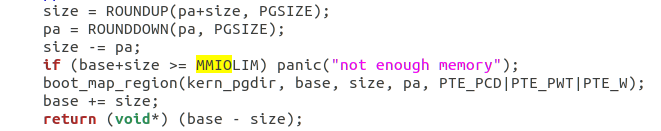
练习1：

在lapic\_init函数的开头就会调用mmio\_map.

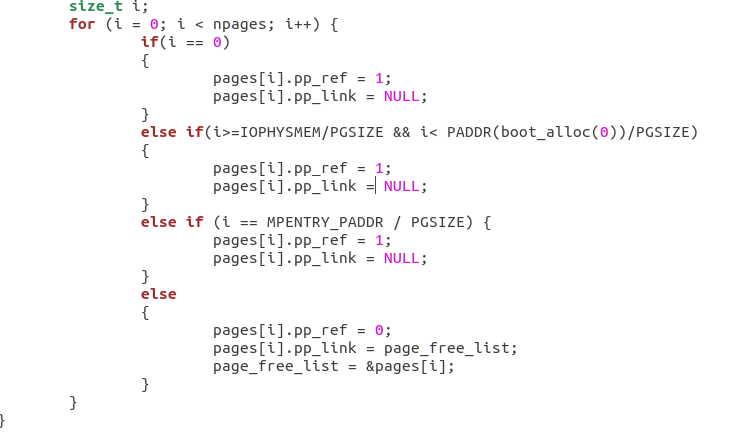


在kern/pmap.c中，有具体的提示，设置个静态变量记录每次变化后的虚拟基地址，使用boot\_map\_region函数将[pa,pa+size)的物理地址映射到[base,base+size)，记得把size roundup到PGSIZE。由于这是设备内存并不是正常的DRAM，所以使用cache缓存访问是不安全的，我们可以用页的标志位来实现。



练习2：

在boot\_aps函数中将启动代码放到了MPENTRY\_PADDR处，而代码的来源则是在kern/mpentry.S中，功能与boot.S中的非常类似，主要就是开启分页，转到内核栈上去，当然这个时候实际上内核栈还没建好。在执行完mpentry.S中的代码之后，将会跳转到mp\_main函数中去。而这里需要提前做的，就是将MPENTRY\_PADDR处的物理页表标识为已用，这样不会讲这一页放在空闲链表中分配出去。只需要在page\_init中添加一个判断就可以。



问题1：

在AP的保护模式打开之前，是没有办法寻址到3G以上的空间的，因此用MPBOOTPHYS是用来计算相应的物理地址的。

但是在boot.S中，由于尚没有启用分页机制，所以我们能够指定程序开始执行的地方以及程序加载的地址；但是，在mpentry.S的时候，由于主CPU已经处于保护模式下了，因此是不能直接指定物理地址的，给定线性地址，映射到相应的物理地址是允许的。

