**Lab 4**

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

**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.

Lapic\_init的起始处调用了mmio\_map\_region，

**lapic = mmio\_map\_region(lapicaddr, 4096);**

这个函数的功能是映射物理地址pa到虚拟地址base上，大小为size，根据注释调用boot\_map\_region函数即可，如图1所示。

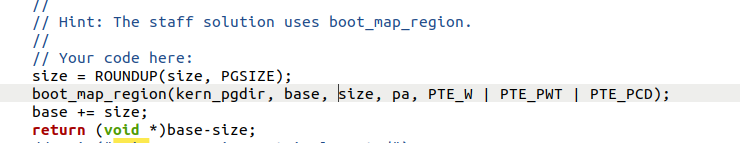


图1

**Exercise 2.** Read boot\_aps() and mp\_main() in kern/init.c, and the assembly code in kern/mpentry.S. Make sure you understand the control flow transfer during the bootstrap of APs. Then modify your implementation of page\_init() in kern/pmap.c to avoid adding the page at MPENTRY\_PADDR to the free list, so that we can safely copy and run AP bootstrap code at that physical address. Your code should pass the updated check\_page\_free\_list() test (but might fail the updated check\_kern\_pgdir() test, which we will fix soon).

添加一个if判断条件即可，如图2所示

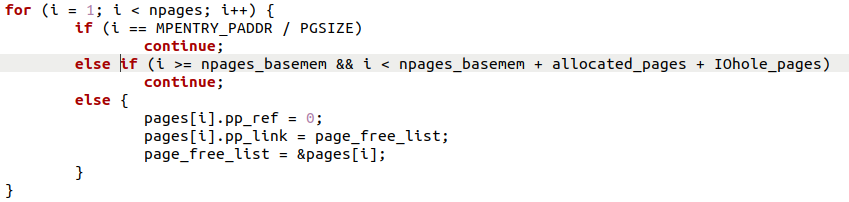


图2

**Question**

**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.**

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，需要通过给定线性地址，然后映射到相应的物理地址。

**Exercise 3.** Modify mem\_init\_mp() (in kern/pmap.c) to map per-CPU stacks starting at KSTACKTOP, as shown in inc/memlayout.h. The size of each stack is KSTKSIZE bytes plus KSTKGAP bytes of unmapped guard pages. Your code should pass the new check in check\_kern\_pgdir().

实现 mem\_init\_mp()函数去映射每个cpu的内核栈，cpu的内核栈如图3所示

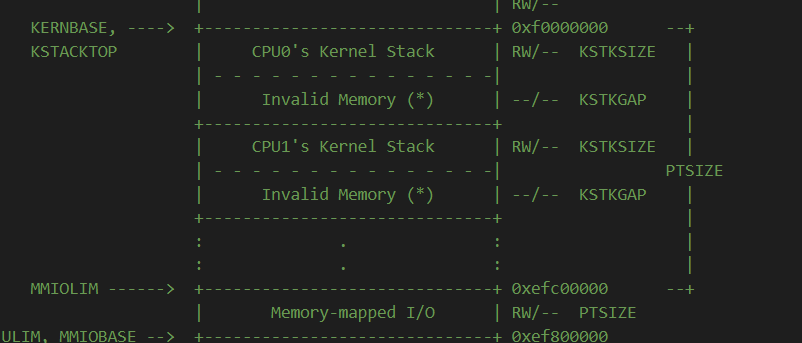


图3

（kstacktop\_i - KSTKSIZE, kstacktop\_i）是需要映射的，然后经过一段长度KSTKGAP的内存不需要映射，因此代码如图4所示

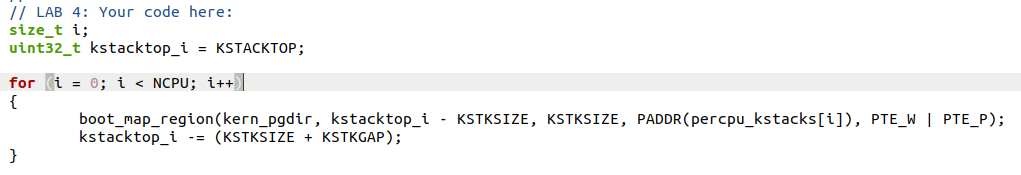


图4

**Exercise 4.** The code in trap\_init\_percpu() (kern/trap.c) initializes the TSS and TSS descriptor for the BSP. It worked in Lab 3, but is incorrect when running on other CPUs. Change the code so that it can work on all CPUs. (Note: your new code should not use the global ts variable any more.)

