

**操作系统原理课程设计报告**

|  |  |
| --- | --- |
| 姓 名： | 苏墨馨 |
| 学 院： | 计算机科学与技术学院 |
| 专 业： | 计算机科学与技术 |
| 班 级： | ACM1601 |
| 学 号： | U01614831 |
| 指导教师： | 张杰 |

|  |  |
| --- | --- |
| 分数 |  |
| 教师签名 |  |

2019年 2月 28 日

[Lab1 Booting a PC 3](#_Toc2897976)

[1 实验目的 3](#_Toc2897977)

[2 实验内容 3](#_Toc2897978)

[3 实验问题。 18](#_Toc2897979)

[Lab2 Memory Management 22](#_Toc2897980)

[1 实验目的 22](#_Toc2897981)

[2 实验内容 22](#_Toc2897982)

[3 实验问题 27](#_Toc2897983)

[Lab3 User Environments 29](#_Toc2897984)

[1 实验目的 29](#_Toc2897985)

[2 实验内容 29](#_Toc2897986)

[3 实验问题 41](#_Toc2897987)

[Lab4 Preemptive Multitasking 43](#_Toc2897988)

[1 实验目的 43](#_Toc2897989)

[2 实验内容 43](#_Toc2897990)

[3 实验问题 54](#_Toc2897991)

### Lab1 Booting a PC

#### 1 实验目的

1. 熟悉x86汇编语言，计算机引导过程，并熟悉实验所需环境QEMU模拟器以及使用QEMU和GDB进行调试的过程；
2. 熟悉本实验环境中的引导程序，熟悉计算机操作系统引导过程；
3. 初步熟悉本实验环境所使用的操作系统内核JOS。

#### 2 实验内容

##### 2.1 Part 1: PC Bootstrap

配置qemu环境，下载lab1之后make产生一个kernel.img的镜像，输出如图1.1所示。执行make qemu即可在qemu里执行JOS kernel镜像。

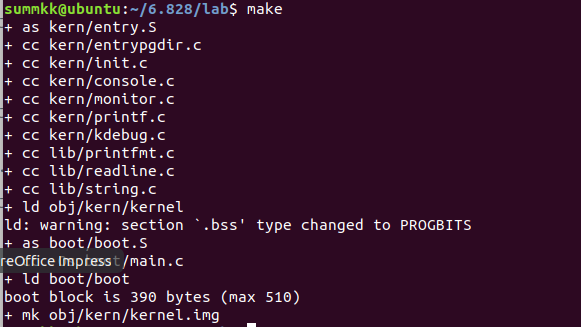


图1.1 编译lab

PC的物理地址空间如图1.2所示，JOS只用到PC的物理内存的第一个256MB，我们假设PC仅有32位物理地址空间。

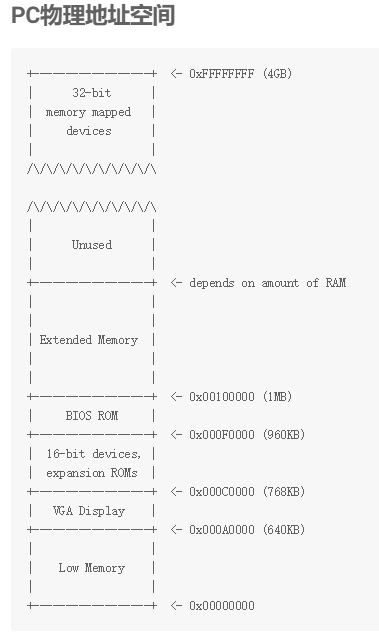


图1.2 PC物理地址空间

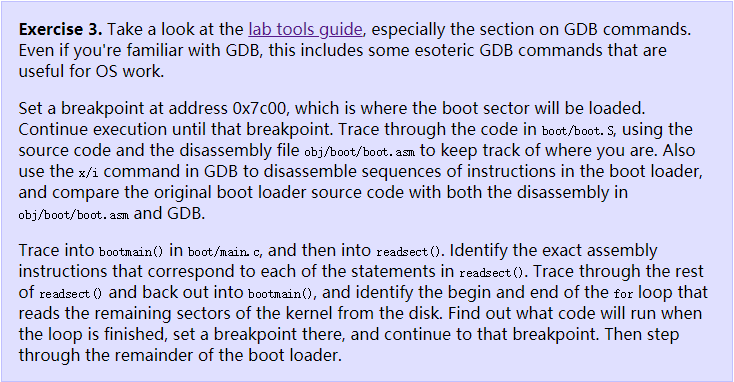
当处理器在上电启动的时候，BIOS首先控制机器，这时候没有其他程序运行，BIOS运行时设置一个中断描述符表(IDT)并初始化多个硬件设备。初始化PCI总线以及所有重要设备之后，BIOS会搜索可引导设备(bootable device)。当找到可引导磁盘之后，BIOS从磁盘读取引导加载程序(boot loader)，并将控制权转移至boot loader。

##### 2.2 Part 2: The Boot Loader

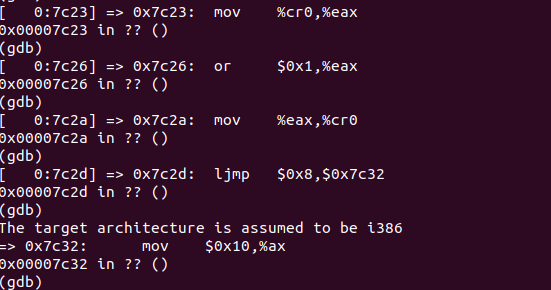
PC的软盘和硬盘分为512个字节区域，称为扇区。扇区是磁盘的最小传输粒度：每个读取或写入操作必须是一个或多个扇区，并在扇区边界上对齐。如果磁盘是可引导的，则第一个扇区称为引导扇区(boot sector)，因为这是boot loader代码所在的位置。当BIOS找到可引导的软盘或硬盘时，它将512字节的引导扇区加载到物理地址0x7c00到0x7dff的内存中，然后使用jmp指令将CS：IP设置为0000：7c00，将控制权传递给boot loader。

Boot loader第一个功能是将处理器从实模式切换到32位保护模式，在保护模式下，软件才能访问处理器物理地址空间1MB以上的所有内存。第二个功能是通过x86的特殊I / O指令直接访问IDE磁盘设备寄存器，从硬盘读取内核。

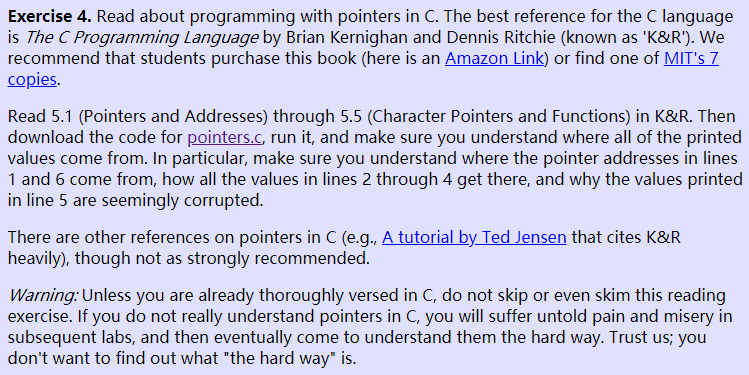
###### Exercise3:



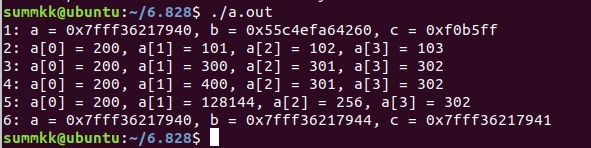
根据exercise3的要求。在0x7c00处设置断点，结合boot/boot.S以及obj/boot/boot.asm在GDB里调试并观察boot loader的执行过程。



###### Exercise4：



学习C语言中有关指针的部分，下载pointers.c的代码并执行，执行结果如下。



根据代码解释各行的输出原因如下。

1：a是数组首地址 b是分配的地址 c是未初始化的指针的地址

2：因为c指向a，修改c指向的值为200，则a[0]为200，其他的按照循环赋值的方式赋值为101 102和103

3：c[1]=a[1] \*(c+2) = a[2] 3[c] = a[3] 修改对应数组的值

4：c = c+1因此c指向a[1] 修改a[1]的值为400，其他不变

5：500=0x01f4 ，c由于先转换为char类型的指针再加一，c移动到a[1]的第二个字节处，再转换为int类型赋值0x1f4，a[1]的值由0x190变为0x1f490，a[2]的值由0x12D变为0x100

6：a仍为数组首地址。b为a增加一个int的长度，即增加4个字节。c为a增加一个char的长度，即增加1个字节。

当编译和链接C程序时，编译器将.c文件转换为包含硬件期望的二进制格式编码的汇编指令的.o文件。然后linker将所有编译的目标文件组合程单个二进制映像，成为ELF格式的二进制文件，即可执行和可链接格式。在本实验中，可以将ELF可执行文件视为带有加载信息的标头，后跟几个程序部分，每个程序部分是一个连续的代码块或数据，用于加载到指定地址的内存中。Boot loader不会修改代码或数据，它将其加载到内存中并开始执行它。

ELF二进制文件以固定长度的ELF头开始，后跟可变长度的程序头，列出要加载的每个程序段。这些ELF头的C定义在inc / elf.h中。我们感兴趣的部分是：

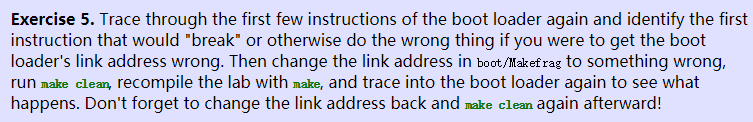
.text：程序的可执行指令。

.rodata：只读数据，例如C编译器生成的ASCII字符串常量。 （但是，我们不会费心设置硬件来禁止写入。）

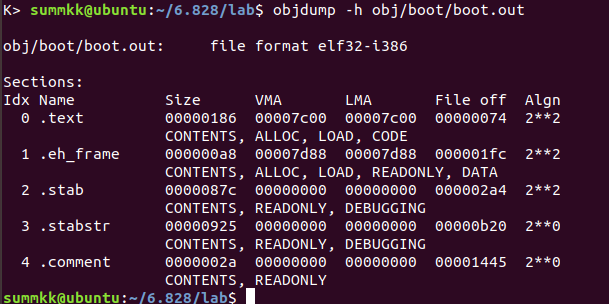
.data：数据部分保存程序的初始化数据，例如使用初始化程序（如int x = 5;）声明的全局变量。

当链接器计算程序的内存布局时，它会在一个名为.bss的部分中为未初始化的全局变量（如int x;）保留空间，该部分紧跟在内存中的.data之后。 C要求“未初始化”的全局变量以零值开头。因此，不需要在ELF二进制文件中存储.bss的内容，相反，链接器只记录.bss部分的地址和大小。加载程序或程序本身必须安排将.bss部分归零。

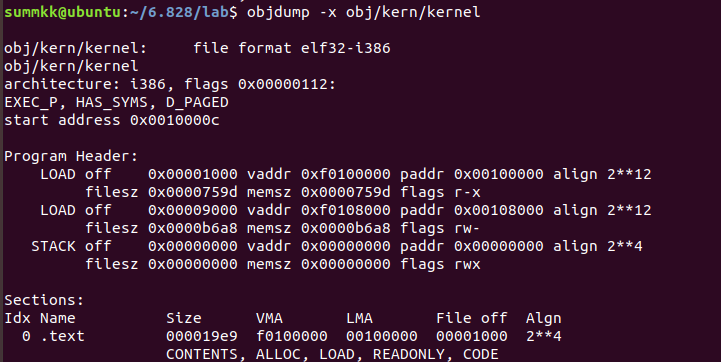
###### Exercise5：



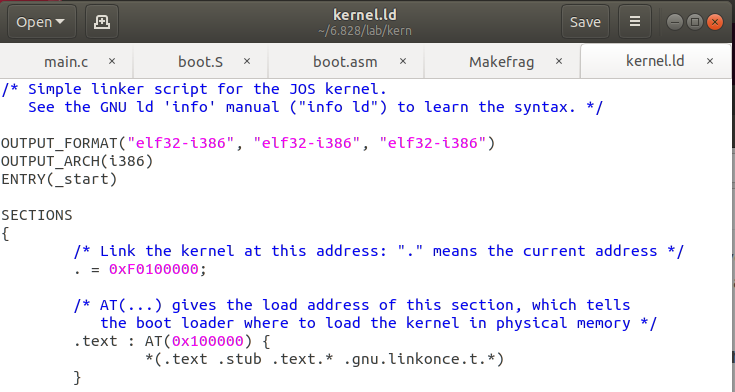
输入objdump –h xx.out可以查看可执行文件xx.out的所有部分的名称、大小和链接地址的完整列表。注意.text段的VMA(链接地址)和LMA(加载地址)。段的链接地址是段期望执行的内存地址。链接器以各种方式对二进制文件中的链接地址进行编码，例如当代码需要全局变量的地址时，如果二进制文件从未链接的地址执行，则二进制文件通常不起作用。通常，链接和加载地址是相同的。查看boot的.text部分如图，它的VMA和LMA相同。



