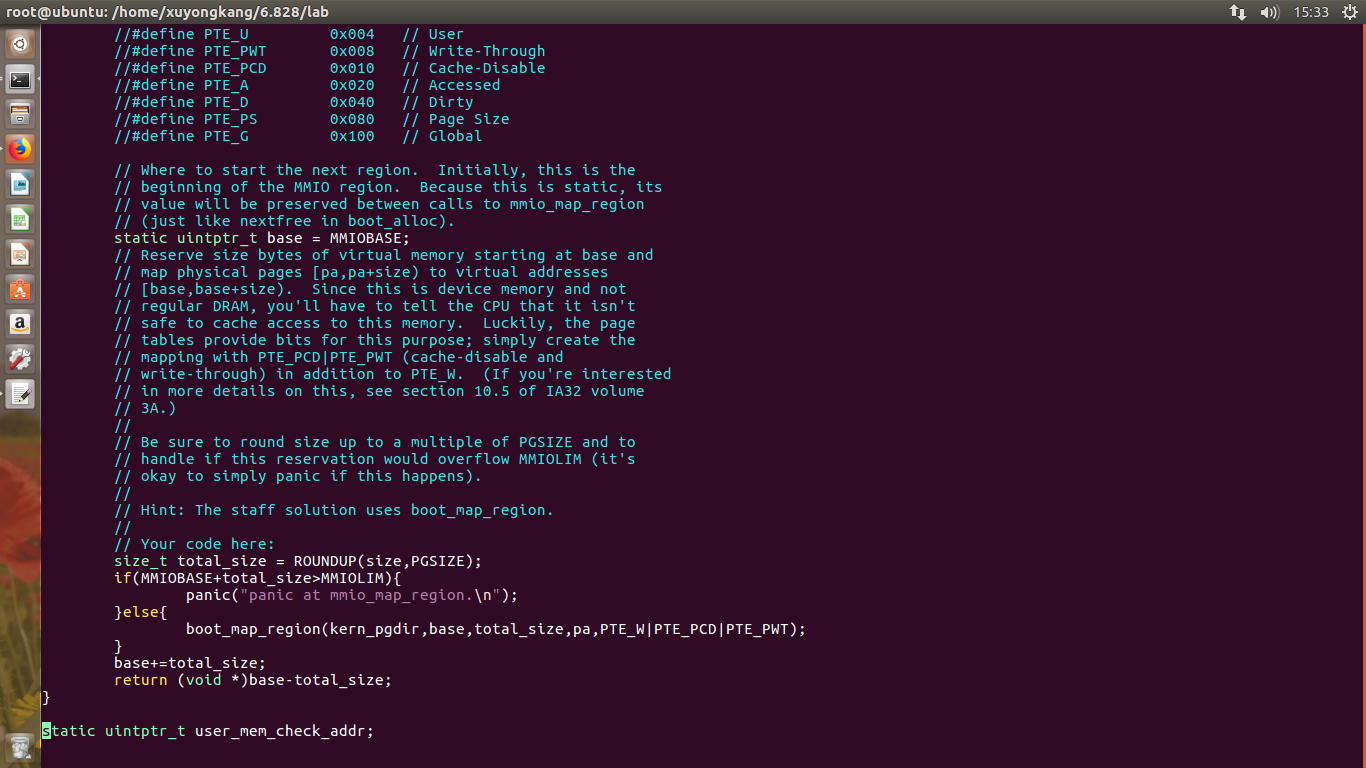
**Part A: Multiprocessor Support and Cooperative Multitasking**

**Exercise 1.**首先是mmio\_map\_region()函数。



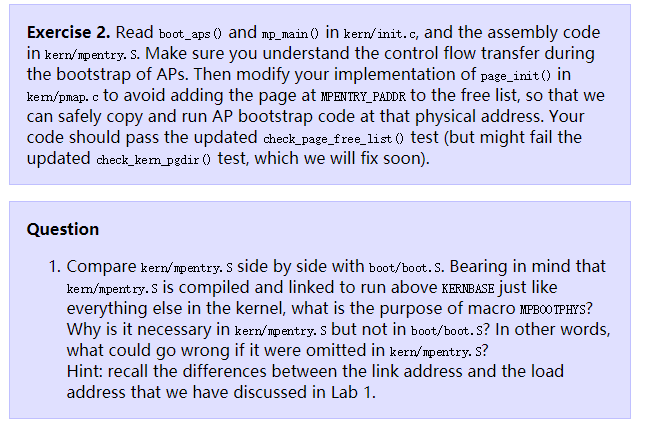
如果再boot\_map\_region()中少加了PTE\_W那么会在接下来的运行过程中：产生lapicw（）函数运行的错误，从而操作系统产生页错误。

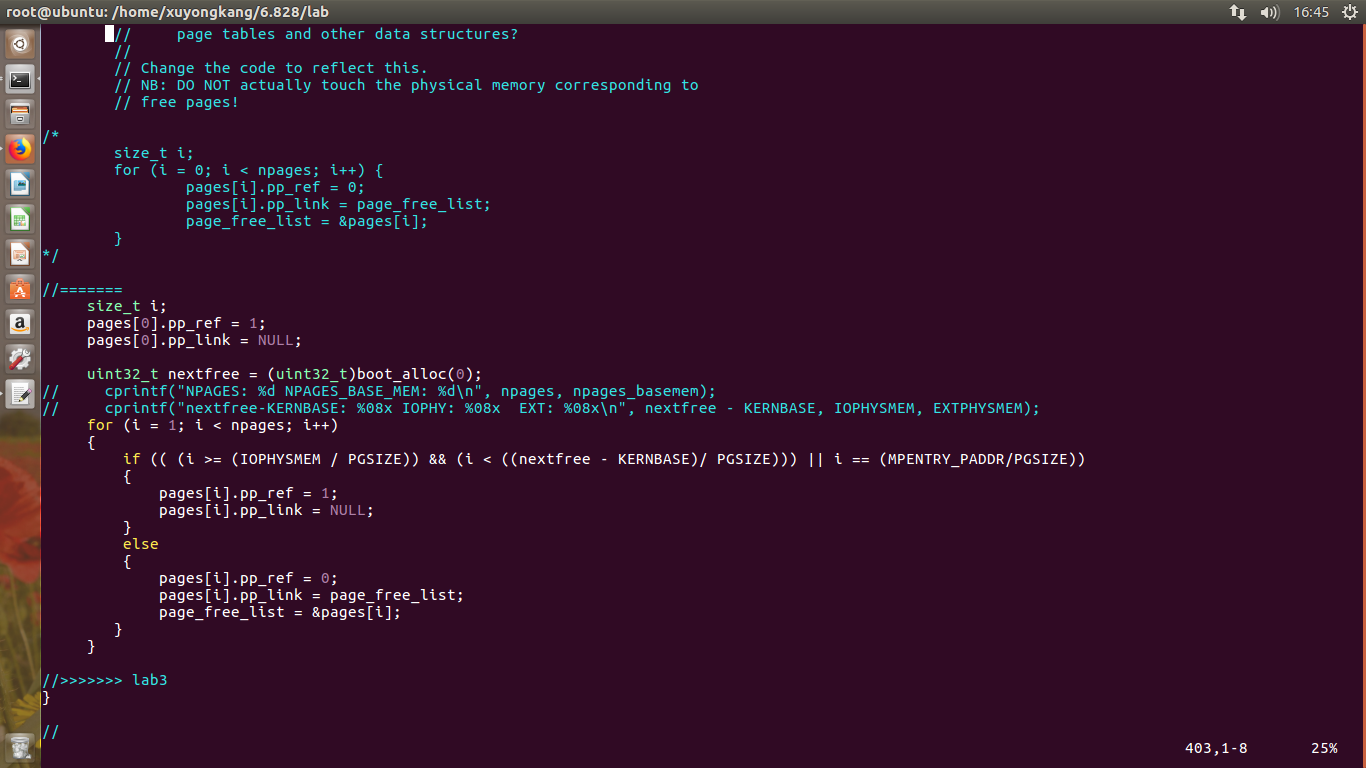
#### Application Processor Bootstrap

启动Aps之前，BSP首先收集多核系统的信息，如总的CPU个数，APIC IDs和LAPIC的MMIO地址。Kern/mpconfig.c通过读取BIOS区域内存的MP信息配置表获得这些信息。

Kern/init.c中的boot\_aps（）函数驱动AP bootstrap进程。Aps在实模式情况下启动，就像bootloader在boot/boot.S中启动的那样。所以boot\_aps()拷贝AP的入口地址(kern/mpentry.S)到一个实模式中可寻址的内存地址。不像bootloader那样必须拷贝到固定地址，我们能够控制AP从哪里开始执行代码，我们拷贝入口代码到0x7000（MPENTRY\_PADDR），但是任何低于640KB没有被占用的，页对齐的物理地址都可以工作。

之后，boot\_aps()通过IPIs向LAPIC单元发送STARTUP一个一个的激活对应于LAPIC单元的AP，同时给他们一个对应的初始化的CS：IP对应着它的入口地址(MPENTRY\_PADDR)。这个kern/mpentry.S中的入口地址跟boot/boot.S中的非常相似，经过简单的setup，它将这个AP进入到保护模式同时开启页保护然后跳到（kern/init.c）中的c启动函数mp\_main()。之后boot\_aps()在将要去唤醒下一个AP之前等待AP发送一个CPU\_STARTED标志，在CpuInfo结构体中的cpu\_status域中。





