\_env\_set\_pgfault\_upcall，这里可以去trap.c里面看一下，在处理缺页异常的函数里面，还有一段需要增加的代码，但那应该是在后续的练习中实现。



此时能通过前两个测试。



#### Exercise 9



在用户异常栈中建立栈帧并跳转。要注意如果异常的栈帧本身就来自用户异常栈，那么需要在栈帧下继续而不是从UXSTACKTOP重新开始。首先先填充空的32位字然后继续。

注释提示里提到了使用env\_run和user\_mem\_assert。其实后者也在pmap.c中实现了。

这里最后调用env\_run函数，所以处理缺页的函数是用户态的。

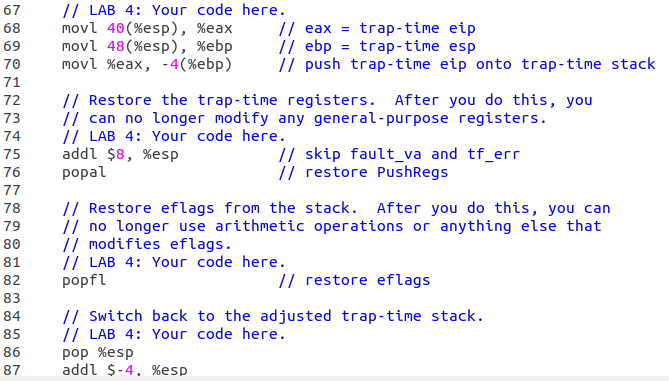


暂时先这样吧，这段代码还有些疑惑。

上面这段改过了，本来的话是直接没有减去sizeof(struct UTrapframe)，那样肯定不对。要注意变量赋值给出的其实是低地址

#### Exercise 10

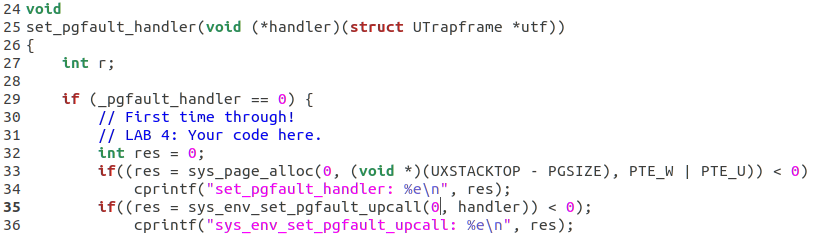
这一段汇编太精彩了。



#### Exercise 11

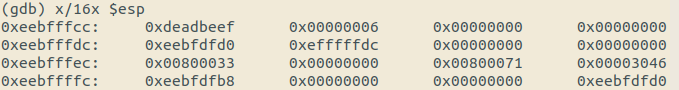


但是通过不了测试。

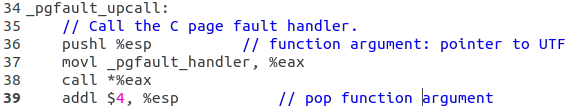


上面那显然太少了，根本没给当前进程设置缺页处理函数。

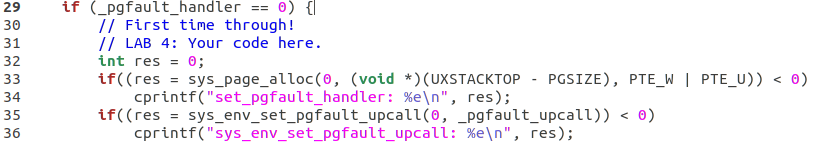
结果陷入死循环。



检查发现，handler函数的第一个参数没有压入栈。原因是上面的代码不能直接设置handler为用户的缺页处理函数，而应该设置\_pgfault\_upcall为处理函数，然后再由\_pgfault\_upcall来调用\_pgfault\_hadler。因为把参数压栈的操作是由\_pgfault\_upcall来完成的。



