**LAB4**

**PART A**

**exercise 1**

处理器通过 映射于内存的输入输出 (memory-mapped I/O, MMIO) 来访问它的 LAPIC。在 MMIO 模式中，物理内存的一部分被硬连线于一些 I/O 设备的寄存器上（译注：I/O 设备的寄存器和内存被映射在物理内存的一些区域）， 所以通常用于访问内存的存取指令也可以同样用于访问设备寄存器。

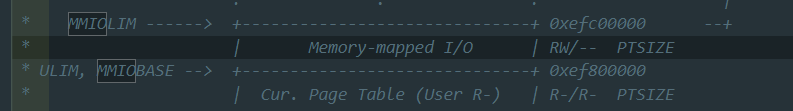
LAPIC 被连接在物理地址 0xFE000000 (距 4GB 还有 32MB 的位置) 的 I/O hole 上，这个地址对我们当前在 KERNBASE 上直接映射来说太高了， 不过 JOS 虚拟内存映射表 在 MMIOBASE 位置留了 4MB 的空隙，所以我们可以将这样的设备映射到这里。



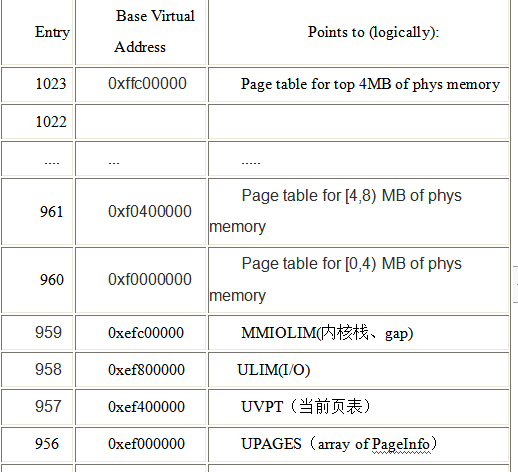
实际上需要将该物理地址映射到相应的虚拟地址即可。故相应的实现应该为：



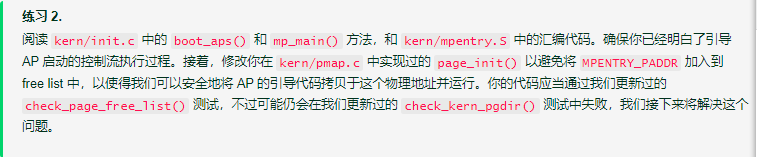
其空间为：

[](https://github.com/leliyliu/jos_work/blob/master/lab4/fig/code1.png)

需要检测其实际空间是否超过了MMIOLIM，然后需要调用之前在lab2中实现的函数（下图），将相应的虚拟地址空间和物理地址建立映射，并且相应的权限，根据代码的提示，应该为：PTE\_PCD | PTE\_PWT | PTE\_W。