这个练习是实现多个cpu下的trap\_init\_percpu()函数，源代码是单个cpu下的对TSS和TSS描述符的初始化，根据注释进行修改即可，如图5所示

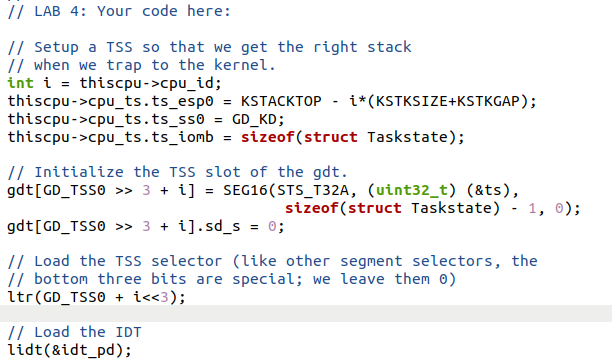


图5

Exercise 5是对一些函数中添加获得锁（lock\_kernel()）和释放锁（unlock\_kernel()）操作，比较简单，在此不再赘述

**Question**

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.

当触发了一个异常或中断时，内核会在处理程序获取大内核锁之前将一些信息push堆栈。因此，当一个CPU处于内核态下时，其他CPU可能会push自己的异常信息进栈。如果所有 CPU 共享一个内核堆栈，则可能会出现问题，比如说CPU0正在处理异常，此时，如果CPU1上的异常被触发，就会将其一些异常信息push堆栈。当CPU0完成这项工作时，它将弹出CPU1 push进来的堆栈而不是它自己的堆栈。在这种情况下，内核堆栈将被破坏。

**Exercise 6.** Implement round-robin scheduling in sched\_yield() as described above. Don't forget to modify syscall() to dispatch sys\_yield().

Make sure to invoke sched\_yield() in mp\_main.

Modify kern/init.c to create three (or more!) environments that all run the program user/yield.c.

Exercise 6是在sched\_yield()中实现RR调度算法，找到curenv之后status为RUNNABLE的第一个env，并且去运行它，如图6所示

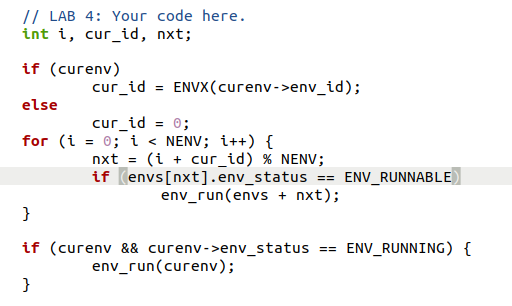


图6

然后输入make qemu CPUS=2命令，就会出现和官网上描述相同的内容，如图7

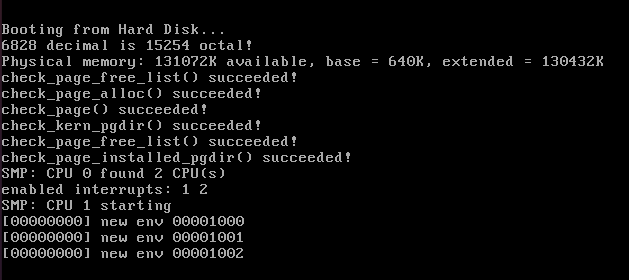


图7

**Question**

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?
2. 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?
3. 因为[UENVS， UENVS+PTSIZE)的映射物理地址都是一样的，所以lcr3切换了页表但是对于进程的e指针还是不用变
4. 切换环境时肯定要保存原来的寄存器状态，不然无法恢复到原来的环境中。发生环境切换时会触发trap handler，将寄存器状态存储到env\_tf中