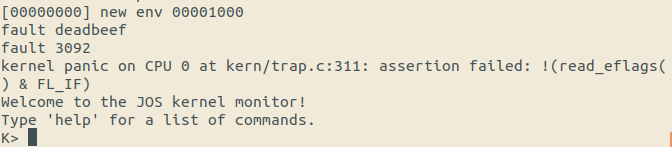
所以这里需要改成下面这样。



这样终于通过了faultdie，但是下面的测试还是有问题。

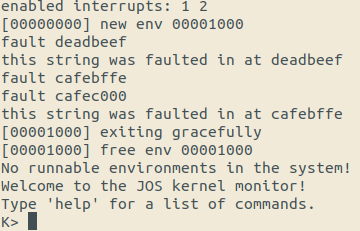


Make run-faultalloc直接整出一个开中断的问题，这问题大了。

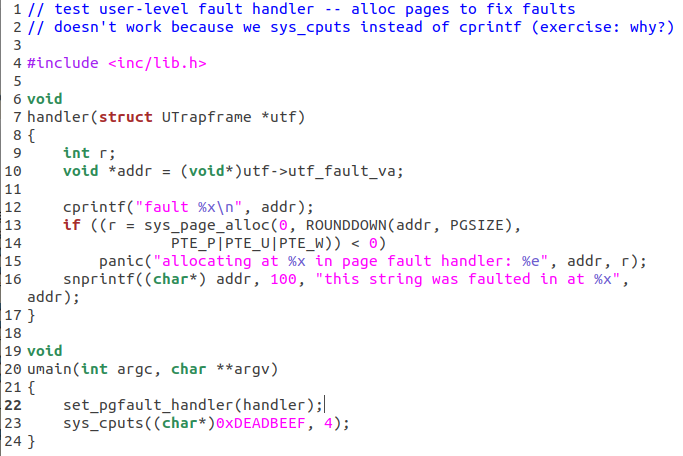




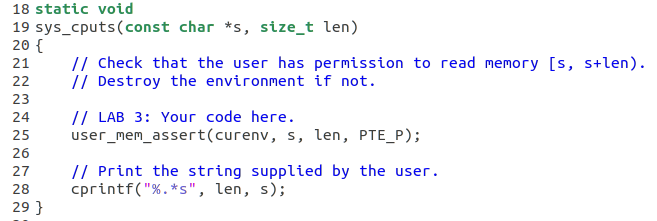
原因因该是忘记在popf之前还有一个eip需要跳过。



这一下行了



在faultallocbad里面，使用sys\_cputs替换了cprintf，然后就会无法通过内核的检查。这里有一个小练习，就是弄清楚为什么会这样。

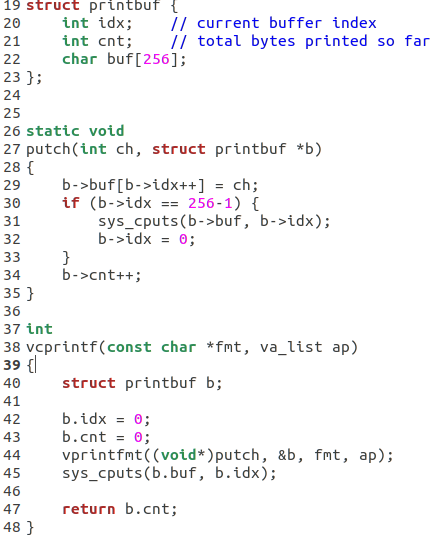


对于syscall来说，它此时已经进入内核态了，如果这时候发现当前用户没有权限访问地址的话，就会直接销毁这个进程，这可以由上面的代码看出。

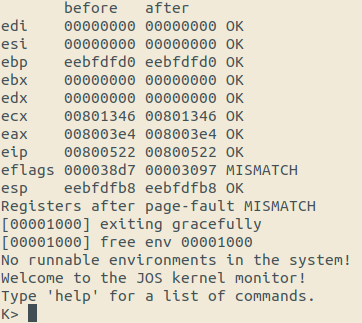
然后如果用户是调用cprintf的话，可以看lib/printf.c和lib/fmtprintf.c这两个文件，可以发现给用户的printf函数，其中的putch函数是先将用户要输出的送入循环缓冲区，等到缓冲区满了或者字符串结束了，再调用sys\_cputs函数。因此，在输出到缓冲区的过程中，就会检测到缺页异常，然后就会触发缺页处理函数。