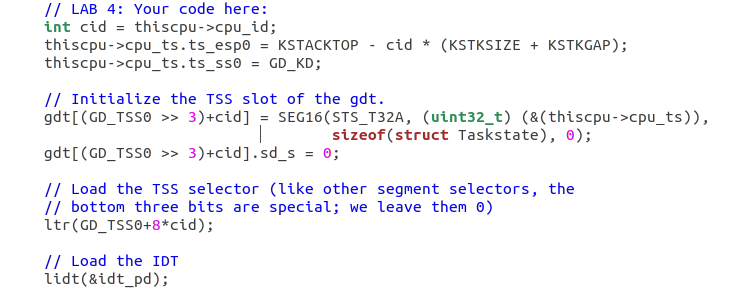
练习3：

需要为每个核都分配一个内核栈，每个内核栈的大小是KSTKSIZE，而内核栈之间的间距是KSTKGAP，起到保护作用。



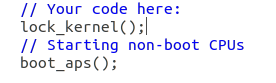
练习4：

由于有多个CPU，所以在这里不能使用原先的全局变量ts，应该利用thiscpu指向的CpuInfo结构体和cpunum函数来为每个核的TSS进行初始化。



练习5：

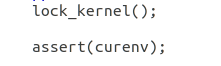
i386\_init



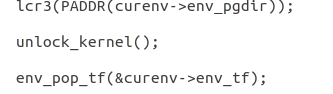
mp\_main



trap



env\_run

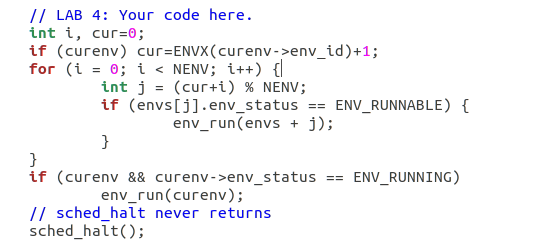


问题2：

因为不同的内核栈上可能保存有不同的信息，当1个CPU从内核退出来之后，有可能在内核栈中留下了一些将来还有用的数据，所以一定要有单独的栈。

练习6：

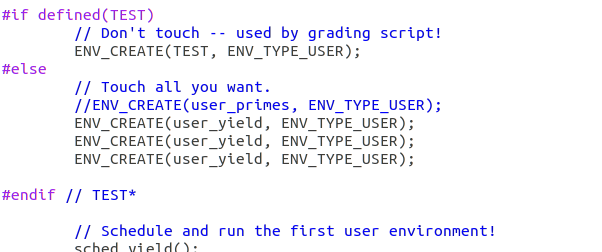
实现sched\_yield函数，并添加系统调用。



syscall() 函数分配 sys\_yield()：



修改kern/init.c的i386\_init()创建3个进程：



问题3：

因为当前是运行在系统内核中的，而每个进程的页表中都是存在内核映射的。每个进程页表中虚拟地址高于UTOP之上的地方，只有UVPT不一样，其余的都是一样的，只不过在用户态下是看不到的。所以虽然这个时候的页表换成了下一个要运行的进程的页表，但是curenv的地址没变，映射也没变，还是依然有效的。

问题4：

因为不进行保存，旧进程运行时的状态就丢失了，运行就不正确了。每次进入到内核态的时候，当前的运行状态都是在一进入的时候就保存了的。如果没有发生调度，那么之前trapframe中的信息还是会恢复回去，如果发生了调度，恢复的就是被调度运行的进程的上下文了。

挑战3：

给中断加入保存浮点寄存器的功能。

　　1、给inc/trap.h文件中的Trapframe结构新增char tf\_fpus[512]成员，并增加uint32\_t tf\_padding0[3]来对齐

2、修改kern/trapentry.S文件中的\_alltraps函数，加入保存fpu寄存器的功能

\_alltraps:

pushl %ds

pushl %es

pushal

// save FPU

subl $524, %esp

fxsave (%esp)

movl $GD\_KD, %eax

movl %eax, %ds

movl %eax, %es

push %esp

call trap

3、修改kern/env.c文件中的env\_pop\_tf函数，加入恢复fpu功能

\_\_asm \_\_volatile("movl %0,%%esp\n"

"\tfxrstor (%%esp)\n"

"\taddl $524,%%esp\n"

"\tpopal\n"

"\tpopl %%es\n"

"\tpopl %%ds\n"

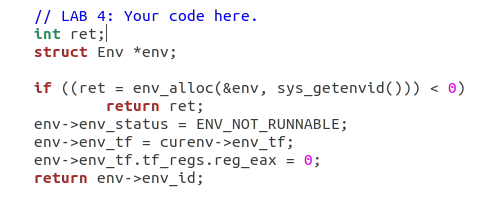
"\taddl $0x8,%%esp\n" /\* skip tf\_trapno and tf\_errcode \*/

"\tiret"

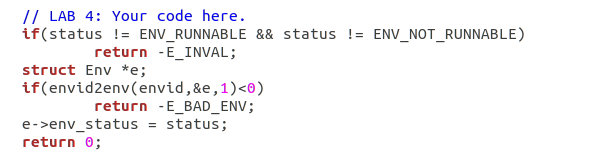
: : "g" (tf) : "memory");