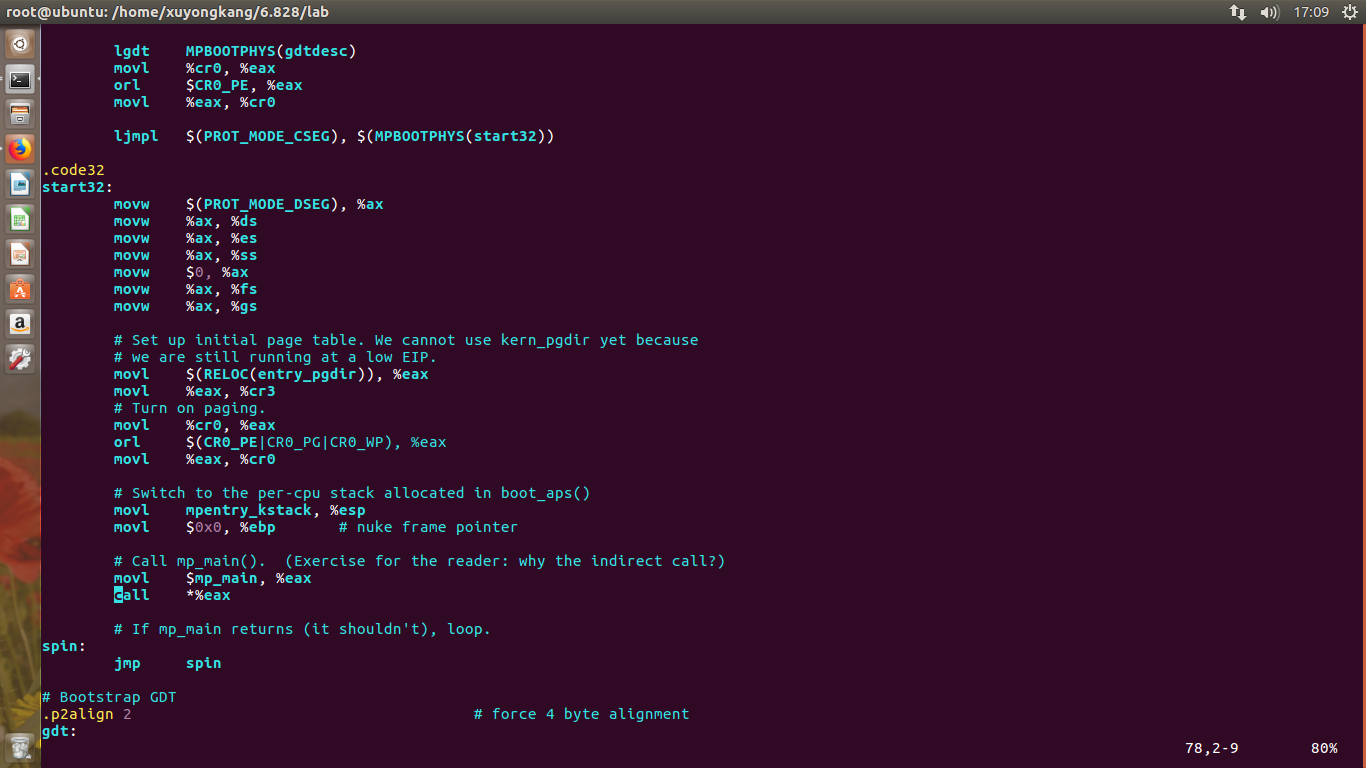
这段代码中是我抄的网上的，其中两部分到现在已经理解了

(i>=IOPHYSMEME)这段的意思是在映射的时候要避开系统的IO接口映射的地址

(I == (MEPENTRY\_PADDR))这段意思是在这里，避开Aps的入口地址

Q/A?

MPBOOTPHYS是一个宏我认为这个宏的作用是将现在的kaddr转变为虚拟地址，这样以来当系统进入到保护模式启动分页模式之后，才能访问到这块地址。



从上面的代码我们可以看到设置分页寄存器的时候，还是使用的与BSP核使用的变量一样的就是entry\_pgdir。所以可以说虽然是多个CPU核其实相当于1个核乘以n他们在结构和功能上都是相同的，访问某些地址的权限等。

**Per-CPU State and Initialization**

当写一个多处理器的OS的时候，要区分清除哪些cpu状态是每个cpu私自占有的，哪些是全局变量，被整个操作系统共享的。Kern/cpu.h定义了大多数每个cpu的状态，包括储存了每个cpu变量的struct CpuInfo。Cpunum()通常返回调用它的CPU的ID，这个id能够作为cpus数组的索引。另外，thiscpu宏是当前CPU的struct CpuInfo的简写.

以下是我们应该直到的CPU状态

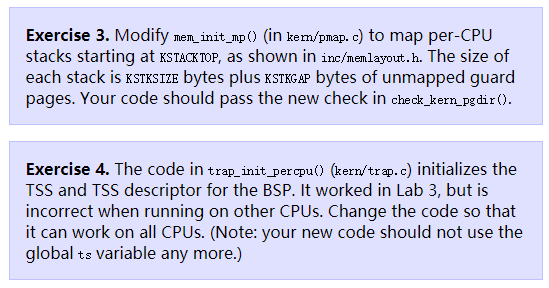
* Per-CPU kernel stack

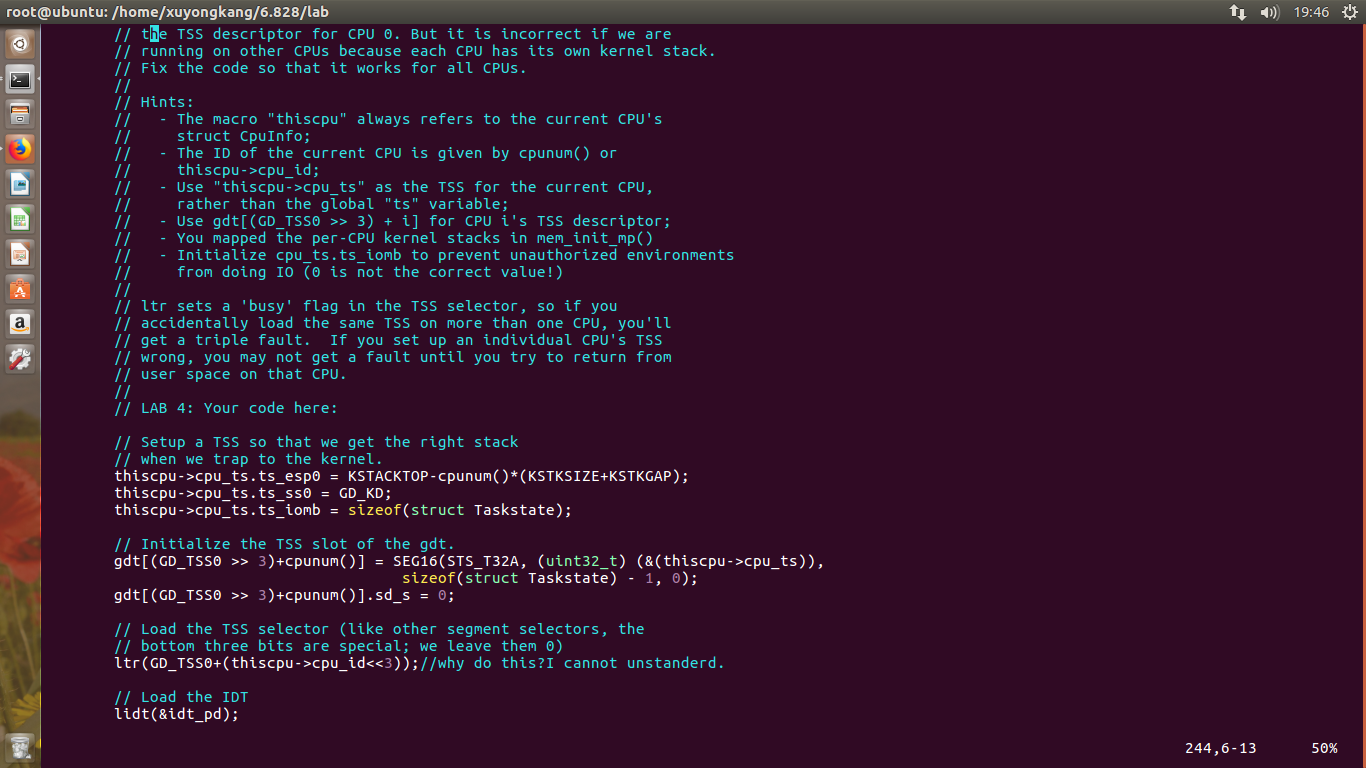
因为可能有多个CPU同时陷入内核，因此对于每个处理器我们需要一个分离的内核堆栈来防止他们干扰彼此的运行。数组percpu\_kstacks[NCPU][KSTKSIZE]为NCPU的内核堆栈保留了空间