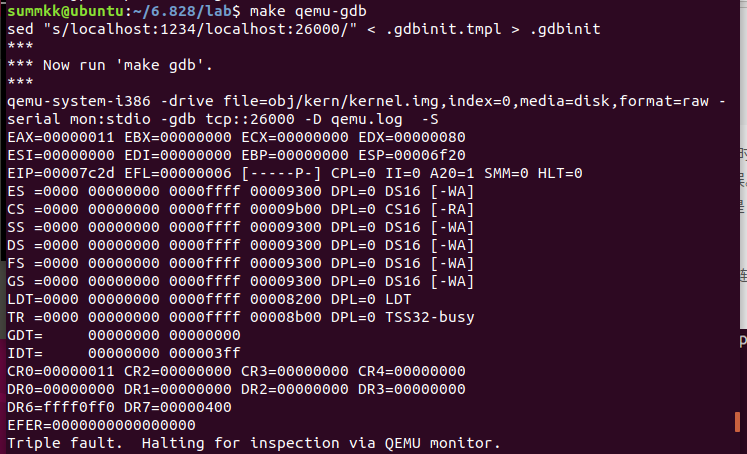
输入objdump –x xx可以检查program headers，输出LOAD(需要加载的区域)，以及vaddr(虚拟地址),paddr(物理地址), memsz和filesz (加载区域的大小)等信息。Kernel的.text段VMA和LMA不相同。boot loader 将kernel加载到低地址的内存中，但它实际是从高地址开始执行的。



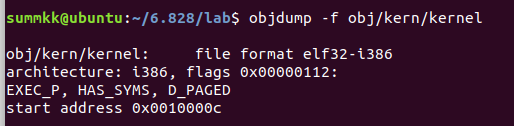
链接和加载地址在kernel.ld文件里，我们通过-Ttext重新设置链接，修改为0x7c80，重新make，然后用gdb进行单步调试。



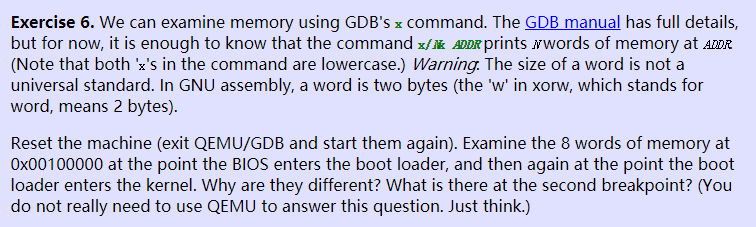
Gdb单步调试时无法继续，qemu端报错如下

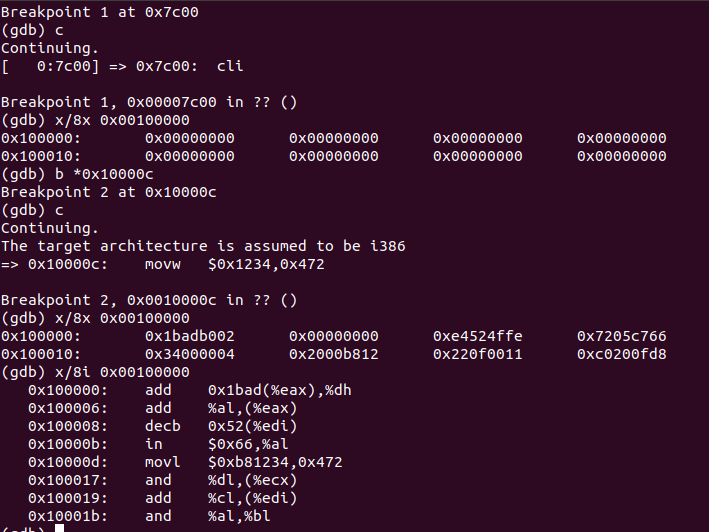


输入objdump -f xx的命令可以查看e\_entry的信息，它保存程序中入口点的链接地址，即程序开始执行的程序文本部分中的存储器地址。可以看到程序开始的地址为0x10000c。

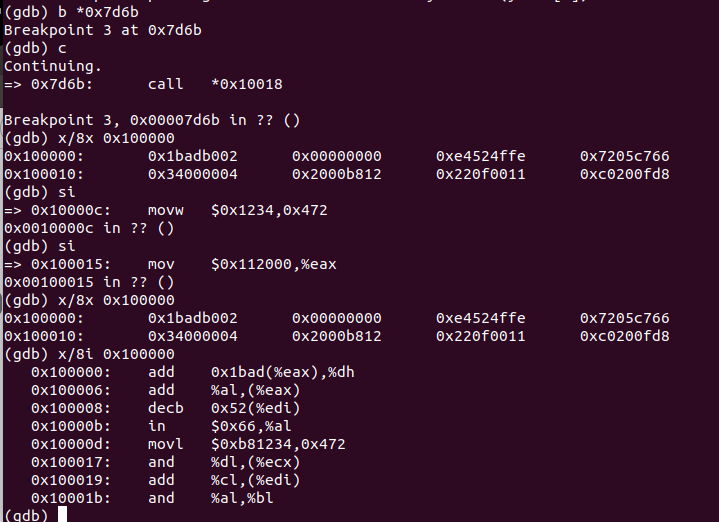


###### Exercise6：





进入引导加载程序的地址为0x7c00，进入内核的第一条指令在0x1000c，在这两处分别设置断点，可以看到，0x100000处存放的是加载的kernel程序内容。然后在0x7d6b调用函数，进入内核程序。

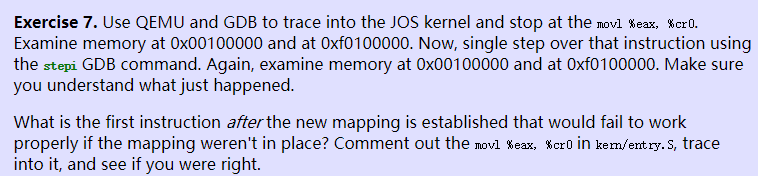


##### 2.3 Part 3: The Kernel

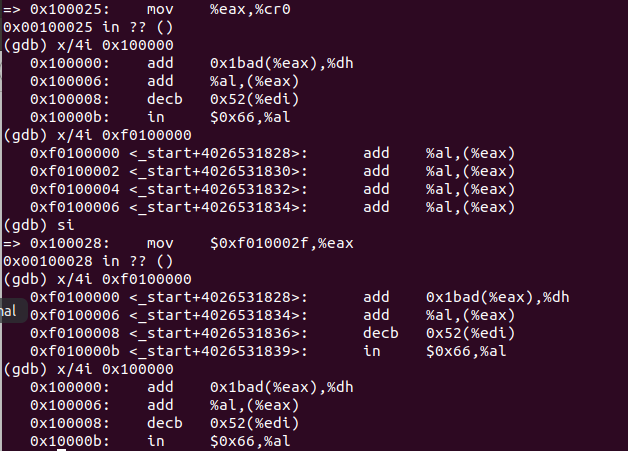
使用虚拟内存可以解决位置依赖的问题。在之前的实验中看到kernel的链接地址和加载地址差别很大，OS kernel一般选择链接在高的虚拟地址下运行以便留下处理器虚拟地址空间的下半部分供用户程序使用。许多机器在地址0xf0100000处没有任何物理内存，因此我们无法指望能够在那里存储内核。相反，我们将使用处理器的内存管理硬件将虚拟地址0xf0100000（内核代码期望运行的链接地址）映射到物理地址0x00100000（引导加载程序将内核加载到物理内存中）。这样，虽然内核的虚拟地址足够高，可以为用户进程留出足够的地址空间，但它将被加载到PC RAM的1MB点的物理内存中，就在BIOS ROM上方。这种方法要求PC至少有几兆字节的物理内存（因此物理地址0x00100000可以工作），但这可能适用于1990年以后建立的任何PC。

事实上，在下一个实验中，我们将把物理地址0x00000000到0x0fffffff的整个底部256MB的物理地址空间分别映射到虚拟地址0xf0000000到0xffffffff。现在，我们只需映射前4MB的物理内存，这足以让我们启动并运行。我们使用kern / entrypgdir.c中手写的，静态初始化的页面目录和页表来完成此操作。直到kern / entry.S设置CR0\_PG标志，内存引用被视为物理地址。一旦设置了CR0\_PG，内存引用就是虚拟内存硬件转换为物理地址的虚拟地址。我们建立一个页表将虚拟地址的[KERNBASE, KERNBASE+4MB)转换到物理地址的[0, 4MB)。entry\_pgdir将0xf0000000到0xf0400000范围内的虚拟地址转换为物理地址0x00000000到0x00400000，以及虚拟地址0x00000000到0x00400000到物理地址0x00000000到0x00400000。任何不属于这两个范围之一的虚拟地址都会导致硬件异常，转入中断处理（未编写）。

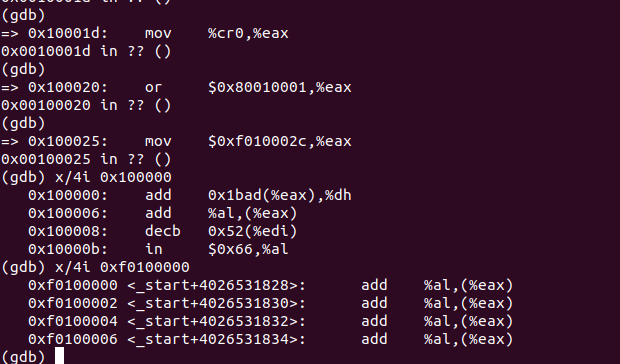
###### Exercise7：



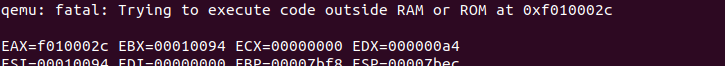
使用gdb检查内存0x100000和0xf0100000处的内容。到达指定语句时查看链接地址和加载地址的内容，不同。再单步执行一步后两个地址的内容相同。即完成了虚拟地址和物理地址的映射。



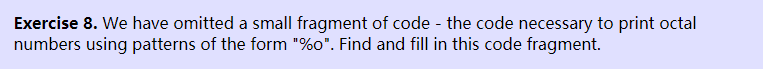
注释掉汇编语句mov %eax,%cr0，重新编译运行。



执行完同样的mov $0xf010002c,%eax后，两个地址的内容依旧不同，即映射失败。继续执行，qemu端出现报错，访问地址超出内存。



###### Exercise8：

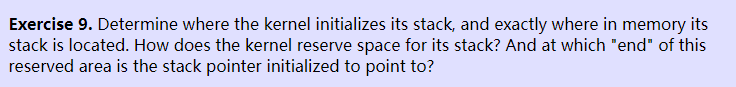


printf.c里调用vprintfmt函数，这个函数定义在printfmt.c里；printf.c和printfmt都调用putch函数，定义在console.c里。

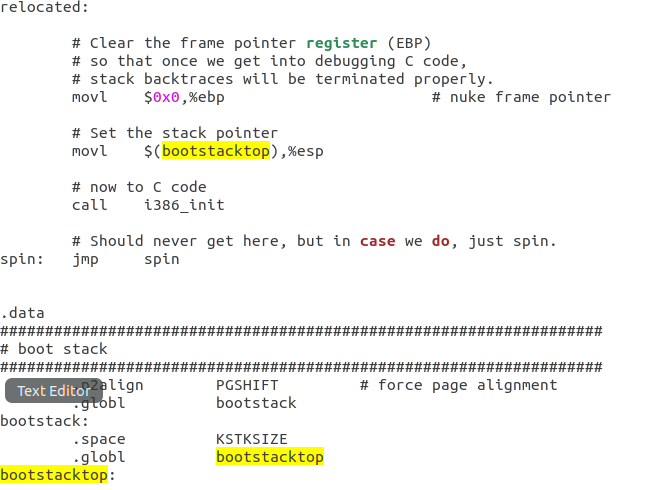
修改printfmt里的vprintfmt函数中case’o’的部分，重新make qemuzhih输出改变。



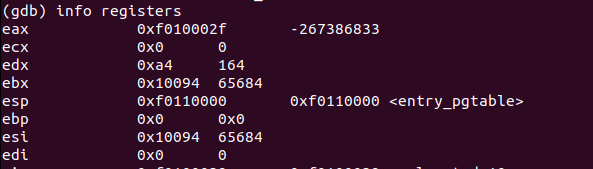
###### Exercise9：



初始化堆栈的代码部分在entry.s，根据设置堆栈指针esp的语句，bootstacktop为栈顶指针。根据bootstacktop的声明找到了.data段，bootstack中的KSTKSIZE 应该是堆栈的大小。



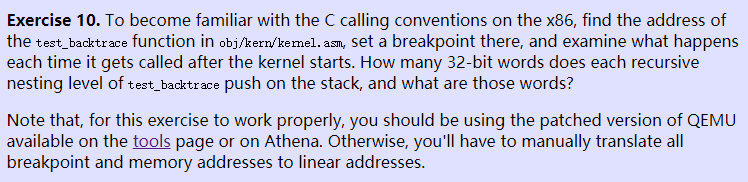
在gdb调试到对应位置mov语句后，查看esp和ebp的值。此时esp指向栈顶。

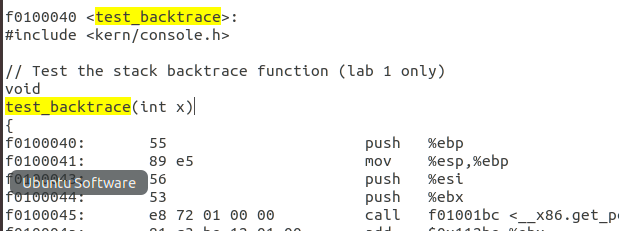


x86堆栈指针（esp寄存器）指向当前正在使用的堆栈上的最低位置。保留给堆栈的区域中该位置以下的所有内容都是空闲的。将值压入堆栈涉及减少堆栈指针，然后将值写入堆栈指针指向的位置。从堆栈中弹出一个值包括读取堆栈指针指向的值，然后增加堆栈指针。在32位模式下，堆栈只能保存32位值，esp总是可以被4整除。