练习7：

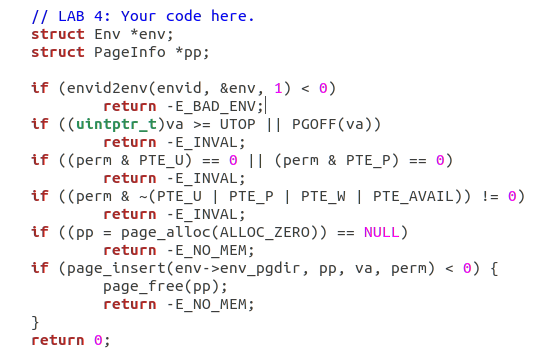
首先是sys\_exofork函数，这个系统调用将创建1个新的空白进程，没有映射的用户空间且无法运行。在调用函数时新进程的寄存器状态与父进程相同，但是在父进程会返回子进程的ID，而子进程会返回0。通过设置子进程的eax为0，来让系统调用的返回值为0。



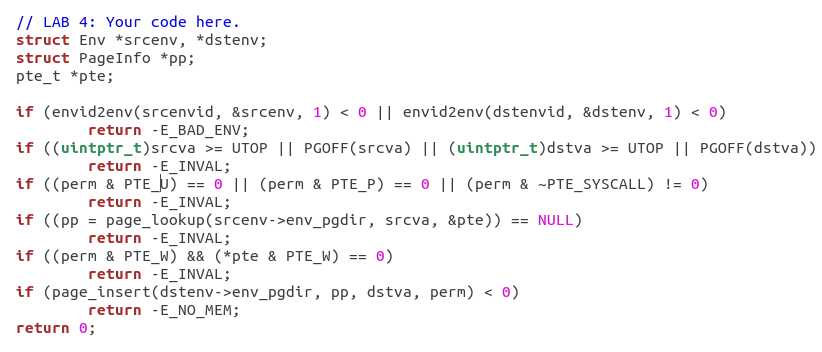
接着是sys\_env\_set\_status函数，设置进程的状态为ENV\_RUNNABLE或者ENV\_NOT\_RUNNABLE。



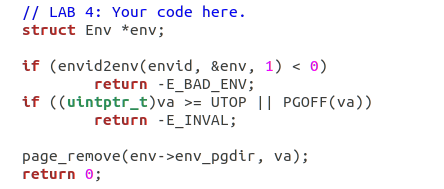
然后是env\_page\_alloc函数，分配1个物理页并映射到给定进程的进程空间的虚拟地址。



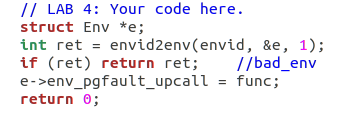
接着是sys\_page\_map函数，从1个进程的页表中拷贝1个页映射到另1个进程的页表中。将进程id为srcenvid的进程的srcva处的物理页的内容，映射到进程id为dstenvid的进程的dstva处。