**exercise 2**

****

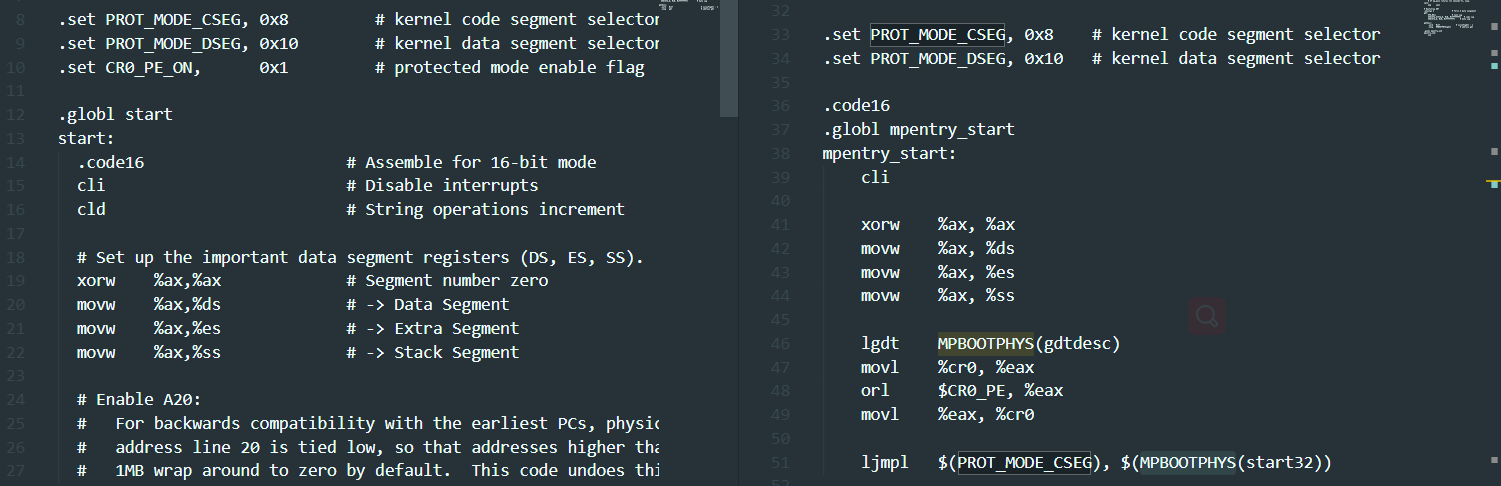
可以看到，对于boot\_aps()函数，其具体为：循环所有的cpu，当某个cpu未启动的时候，将相应的mpentry\_kstack进行保存，并调用lapic\_startap来对其进行相应的处理，一直等到该CPU开启。

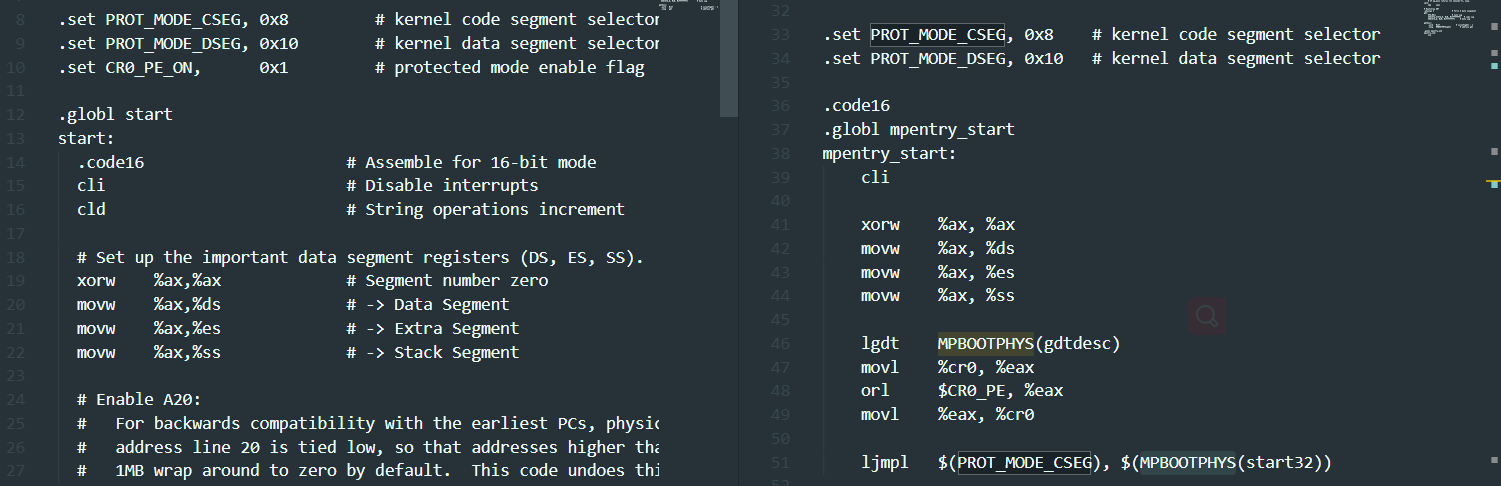
对于mp\_main函数而言，其进行相应的初始化操作，并启动CPU，发送信号给boot\_aps告诉当前状态。