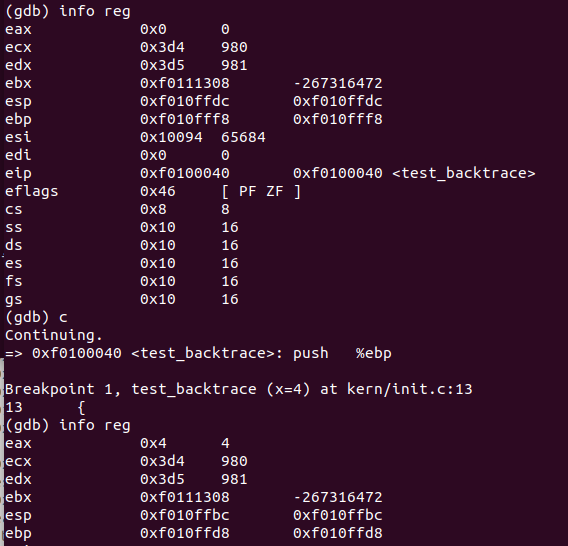
相反，ebp（基指针）寄存器主要通过软件约定与堆栈相关联。在进入C函数时，函数的prologue code通常通过将其推入堆栈来保存先前函数的基本指针，然后在函数持续时间内将当前esp值复制到ebp中。如果程序中的所有函数都遵循这个约定，那么在程序执行期间的任何给定点，都可以通过跟踪保存的ebp指针链并确切地确定嵌套的函数调用序列。

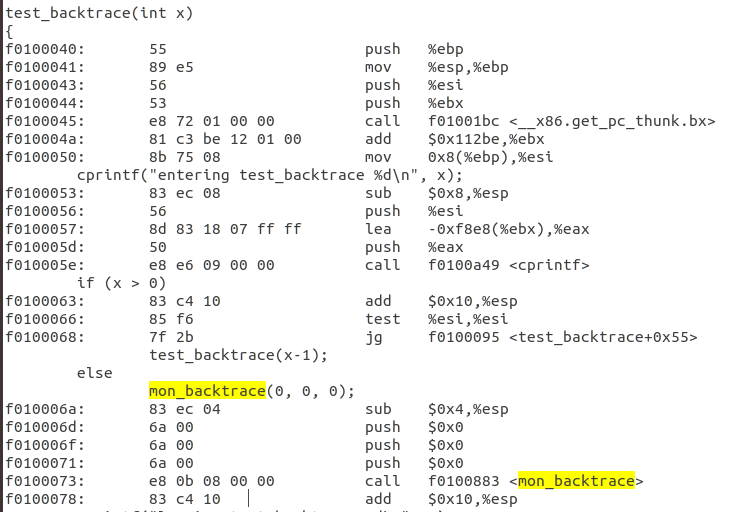
###### Exercise10：



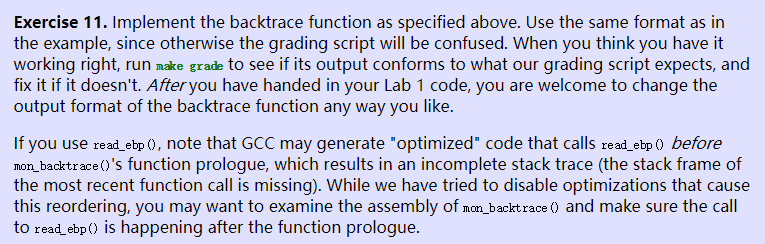


在obj / kern / kernel.asm中找到test\_backtrace函数，gdb运行到断点处观察寄存器esp和ebp的值，结合kernel.asm中的部分代码发现：每一次递归调用test\_backtrace，入栈参数，返回地址，ebp，然后更新ebp的值为esp的值，入栈部分存有变量的寄存器。最后移动esp，扩大栈的空间。前后esp变化了0x20即每次入栈8个字。

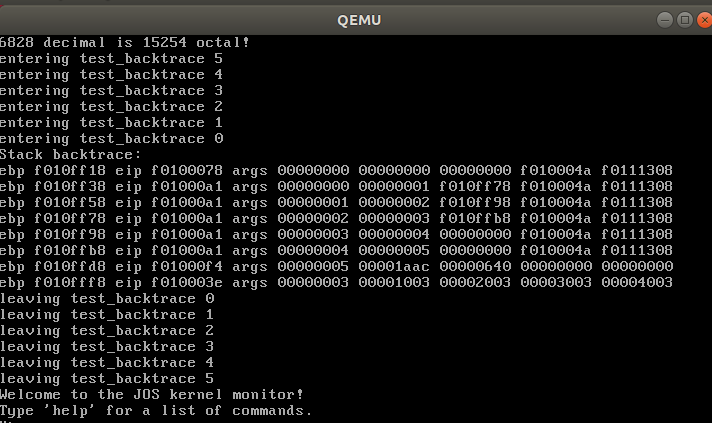


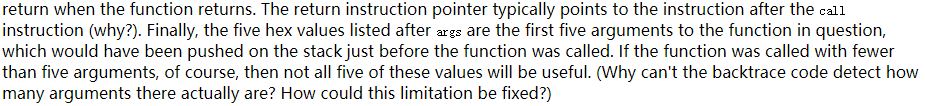


###### Exercise11：



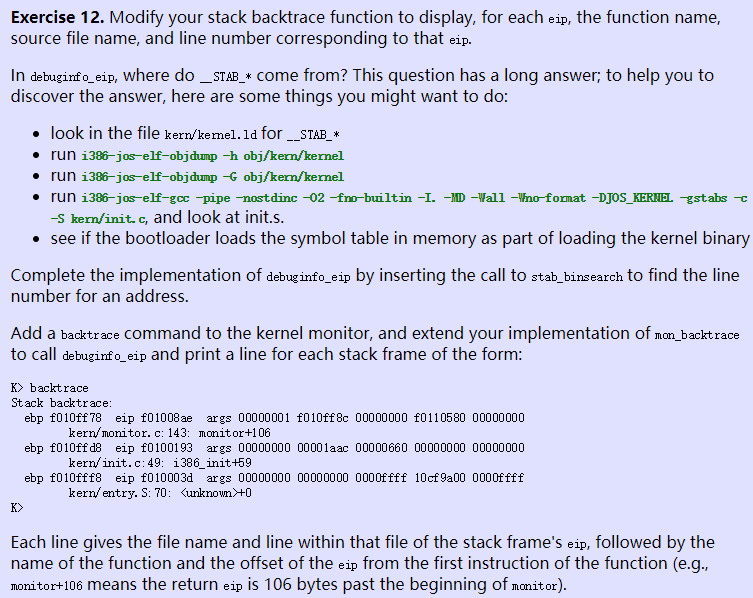
实现回溯函数。由于编译器的优化，所以需要检查mon\_backtrace()确保read\_ebp()的调用在function prologue之后。根据参数入栈顺序可以根据ebp的指针地址运算获得eip以及参数。执行make qemu之后可以看到stack traceback的输出如下。





call func在执行完func之后根据eip返回上一级函数执行。因为backtrace的时候是根据ebp的偏移取参数，而我们并不知道压入了几个参数所以不能实际检测，但是可以加入参数的个数入栈从而得到参数的个数。

###### Exercise12：

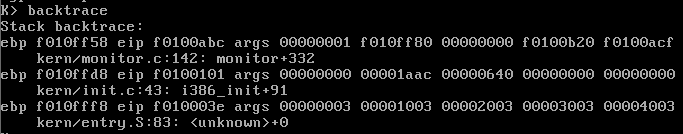


在monitor.c中添加backtrace命令，修改kdebug.c的debuginfo\_eip()和monitor.c中的mon\_backtrace()。

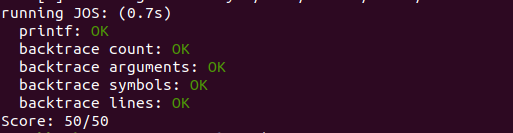
stab中存着完整的调试信息，根据stab表的内容可以知道函数行号在stabs[lline].n\_desc存储。



首先需要根据提示查找stab表，调用已经提供的stab二分查找。可以直接根据查找结果返回行号。在monitor中模仿help命令直接添加backtrace以及修改mon\_backtrace调用的函数即可。在qemu monitor中执行的结果如下。

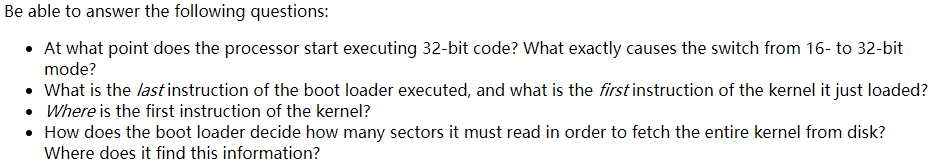


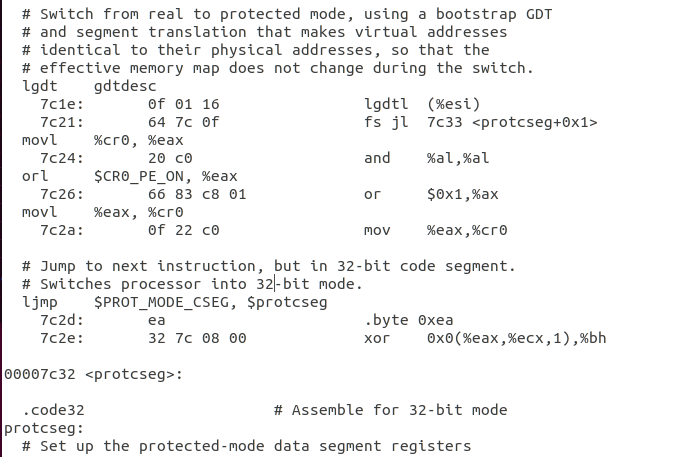
运行make grade测试结果如图



#### 3 实验问题。

Exercise3后的question



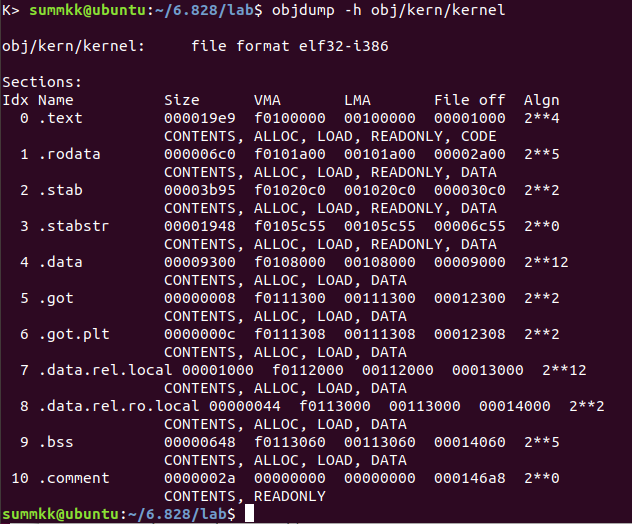


* 32位保护模式的开始在哪里？

Boot.asm中使用bootstrap GDT从真实模式切换到保护模式并进行虚拟地址的段转换，与其物理地址相同，以便有效内存映射在切换期间不会更改。

在7c2d处的跳转进行了切换。具体语句为：跳转到下一条指令，但是在32位代码段中。将处理器切换为32位模式。 修改寄存器的PE值，进入保护模式xor转移到32位

将在寄存器cr0中设置CRO\_PE后（enable保护模式）CPU将虚拟地址转换为物理地址的过程并不发生变化，直到执行ljmp，新值载入到CS段寄存器之后，处理器才读取GDT并修改虚拟地址到物理地址的转换方式。

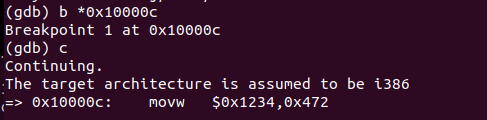


* 引导加载程序的最后一条指令是

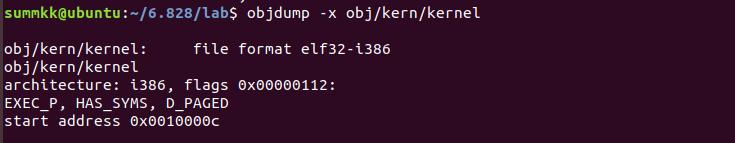
boot loader 最后一步是加载kernel，所以在 boot/main.c 中可以找到 ((void (\*)(void)) (ELFHDR->e\_entry))(); 这行代码，上面的注释 call the entry point from the ELF header 表明这是准备读取ELF头。

* 刚加载的内核的第一条指令是

通过 objdump -x obj/kern/kernel 可以查看kernel的信息，其中开头就有 start address 0x0010000c，通过 b \*0x10000c然后在 c 能得到执行的指令是 movw $0x1234,0x472，当然在 kern/entry.S 中也能找到这个指令。