Exercise 7是实现一些函数，并在syscall()中进行相应的处理

**sys\_exofork():**通过当前运行的环境创造一个子环境，子环境和父环境有相同的寄存器状态、env\_status标记为ENV\_NOT\_RUNNABLE，并且把子环境的返回值设为0，如图8

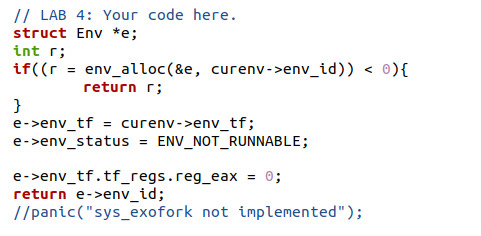


图8

**sys\_env\_set\_status():**将给定环境的status设置为ENV\_RUNNABLE 或 ENV\_NOT\_RUNNABLE，如图9

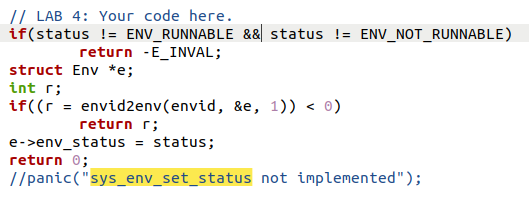


图9

**sys\_page\_alloc():**给环境分配页并在虚拟地址上进行映射，如图10

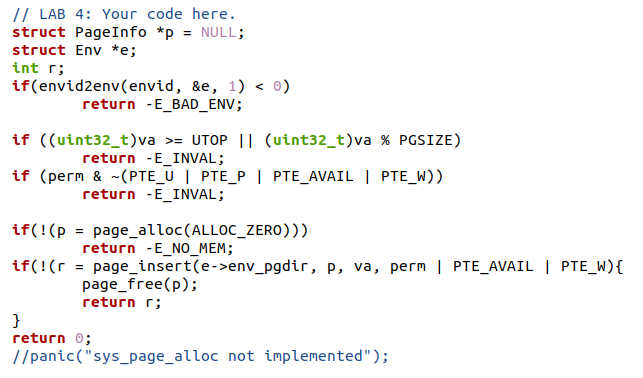


图10

**sys\_page\_map():**从一个环境中的地址空间中复制一个页面映射到另一个环境中的地址空间中，跟着注释一步一步写就能写出来，类似上一个函数，代码如图11所示

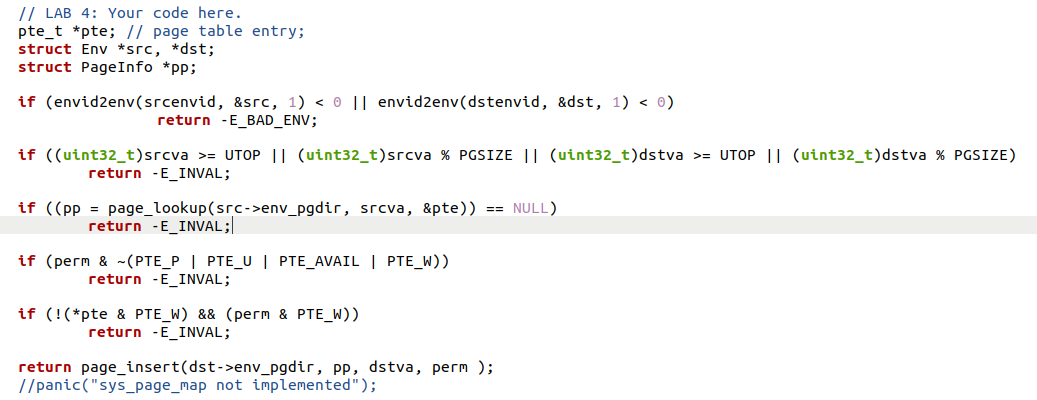


图11

**sys\_page\_unmap():**类似上面几个函数，判断条件更少了，比较简单，如图12

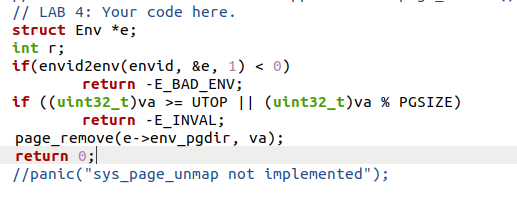


图12

## **Part B: Copy-on-Write Fork**

**Exercise 8.** Implement the sys\_env\_set\_pgfault\_upcall system call. Be sure to enable permission checking when looking up the environment ID of the target environment, since this is a "dangerous" system call.