而如果直接调用sys\_cputs的话，就直接进入内核态再检查，此时检查出现问题就直接销毁进程了。

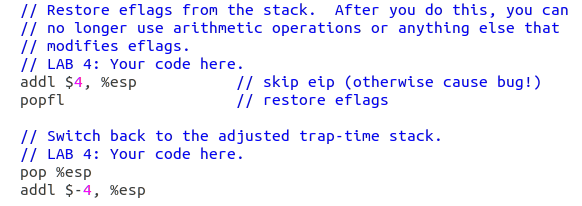
这就是两种写法的区别。



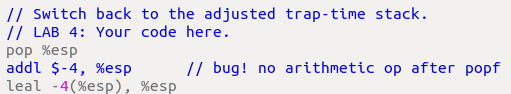
最后，对于faultregs还是不行



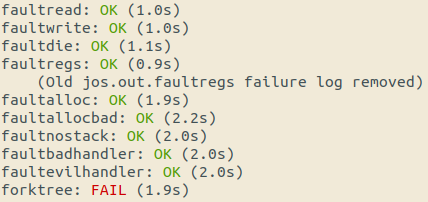
可以看到，eflags的值有问题。



这是因为在恢复了eflags之后还进行了算数运算！



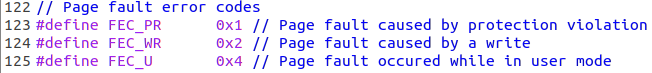
改用LEA指令实现减法，就不会修改eflags了！



只剩下forktree了，也就是完成Exercise 12。

#### Exercise 12

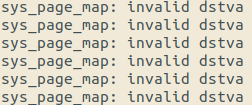
这是Part B最后的练习了，实现写时拷贝技术。



需要用到mmu.h中关于error code的宏。

结果超时，这是第一次遇到这种情况。

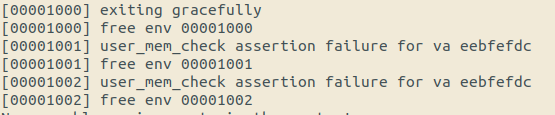




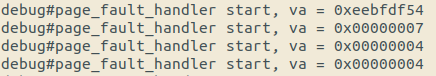
可能是因为这里的addr没有ROUNDDOWN，所以没页对齐导致。



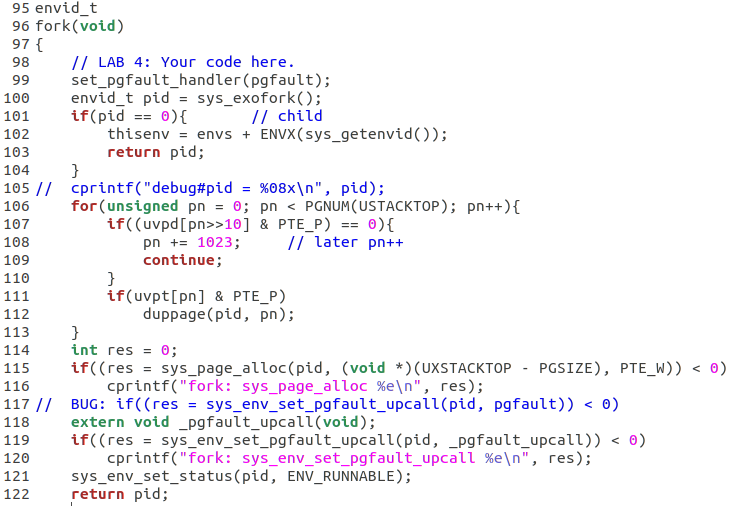
虽然不会死循环了，但还是有问题。

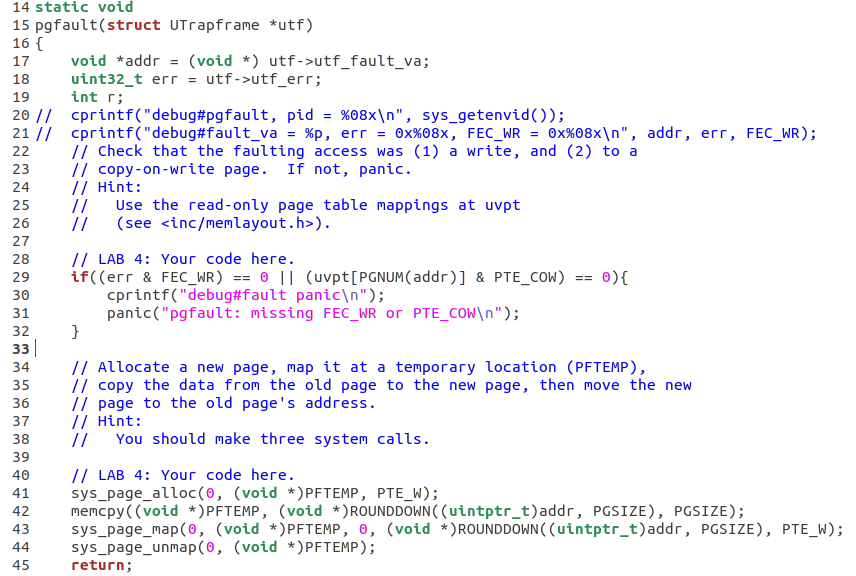


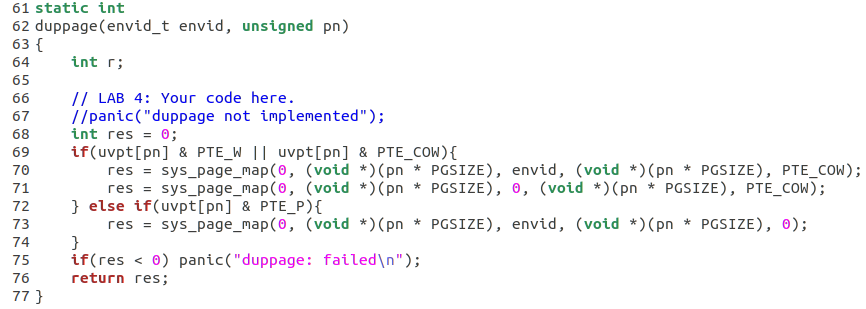
调试了好久，后来把缺页地址打印出来

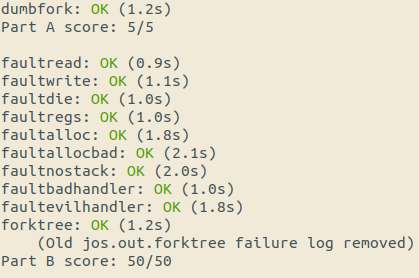


又是老问题了，sys\_env\_set\_pgfault\_upcall函数不能直接掉目标函数，而是应该调用\_pgfault\_upcall！









不容易啊！终于完成Part B了，太爽了。但真的难顶啊。

### Part C

有16个外部中断请求（IRQ），在实验中，它们对应的中断向量号是32-47（其他系统中未必）。

其中，IRQ 0就是时钟中断，其处理函数就是IDT[IRQ\_OFFSET + 0]，这里IRE\_OFFSET = 32，定义在inc/trap.h中。

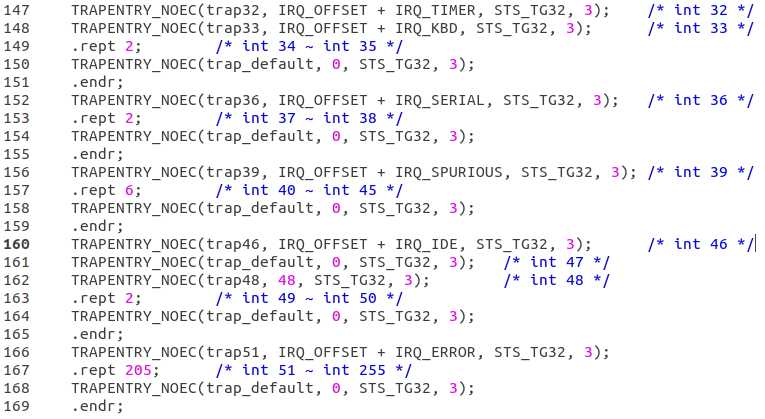
在JOS（而不是xv6 UNIX，这两个是有区别的）中，当运行于内核态时始终关中断，所以外部中断请求都被忽略。

自从在boot中关中断以来，其实中断标志位FL\_IF就一直没有打开过，直到现在这个实验，终于要开中断了。这也更加加深了一个概念，开中断和关中断是对于外部中断而言的，而之前处理的所有中断都是内部中断。

#### Exercise 13

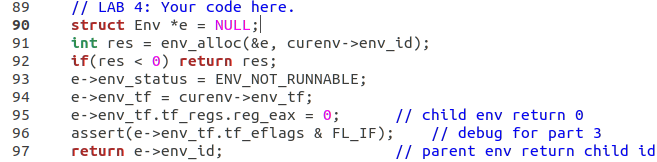
修改trapentry.S和trap.c，增加外部中断向量。由于之前完成了challenge，所以中断向量表的构建基本都在trapentry.S中完成，这里只需要修改前者即可。

Trapentry.S修改如下，trap.c不变。

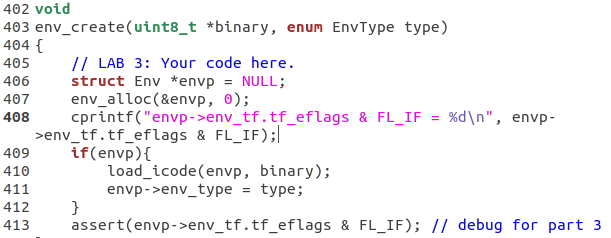


接下来修改env\_alloc()，但这里有个疑惑，上面的外部中断都是trap，那么通过陷阱门时不会自动重置IF，这样进入内核之后不就会保持开中断的状态了吗？感觉会出问题。