* 内核的第一条指令在哪里？

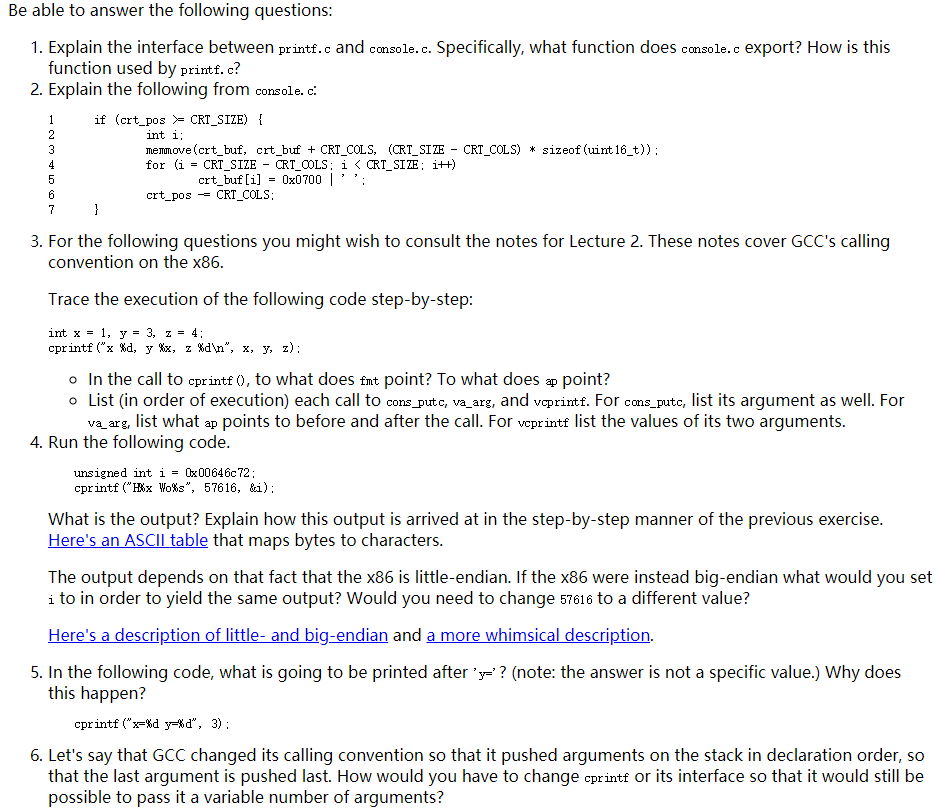


如上图所示，在0x0010000c处。

* 引导加载程序如何决定从磁盘获取整个内核必须读取多少扇区？ 它在哪里找到这些信息？

首先关于操作系统一共有多少个段，每个段又有多少个扇区的信息位于操作系统文件中的Program Header Table中。程序头指定要加载到内存中的ELF对象的哪些部分以及每个应占用的目标地址。这个表中的每个表项分别对应操作系统的一个段。并且每个表项的内容包括这个段的大小，段起始地址偏移等等信息。这个表存放在操作系统内核映像文件的ELF头部信息中。

Exerrcise8后的question



1. console.c中的接口cputchar(int c)提供给printf.c中的putch函数调用。

2. 判断语句判断CRT\_SIZE,即当前页写满，然后将1到n行的内容复制到0到n-1，将第n行覆盖为空格，情况最后一行同步crt\_pos。

3. fmt指向”x %d, y %x, z %d\n”的格式串,ap指向参数x。调用的顺序为putch(int ch, int \*cnt) -> cputchar(int c) -> cons\_putc(int c)

4.输出为He110 World。5776以16进制输出为e110，将i转换为字符串输出，根据它在内存里的存储输出rld\0。

5.输出为一个不确定的数，因为缺少一个参数，输出最后为栈内比3高一个地址的数。

6.将参数数目在eip之后入栈然后读取。

### Lab2 Memory Management

#### 1 实验目的

1. 了解计算机的内存管理技术以及相关的硬件；
2. 理解物理内存和虚拟内存的关系，理解页表的作用；
3. 在操作系统JOS上实现物理内存分配以及虚拟内存的管理。

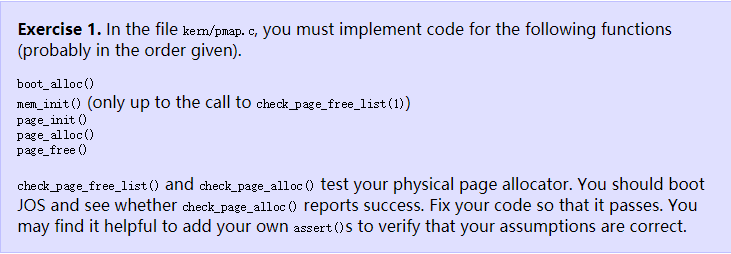
#### 2 实验内容

##### 2.1 Part 1: Physical Page Management

操作系统必须跟踪物理RAM的状态，即哪些部分是在使用的，哪些部分是空闲的。JOS以page granularity管理物理内存，使用MMU映射和保护已分配的内存。

JOS在启动之后调用i386\_init()函数，它调用cons\_init()函数设置好屏幕显示设备，mem\_init()函数初始化内存管理函数。

###### Exercise1



练习1要求填写以下五个函数完成物理页的分配，并通过测试函数。

boot\_alloc()

boot\_alloc()函数是一个简单的物理内存分配函数，仅在JOS建立它的虚拟内存系统的时候调用，在page\_free\_list建立后物理内存的分配使用page\_alloc()函数实现。boot\_alloc函数根据参数n进行分配。当n>0时，分配足够存储n bytes的连续物理页面并返回kernel虚拟地址；当n=0时，返回下一个空闲的内存并且不进行任何分配。如果内存溢出，panic结束boot\_alloc()。

nextfree作为空闲内存的next byte的虚拟地址，初始化已经由boot\_alloc()给定的代码完成了。需要分配空间并更新nextfree确保它是按照页面对齐的。

inc/tpye.h里ROUNDUP的作用是用于对齐，因此可以直接写boot\_alloc的代码。

mem\_init()

mem\_init()函数建立kernel内的两级页表结构，kern\_pgdir是页表根的虚地址。[UTOP,ULIM]之间user只读不可写，ULIM之上user不可读也不可写。

在exercise1里，只需要填写给pages分配内存的部分。用boot\_alloc给pages分配n个页面的内存然后置为0即可。

page\_init()

page\_init()函数初始化page的结构体和内存空闲链表page\_free\_list。内存的使用情况有四种：1)物理页0已被使用，用于保存实模式IDT和BIOS结构。2) [PGSIZE, npages\_basemem\*PGSIZE )是空闲的 3) IO hole [IOPHYSMEM, EXTPHYSMEM)不可以分配。 4) extended memory [EXTPHYSMEM, ...)扩展内存中有被使用的内存也有未被使用的内存。可以根据boot\_alloc(0)返回的页面地址来判断是否被使用。根据以上情况初始化pages和page\_free\_list。用到pmap.h中的宏PADDR将虚拟地址转换为对应的物理地址。

page\_alloc()

page\_alloc()函数分配物理页并返回。如果标志为alloc\_flags && ALLOC\_ZERO，则返回的页面置0。置0的时候用到函数page2kva获取物理页的内核虚拟地址。不需要增加引用计数，分配后的页面pp\_link指针设置为NULL。

page\_free()

page\_free()函数将页面加入到free list中。当pp\_ref非零或者pp\_link非空的时候panic。根据注释直接修改指针即可。

练习完成后，测试结果如图2.1所示，通过了check\_page\_alloc()。

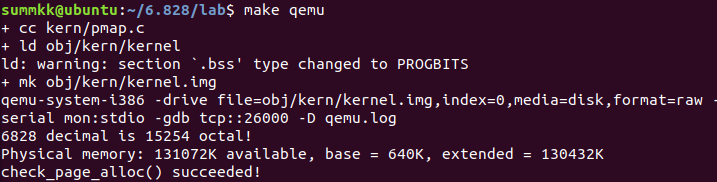


图2.1 exercise1测试结果

##### 2.2 Part 2: Virtual Memory

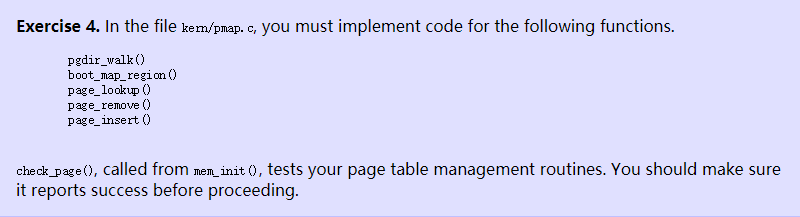
###### Exercise2

Intel 0386 Perference Manual的第五章和第六章介绍了内存管理和保护模式。讲述了段、页的翻译以及段页的保护模式。JOS使用的是二级页表。

###### Exercise3

qemu中info pg 观察页表情况，info mem观察虚拟内存的权限。在gdb中我们看到的是逻辑地址，在qemu monitor中用的是物理地址。

###### Exercise4



练习4要求实现五个函数从而实现页表管理。

pgdir\_walk()

pgdir\_walk()函数需要从二级页表中返回形参虚拟地址va的page table entry。页表的翻译过程如图2.2所示。检查虚拟地址(应该是线性地址)va 已经能够用页表(页目录+页表的体系)翻译,如果能够,则返回该地址对应的页表项的地址;如果不能,同时 create=0 的话,则返回空(NULL);但是,如果 create=1 的话,为该地址创建对应的页表(因为没有实际物理页面相对应,即使创建,返回的页表项中的地址部分也为空!),并返回 va 所对应的页表项的地址。

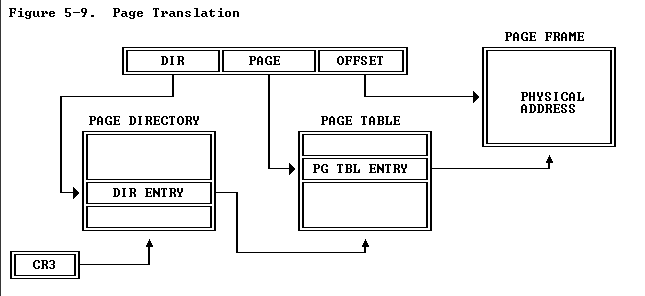


图2.2页表的寻址过程

根据定义的宏PDX和PTX取得线性地址的页目录项和页表项地址部分。PTE\_ADDR取得页表项中的物理地址部分。注意给页表合适的权限位。

boot\_map\_region()

在页表中将线性地址[va, va+size)映射到物理地址[pa, pa+size)，Size是PGSIZE的整数倍，va和pa是页对齐的。

通过前面的pgdir\_walk()函数可以实现线性地址和物理地址的转换。

page\_lookup()

在页式地址翻译机制中查找线性地址 va 所对应的物理页面,如果找到,则返回该物理页面,并将对应的页表项的地址放到 pte\_store 中；如果找不到，则返回NULL。用pa2page返回pa所在的物理页面对应的页面结构。

page\_remove()

删除线性地址va所对应的物理页面。通过函数page\_lookup()得到va对应的物理页面，再用page\_decref()函数删除页面。最后需要调用 tlb\_invalidate()更新tlb。

page\_insert()

该函数的功能是将页面管理结构 pp 所对应的物理页面分配给线性地址va并将对应的页表项的permission即低12位设置成PTE\_P&perm。如果va已经指向了一个物理页面且不是这个函数需要的物理界面，则需要调用page\_remove()删除物理页面。如果成功返回0，否则返回-E\_NO\_MEM。

练习完成后测试如图，通过了check\_page()

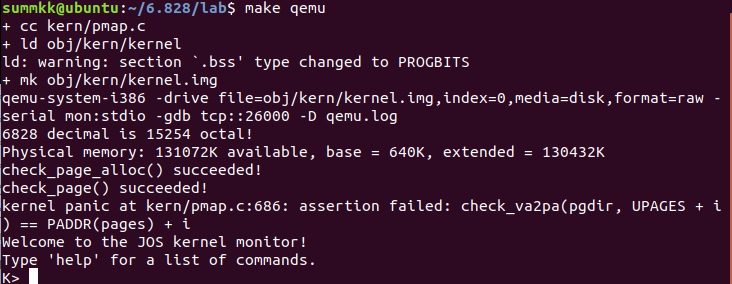


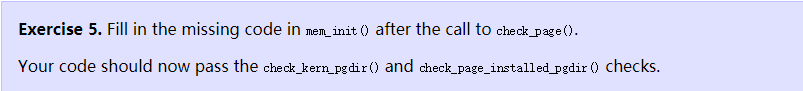
图2.3 exercise4测试结果

##### 2.3 Part 3: Kernel Address Space

JOS将处理器的32位线性地址空间分为两部分。ULIM以上的部分由内核完全控制，lab3中加载的用户环境(进程)将控制ULIM以下的部分的布局和内容。kernel保留256MB的虚拟地址空间。

由于kernel和user内存都存在于环境的地址空间中，需要用页表中的权限位控制user访问的地址空间。用户环境对ULIM以上的任何内存都没有权限，而内核将能够读写该内存。对于地址范围[UTOP，ULIM]，kernel和user都具有相同的权限：它们可以读取但不能写入此地址范围。此范围的地址用于将某些内核数据结构以只读方式暴露给user。最后，UTOP下面的地址空间供user使用;user将设置访问此内存的权限。

###### Exercise5：



Exercise5要求完成mem\_init()从而实现带有权限的地址空间设置。

根据注释，分成三部分进行映射。1)pages的线性地址为UPAGES，权限只读。2)bootstack即kernel栈的物理内存线性地址为KSTACKTOP-KSTKSIZE，权限为kernel可读写。3)映射全部的物理内存，线性地址为KERNBASE，权限为kernel可读写。直接调用之前的boot\_map\_region进行映射。