在Lab2中，你将bootstack引用的物理内存映射为KSTACKTOP下方的内核堆栈。相似的，在这个实验中，你将映射带有guard pages的内核堆栈映射到这个地方，这些映射的堆栈和guard pages的排列就像数组一样CPU0‘的堆栈从KSTACKTOP开始往下生长，CPU1的堆栈在CPU0堆栈再往下KSTKGAP的位置向下生长，其他类似inc/memlayout.h展示了映射的结构。

* Per-CPU TSS and TSS descriptor  
  为了明确每个CPU的堆栈的位置，每个CPU的TSS是必须知道的。CPU i的TSS存储在cpus[i].cpu\_ts，并且对应的TSS描述段GDT入口为：gdt[(GD\_TSS0>>3)+i]。定义再kern/trap.c中的全局变量ts将不会再起作用。
* Per-CPU current environment pointer  
  因为每个CPU可以同时运行不同用户的进程，我们重新定义了符号curenv，将curenv指向cpus[cpunum()].cpu\_env（或者thiscpu->cpu\_env），这个变量指出当前长在执行的CPU正在运行的环境
* Per-CPU system registers  
  所有的寄存器（包括系统寄存器）都是CPU专用的。因此，初始化这些寄存器的指令，例如lcr3(),ltr(),lgdt(),lidt()等，必须在每个CPU上执行一次。函数env\_init\_percpu()和trap\_init\_percpu()是为此目的而定义的。

除此之外，如果你在解决方案中增加了任何额外的每CPU状态或执行了任何额外的CPU特定初始化（例如，在CPU寄存器中设置新位）来挑战早期实验中的问题，请务必复制他们在这里的每个CPU上。





上面的图片是kern/trap.c中的trap\_init\_percpu(void)函数在这个函数函数中自己改错的一点是：

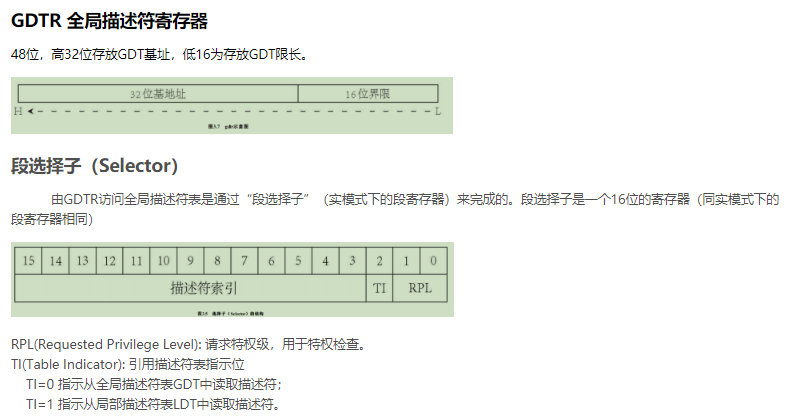
ltr(GD\_TSS0+(thiscpu->cpu\_id<<3))

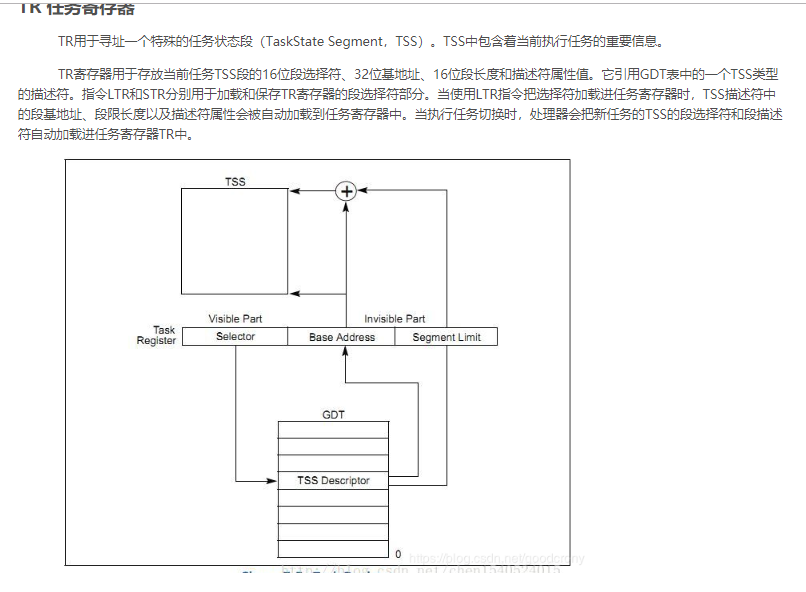
就是不知道为什么要加上(thiscpu->cpu\_id<<3)

这里就要对计算机的相关原理有相当程度的了解：

<https://blog.csdn.net/goodcrony/article/details/88122934#TR%20%E4%BB%BB%E5%8A%A1%E5%AF%84%E5%AD%98%E5%99%A8>

tr是一个任务段寄存器，寄存的是段选择子





注意段选择符的后三位是特殊的标志位，当加载tr寄存器（任务寄存器）的时候，任务寄存器中的内容会自动更换。

**Locking**

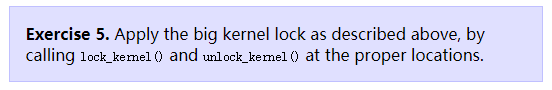
我们当前的代码在初始化mp\_main（）中的AP后开始运行。 在让AP进一步发展之前，我们需要首先解决当多个CPU同时运行内核代码时的竞争条件。 实现此目的的最简单方法是使用大内核锁。 大内核锁是一个全局锁，它在环境进入内核模式时保持，并在环境返回用户模式时释放。 在此模型中，用户模式下的环境可以在任何可用的CPU上并发运行，但在内核模式下只能运行一个环境; 尝试进入内核模式的任何其他环境都被迫等待。

Kern/spinlock.h声明了一个大内核锁，叫做kernel\_lock。它也提供了lock\_kernel()和unlock\_kernel()，用于获取和释放锁的快捷方式。你应该在四个位置应用大内核锁。

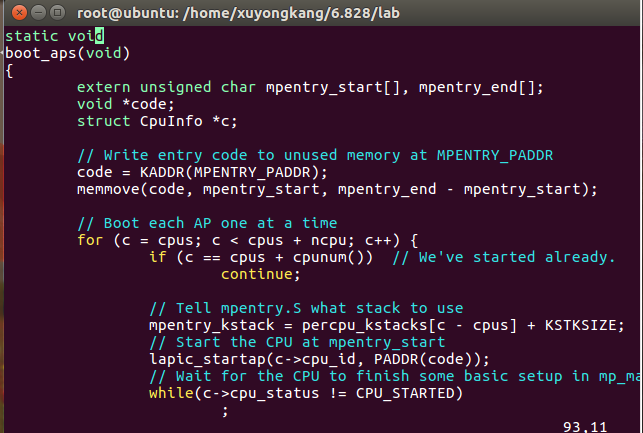
在i386\_init()中,在BSP唤醒其他CPU之前获取锁

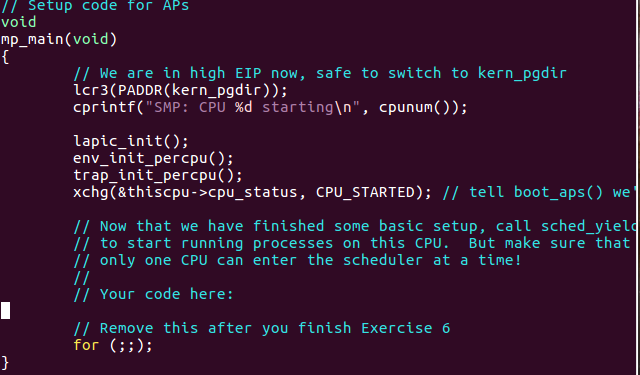
在mp\_main()，在初始化AP后获取锁，然后调用sched\_yield()在这个AP上开始运行环境。

在trap()中，当从用户模式陷入到trap()函数的时候，获取锁定。要确定陷阱是在用户模式还是在内核模式下发生的，请检查tf\_cs的低位。

在env\_run()中，在切换到用户模式之前释放锁定。不要太早或者太晚，否则你会遇到竞争：wq或者死锁。

解答：



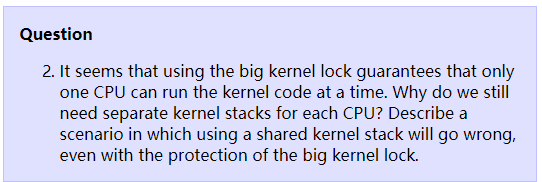


BSP将APs启动之后是将各个内核保持空转的状态。

自己分别在练习中要求的几个地方加上了对内核的控制，如果不正确还需要后续的更改。

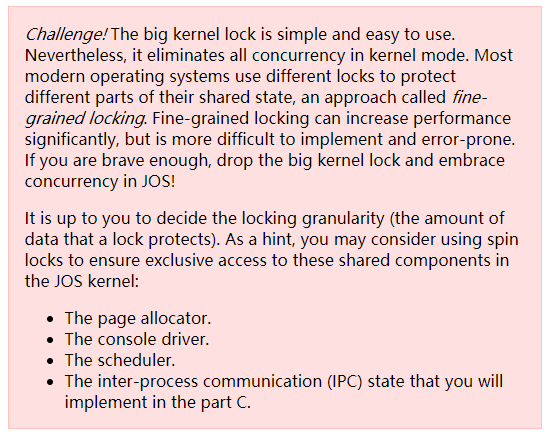
解答完毕。

怎样测试你的锁定是否正确吗？目前是没办法进行测试的。但是在你在下一个练习中拓展了调度器（scheduler）后就能进行检验了。



看起来使用“大内核锁”保证了同一个时刻只有一个CPU能运行内核代码。为什么我们仍然需要为每个内核的堆栈分配空间？描述一个当我们使用大内核锁的时候同时使用共享堆栈内存仍然会出现错误的的情况。