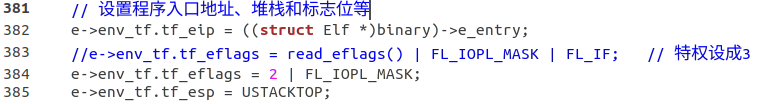


果然有问题，经过长时间的检查发现，fork之后的环境没有置FL\_IF位，在exofork里面加入assert，发现这里有问题。但其实这是复制之前环境的问题。现在的关键在于，第一个环境为什么没有开中断。

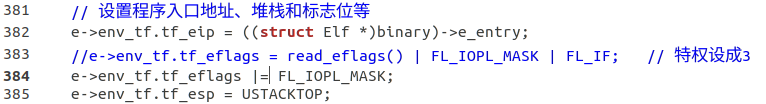
在这里插入assert居然发现在load\_icode前后变了！



最后发现真的是当年留下的祸患！



这里把FL\_IF清了。。。改成下面这样。



然后果然进入内核没有关中断。这里我感觉得设置成Int而不能是Trap。不能全部按那个表来。

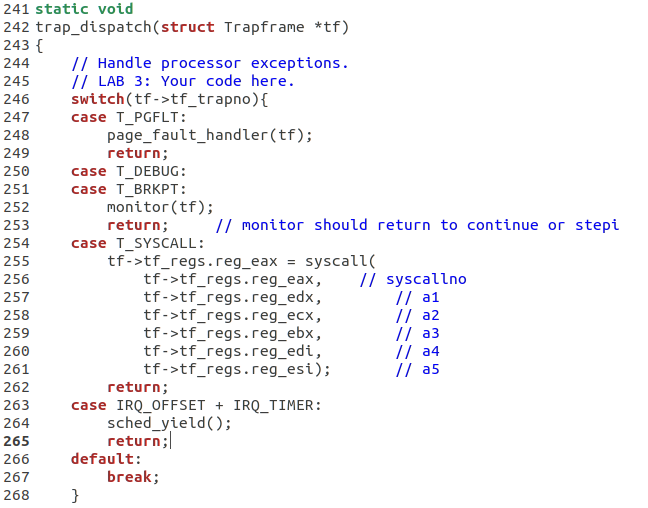




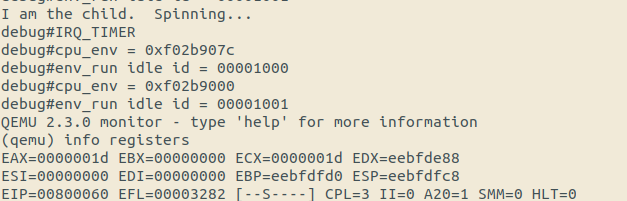
改成STS\_IG32，这样就可以进入内核自动关中断了。其它的中断也设置成这个。

#### Exercise 14

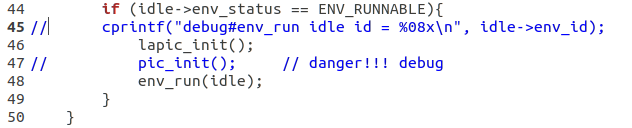
修改dispatch，这个没问题。



这次真有些没辙了，因为检查寄存器发现IF也是开着的，没问题啊。

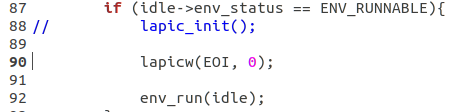


非常诡异！

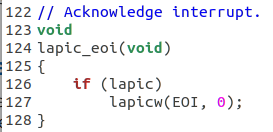


在sched.c里面的循环中加入lapic\_init函数，居然可以通过测试！这应该不是正常的做法，但本来确实也找不到什么问题。这个地方太诡异了。

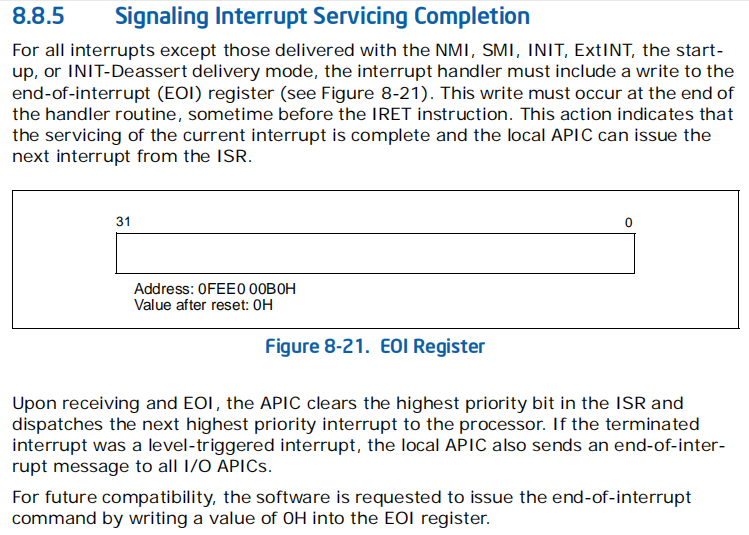
另外，stresssched也没有通过，肯定还有什么我没意识到的问题。



经过反复尝试，发现必须在sched.c里面加上lapicw(EOI,0)这一行才行。现在范围缩小到这里了。



再回到lapic.c中，发现确实提供了这个接口。在x86手册中也介绍了这个LAPIC寄存器EOI。当处理外部中断结束后，需要将EOI寄存器清0，这样LAPIC才能够继续处理下一个中断。