测试结果如图2.4所示

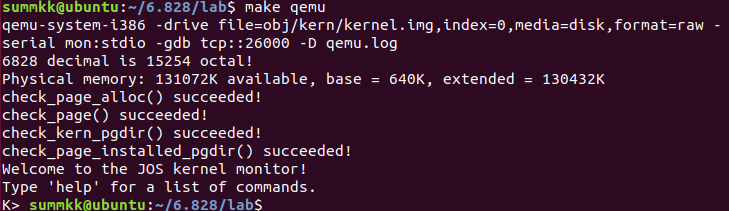


图2.4 exercise5测试

全部exercise完成后make grade如图2.5所示

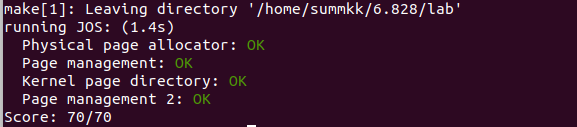
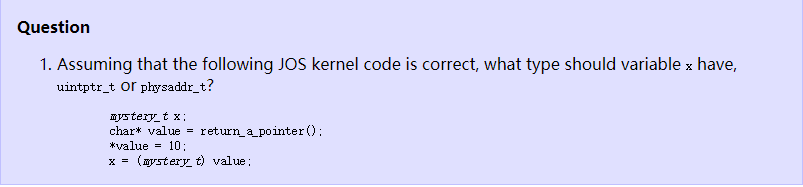
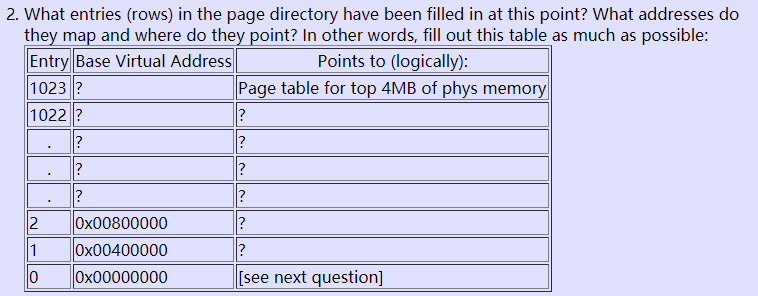


图2.5 lab2测试

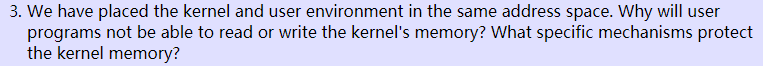
#### 3 实验问题



C中的指针都是虚拟地址，在对physaddr\_t进行强制类型转换的时候会将物理地址视为虚拟地址进行类型转换，因此，mystery\_t应该为虚拟地址。



|  |  |  |
| --- | --- | --- |
| Entry | Base Virtual Address | Points to (logically) |
| 1023 | 0xffc00000 | Page table for top 4MB of phys memory |
| … | … | … |
| 960 | 0xf0000000 | Page table for bottom 4MB of phys memory |
| 959 | 0xefc00000 | Kernel stack |
| 958 | 0xef800000 | Memory-mapped I/O |
| 957 | 0xef400000 | page table |
| 956 | 0xef000000 | page structurers |
| … | … | … |
| 0 | 0x00000000 |  |



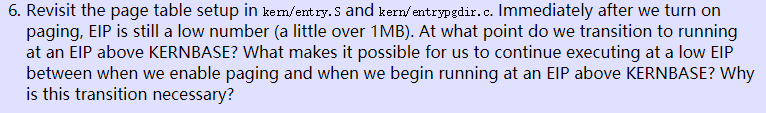
通过页表项以及页目录项的权限位(permission bits)控制用户的读写，如果没有设置PTE\_U为1则用户没有权限。

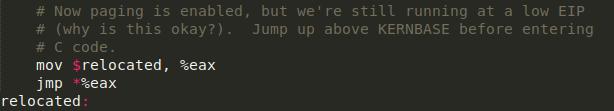


最大可以支持的物理内存为(PTSIZE/sizeof(PageInfo))\*212 = 2GB,因为我们映射pages数组的大小为PTSIZE即4MB，每一个PageInfo(8Byte)对应一个物理页即212字节。



JOS为页表需要2MB，页目录需要4KB，PageInfo需要4MB，总共约为6MB。





如上图所示，在jmp \*%eax之后完成了跳转。因为entry\_pgdir同样映射到了[0, 4M)，所以我们可以继续以低EIP执行。但是如果不进行跳转的话，kern\_pgdir被加载的时候会出现冲突。

### Lab3 User Environments

#### 1 实验目的

1. 了解计算机的进程管理过程；

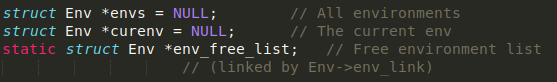
2. 了解中断的产生原因和中断和异常的处理机制；

3. 熟悉用户模式和kernel模式下的权限控制。

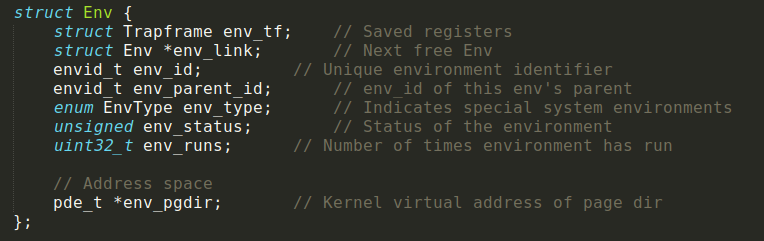
#### 2 实验内容

##### 2.1 Part 1: User Environment and Exception Handling

在inc/env.h里可以看到kernel维护三个与环境相关的全局变量。envs是所有环境，curenv是当前环境，env\_free\_list是空闲的环境链表。envs指向一个Env结构数组表示系统中所有环境。JOS支持NENV各环境(进程)同时活动。不活动的Env结构保存在env\_free\_list上。在启动运行第一个环境之前，curenv设置为NULL。

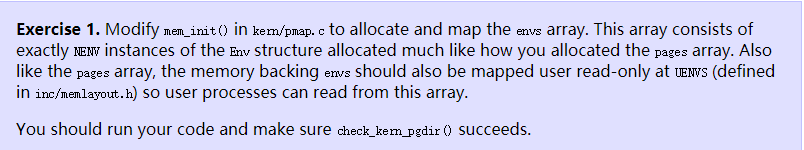


Env结构以及里面各个字段的用途如图所示。



JOS环境耦合了线程和地址空间的概念，线程由保存的寄存器(env\_tf字段)定义，地址空间由env\_pgdir指向的页表目录和页表定义。环境在运行之前内核设置CPU必须保存寄存器并选择合适的地址空间。因为JOS系统每次仅支持一个JOS环境活动，因此JOS只需要一个单独的kernel stack。

###### Exercise1

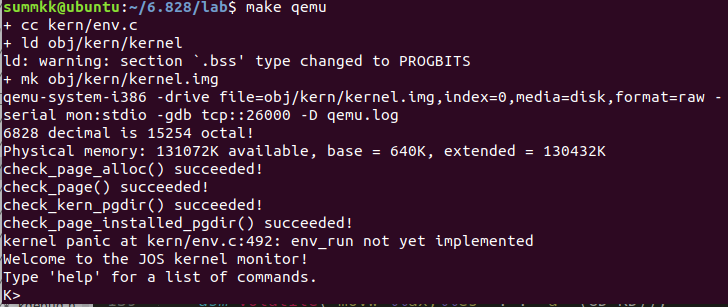


练习1要求修改mem\_init给envs数组分配空间。

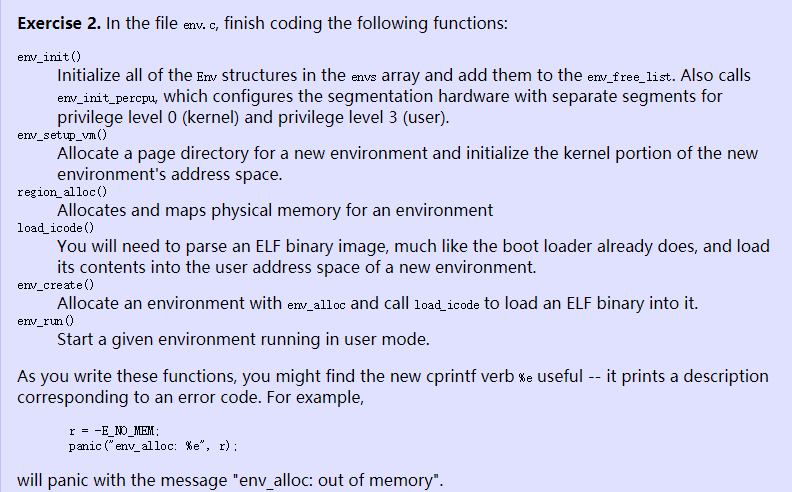
mem\_init()

mem\_init()函数给envs分配空间，和lab2中给pages分配空间思路一致。可以按照给pages分配空间的方法来编写代码。

测试结果如图，check\_kern\_pgdir()成功



###### Exercise2



在kern/init.c的i386\_init()初始化环境并允许一个二进制映像的代码。需要填写env.c文件里的函数完成环境的初始化，建立以及运行等功能。

env\_init()

env\_init()函数初始化env数组中的Env结构并将它们添加到env\_free\_list。然后调用env\_init\_percpu()函数为不同的特权级设置分段硬件。

初始化env数组只需要将数组中各个字段赋值为最开始还没有环境运行时的值，并更新env\_free\_list，因为env分配的时候返回envs[0]，所以在初始化env\_free\_list的时候可以倒序遍历envs。

env\_setup\_vm()

env\_setup\_vm()函数为新环境e分配页面目录并初始化新环境的地址空间的内核部分。但不要将任何内容映射到环境的虚拟空间地址的用户部分。如果成功则返回0，否则返回的值小于0(当页目录或者页表还没有分配的时候返回-E\_NO\_MEM)。注意对于 UTOP 以上的虚拟地址空间对于所有的进程来说都是相同的,用户进程无法访问和使用这一部分虚拟地址空间,所以只要将这部分地址空间的映射关系由kern\_pgdir中完全拷贝到 e->env\_pgdir中，然后修改UVPT部分即可。在分配完页面之后需要将该页面的逻辑地址所指向的内存清零，因为这块地址将会用来控制该环境所有内存页式分配的页目录。需要增加页引用。

region\_alloc()

region\_alloc()函数为环境分配长度为len的内存并映射物理内存。注意映射的页面不需要初始化或者置零。Pages应该是user和kernel都可读写的。需要考虑页面对齐，va向下对齐，va+len向上对齐。

load\_icode()

load\_icode()函数解析ELF二进制映像，就像引导加载程序那样，并将其内容加载到新环境的用户地址空间中。可以根据boot/main.c编写load\_icode()函数，

这个函数仅在内核初始化期间调用，之后再运行用户的第一个环境。它将ELF二进制映像中的所有可加载段加载到环境的用户内存中，从ELF程序头中读出相应的虚拟地址。它同时将标记为program header的地方清零，实际上这些地方例如.bss段并不存在于ELF文件中。

env\_create()

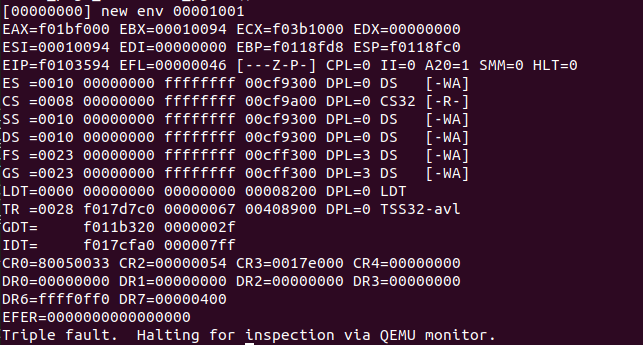
env\_create()函数使用env\_alloc分配环境并调用load\_icode以将ELF二进制文件加载到其中。这个函数仅在kernel初始化的时候调用，在运行第一个用户程序之前。新的env的父ID设为0.

env\_run()

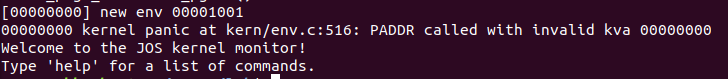
env\_run()函数启动以用户模式运行的给定环境。这个函数不会返回。

根据注释里的提示可以看到，首先需要设置当前的curenv的状态为ENV\_RUNABLE，将e设置为当前的curenv，设置它的状态为ENV\_RUNNING,更新env\_runs计数器然后调用lcr3函数切换地址空间。lcr3()这个函数，通过lcr3指令把页目录表的起始地址存入CR3寄存器。这里的拷贝到指定的虚地址处，是指用户空间的虚地址，而不是内核空间的虚地址，所以还需要用lcr3函数加载用户空间的页表目录才能将地址转换为用户空间地址。然后使用 env\_pop\_tf()恢复环境的寄存器并进入环境中的用户模式。

全部代码编写完成后执行会引发段错误如图

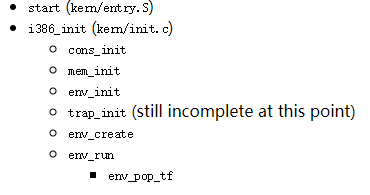


这个段错误由于curenv没有判断是否为null，即第一次运行env\_run时，curenv为null，如果不进行判断会引发段错误。前面给出的代码是修改后的代码。

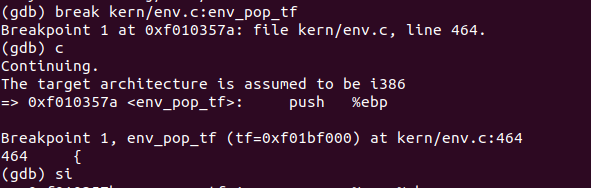


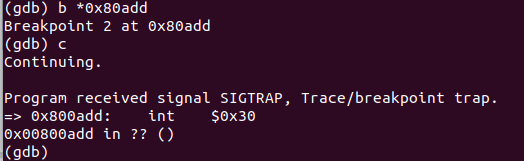
然后出现envs初始化出错误如图是因为前面的函数env\_init出现错误，用i作为下标但是用i>0作为判断条件修改还执行成功引发报错。