不会



挑战！这个大内核锁是简单又容易使用的。然而，它消除了内核模式中的所有并发性。大多数现代的操作系统使用不同的锁去保护其共享状态的不同部分一种称为细粒度锁定可以显著提高性能，但更难以实现并且容易出错。如果你足够勇敢放弃大内核锁并在JOS中拥抱并发

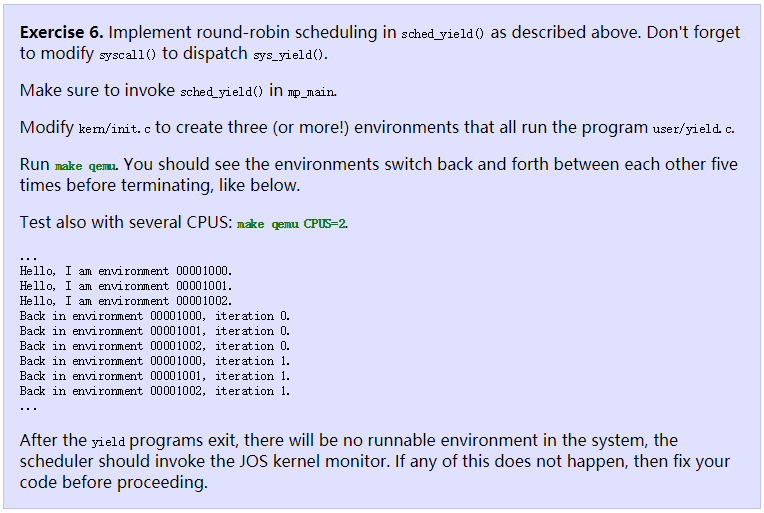
由你决定锁定粒度（锁保护的数据量）却决于你，作为提示，你可以考虑使用自旋锁来确保对JOS内核中这些共享组件的独占访问。

* The page allocator
* The console driver
* The scheduler
* The inter-process communication(IPC) state that you will implement in the part C.

**Round-Robin Scheduling(循环调度)**

在这个实验中你的下一个任务是去更改JOS内核，以便于它可以以“循环”方式在多个环境之间切换。JOS中的循环调度工作原理如下：

* Kern/sched.c中的Sched\_yield()函数负责选择一个新的环境去运行。它以循环方式顺序搜索envs[]数组，从之前运行的环境开始(或者如果没有以前运行的环境，则在数组的开头)，选择它找到的状态为ENV\_RUNNABLE的第一个环境(参见inc/env.h)，并调用env\_run()跳转到该环境。
* Sched\_yield()不得同时在两个CPU上运行同一个环境。我们可以辨别一个环境是否运行在某个CPU上（也可能就是当前这个CPU上）因为这个环境的状态将会是ENV\_RUNNING
* 我们已经为你实现了一个新的系统调用叫做sys\_yield()，用户环境可以调用这个函数去调用内核的sched\_yield()函数，这样一来自动放弃自己占用的CPU给不同的环境



在函数sched\_yield()中实现上面描述的循环调度，不要忘了去修正syscall()去调度sys\_yield()；

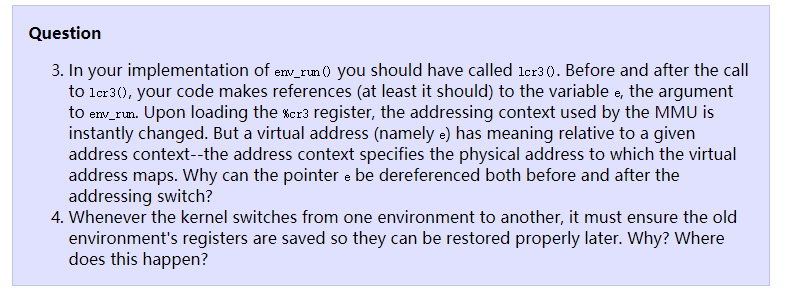
请保证在mp\_main()调用sched\_yield()

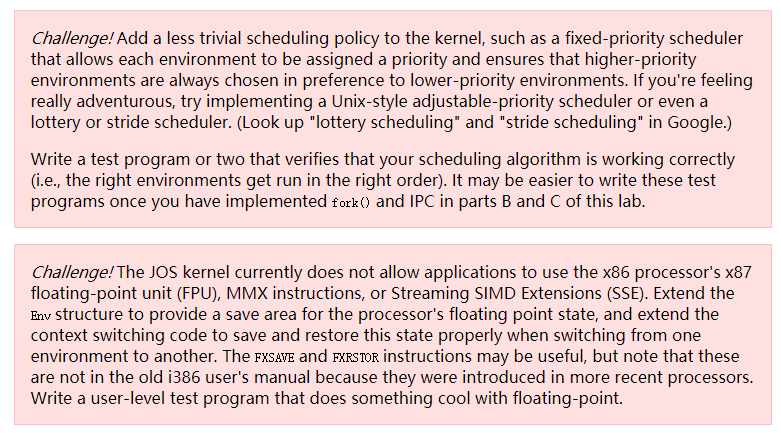
修正kern/init.c去创建三个（或者更多）的都是运行uer/yield.c的环境。

运行make qemu。你应该看环境在终止之前来回切换5次，如下所示：

…

在yield程序推出之后，系统将不会由可运行的环境，这个调度器应该调用JOS内核监视器。如果这些事有一件没有发生请修复之后再往下进行。



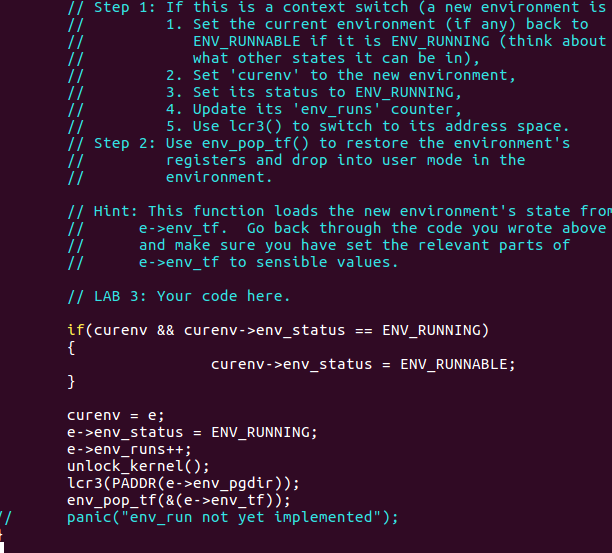


在你的env\_run()实现中你应该调用lcr3()。在调用lcr3()之前和之后，你的代码对变量e（env\_run的参数）进行引用（至少它应该）。加载%cr3寄存器后，MMU使用的寻址上下文立刻更改。但是虚拟地址（即e）具有相对于给定地址上下文的含义-地址上下文指定虚拟地址映射到的物理地址。为什么指针e可以在寻址开关之前和之后解除引用？