所以把这个重置EOI寄存器放在中断分发dispatch函数里面好了。因为其他NMI之类的中断，或者进程主动放弃CPU，是不需要重置EOI寄存器的。



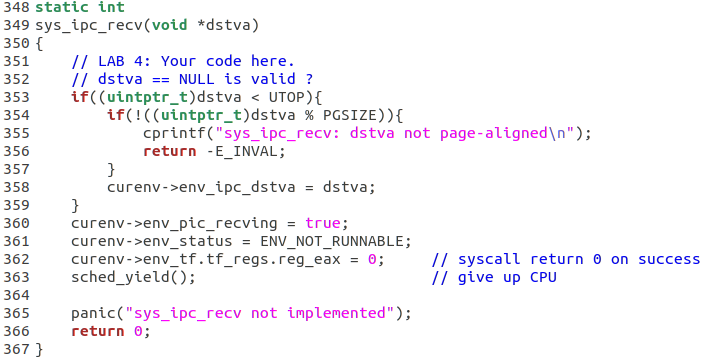
这样本来通不过的stresssched现在也可以通过了。不得不说，这个坑实在太大了。



有65/80分了，接下来只有一个进程间通信了。

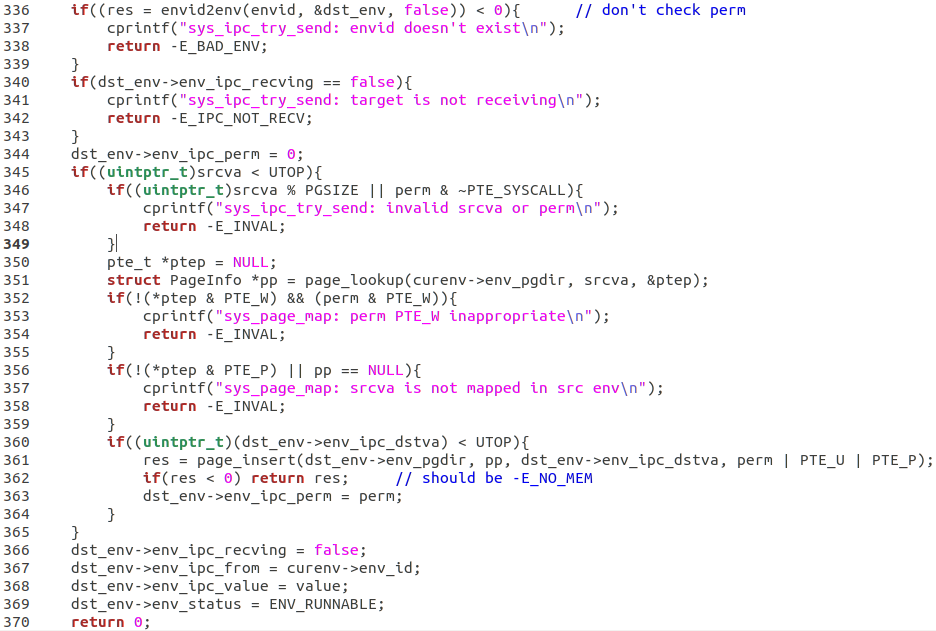
#### Exercise 15

先实现kern/syscall.c的两个IPC处理函数，然后实现lib/pic.c中的IPC库函数。

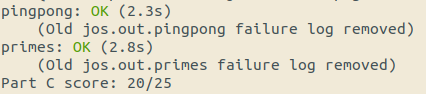


先完成了sys\_ipc\_recv。这里有个疑惑，就是如果输入的指针是NULL的话，也满足<UTOP的条件，那也算能接受共享内存映射吗？其实之后可以看到，在库函数中判断是否NULL。

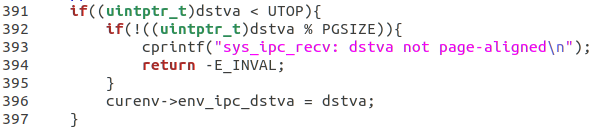
然后在完成sys\_ipc\_ret\_send，这个函数主要也是检查参数的有效性。



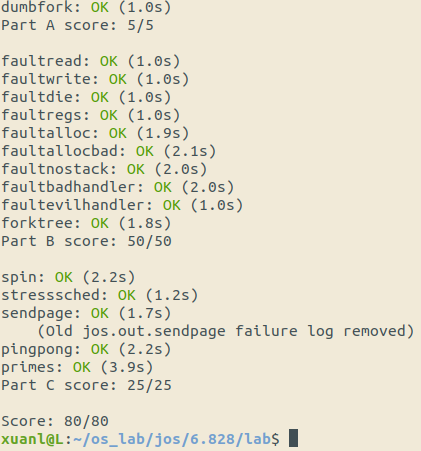
基本可以了，但前面的snedpage还是有问题。最后还差5分。

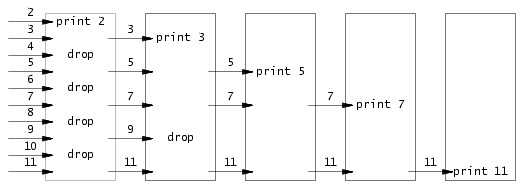


又是感叹号的问题，这里应该是模PGSIZE之后不等于0，所以不用感叹号。用!=就很清晰。



把感叹号去掉，终于完成了lab 4！太不容易了。



下面这是关于最后的prime测试程序的解释。每个进程把最先收到的数作为自己的素数，然后之后每一次收到数，都判断是否是自己这个素数的倍数，如果不是的话就传下去。

#### 疑惑1

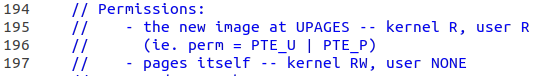


在lab 2里面，把pages数组映射成没有PTE\_U权限。但是在这个实验里面命名在用户态使用了pages数组？

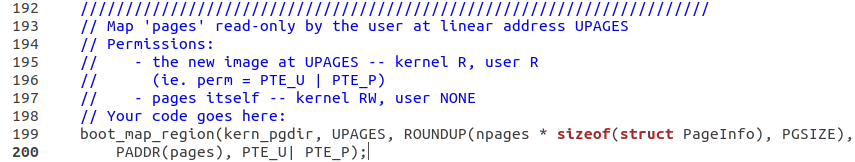
其实用户访问的upvt是当前用户页目录，而上面的pages数组是UPAGES，在这个lab 4里面也没有用到。但根据注释，还是把上面这个地方的PTE\_W改成PTE\_U比较好。因为注意到，内核对于任意一个页的读写权限是根本不受页表权限位的限制的，即使没有PTE\_W，内核也能修改内容。PTE\_W是给用户加的权限！