kern/mpentry.s 的代码实际上是对于引导的处理，是一个相关的启动代码。其类似于boot.s代码，主要做了一些控制上的处理。由于相应的要求，故需要排除掉将==MPENTRY\_PADDR==加入到其中，即**需要加一个if语句，判断 i == mp\_page**，如果满足，则不将其加入其中。

看到其结果为：check\_page\_free\_list能够测试成功，但是在测试check\_kern\_pgdir时出现错误，可以看到，有kernel panic。

**问题1：**

*逐行比较 kern/mpentry.S 和 boot/boot.S。牢记 kern/mpentry.S 和其他内核代码一样也是被编译和链接在 KERNBASE 之上运行的。那么，MPBOOTPHYS 这个宏定义的目的是什么呢？为什么它在 kern/mpentry.S 中是必要的，但 在 boot/boot.S 却不用？换句话说，如果我们忽略掉 kern/mpentry.S 哪里会出现问题呢？ 提示：回忆一下我们在 Lab 1 讨论的链接地址和装载地址的不同之处。*[](https://github.com/leliyliu/jos_work/blob/master/lab4/fig/code2.png)



对比代码可以发现boot/boot.S与kern/mpentry.S中的不同。

mpentry.S代码mpentry\_start, mpentry\_end的地址都在KERNBASE(0xf0000000）之上，实模式无法寻址，所以需要通过MPBOOTPHYS来寻址。而boot.S加载的位置本身就是实模式可寻址的低地址，所以不用额外转换。

**exercise 3**



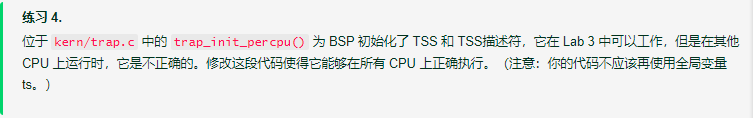
根据要求，将相应的堆栈映射到相应的地址空间即可。



将相应的CPU根据序号分配到相应的地址空间 ==KSTACKTOP - KSTKSIZE - i \*(KSTKSIZE + KSTKGAP)== 即可。同时，需要赋予的权限为：PTE\_W。

这时候运行make grade，可以看到check\_kern\_pgdir()已经可以成功运行。

**exercise 4**



具体代码实现如下图所示：为了使得其能够在所有的CPU上运行，即需要使用thiscpu来指示当前的cpuid,tss等属性，根据注释要求完成代码。

完成之后，运行make qemu CPUS=4，其最终结果就如同的实验要求所示。

**exercise 5**

****

加锁位置1：i386\_init(void)



加锁位置2：mp\_main(void)



加锁位置3：trap(struct Trapframe \*tf)



释放锁位置：env\_run(struct Env \*e)



根据前述的要求，需要加锁和解锁的位置已经相应，对于i386\_init()中，显然需要在boot\_aps()之前加锁，这样才能在运行AP之前加上锁。而对于mp\_main()中而言，显然需要其在继续执行之前加上锁，即当其传递了相应信号，但是还未执行之前加上锁。而对于trap()而言，在检测当前的运行环境的时候(进程)，就需要加上锁。最后一个释放锁的位置，应该在env\_pop\_tf函数设置进程的寄存器前。在这个函数执行后，回到用户进程。

**问题2：**

*看起来使用全局内核锁能够保证同一时段内只有一个 CPU 能够运行内核代码。既然这样，我们为什么还需要为每个 CPU 分配不同的内核堆栈呢？请描述一个即使我们使用了全局内核锁，共享内核堆栈仍会导致错误的情形。*