最后是sys\_page\_unmap函数，解除指定进程中的1个页映射。

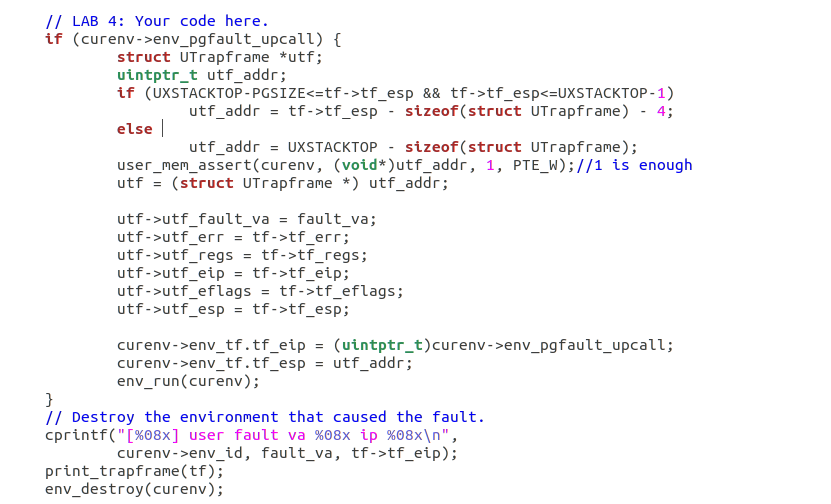


练习8：



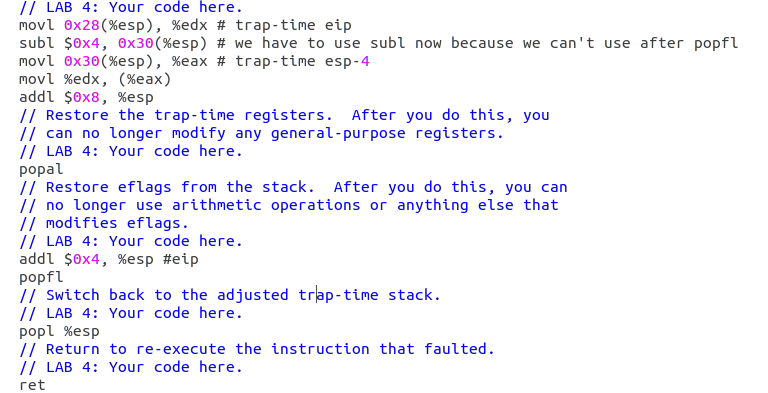
练习9：

如果当前已经在用户错误栈上了，那么需要留出4个字节，否则不需要，具体和跳转机制有关系。简单说就是在当前的错误栈顶的位置向下留出保存UTrapframe的空间，然后将tf中的参数复制过来。修改当前进程的程序计数器和栈指针，然后重启这个进程，此时就会在用户错误栈上运行中断处理程序了。当然，中断处理程序运行结束之后，需要再回到用户运行栈中，这个就是异常处理程序需要做的了。



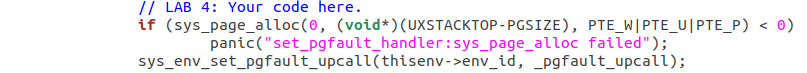
练习10：

\_pgfault\_upcall是所有用户页错误处理程序的入口，在这里调用用户自定义的处理程序，并在处理完成后，从错误栈中保存的UTrapframe中恢复相应信息，然后跳回到发生错误之前的指令，恢复原来的进程运行。