注意理解下面这一段英文，这里所谓的image at UPAGES，是说用户也能够访问内核的pages数组并读取内容，也就是获取权限之类的，还有物理地址之类的。但是用户不能够访问某一个页表项对应的那一页的内容！这就是所谓的pages itself，页表项可以读，页本身不能读。



所以这一段还是改成PTE\_U，符合要求，但到现在为止都没有用到。因为ULIM以上的页表项用户不需要访问，ULIM一下的页表项内核也没有，都在用户自己那儿。



#### 疑惑2

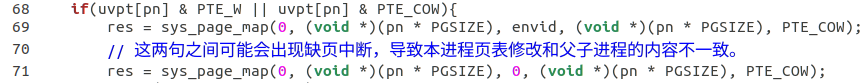
在exercise 12的指导和注释中都提到了duppage的实现过程中要注意map的顺序，还有要注意一定要记得remap，即使本来已经是PTE\_COW了。





这个地方我确实有些困惑的，因为如果只看page\_map系统调用的实现的话，其实这个操作和顺序没有关系。而且本来已经是PTE\_COW了，为什么还要再remap呢？我本来觉得会不会是避免有一页同时有PTE\_W和PTE\_COW的情况，但一方面这种情况本身不会发生，另一方面即使发生了也并不会对目前的实现造成影响。

之后，我想到了一种可能的合理一些的情况，那就是在给子进程map和给自身remap的这两个系统调用之间，是否会发生缺页中断，导致自身页表权限的修改呢？



确实有可能发生，也就是duppage函数正在复制的这一页就是用户栈本身！这个时候，对res变量的写操作会触发缺页中断和权限位的变动。这样就能回答上面的问题了。

1. 为什么不能交换顺序

交换顺序必定出错，因为在duppage复制用户栈这一页的时候，先把这一页的权限改成了PTE\_COW，然后写入res局部变量导致缺页中断。此时就把刚刚改成PTE\_COW的页又改了回去！因此不能交换顺序

1. 为什么即使原来是PTE\_COW也需要重新map

这种情况可能出错，特别是在duppage复制用户栈这一页的时候。原因和上面相同，本来是PTE\_COW，但是map了之后修改了res变量，导致PTE\_COW位被清除，因此需要重新map。

解答这个问题的关键是要对内核态和用户态有清晰的认识，fork和duppage函数虽然对于程序员来说是系统调用，但本质还是用户态的函数。只要处于用户态，就有可能被中断打断，因此用户态的语句之间要格外注意是否会因为中断而产生意料之外的影响。

Lab4的challenge非常多，一时想要全部做完肯定是不现实的。先挑一些简单的做了吧。

#### Challenge 1

The big kernel lock is simple and easy to use. Nevertheless, it eliminates all concurrency in kernel mode. Most modern operating systems use different locks to protect different parts of their shared state, an approach called fine-grained locking. Fine-grained locking can increase performance significantly, but is more difficult to implement and error-prone. If you are brave enough, drop the big kernel lock and embrace concurrency in JOS!

It is up to you to decide the locking granularity (the amount of data that a lock protects). As a hint, you may consider using spin locks to ensure exclusive access to these shared components in the JOS kernel:

The page allocator.

The console driver.

The scheduler.

The inter-process communication (IPC) state that you will implement in the part C.

使用内核细加锁技术，来替代实验中的大内核锁。感觉这个也不简单，bug会有不少。而且也不清楚要怎么测试。

#### Challenge 2

Add a less trivial scheduling policy to the kernel, such as a fixed-priority scheduler that allows each environment to be assigned a priority and ensures that higher-priority environments are always chosen in preference to lower-priority environments. If you're feeling really adventurous, try implementing a Unix-style adjustable-priority scheduler or even a lottery or stride scheduler. (Look up "lottery scheduling" and "stride scheduling" in Google.)

Write a test program or two that verifies that your scheduling algorithm is working correctly (i.e., the right environments get run in the right order). It may be easier to write these test programs once you have implemented fork() and IPC in parts B and C of this lab.

增加优先级调度，这个也是得在Env数据结构里添加些东西。

如果想简单点实现的话，也就是在sched里面的循环上做文章。必须把所有的Env都遍历完一遍之后，再决定选择哪个Env来执行。

可能还需要提供一些系统调用来修改优先级？

#### Challenge 3

The JOS kernel currently does not allow applications to use the x86 processor's x87 floating-point unit (FPU), MMX instructions, or Streaming SIMD Extensions (SSE). Extend the Env structure to provide a save area for the processor's floating point state, and extend the context switching code to save and restore this state properly when switching from one environment to another. The FXSAVE and FXRSTOR instructions may be useful, but note that these are not in the old i386 user's manual because they were introduced in more recent processors. Write a user-level test program that does something cool with floating-point.

这个就是把浮点单元的一些寄存器状态也保存在trap frame和Env结构当中？要使用新的浮点指令。关键实验中也没实现打印浮点数的printf好像，也不清楚怎么测试。

#### Challenge 4

Add the additional system calls necessary to read all of the vital state of an existing environment as well as set it up. Then implement a user mode program that forks off a child environment, runs it for a while (e.g., a few iterations of sys\_yield()), then takes a complete snapshot or checkpoint of the child environment, runs the child for a while longer, and finally restores the child environment to the state it was in at the checkpoint and continues it from there. Thus, you are effectively "replaying" the execution of the child environment from an intermediate state. Make the child environment perform some interaction with the user using sys\_cgetc() or readline() so that the user can view and mutate its internal state, and verify that with your checkpoint/restart you can give the child environment a case of selective amnesia, making it "forget" everything that happened beyond a certain point.