实现sys\_env\_set\_pgfault\_upcall系统调用，类似上面几个函数，如图13

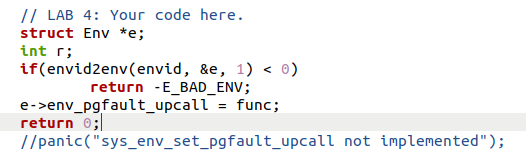


图13

**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?)

实现page\_fault\_handler函数，这个函数相对比较复杂，首先需要判断当前环境下的page fault upcall是否存在，否则就destroy这个环境；其次是判断UTrapframe的位置。如果它不是由另一个页面错误引起的，它就在 UXSTACKTOP的下方。否则它应该在前一个堆栈的下方，先压入一个32比特的空字；然后调用user\_mem\_assert()判断当前环境是否有权限访问UTrapframe的位置；最后是设置该UTrapframe的一些变量然后转换环境，代码如图14所示

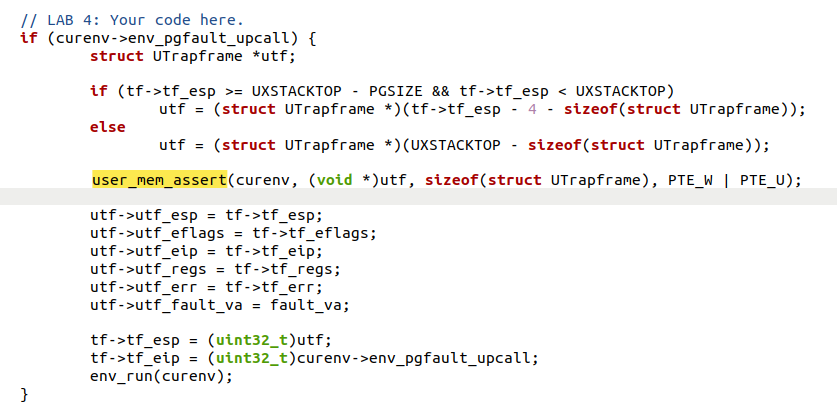


图14

**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.

上面的Exercise要求实现一段汇编，其中调用缺页中断处理程序的代码已经写好，我们需要实现从处理程序返回到原来触发缺页终端的正常程序，这一部分对我来说比较困难，这一部分我是从网上查阅资料得到的，如图15所示

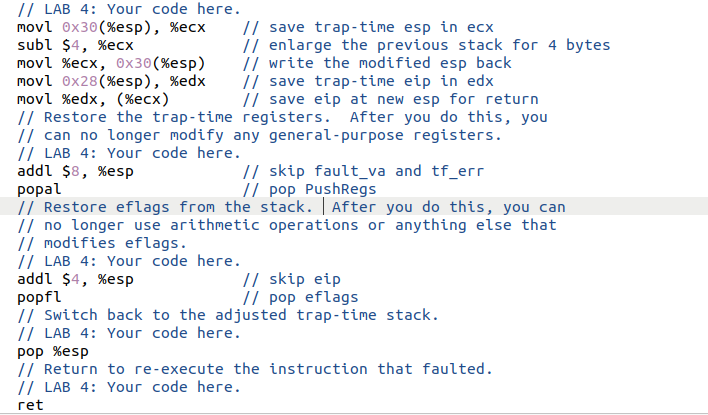


图15

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

完善set\_pgfault\_handler()函数，需要补全分配页面和设置缺页中断处理的系统调用，增加两个判断条件即可，如图16

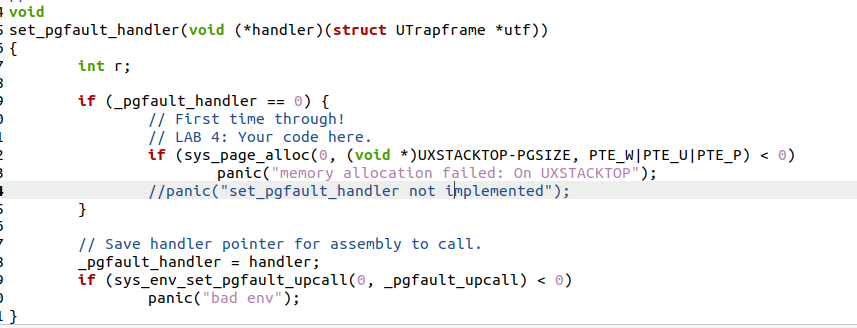


图16

**Exercise 12.** Implement fork, duppage and pgfault in lib/fork.c.

**fork():**这个函数的作用是创建一个新环境，然后扫描父环境的整个地址空间并在子环境中设置相应的页面映射，函数的基本流如下：

1. 父环境调用set\_pgfault\_handler()函数将 pgfault()作为页面错误处理程序。
2. 父环境调用sys\_exofork()来创建子环境。
3. 对于在UTOP以下地址空间中的每个可写或写时复制的页面，父环境调用 duppage，它会将写时复制页面映射到子级的地址空间，然后重新映射页面写时复制自己的地址空间。
4. 父环境设置子环境的用户页面错误入口点。
5. 父环境将子环境标记为可运行。