这是因为虽然全局内核锁限制了多个进程同时执行内核代码，但是在陷入trap()之前，CPU硬件已经自动压栈了SS, ESP, EFLAGS, CS, EIP等寄存器内容，而且在trapentry.S中也压入了错误码和中断号到内核栈中，所以不同CPU必须分开内核栈，否则多个CPU同时陷入内核时会破坏栈结构，此时都还没有进入到trap()的加大内核锁位置。

**exercise 6**

**轮盘调度算法**：

你的下一个任务是修改 JOS 内核以使其能够以 轮转 的方式在多个进程中切换。JOS 的轮转调度算法像这样工作：

* kern/sched.c 中的 sched\_yied() 函数负责挑选一个进程运行。它从刚刚在运行的进程开始，按顺序循环搜索 envs[] 数组（如果从来没有运行过进程，那么就从数组起点开始搜索），选择它遇到的第一个处于 ENV\_RUNNABLE（参考 inc/env.h）状态的进程，并调用 env\_run() 来运行它。
* sched\_yield() 绝不应当在两个CPU上同时运行同一进程。它可以分辨出一个进程正在其他CPU（或者就在当前CPU）上运行，因为这样的进程处于 ENV\_RUNNING 状态。
* 我们已经为你实现了新的系统调用 sys\_yield()，用户进程可以调用它来触发内核的 sched\_yield() 方法，自愿放弃 CPU，给其他进程运行。





同时需要修改mp\_main(void)中的代码：



代码需要查看当前是否有进程在运行，并从其开始，进行相应的搜索，查看是否可以找到ENV\_RUNNABLE的进程。如果没有，当前进程如果还需要运行则继续运行，否则halt。

在kern/init.c中创建三个或更多进程，运行user/yield.c

修改syscall.c



修改i386\_init(void)



可以看到，能够正常运行，并且运行make qemu CPUS=2则会出现练习中的错误。

**问题3**

*在你实现的 env\_run() 中你应当调用了 lcr3()。在调用 lcr3() 之前和之后，你的代码应当都在引用 变量 e，就是 env\_run() 所需要的参数。 在装载 %cr3 寄存器之后， MMU 使用的地址上下文立刻发生改变，但是处在之前地址上下文的虚拟地址（比如说 e ）却还能够正常工作，为什么 e 在地址切换前后都可以被正确地解引用呢？*

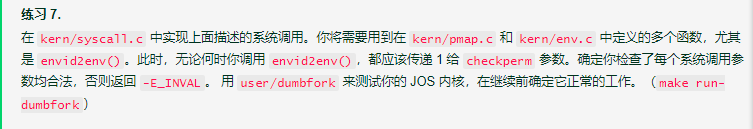
因为在env\_setup\_vm()函数中，设置所有的进程env\_pgdir跟kern\_pgdir是一样的，只有UVPT部分不一样。因此虚拟地址e映射到相同的物理地址。

**问题4**

*无论何时，内核在从一个进程切换到另一个进程时，它应当确保旧的寄存器被保存，以使得以后能够恢复。为什么？在哪里实现的呢？*

保存寄存器状态，是为了知道下一条指令地址以及进程栈的状态，否则我们不知道从哪里继续运行。保存寄存器状态的代码是 trap.c 中的 curenv->env\_tf = \*tf;