挑战！不做，因为不会。

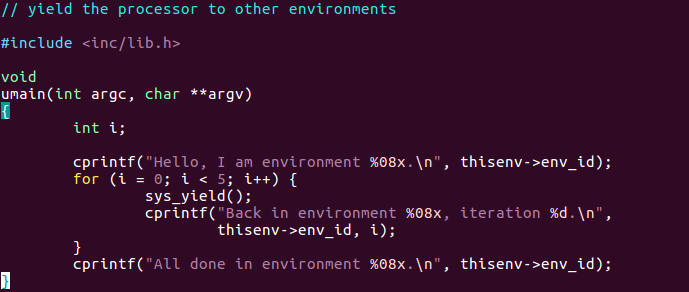
不会，或许是因为两个环境中过的kern\_pgdir是相同的

疑问！



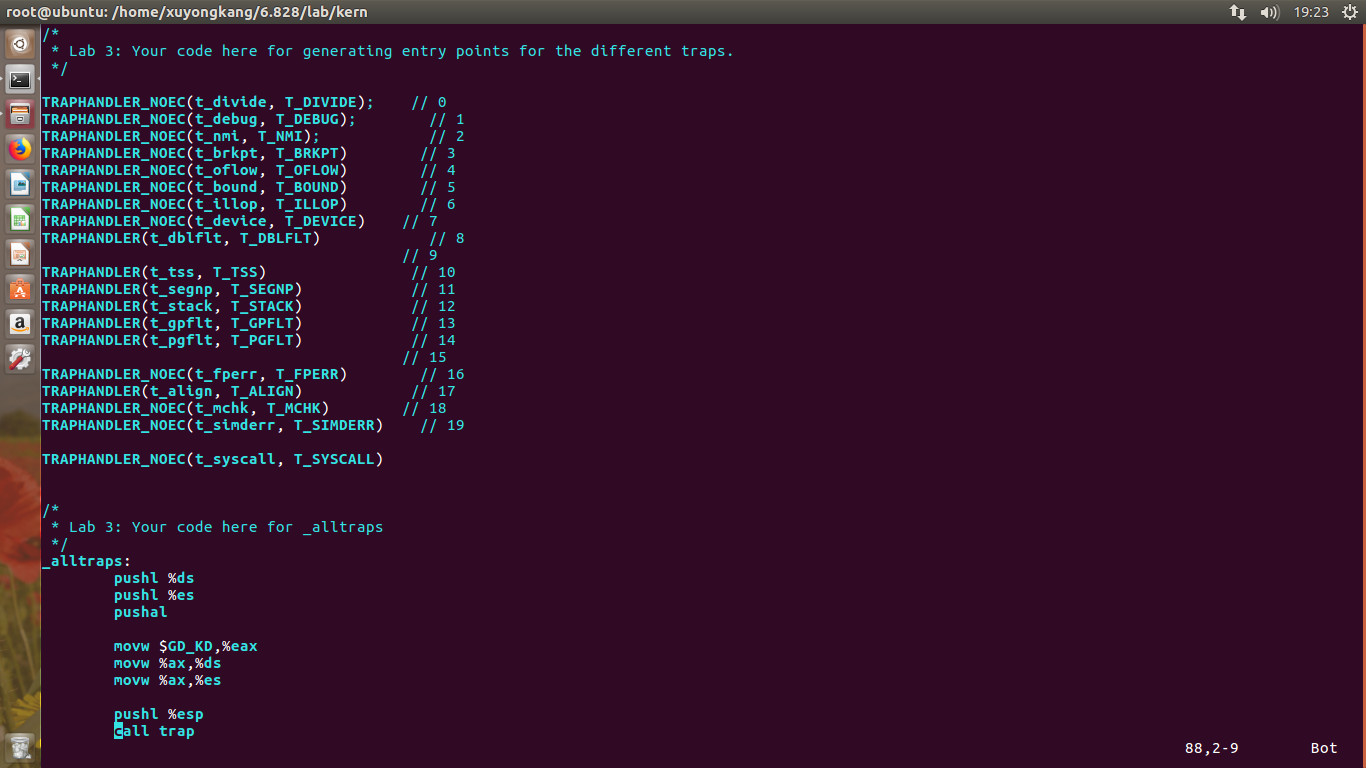
在这里我们探讨以下为什么只要最后实现一个sched\_yield()（kern/sched.c）函数就能够实现在一个CPU上进行多个运行环境的切换？

我们可以看到user/yield.c中代码：

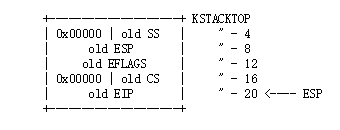


在这里函数使用了系统调用sys\_yield()

一旦进行系统调用，程序就会从用户模式陷入到内核模式：



在程序进入\_alltraps:之前先进行了一些处理包括：

1. 处理器切换到由TSS的SS0和ESP0字段定义的堆栈，其在JOS中将分别保持值GD\_KD和KSTACKTOP
2. 处理器从地址KSTACKTOP开始在内核栈上推送异常参数  
   
3. 因为我们正在处理除法错误，即x86上的中断向量0，所以处理器读取IDT条目0并将CS：EIP设置为指向条目描述的处理程序函数。（这里我们可以类比得到当调用系统调用的时候，计算机也是从IDT中读取对应的运行程序的地址并运行。）

注意：在kern/trapentry.S中所有的中断和异常都最终使用统一的代码处理就是

\_alltraps,在这里之前已经有部分变量包括

0x00000|old SS

Old ESP

Old EFLAGS

0x00000|old CS

Old EIP

同样的在\_alltraps中又进行了

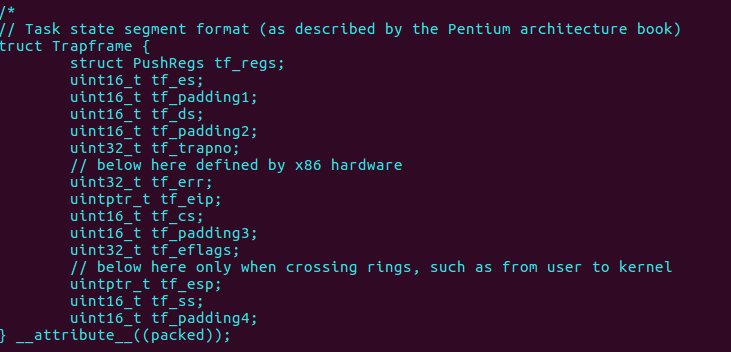
Push ds

Push es

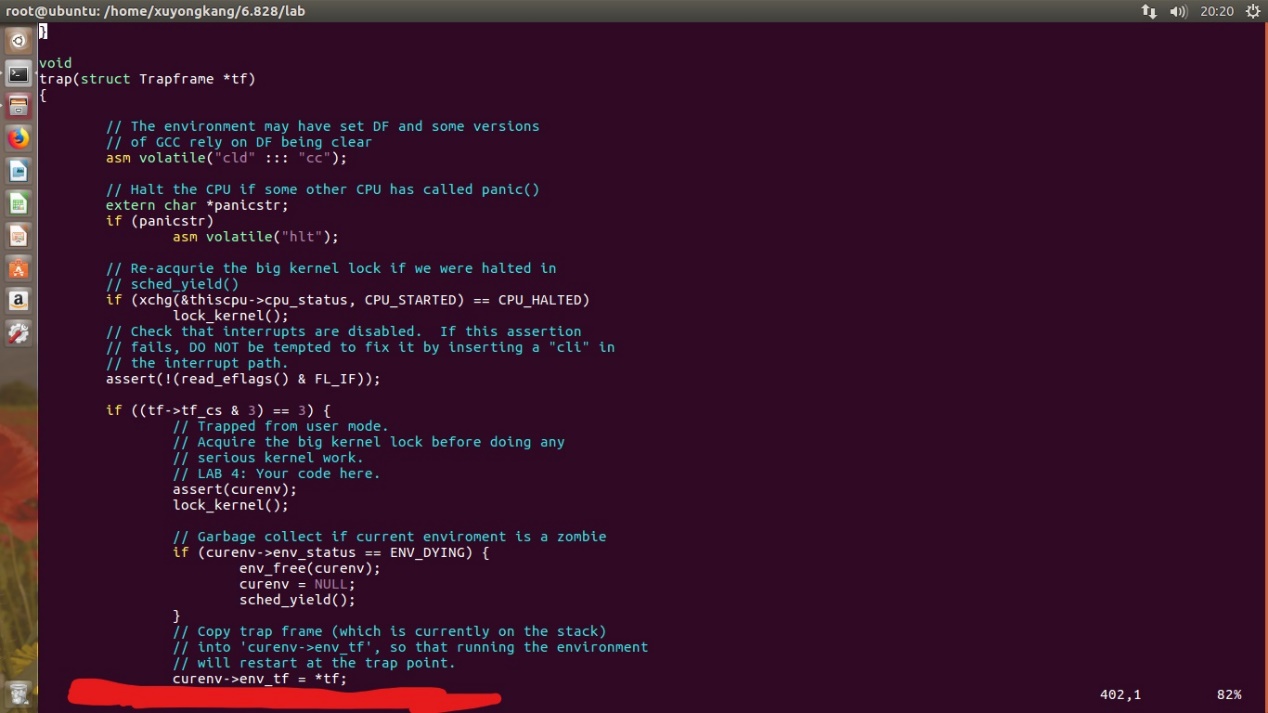
Push al

这些所有的变量都是被推入到了当前任务段TSS中相关字段定义的:段基址+偏移地址的位置。

这样最终就相当于在KSTACKTOP的位置形成了一个trapframe



这样我们就可以知道大致的借用系统调用实现进程切换运行的原理：

1. 一个环境正在运行
2. 调用切换任务的进程，在进入中断陷阱之前，相关程序会将所有的当前环境的状态保存到一个trapframe \*tf中（注意这个tf只是一个地址，并没有函数申请实体，只是程序运行的过程不断的压栈处理从而形成这样一个变量，而这个变量就是保存在进行进程切换的前一个某个进程当时所处的环境）
3. 在中断，中运行一个被选中的环境
4. 在这个新的环境可能还会进行系统调用切换到另一个不同的程序；然后可以不断的进行程序的切换。根据这个原理就能实现进程的切换，即多任务的处理模式。  
   

注意最后一句代码将进入一个正在运行的环境进入中断后的所有状态都保存下来，并更新当前环境中的trapframe->env\_tf。这样如果下一次再调度到这个函数的时候，就能刚好回到这个环境进入中断之前的状态，从而继续运行。而且我们可以注意这个tf保存的位置刚好一个环境对应一个任务段，TSS所定义的堆栈的最顶端永远保存这这个运行环境的所有的计算机内部的瞬间状态，能够随时恢复到之前的运行状态。（可以仔细思考一下：当前环境有对应的TSS，这个TSS定义了堆栈，使用自己的堆栈保存自己现在的状态，那么在将环境当前状态推入到这个堆栈之前，这个位置保存的是自己的环境前一次的运行被中断时的状态。）