这个是一个user code在运行的时候的调用图，如果执行正确的话系统应该进入用户环境并执行hello的二进制文件。



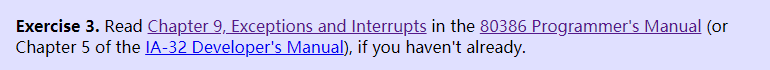
hello中用int指令进行系统调用，但我们还没有处理中断，因此会引发段错误，使用gdb来检查我们是否正在进入用户模式。找到obj/hello.asm中sys\_cputs函数的位置在0x800ac6，指令int $0x30在0x80add处，设置断点发生中断引发的段错误。Exercise3完成。





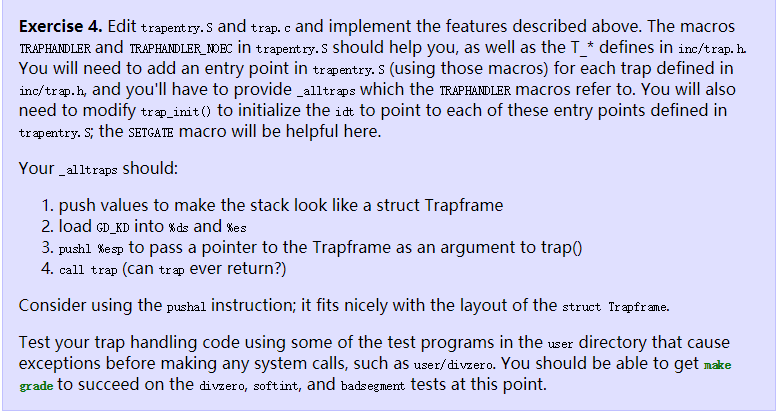
接下来需要实现基本异常和系统调用处理，因为处理器一旦进入了用户模式就无法退出。

###### Exercise3

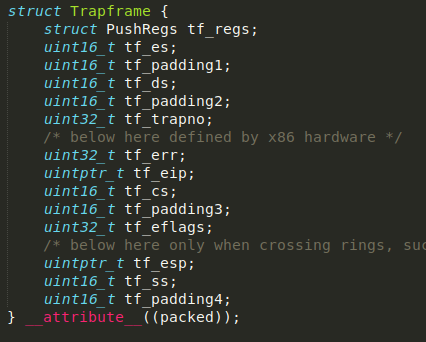


了解关于异常和中断的基本内容。

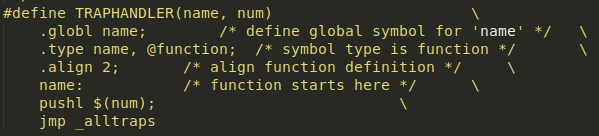
###### Exercise4



练习4需要实现IDT表的创建，并在栈上新建一个Trapframe，以便后面调用其地址进行异常处理。按照exercise描述的提醒，找到inc/trap.h里的Trapframe，结构如下。



宏TRAPHANDLER和TRAPHANDLER\_NOEC的区别在于是否压入error code。TRAPHANDLER内容如下，它进行了一部分数据的入栈。

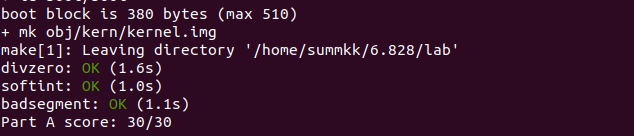


在trapentry.S中调用TRAPHANDLER的宏进行函数声明。

在\_alltraps阶段需要根据TRAPHANDLER已经压入的值和Trapframe需要压入的寄存器值进行push，然后更改数据段为内核的数据段。再调用trap进行异常处理。

在trap.c里需要建立IDT表，根据trapentry声明的函数调用宏SETGATE建立中断向量表。SETGATE宏中的各部分意义在inc/mmu.h文件里有定义。在C文件里需要再次声明函数。

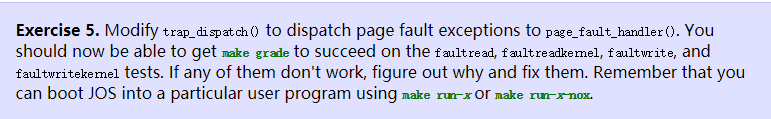
对partA部分进行make grade测试，结果如图

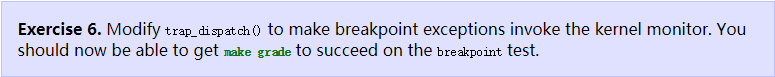


##### 2.2 Part 2: Page Fault, Breakpoints Exception, and System Call

当处理器发生Page fault页面错误时，它将导致故障的线性(虚拟)地址存储在特殊处理器控制寄存器CR2中。在trap.c中，page\_fault\_handler()用于处理页面错误。

###### Exercise5、6：

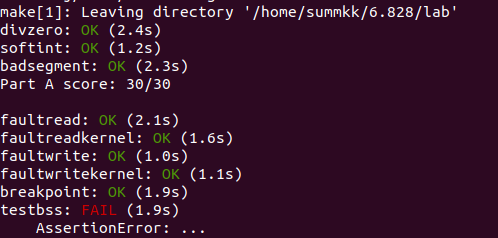




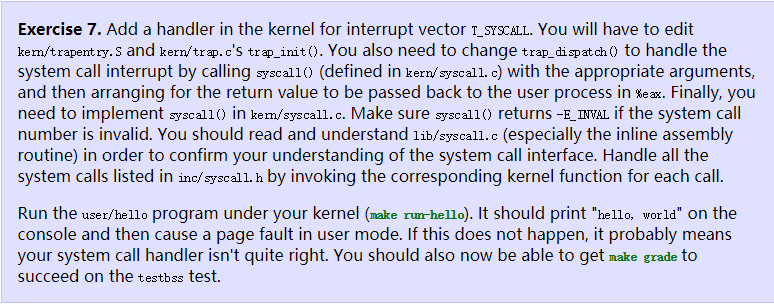
修改trap\_dispatch()将页面错误分派给page\_fault\_handler()。trap\_dispatch()函数通过Trapframe里的数据判断错误类型并且调用对应的处理函数。

因为exercise5和6都是修改trap\_dispatch()函数调用对应的处理函数，处理方式差不多，放在一起写之后进行测试。Exercise6需要注意断点异常的特权级为用户级，即SETGATE的最后一个参数为3，否则会引发保护异常。

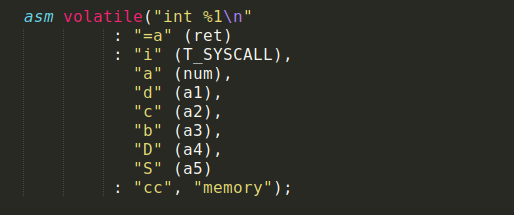
make grade测试通过断点之前的测试，结果如图



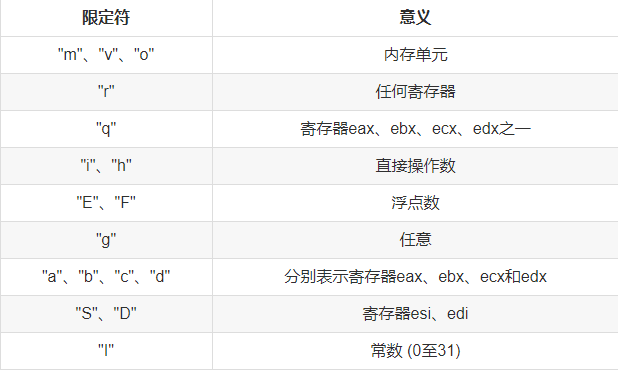
###### Exercise7：



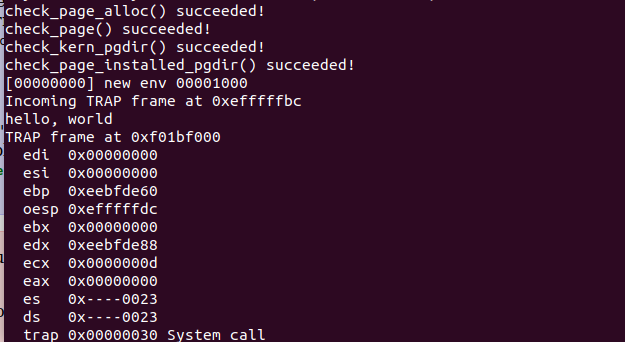
在lib/syscall.c文件里的汇编语句是系统调用，并且它指出了我们在进行系统调用的时候需要的参数顺序。

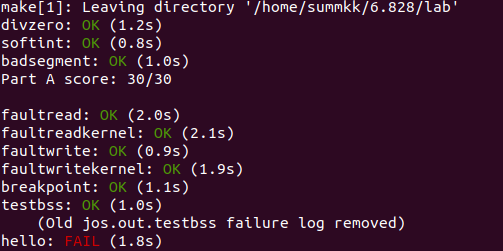


参考内联汇编的语法可以知道压入的参数然后在trap\_dispatch()函数中加入对应的调用。

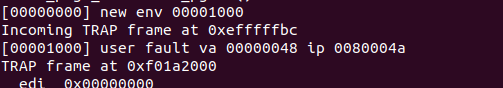
再根据syscall的类型选择需要调用的处理函数完成kern/syscall.c函数。

测试运行make run-hello打印出hello,world

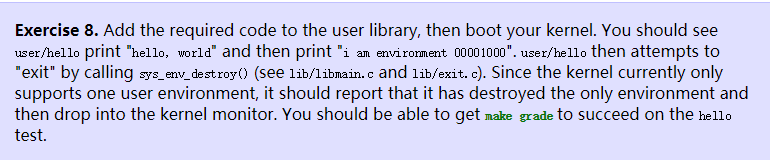




如果没有inc/syscall.c会引发user fault，可以看出jos系统调用是用户进程使用inc目录下的接口，然后在lib/syscall中进行参数的传递，引发int $0x30中断，trap.c调用对应的中断处理函数，由kern/syscall.c进行中断处理。



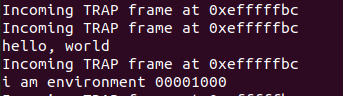
###### Exercise8：



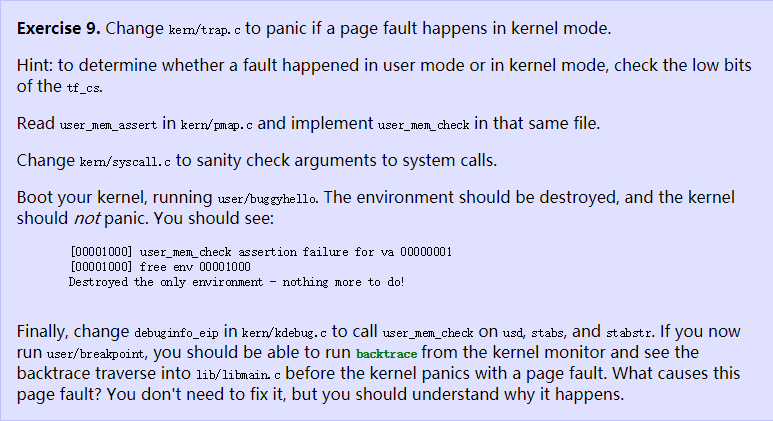
在lic/libmain.c中设置当前活动的进程即可



设置之后make run-hello输出如下信息，和exercise8中一致。

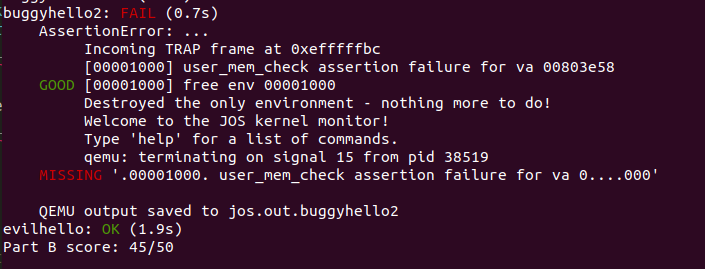


###### Exercise9：

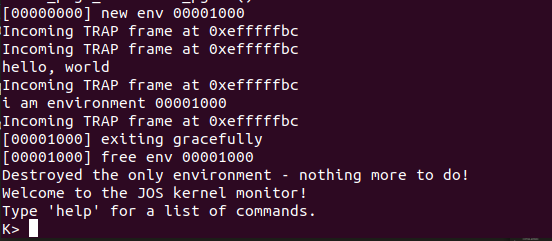


为了实现在kernel mode的时候发生page fault产生panic，根据tf\_cs的地位判断现在的mode。

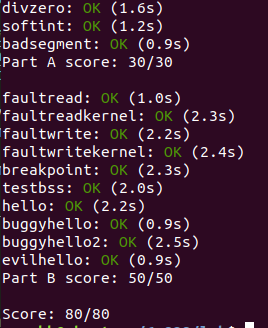
填写pmap.c里的内存检查函数。需要检查内存[va,va+len]是否有权限，va和len不一定是页面对齐的。用户可以访问的虚拟地址要求地址低于ULIM并且在页表中有相应的权限。最开始用page\_lookup因为地址对齐的问题导致无法通过测试如图。修改用pgdir\_walk之后完成。