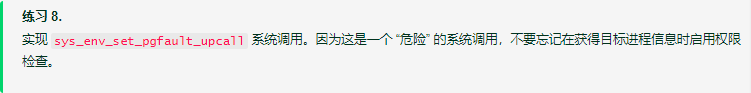
**exercise 7**

****

根据要求，实现相应的调用，参数的调用根据提示确定。然后，进行make grade测试，其最终结果为5分，part A部分已经完成。

**PART B**

**exercise 8**

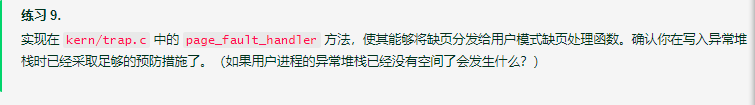


完成sys\_env\_set\_pgfault\_upcall，如下：



Env结构体中成员env\_pgfault\_upcall记录缺页处理函数入口点。

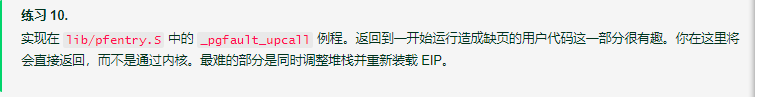
**exercise 9**

****



根据前面所提出的要求，首先需要判断用户进程在缺页发生时是否已经运行在异常堆栈上，并根据此确定utf的具体初始地址。然后根据当前的tf给utf赋值即可，当然前期需要判断是否有足够的空间，并设置当前进程的相应属性。

**exercise 10**

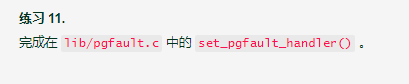


根据相应提示，可以实现代码如下：



在esp中存储了当前产生page\_fault的utf内容。pgfault\_upcall实际上被相应地调用，需要返回eip。实际上是对于异常处理之后返回的相应处理。故需要根据保存的esp找到相应的eip。由于需要将相应的寄存器都占用，故而不能先将eip保存在其它任何通用寄存器中，只能保存在堆栈中，故而需要将其保留在起初的esp的位置。

**exercise 11**



完成代码如下：



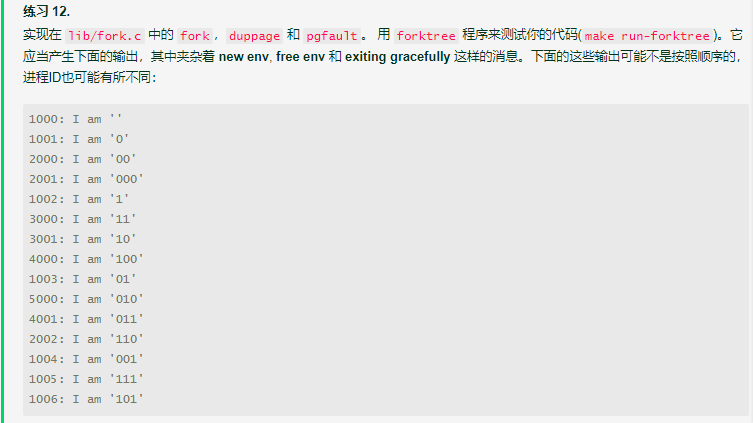
第一次设置页面错误处理函数时，做两件事：(1)为异常栈分配空间；(2)告诉内核发生缺页错误是调用\_pgfault\_upcall。

总结：

用户进程调用pgfault.c 的 set\_pgfault\_handler()（练习十一），设置异常栈，通过系统调用sys\_env\_set\_pgfault\_upcall（练习八），设置用户模式下的缺页处理程序入口为\_pgfault\_upcall；设置缺页处理程序\_pgfault\_handler = handler;

用户进程出现缺页异常时，trap\_dispatch()将这个异常分发给page\_fault\_handler()（练习九）处理，初始化当前异常栈帧，检查它是否出错，在这个栈帧中存入utf结构体，切换到这个异常栈帧；并将eip修改为\_pgfault\_upcall（练习十）；重新run用户进程，在用户模式下运行\_pgfault\_upcall，它调用用户程序\_pgfault\_handler具体处理缺页异常