至此，多任务，进程切换暂时结束

**System Calls for Environment Creation**

虽然你的内核现在能够在多个用户级环境之间运行和切换，但它仍然限于运行内核最初设置的环境。你现在将实现必要的jos系统调用，以允许用户创建和启动其他新的用户环境。

Unix提供fork（）系统调用作为其进程创建原语。Unix fork（）复制调用进程（父进程）的整个地址空间来创建一个新进程（子进程）。两个可观察用户空间之间的唯一区别是它们的进程ID和父进程ID（由getpid和getppid返回）。在父级中，fork()返回子进程ID，而在子级中，fork()返回0.默认情况下，每个进程都有自己的私有地址空间，并且进程对内存的修改对另一个进程都不可见。您将提供一组不同的，更原始的JOS系统调用来创建新的用户模式环境。 通过这些系统调用，您将能够在用户空间中实现类似Unix的fork（），以及其他样式的环境创建。 您将为JOS写的新系统调用如下：

sys\_exofork: 此系统调用创建一个几乎为空白平板的新环境：没有任何内容映射到其地址空间的用户部分，并且它不可运行。 在sys\_exofork调用时，新环境将具有与父环境相同的寄存器状态。 在父级中，sys\_exofork将返回新创建的环境的envid\_t（如果环境分配失败，则返回否定错误代码）。 但是，在子节点中，它将返回0.（由于子节点开始标记为不可运行，因此sys\_exofork实际上不会在子节点中返回，直到父节点通过使用标记子节点运行来明确允许它为止）。

sys\_env\_set\_status: 将指定环境的状态设置为ENV\_RUNNABLE或ENV\_NOT\_RUNNABLE。 一旦其地址空间和寄存器状态已完全初始化，此系统调用通常用于标记准备运行的新环境。

sys\_page\_alloc: 分配一页物理内存并将其映射到给定环境的地址空间中的给定虚拟地址。

sys\_page\_map: 将页面映射（不是页面的内容！）从一个环境的地址空间复制到另一个环境，留下内存共享安排(?)，以便新旧映射都指向物理内存的同一页面。

sys\_page\_unmap: Unmap a page mapped at a given virtual address in a given environment.

对于接受环境ID的上述所有系统调用，JOS内核支持值为0表示“当前环境”的约定。 该约定由kern / env.c中的envid2env（）实现。

我们在测试程序user / dumbfork.c中提供了一个非常原始的类Unix（）的实现。 此测试程序使用上述系统调用来创建和运行具有其自己的地址空间副本的子环境。 然后，这两个环境使用sys\_yield来回切换，如上一练习中所示。 父母在10次迭代后退出，而孩子在20次退出后退出。

COW(copy on write 技术)

在linux程序中，fork()会产生一个和父进程完全相同的子进程，但子进程在此后多会exec系统调用（系统调用exec是以新的进程去代替原来的进程，但进程的PID保持不变。因此可以这样认为，exec系统调用并没有创建新的进程，只是替换了原来进程的上下文的内容。原进程的代码段，数据段，堆栈段被新的进程所代替），出于效率考虑，linux中引入了”写时复制”技术，也就是只有进程空间的各段的内容要发生变化时，才会将父进程的内容复制一份给子进程。

那么子进程的物理空间没有代码，怎么去取指令执行exec系统调用呢？

在fork之后exec之前两个进程用的是相同的物理空间（内存区），子进程的代码段、数据段、堆栈都是指向父进程的物理空间，也就是说，两者的虚拟空间不同，但其对应的物理空间是同一个。当父子进程中有更改相应段的行为发生时，再为子进程相应的段分配物理空间，如果不是因为exec，内核会给子进程的数据段，堆栈段分配相应的物理空间(至此两者有各自的进程空间，互不影响)，而代码段继续共享父进程的物理空间（两者的代码完全相同）。而如果是因为exec，由于两者执行的代码不同，子进程的代码段也会分配单独的物理空间。

这种技术也被广泛的用于各种语言特性中，如C++语言中可以使用shared\_ptr来实现这样的操作。当多个用户共享一块相同的数据时，如果其中某个用户要求对数据进行修改，系统会把这块数据复制一份然后进行修改，修改完成后让该用户的记录指向新修改的数据，其他用户看到的还是原来的数据而该用户看到的是已经修改的数据。如果数据不写只读的话，不会被复制，这样可以节省存储空间。

更详细的linux中的COW技术：https://blog.csdn.net/qq\_37375427/article/details/83034965

https://blog.csdn.net/u010120125/article/details/78427597

（这段COW技术还不是很了解）

任务段使用的数据结构tss（清楚里面的结构）和中断使用的数据结构tf（清楚结构和进程切换过程）一定要分清楚。一个一旦陷入中断（无论是进行系统调用还是页错误或者其他形式的进入trap()都是要形成一个trapeframe的结构,然后再进行处理）。

首先：trapframe的结构：

struct Trapframe {

struct PushRegs tf\_regs;

uint16\_t tf\_es;

uint16\_t tf\_padding1;

uint16\_t tf\_ds;

uint16\_t tf\_padding2;

uint32\_t tf\_trapno;

/\* below here defined by x86 hardware \*/

uint32\_t tf\_err;

uintptr\_t tf\_eip;

uint16\_t tf\_cs;

uint16\_t tf\_padding3;

uint32\_t tf\_eflags;

/\* below here only when crossing rings, such as from user to kernel \*/

uintptr\_t tf\_esp;

uint16\_t tf\_ss;

uint16\_t tf\_padding4;

} \_\_attribute\_\_((packed));

struct PushRegs {

/\* registers as pushed by pusha \*/

uint32\_t reg\_edi;

uint32\_t reg\_esi;

uint32\_t reg\_ebp;

uint32\_t reg\_oesp; /\* Useless \*/

uint32\_t reg\_ebx;

uint32\_t reg\_edx;

uint32\_t reg\_ecx;

uint32\_t reg\_eax;

} \_\_attribute\_\_((packed));

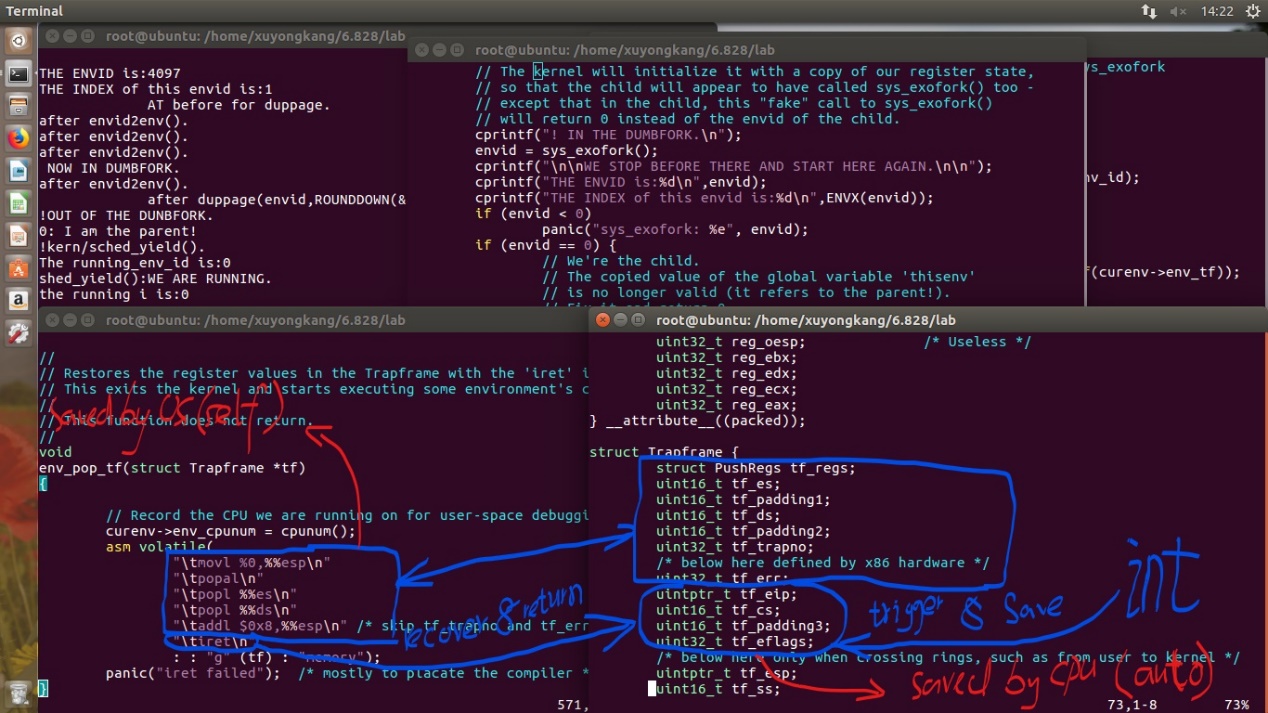
注意：iret的整个操作过程

Popal汇编指令的过程

Pushal和popal汇编指令就是将struct PushRegs中定义相应的在当前计算机中的寄存器中的值放入到堆栈中，相反的做法，popal是将堆栈中的值弹出到相应的当前计算机的寄存器中。