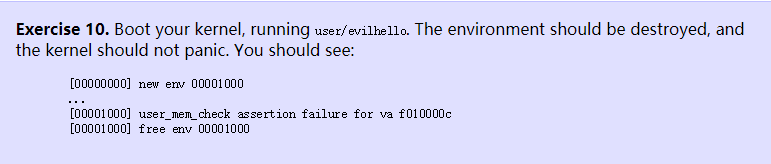
在kernel/syscall.c中的sys\_cputs()，kern/kdebug.c 中的 debuginfo\_eip()中加入user\_mem\_check()函数进行内存检查。完成后执行make run-buggyhello产生输出如图



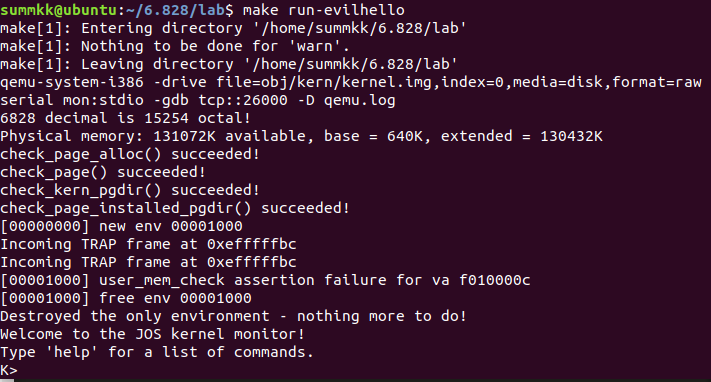
make grade测试结果如图



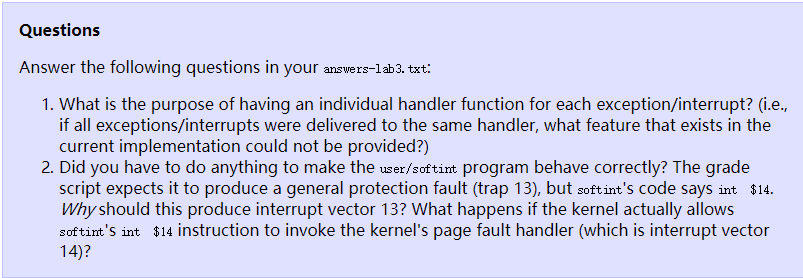
###### Exercise10：



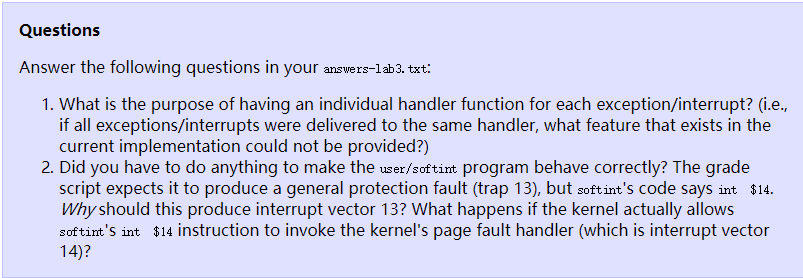
make run-evilhello的测试结果如下，envrionmen destroyed成功，并没有发生panic。



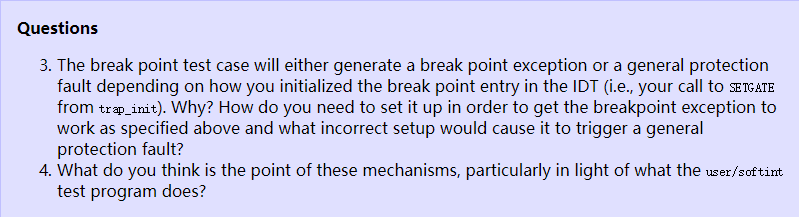
#### 3 实验问题



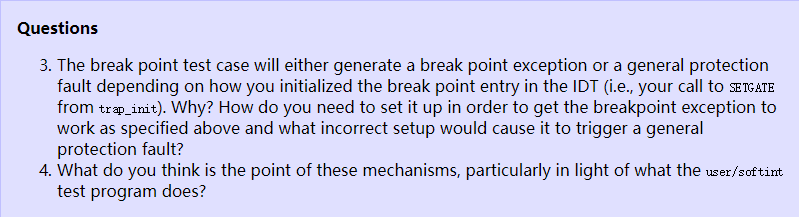
因为每个中断和异常的处理方式不同，特权级也不同，不能用同一个handler进行处理。



softint期望产生一个缺页异常14，但是实际产生的是通用保护异常13。因为目前系统在用户态，权限为3，而int系统调用的权限为0。因此会产生通用并保护异常。



因为断点异常是一个用户级的异常，因此如果特权级置为0会产生通用保护异常，用户没有权限访问异常处理代码。如果特权级置为3则会继续执行，产生断点异常。



缺页异常为了在缺少页面的时候可以新建页面或者调入页面，断点异常方便用户进行程序的调试，异常和中断机制是为了可以让机器从用户态转换回内核态进行处理，防止用户意外访问内核代码导致OS异常，保护内核。

### Lab4 Preemptive Multitasking

#### 1 实验目的

1. 了解多处理器的工作方式以及进程调度；

2. 理解用户模式下发生页面错误之后的处理方式以及用户权限和kernel权限的转换；

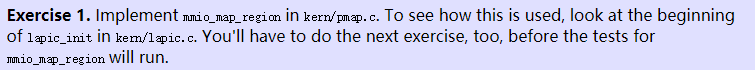
3. 知道如何进行进程间通信。

4. 知道进程创建子进程的方式以及父子进程的关系。

#### 2 实验内容

##### 2.1 Part 1: Multiprocessor Support and Cooperative Multitasking

###### Exercise1



###### mmio\_map\_region()

为MMIO预留size的空间并将[pa,pa+size)映射到这个位置。返回预留的空间的base地址。注意size不一定是PGSIZE对齐的。

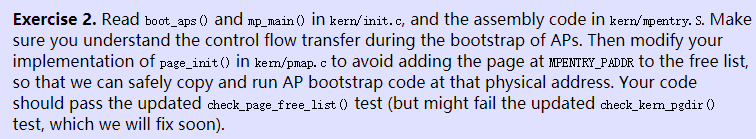
因为这个需要进行映射的地址不是常规的DRAM，除了PTE\_W之外还需要设置页表的权限位为PTE\_PCD|PTE\_PWT即cache-disable和write-through位。需要将size向上按照PGSIZE对齐，处理以避免它溢出MMIOLIM。像之前一样直接用boot\_map\_region()函数进行映射即可。更改base的值使每次映射到新的界面。

在启动 APs 之前，BSP 需要先搜集多处理器系统的信息，例如 CPU 的总数，CPU 各自的 APIC ID，LAPIC 单元的 MMIO 地址。kern/mpconfig.c 中的 mp\_init() 函数通过阅读 BIOS 区域内存中的 MP 配置表来获取这些信息。

boot\_aps() 函数驱动了 AP 的引导。APs 从实模式开始，如同 boot/boot.S 中 bootloader 的启动过程。因此 boot\_aps() 将 AP 的入口代码 (kern/mpentry.S) 拷贝到实模式可以寻址的内存区域 (0x7000, MPENTRY\_PADDR)。

此后，boot\_aps() 通过发送 STARTUP 这个跨处理器中断到各 LAPIC 单元的方式，逐个激活 APs。激活方式为：初始化 AP 的 CS:IP 值使其从入口代码执行。通过一些简单的设置，AP 开启分页进入保护模式，然后调用 C 语言编写的 mp\_main()。boot\_aps() 等待 AP 发送 CPU\_STARTED 信号，然后再唤醒下一个。

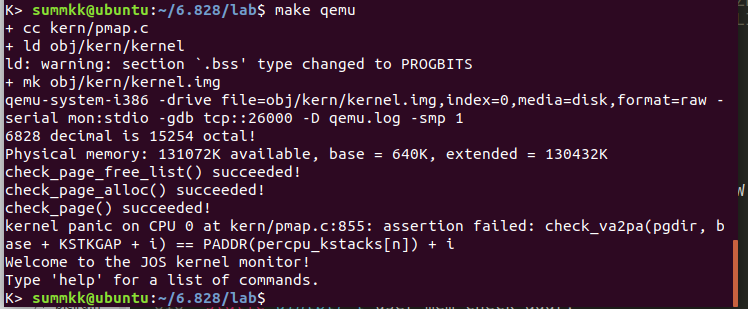
###### Exercise2



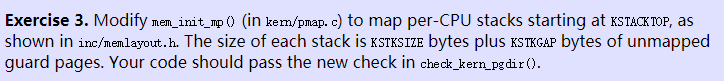
了解boot\_aps()和mp\_main()之后修改page\_init()函数避免添加MPENTRY\_PADDR的页面到空闲页表。在这个物理地址处需要进行AP bootstrap的拷贝和运行。

可以在原来的page\_init()函数最后修改page\_free\_list，去掉MPENTRY\_PADDR对应的页面即可。

完成exercise2后check\_page()成功。



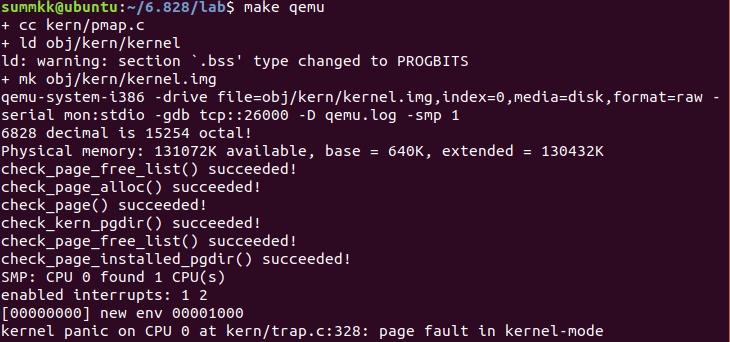
###### Exercise3：



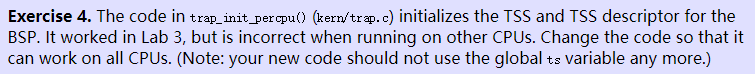
将每个CPU的栈映射到KSTACKTOP，具体的内存结构在inc/memlayout.h里。CPU的kernel 栈从虚拟地址kstacktop\_i = KSTACKTOP - i \* (KSTKSIZE + KSTKGAP)开始向下增长。注意如果堆栈溢出之后，它将会引发fault，这就是保护页。kernel的权限为RW，user没有权限。

同样可以通过boot\_map\_region()函数进行内存的映射，每次栈的地址向下增长即可。

完成exercise3之后check\_kern\_pgdir()成功。

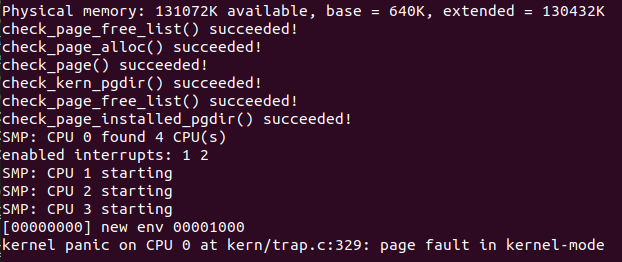


###### Exercise4：



初始化每个CPU的进程，根据它已经给了的用ts初始化的过程进行修改。首先建立TSS然后进入kernel，初始化gdt的TSS槽，加载TSS selector，因为低位比较特殊，所以低位需要置为0，最后加载IDT。

编写完成后出现page fault



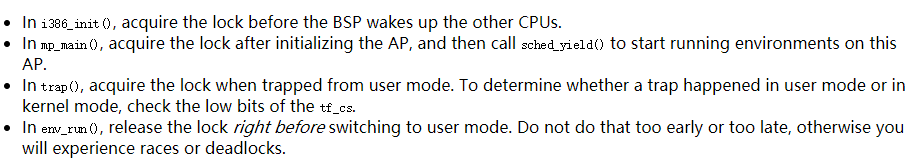
注意：lab3里env\_setup\_vm()在低于UTOP的地方为0，不能直接用memcpy复制kern\_pgdir，修改完成后正确输出如下



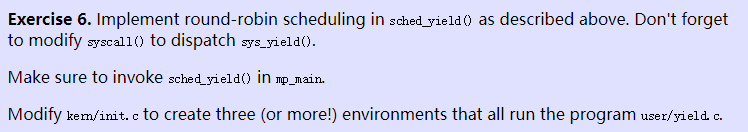
###### Exercise5



根据MIT文档中的这一段话，直接在对应位置添加lock\_kernel()或者unlock\_kernel()即可。



###### Exercise6

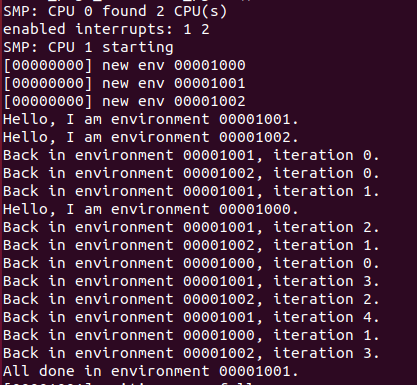


在shed\_yield()中实现round-robin调度，在mp\_main()中调用sched\_yield()，修改kernel/init.c建立一个或多个环境执行user/yield.c程序。