这个实在有些无从下手。如果要能保存一个进程在执行过程中的快照，并能够进行恢复的话，我觉得要保存大量的信息啊？比如堆栈的数据、data段的数据之类的，这些都得保存，也不知道保存到哪里去。难道先保存到一些TEMP区域？这个确实挺难的，而且debug恐怕也不容易。

#### Challenge 5

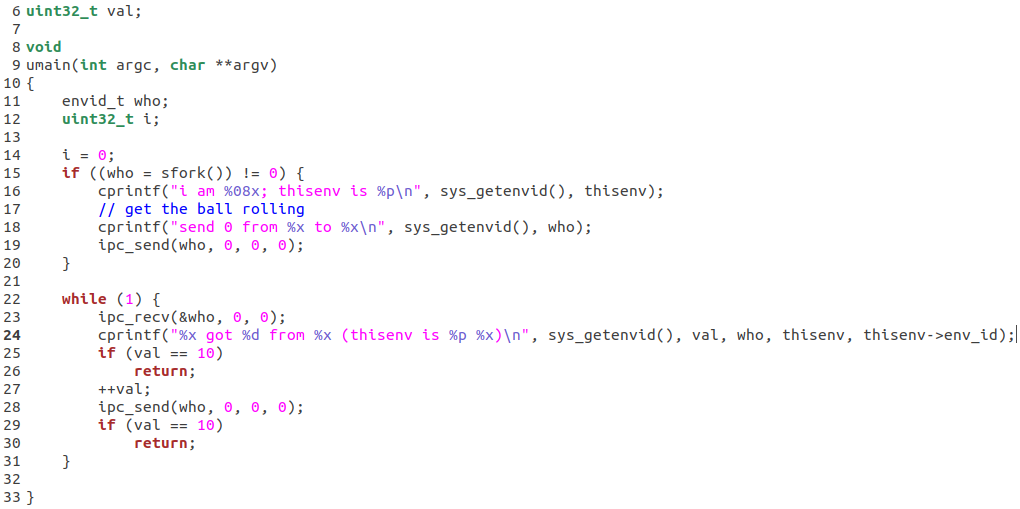
Extend your kernel so that not only page faults, but all types of processor exceptions that code running in user space can generate, can be redirected to a user-mode exception handler. Write user-mode test programs to test user-mode handling of various exceptions such as divide-by-zero, general protection fault, and illegal opcode.

也就是其它的异常也可以配置用户用户处理函数。这个感觉工作量比较大。

#### Challenge 6

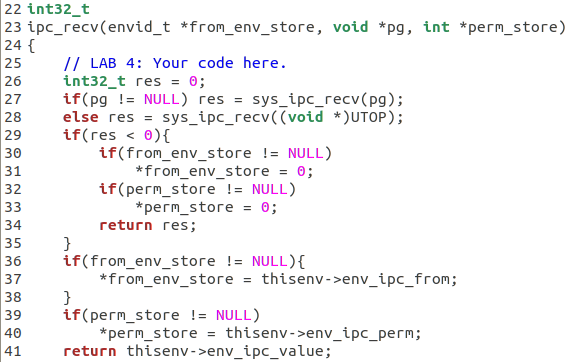
Implement a shared-memory fork() called sfork(). This version should have the parent and child share all their memory pages (so writes in one environment appear in the other) except for pages in the stack area, which should be treated in the usual copy-on-write manner. Modify user/forktree.c to use sfork() instead of regular fork(). Also, once you have finished implementing IPC in part C, use your sfork() to run user/pingpongs. You will have to find a new way to provide the functionality of the global thisenv pointer.

这个其实还比fork更简单了，因为不需要考虑写时拷贝的问题。但确实还需要考虑怎么处理thisenv全局变量指针。我感觉实在不行就直接每次都调用sys\_getenvid()来获取当前环境好了，功能是肯定可以实现的。



但这个程序里面好像非要用thisenv来整，这有点没办法了

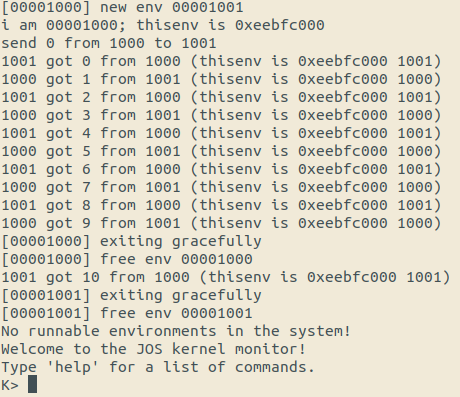
考虑把thisenv指向的变量转移到其他地方（暂时用USTACKTOP - 2\*PGSIZE来存放），但这会导致数据不一致的问题（脏数据），所以可以考虑在UENV这一页设置PTE\_COW标志。



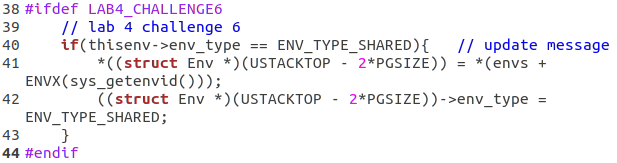
目前的想法是这样的，要使用一个新的缺页处理库函数spgfault。首先对于UENV这一页得设置成PTE\_COW，然后当用户态企图修改。但这种方案应该行不通，因为内核直接使用的是内核页表，不会触发中断。

我感觉得修改内核，来标记这种共享内存类型的进程。目前考虑先增加一种Env 的类型来实现。

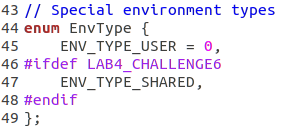
完成pingpongs进程的测试。这种实现下，thisenv的地址恒定，值也恒定，但因为两个进程在这一页上的页表项不同，导致最终的物理地址是不同的。此外还需要在ipc.c中对脏数据进行维护。