**exercise 12**



对三个函数实现如下



根据相应的注释和之前所给出的提示，在lib/entry.S中设置了uvpt 和 uvpd 所以你可以很容易地在 lib/fork.c 中找到页表的信息（即page table 和 page directory)，故根据此，可以去检查相应的权限是否正确。然后根据allocate，map和unmap等操作，具体对每一步进行相应的检测即可。



duppage函数将相应的权限一直映射到相应的内存，并检测是否能映射成功即可。

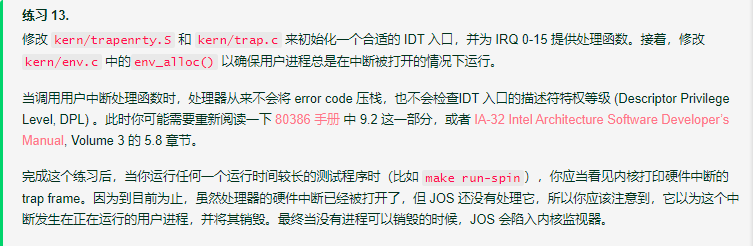


fork函数需要完成pgfault和sys\_exofork()并且对于其相应地址空间调用一次duppage，并且需要检测当前的进程状态。

make grade，part B完成。

**PART C**

**exercise 13**



修改kern/trapentry.S进行相应的初始化。



根据相应的手册中的说明，可以知道，其具体的对应可以在kern/trapentry.S中设置为此。

修改kern/trap.c进行相应的对应处理：

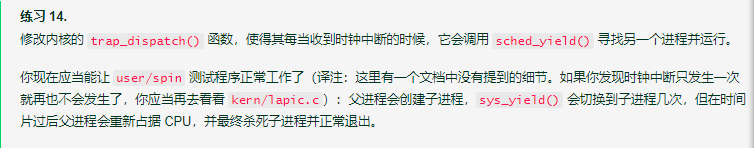


因此，这样已经将相应的IRQ与其中的处理对应起来，接下来需要修改env\_alloc()函数，来确保用户进程总是在中断被打开的情况下运行。



即设置当前用户进程的eflags中的EL\_IF始终被置位即可。（根据80386手册中相应的9.2这一部分的说明，可以看到其具体调用用户处理函数时处理器的相应操作。）

**exercise 14**



实际上只需要查看相应的调用即可，因此，修改trap\_dispatch()函数如下：

if (tf->tf\_trapno == IRQ\_OFFSET + IRQ\_TIMER) {

// cprintf("irq timer\n");

lapic\_eoi();

sched\_yield();

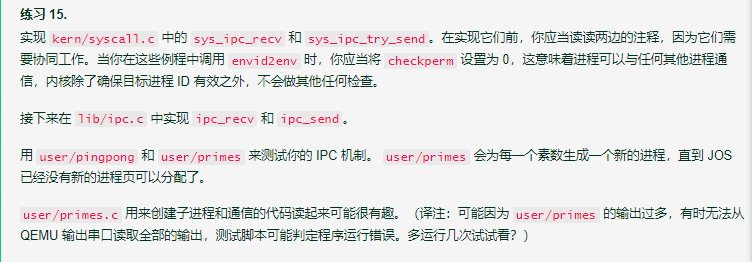
return;

}

使用lapic\_eoi()函数中断；调用调度函数。

现在进行make run-spin，已经可以直接通过。

**exercise 15**



关于在kern/syscall.c中的两个函数的实现，其具体代码为：



根据相应的提示，调用envid2env，然后查看是否有在receive的进程，并且根据注释进行相应的检测，主要是对于当前进程权限的检测。



对于sys\_ipc\_recv函数，实际上只需要判断当前的地址空间是否满足，然后设置当前的env\_ipc\_recving位为1即可，同时需要当前进程的状态即可，最后需要调用sys\_yield函数。

然后需要在lib/ipc.c中实现两个相应函数：



对于ipc\_recv而言，只需要**调用**系统中的ipc的通信接收过程，然后根据最终接收到的通信结果和相应函数的返回来进行处理即可。



而对于ipc\_send而言，也同样很简单，只需要**判断**发送是否成功，如果不成功，panic即可，否则，调用sys\_yield()函数。

最终，完成本次实验。