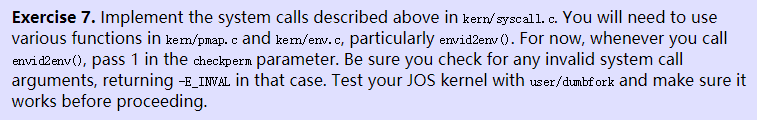
round-robin调度顺序遍历所有的用户环境，从当前环境开始跑，直到找到第一个状态为ENV\_RUNNABLE的进程，并修改它的状态到运行。如果没有其他的envs没有可以运行的，可以继续执行当前的env。不要选择正在其他CPU上执行的程序，如果没有可以执行的程序则CPU进入sched\_halt()函数暂停工作。

根据提示可以直接写代码，如果当前curenv为null的话，找到它的env\_id作为遍历的开始，否则以0为开始进行遍历。遍历结束后没有找到可以运行的程序则继续运行当前的curenv(如果存在)否则暂停。

测试结果如下



###### Exercise7



sys\_exofork()

这个系统调用创建一个几乎为空白的新环境，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，则返回负的错误代码）。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，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）。

因为需要创建的新环境机会为空白的因此直接调用env\_alloc()后设置一下新环境的状态以及寄存器就可以返回了。注意因为它需要在子节点中返回0，而且是通过父节点返回再进行子结点返回，因此需要将新环境的寄存器eax设为0。

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。

调用kern/env.c中的envid2env函数将envid转换为Env结构，它的第三个参数设为1，用来检查当前环境的权限是否有权限设置env状态。根据提示可以直接编写代码。

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。页面内容设置为0。注意权限位的设置。

这个系统调用主要是page\_alloc()和page\_insert()函数的封装，需要处理这两个函数失败的情况并报错即可，如果page\_insert()失败，则。

sys\_page\_map()

将页面映射（不是页面内容）从一个环境的地址空间复制到另一个环境，设置内存共享管理，以便新旧映射都指向物理内存的同一页面。

这个系统调用主要是page\_lookup()和page\_insert()的封装，需要检查页面的权限。

sys\_page\_unmap()

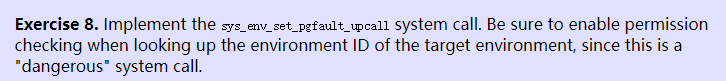
取消映射在给定环境中给定虚拟地址映射的页面。

这个函数主要是page\_remove()的封装。和前面一样检查越界和权限问题即可。

##### 2.2 Part 2: Copy-on-Write Fork

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。

###### Exercise8

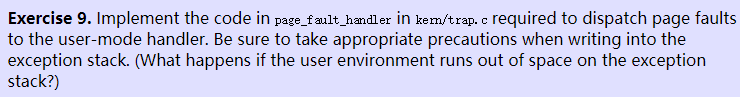


sys\_env\_set\_pgfault\_upcall()

将envid转换为结构Env然后将Env结构中用于保存page fault的处理函数的指针设置为func即可。

我们将处于页错误处理的用户进程状态称为 trap-time。在 lib/pfentry.S 中的\_pgfault\_upcall 汇编函数,首先是调用具体的页错误处理函数,然后利用在 trap.c 中压入的 UTrapframe 结构变换到调用缺页处理进程的上下文。首先变换到调用缺页处理的堆栈,将保存的缺页处理进程的 eip 压入堆栈,然后空一个字出来保存后面的堆栈信息以免覆盖堆栈信息。剩下的就是换回到原来的堆栈,返回到调用缺页处理进程了。

###### Exercise9

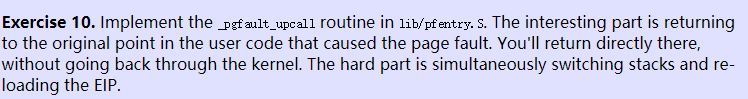


当kernel处理page\_fault时，并不是在kernel栈或者用户栈处理，而是启用了一个新的栈，用户异常栈。并且使用一个新的数据结构UTrapframe来保存触发page\_fault的进程信息。所以现在用户空间的缺页流程是这样的的，用户空间发生缺页，产生中断，陷入到内核中，分发到page\_fault\_handler中，在用户异常栈保存错误进程的信息，以及错误地址(保存到UTrapframe中)，切换到用户异常栈，然后调用用户自定义的pgfault\_upcall，最后再切换到原来错误的地方继续运行。

栈的切换分成两种情况。1)用户进程发生page\_fault。用户栈 -> kernel栈 -> 用户异常栈.2)在page\_fault处理时又发生page\_fault。虽然已经在用户异常栈了，但还是会继续陷入到内核中，重走一遍上面的流程，这里需要注意，压入UTrapframe时，需要空4个字节。所以栈切换顺序为，用户异常栈 -> kernel栈 -> 用户异常栈。

根据上面的情况编写代码即可。

###### Exercise10



汇编代码调用了用户定义的page\_handler()函数然后切换堆栈从而从内核态返回到用户态。

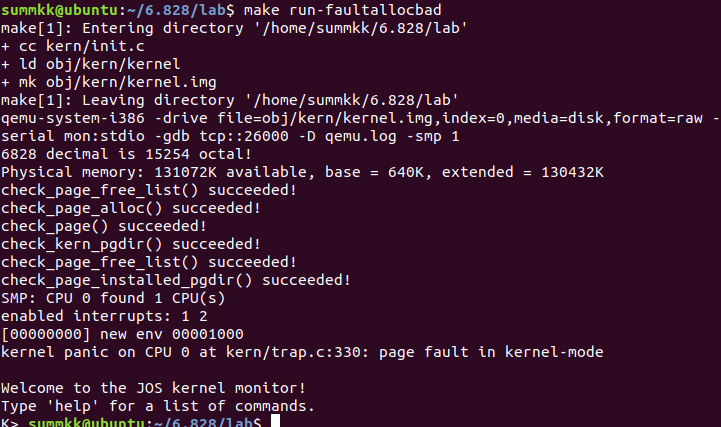
###### Exercise11



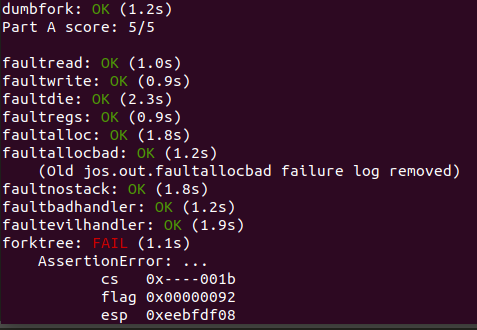
为用户异常栈分配页，注意边界条件的处理。

在完成11之后对它们进行测试。

user\_mem\_check()没有检查pgdir\_walk返回目录项为空的情况导致下图的错误。



修改后通过part B前面的检查



与dumbfork（）一样，fork（）应创建一个新环境，然后扫描父环境的整个地址空间并在子项中设置相应的页面映射。关键的区别在于，当dumbfork（）复制页面时，fork（）最初只会复制页面映射。 fork（）仅在其中一个环境尝试编写时才复制每个页面。

fork（）的基本控制流程如下：

1.父级使用您在上面实现的set\_pgfault\_handler（）函数将pgfault（）安装为C-level page fault handler。

2.父级调用sys\_exofork（）来创建子环境。

3.对于UTOP下面的地址空间中的每个可写或写时复制页面，父进程调用duppage，它应该将页面copy-on-write映射到子进程的地址空间，然后重新映射页面copy-on-write in它自己的地址空间。 duppage设置两个PTE，使页面不可写，并在“avail”字段中包含PTE\_COW，以区分写入时复制页面和真正的只读页面。但是，异常堆栈不会以这种方式重新映射。相反，您需要在子代中为异常堆栈分配一个新页面。fork（）还需要处理存在的页面，但不能写入或写入时复制。

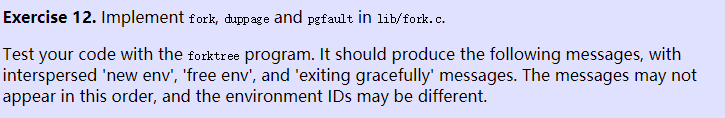
4.父级将子级的用户页面错误入口点设置为自己的。

5.孩子现在已准备好运行，因此父母将其标记为可运行。

每当其中一个环境写入尚未写入的写时复制页面时，就会出现页面错误。这是用户页面错误处理程序的控制流程：

1）内核将页面错误传播到\_pgfault\_upcall，后者调用fork（）的pgfault（）处理程序。2）pgfault（）检查故障是否为写入（检查错误代码中的FEC\_WR），并且页面的PTE标记为PTE\_COW。如果没有，panic错误。3）pgfault（）分配映射到临时位置的新页面，并将错误页面的内容复制到其中。然后，故障处理程序将新页面映射到具有读/写权限的适当地址，而不是旧的只读映射。

###### Exercise12

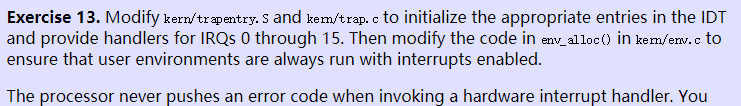


根据前面文档的讲述完成代码，测试结果如图，Part B通过。



##### 2.3 Part 3: Preemptive Multitasking and Inter-Process communication(IPC)

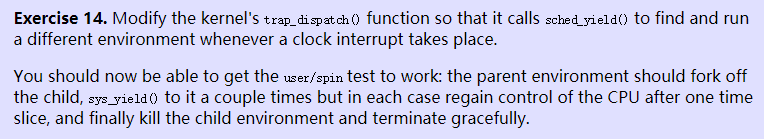
###### Exercises13



修改trap.c和trapentry.S设置IDT，注意它们都不需要入栈error code。和之前的修改trap.c和trapentry.S几乎一致。

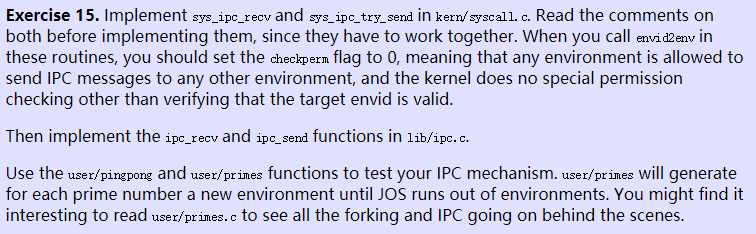
注意istrap必须设为0，否则会引发assert。

Exercise14



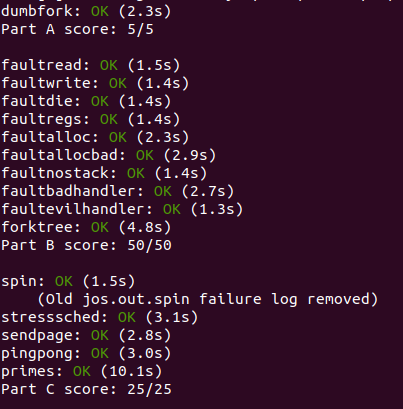
出现时钟中断的时候进行进程调度，判断中断类型然后调用对应的sched\_yield()函数进行进程调度。

###### Exercise15

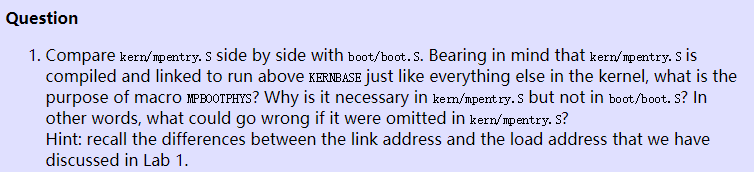


sys\_ipc\_recv()，等待接受信息，只需要改变进程状态，直接调度就会阻塞了。sys\_ipc\_try\_send（）需要注意共享内存时，不能直接使用sys\_page\_map，因为sys\_page\_map查找env时会检查权限。

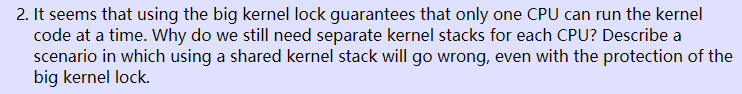
make grade通过检查。



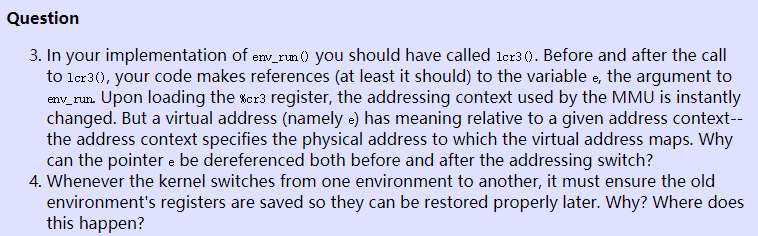
#### 3 实验问题



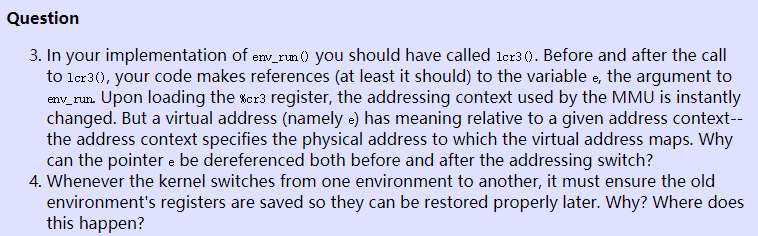
mpentry.S与boot.S的区别在于它不需要驱动A20，并且它通过MPBOOTPHYS宏的转换完成虚拟地址和实际地址的转换。因为此时分页机制和保护模式未开启，因此需要转换为实际的物理地址。



如果多个CPU同时进入中断而栈访问不互斥会导致压入的寄存器信息混乱出现错误。



env\_setup\_vm()函数以kernel的页目录作为模板创建用户的页目录，因此它们页表项是相同的。



Stack中的trapframe保留了寄存器状态。