代码如图17所示

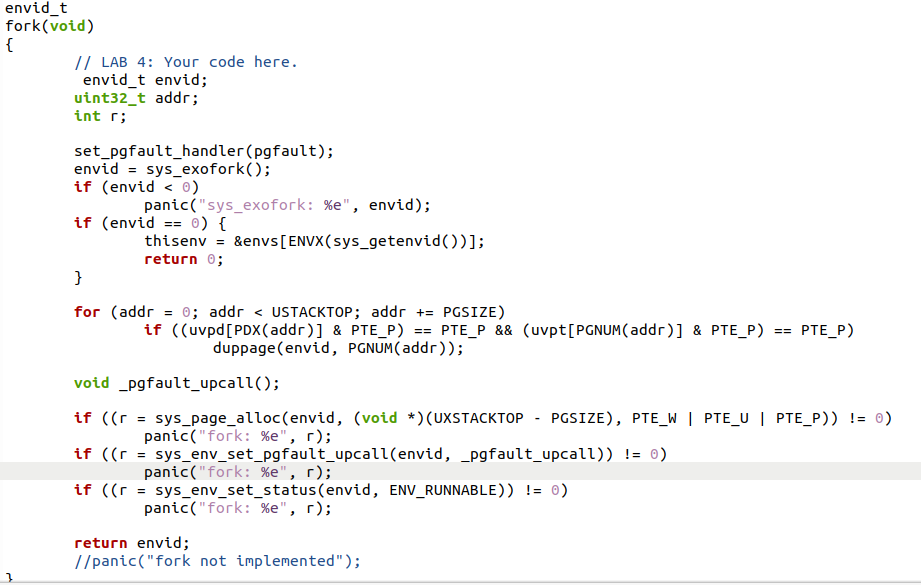


图17

**duppage():**在同一个虚拟地址上将虚拟页面pn映射到目标envid。如果页面可写或写时复制，必须在写时复制创建新映射，然后我们的映射必须是也标记为写时复制。代码如图18所示

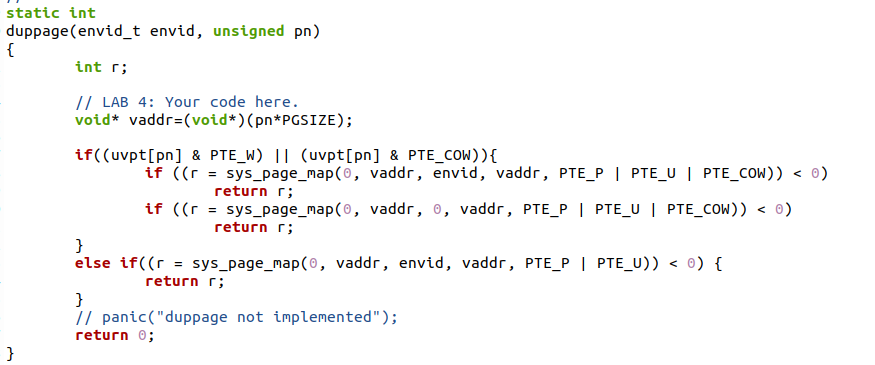


图18

**pgfault():**先判断错误访问的是否是一个可写或者是写时复制的页面，如果不是则需要panic，然后分配一个新页面，并映射到一个临时的地址，并从旧的页面复制数据和页面地址到新页面，如图19

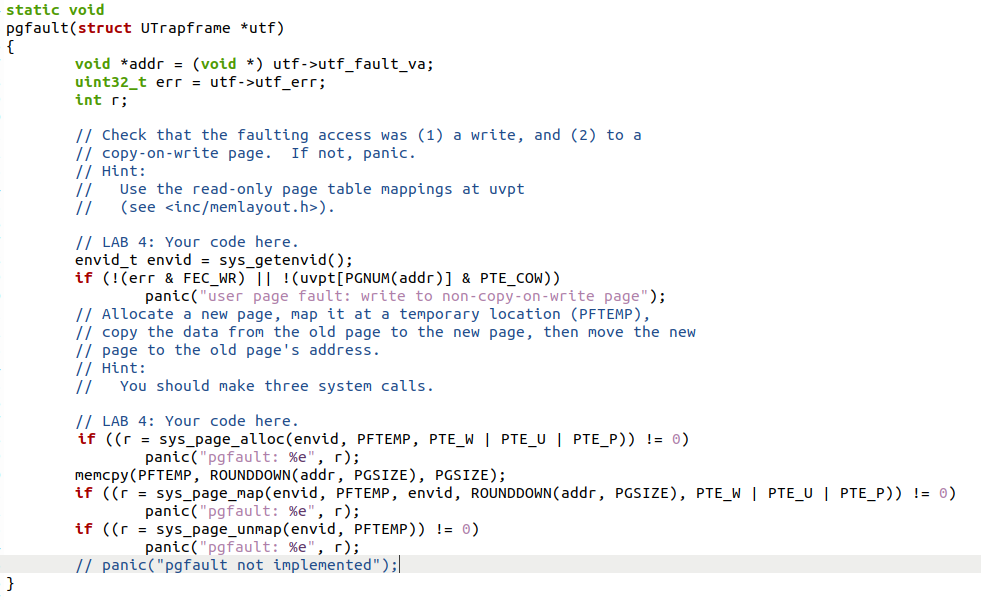


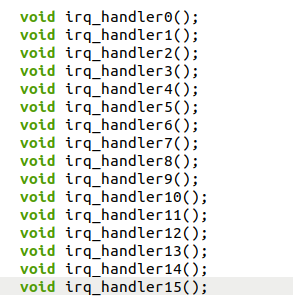
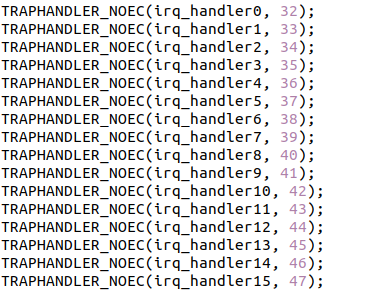
图19

## Part C: Preemptive Multitasking and Inter-Process communication (IPC)

**Exercise 13**. Modify kern/trapentry.S and kern/trap.c to initialize the appropriate entries in the IDT and provide handlers for IRQs 0 through 15. Then modify the code in env\_alloc() in kern/env.c to ensure that user environments are always run with interrupts enabled.

Also uncomment the sti instruction in sched\_halt() so that idle CPUs unmask interrupts.