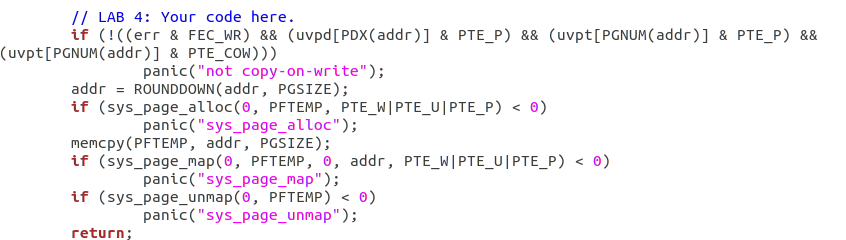
练习11：

进程在运行前注册自己的页错误处理程序，重点是申请用户异常栈空间，最后添加上系统调用号。



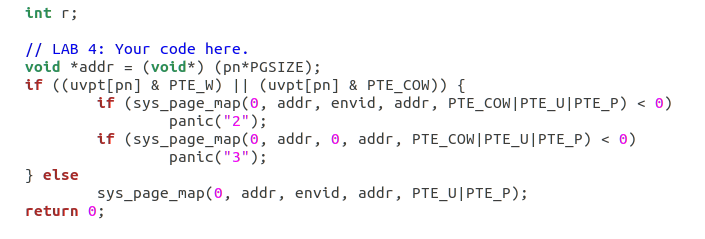
练习12：

首先是pgfault处理page fault时的写时拷贝。

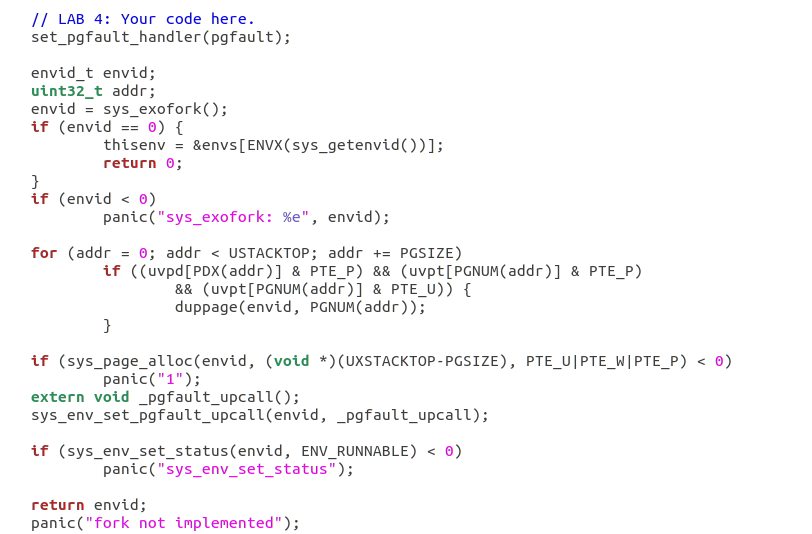


在pgfault函数中先判断是否页错误是由写时拷贝造成的，如果不是则panic。借用了一个一定不会被用到的位置PFTEMP，专门用来发生page fault的时候拷贝内容用的。先解除addr原先的页映射关系，然后将addr映射到PFTEMP映射的页，最后解除PFTEMP的页映射关系。

接下来是duppage函数，负责进行COW方式的页复制，将当前进程的第pn页对应的物理页的映射到envid的第pn页上去，同时将这一页都标记为COW。

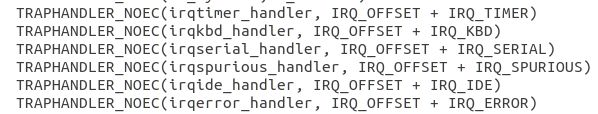


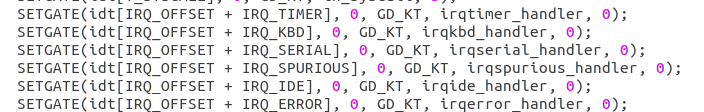
最后是fork函数，将页映射拷贝过去，这里需要考虑的地址范围就是从UTEXT到UXSTACKTOP为止，而在此之上的范围因为都是相同的，在env\_alloc的时候已经设置好了。



练习13：

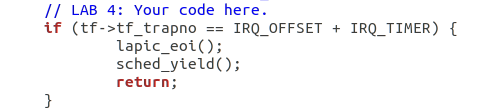
模仿原先设置默认中断向量即可，在kern/trapentry.S中定义IRQ0-15的处理例程。





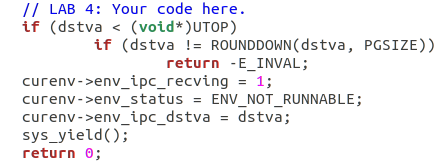


练习14：

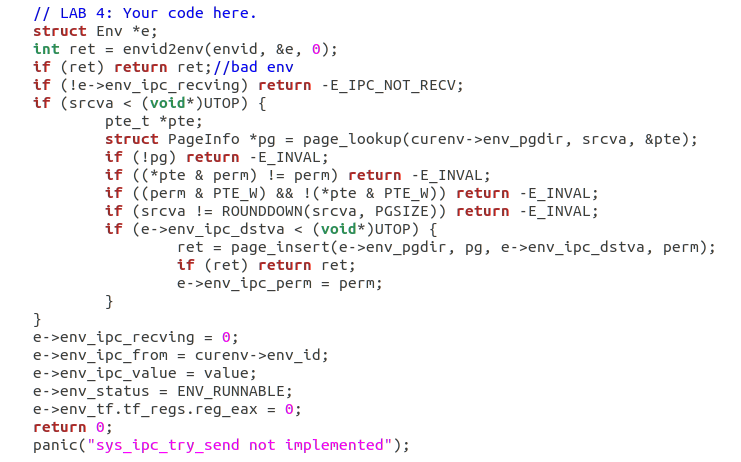


练习15：

首先是sys\_ipc\_recv函数，其功能是当一个进程试图去接收信息的时候，应该将自己标记为正在接收信息，而且为了不浪费CPU资源，应该同时标记自己为ENV\_NOT\_RUNNABLE，只有当有进程向自己发了信息之后，才会重新恢复可运行。最后将自己标记为不可运行之后，调用调度器运行其他进程。



接着是sys\_ipc\_try\_send函数，其实现相对来说麻烦很多，因为有很多的检测项，包括权限是否符合要求，要传送的页有没有，能不能将这一页映射到对方页表中去等等。如果srcva是在UTOP之下，那么说明是要共享内存，那就首先要在发送方的页表中找到srcva对应的页表项，然后在接收方给定的虚地址处插入这个页表项。接收完成之后，重新将当前进程设置为可运行，同时把env\_ipc\_recving设置为0，以防止其他的进程再发送，覆盖掉当前的内容。



最后是2个用户态库函数的实现。