中断与中断恢复

对于iret使用一张图解释：



在cpu进入中断或者陷阱之前，CPU会自动保存少部分变量到堆栈中，但是这些变量不足以恢复现场（这些变量是一些最重要的变量），然后使用iret的时候这条指令会自动从当前的堆栈中弹出某个任务进入中断进入堆栈之前CPU自动保存的一些重要变量到对应的寄存器中然后开始运行这样也就完成了任务进入中断然后再切换回去的这个过程。

续：（接着上面的exercise写）

1. 自己写了sys\_exofork：函数但是由于中间少了一行所以会出错
2. 自己写了sys\_env\_set\_status：写完这个函数，运行user/dumbfork.c暂时没有遇到什么问题。
3. 自己写了sys\_page\_alloc：仍然是运行user/dumbfork.c没有什么问题
4. 自己写了sys\_page\_map:仍然是运行user/dumbfork.c没有什么问题
5. 自己写了sys\_page\_unmap:仍然是运行user/dumbfork.c没有什么问题

在写1遇到的问题：

自己写的函数：



一开始总是错误，因为少了一句：

Newenv\_store->env\_tf.tf\_regs.reg\_eax = 0;

为什么需要这一句可以往下分析一下：

记录整个user/dumbfork的整个过程：

(1):ENV\_CRETA();创造一个环境环境的索引为0；

(2):运行函数sched\_yield();

Sched\_yield()函数来运行索引指示的环境（这个环境就是dumbfork）这个进程的envid是4096（因为当时就这个环境属于用户的环境）

(3):正在运行的dumbfork函数来创建自己的子进程，新创建的子进程为4097

Dumbfork()调用系统调用sys\_exofork()去重新创建一个子环境这个子环境的（1）内存映射（2）上下文环境完全与父环境相同，就是相当于如果子环境如果运行的化完全就是一个父环境的复制。

但是在函数sys\_exofork()中newenv\_store->env\_tf.tf\_regs.reg\_eax = 0;如果少了这一句就会出错，原因可以看(4)之后的步骤

(4)在dumbfork()被调用完之后，当前环境的一个子环境已经通过sys\_exofork()等函数创建了。

（5）在运行完dumbfork()函数之后，就进入了for循环,在for循环中进行了sched\_yield()，由于当前环境已经创建了一个子进程，所以这次sched\_yield()会将当前环境切换到子环境。好了，现在我们现在想想子环境中的数据结构trapeframe tf中的数据保存的就是父亲环境进入系统中断一瞬间保存的数据；现在通过env\_run()调用env\_pop\_tf()去运行这个环境，其实这个子环境从来没有进入过中断指示拷贝了父环境的变量而已，这个整个中断的过程其实就是拷贝的它的父环境的一个过程，所以子环境从”中断”中返回的，返回值就是env\_tf.tf\_regs.reg\_eax;而这个返回值会被当做envid使用，由于是相当于子环境调用sys\_exofork()所以返回值应该为0.刚好解释了为什么要加上那一句才能正确运行user/dumbfork.c

## Part B: Copy-on-Write Fork

我们之前提到过，Unix提供系统调用fork()作为其主要进程创建原语。Fork()系统调用复制调用进程（父进程）的地址空间以创建新进程（子进程）。Xv6 Unix通过将父页面中的所有数据复制到为子节点分配的新页面来实现fork()。这与dumbfork()采用的方法基本相同。将父地址空间复制到子进程中是fork()操作中最昂贵的部分。

但是，对子进程中的exec()调用几乎会立即调用fork(),后者采用新程序替换子进程的内存。例如，这就是shell通常所做的事情。在这种情况下复制父地址空间所花费的时间在很大程度上被浪费了，因为子进程在调用exec()之前将使用很少的内存。

出于这个原因，Unix的更高版本利用虚拟内存硬件来允许父和子共享映射到其各自地址空间的内存，知道其中一个进程实际修改它。这种技术称为写时复制。为此，在fork()上，内核会将地址空间映射从父级复制到子级而不是映射页面的内容，同时将现在共享的页面标记为只读。当其中一个进程尝试写入其中一个共享页面时，该进程会发生页面错误。此时Unix内核意识到该页面实际上是一个“虚拟”或“写时复制”副本，因此它为错误进程创建了一个新的，私有的，可写的页面副本。这样，在实际写入之前，实际上不会复制各个页面的内容。这种优化使得子句中的fork()后面跟exec()更加便宜，子进程可能只需要在调用exec()之前复制一个页面（其堆栈的当前页）。

在本实验的下一个部分中，您将实现一个“正确的”类似于Unix的fork()，它具有写时复制功能，作为用户空间库历程(user space library routine)。在用户空间中实现fork()和copy-on-write支持具有以下优点：内核更加简单，因此更可能是正确的。它还允许单个用户模式程序fork()定义自己的语义。想要稍微不同的实现的程序（例如像dumbfork()这样的昂贵的始终复制版本，或者之后父和子实际共享内存的程序）可以很容易地提供它自己的。

### User-level page fault handling

用户级copy-on-write fork（）需要知道写保护页面上的页面错误，因此这是您首先要实现的。写时复制只是用户级页面错误处理的许多可能用途之一。

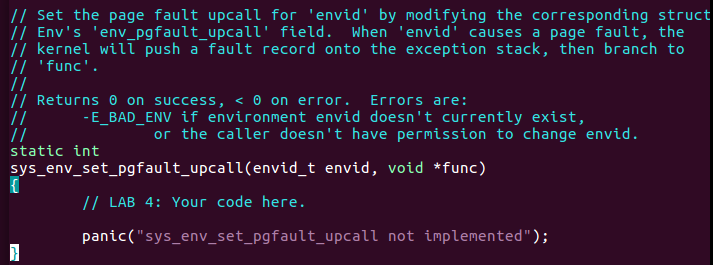
设置地址空间是很常见的，以便页面错误指示何时需要执行某些操作。例如，大多数Unix内核最初只在新进程的堆栈区域中映射单个页面，并在随后“按需”分配和映射其他堆栈页面，因为进程的堆栈消耗增加并导致尚未映射的堆栈地址上的页面错误。典型的Unix内核必须跟踪在进程空间的每个区域中发生页面错误时要采取的操作。例如，堆栈区域中的故障通常将分配和映射物理存储器的新页面。程序的BSS区域中的错误通常会分配一个新页面，用零填充并映射它。在具有按需分页可执行文件的系统中，文本区域中的错误将从磁盘中读取二进制文件的相应页面，然后映射它。

这是内核要跟踪的大量信息。您将决定如何处理用户空间中的每个页面错误，而不是采用传统的Unix方法，其中错误的破坏性较小。这种设计的另一个好处是允许程序在定义内存区域时具有很大的灵活性;稍后您将使用用户级页面错误处理来映射和访问基于磁盘的文件系统上的文件。

#### Setting the Page Fault Handler

为了处理自己的页面错误，用户环境需要使用JOS内核注册页面错误处理程序入口点。 用户环境通过新的sys\_env\_set\_pgfault\_upcall系统调用注册其页面错误入口点。 我们在Env结构中添加了一个新成员env\_pgfault\_upcall来记录这些信息。（要理解）

**Exercise 8.**实现sys\_env\_set\_pgfault\_upcall系统调用。 在查找目标环境的环境ID时，请务必启用权限检查，因为这是一个“危险”的系统调用。

****

（这一部分是自己写的一点总结）在这里使用envid2env(env,&newenv\_store,1)时候，1是必须的因为要进行环境的检查。用于检查调用环境是否具有操纵指定环境的合法权限。如果设置了checkperm，则指定环境必须是当前的环境或者是当前环境的直接孩子。

#### Normal and Exception Stacks in User Environments

在正常执行期间，JOS中的用户环境将在普通用户堆栈上运行：其ESP寄存器开始指向USTACKTOP，并且它推送的堆栈数据驻留在USTACKTOP-PGSIZE和USTACKTOP-1之间的页面上。但是，当在用户模式下发生页面错误时，内核将重新启动在不同堆栈（即用户异常堆栈）上运行指定用户级页面错误处理程序的用户环境。

本质上，我们将使JOS内核代表用户环境实现自动“堆栈切换”，就像x86处理器在从用户模式转换到内核模式时代表JOS实现堆栈切换一样！

JOS用户异常堆栈的大小也是一页，其顶部定义为虚拟地址UXSTACKTOP，因此用户异常堆栈的有效字节是从UXSTACKTOP-PGSIZE到UXSTACKTOP-1。在此异常堆栈上运行时，用户级页面错误处理程序可以使用JOS的常规系统调用来映射新页面或调整映射，以便修复最初导致页面错误的任何问题。然后，用户级页面错误处理程序通过汇编语言存根返回到原始堆栈上的错误代码。