这个练习比较简单，添加外部中断，编号从0-15的IRQ编号映射到IDT表项的32-47，和之前的中断一样，在trapentry.S和trap.c中进行修改即可，然后修改env\_alloc()函数，设置%eflags寄存器的FL\_IF 位，开启中断，代码如图20所示



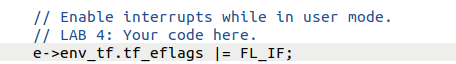
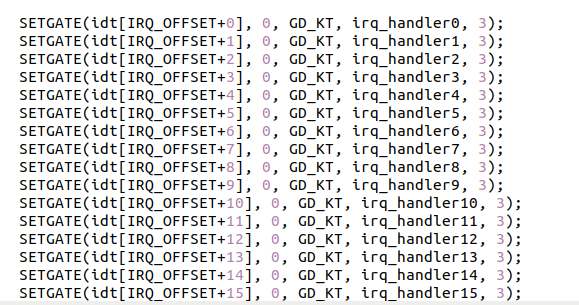


图20

**Exercise 14.** Modify the kernel's trap\_dispatch() function so that it calls sched\_yield() to find and run a different environment whenever a clock interrupt takes place.

在switch中添加一个case即可，如图21

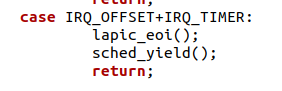


图21

Exercise 15. Implement sys\_ipc\_recv and sys\_ipc\_try\_send in kern/syscall.c. Read the comments on both before implementing them, since they have to work together. When you call envid2env in these routines, you should set the checkperm flag to 0, meaning that any environment is allowed to send IPC messages to any other environment, and the kernel does no special permission checking other than verifying that the target envid is valid.

Then implement the ipc\_recv and ipc\_send functions in lib/ipc.c.