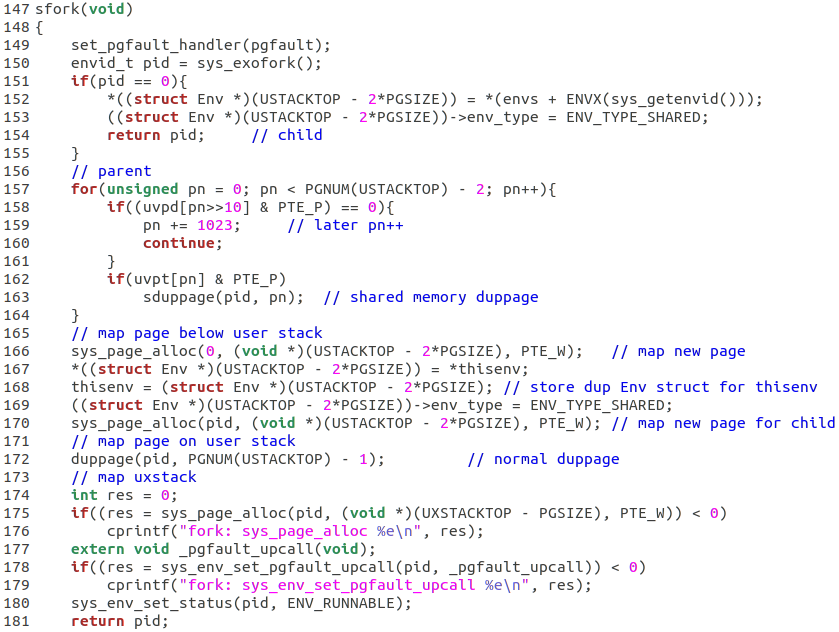
Ipc\_recv中维护脏数据



新的环境类型



最后给出这种方案的sfork实现



以及对应的sduppage复制页函数实现



#### Challenge 7

Your implementation of fork makes a huge number of system calls. On the x86, switching into the kernel using interrupts has non-trivial cost. Augment the system call interface so that it is possible to send a batch of system calls at once. Then change fork to use this interface.

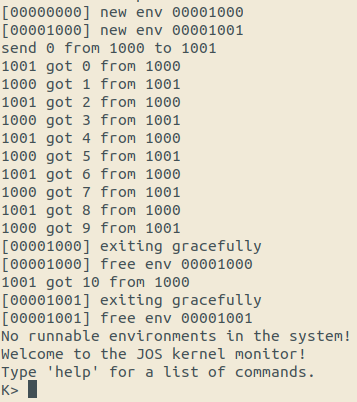
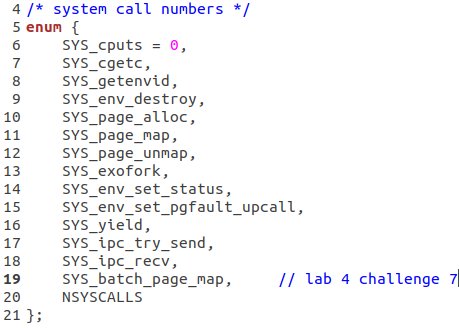
How much faster is your new fork?

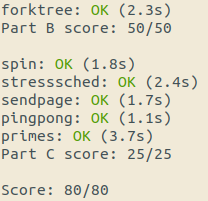
You can answer this (roughly) by using analytical arguments to estimate how much of an improvement batching system calls will make to the performance of your fork: How expensive is an int 0x30 instruction? How many times do you execute int 0x30 in your fork? Is accessing the TSS stack switch also expensive? And so on...

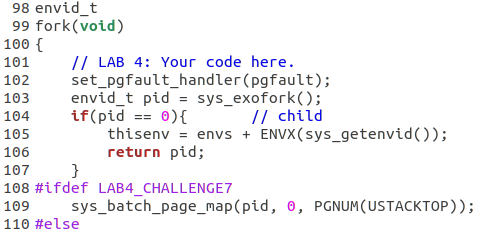
Alternatively, you can boot your kernel on real hardware and really benchmark your code. See the RDTSC (read time-stamp counter) instruction, defined in the IA32 manual, which counts the number of clock cycles that have elapsed since the last processor reset. QEMU doesn't emulate this instruction faithfully (it can either count the number of virtual instructions executed or use the host TSC, neither of which reflects the number of cycles a real CPU would require).

应该就是把那个for循环的部分改用一个接口就行了，这个应该不难。

先增加syscall编号，然后编写sys\_batch\_page\_map函数。通过了测试。



依然能通过所有测试。



#### Challenge 8

Why does ipc\_send have to loop? Change the system call interface so it doesn't have to. Make sure you can handle multiple environments trying to send to one environment at the same time.

这里应该要尝试其它的进程间通信的实现，信箱？还是有什么更好的方案吗。感觉得查不少资料。

#### Challenge 9

The prime sieve is only one neat use of message passing between a large number of concurrent programs. Read C. A. R. Hoare, ``Communicating Sequential Processes,'' Communications of the ACM 21(8) (August 1978), 666-667, and implement the matrix multiplication example.

实现一篇论文

#### Challenge 10

One of the most impressive examples of the power of message passing is Doug McIlroy's power series calculator, described in [M. Douglas McIlroy, ``Squinting at Power Series,'' Software--Practice and Experience, 20(7) (July 1990), 661-683](http://plan9.bell-labs.com/who/rsc/thread/squint.pdf). Implement his power series calculator and compute the power series for sin(x+x^3).

实现一篇论文

#### Challenge 11

Make JOS's IPC mechanism more efficient by applying some of the techniques from Liedtke's paper, [Improving IPC by Kernel Design](http://dl.acm.org/citation.cfm?id=168633), or any other tricks you may think of. Feel free to modify the kernel's system call API for this purpose, as long as your code is backwards compatible with what our grading scripts expect.

实现一篇论文