希望支持用户级页面错误处理的每个用户环境都需要使用A部分中引入的sys\_page\_alloc（）系统调用为自己的异常堆栈分配内存。

#### Invoking the User Page Fault Handler

您现在需要更改kern / trap.c中的页面错误处理代码，以便从用户模式处理页面错误，如下所示。我们将在故障时调用用户环境的状态为陷阱时状态。

如果没有注册页面错误处理程序，则JOS内核会像以前一样使用消息破坏用户环境。否则，内核在异常堆栈上设置一个陷阱框架，看起来像是inc / trap.h中的struct UTrapframe：

<-- UXSTACKTOP

trap-time esp

trap-time eflags

trap-time eip

trap-time eax start of struct PushRegs

trap-time ecx

trap-time edx

trap-time ebx

trap-time esp

trap-time ebp

trap-time esi

trap-time edi end of struct PushRegs

tf\_err (error code)

fault\_va <-- %esp when handler is run

然后内核安排用户环境恢复执行，该错误处理程序在具有此堆栈帧的异常堆栈上运行;你必须弄清楚如何实现这一目标。 fault\_va是导致页面错误的虚拟地址。

如果发生异常时用户环境已在用户异常堆栈上运行，则页面错误处理程序本身已出现故障。在这种情况下，您应该在当前的tf-> tf\_esp而不是UXSTACKTOP下启动新的堆栈帧。你应该首先推送一个空的32位字，然后是一个struct UTrapframe。

要测试tf-> tf\_esp是否已经在用户异常堆栈中，请检查它是否在UXSTACKTOP-PGSIZE和UXSTACKTOP-1之间的范围内。

通过修改相应的struct Env的'env\_pgfault\_upcall'字段，为'envid'设置页面错误upcall。当'envid'导致页面错误时，内核会将错误记录推送到异常堆栈，然后转移到'func'。成功时返回0，错误时返回<0。错误是：-E\_BAD\_ENV如果环境envid当前不存在,或者来电者无权更改envid。

**Exercise 9.** Implement the code in page\_fault\_handler in kern/trap.c required to dispatch page faults to the user-mode handler. Be sure to take appropriate precautions when writing into the exception stack. (What happens if the user environment runs out of space on the exception stack?)

#### User-mode Page Fault Entrypoint

Next, you need to implement the assembly routine that will take care of calling the C page fault handler and resume execution at the original faulting instruction. This assembly routine is the handler that will be registered with the kernel using sys\_env\_set\_pgfault\_upcall().

**Exercise 10.** Implement the \_pgfault\_upcall routine in lib/pfentry.S. The interesting part is returning to the original point in the user code that caused the page fault. You'll return directly there, without going back through the kernel. The hard part is simultaneously switching stacks and re-loading the EIP.

Finally, you need to implement the C user library side of the user-level page fault handling mechanism.

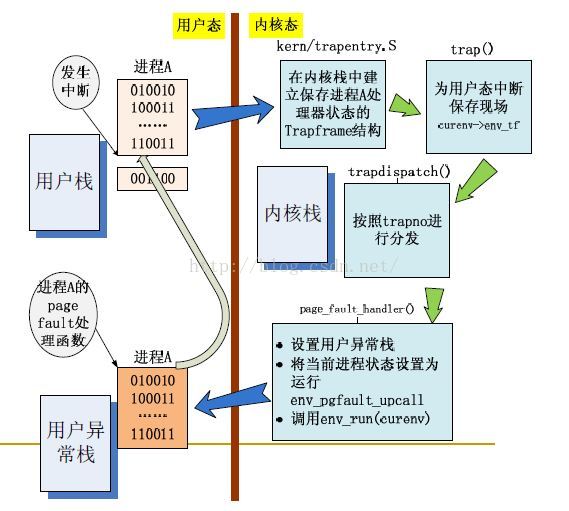
**Exercise 11.** Finish set\_pgfault\_handler() in lib/pgfault.c.

以上的部分内容自己完全不知道实现的思路。所以开始参照网上

以下部分内容是是拷贝自：<https://blog.csdn.net/fang92/article/details/50223659>

要想实现写时复制的fork，首先就是要完成页面异常的程序。

在JOS里面，不考虑内核态的page fault，只考虑用户态的page fault，即只在内核态引发page fault。和其他中断函数不同，当引发page fault中断时,page fault的处理函数是在用户态中执行的，其处理函数也是在用户态时设置的，其中栈换成了异常栈。网上挂了一张图，是整个page fault的执行过程，清晰明了：



用户想实现COW技术的的其中一个要求就是当发生页错误时候，用户环境产生的页面错误想要交给用户自己来处理；而不再由内核处理。

其实整个思路就是，发生了页错误，然后进入了trap()函数，之后trap()函数经过分发将将页错误交给函数page fault handler来处理。

在trap中，首先要判断是否注册了用户自己定义的page fault handler，如果定义了的话，我们将处理交给用户定义的函数。具体来说就是两个操作，一个是更换栈，一个是更换eip。

经过自己理解之后的程序运行的大致思路：

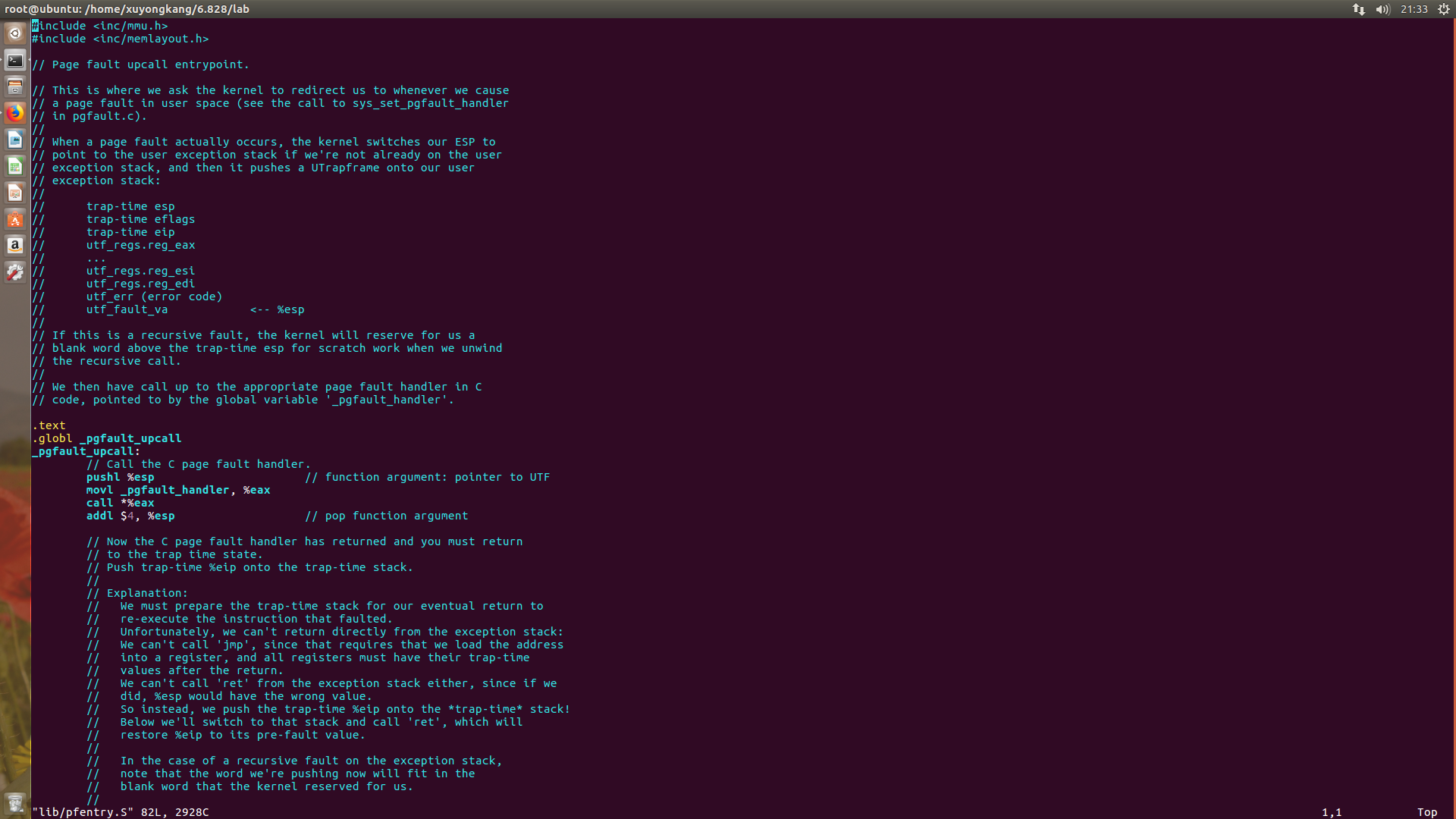
首先用户有一个环境，在这个环境的最前面要先：

set\_pgfault\_handler(handler);

handler是一个用户自己定义的C语言函数

然后

Set\_pgfault\_handler(handler)调用sys\_env\_set\_pgfault\_upcall (sys\_getenvid(), \_pgfault\_upcall)



\_pgfault\_upcall函数会跳到\_pgfault\_handler也就是用户自己定义的handler位置