首先是实现sys\_ipc\_recv和sys\_ipc\_try\_send函数，虽然需要写的代码不少，但是照着注释来也不会很复杂，如图22、图23

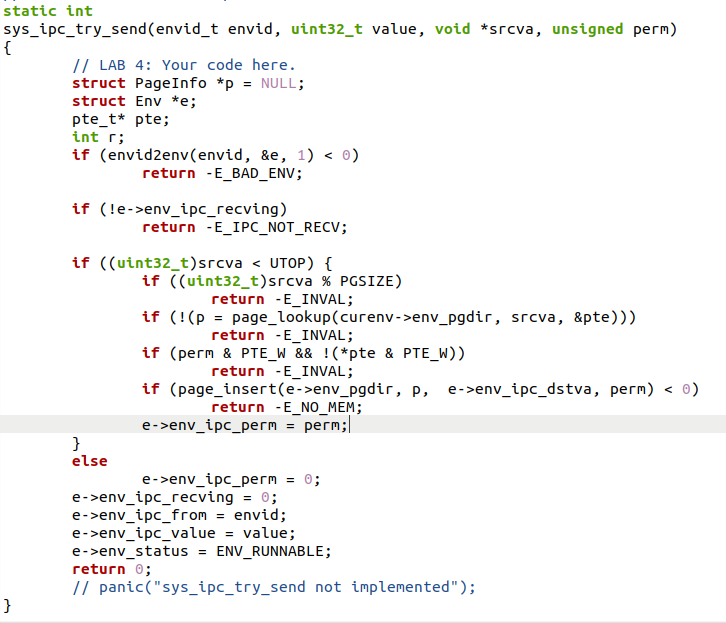


图22

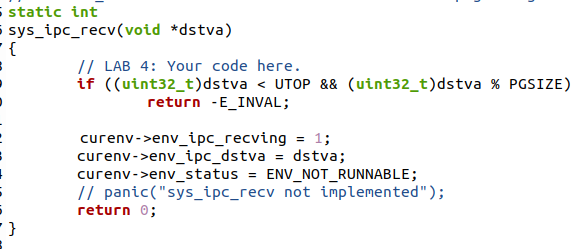


图23

然后是实现ipc\_recv和ipc\_send函数，这两个函数是上面两个函数的包装体，主要是调用上面的两个函数，代码如图24、图25



图24

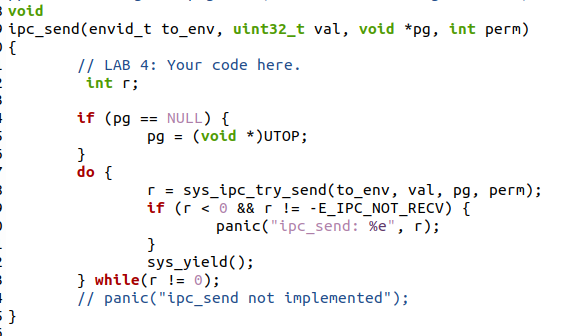


图25

图26为lab 4的结果

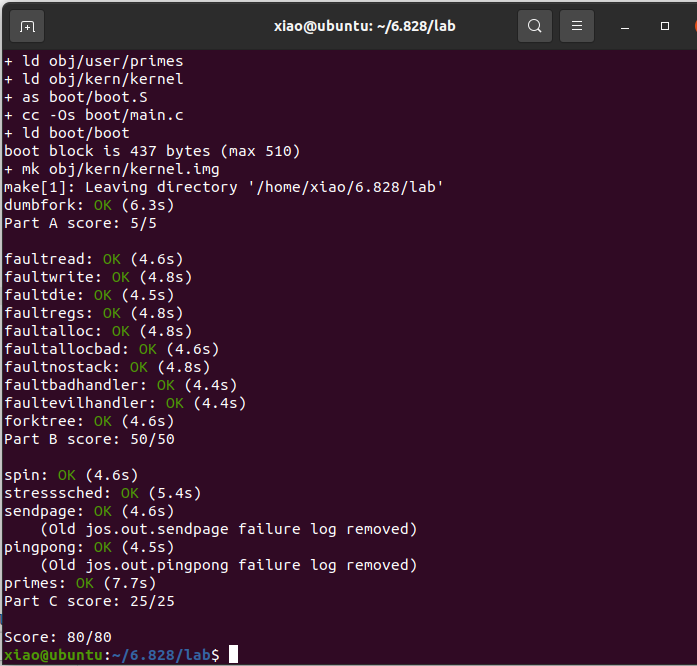


图26