華中科技大學

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

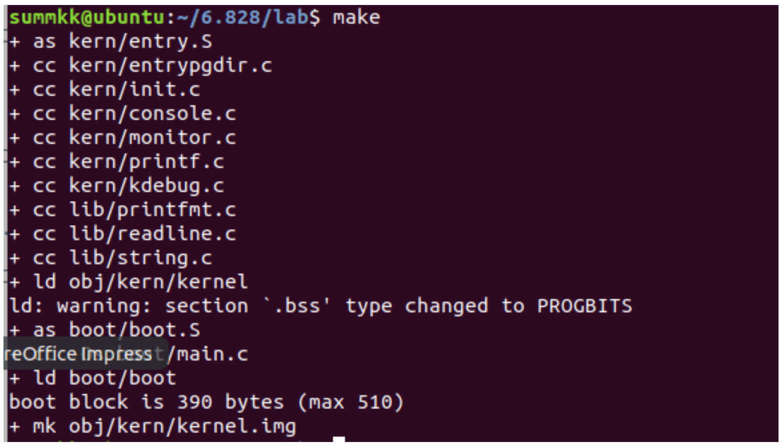
1 实验目的

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。



```
summkk@ubuntu:~/6.828/lab$ make
+ as kern/entry.S
+ cc kern/entrypgdir.c
+ cc kern/init.c
+ cc kern/console.c
+ cc kern/monitor.c
+ cc kern/printf.c
+ cc kern/kdebug.c
+ cc lib/printfmt.c
+ cc lib/readline.c
+ cc lib/string.c
+ ld obj/kern/kernel
ld: warning: section `.bss' type changed to PROGBITS
+ as boot/boot.S
reOffice Impress t/main.c
+ ld boot/boot
boot block is 390 bytes (max 510)
+ mk obj/kern/kernel.img
```

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。

```
+------------------+  <- 0xFFFFFFFF (4GB)
|      32-bit      |
|  memory mapped   |
|     devices      |
|                  |
/\/\/\/\/\/\/\/\/\/\

/\/\/\/\/\/\/\/\/\/\
|                  |
|      Unused      |
|                  |
+------------------+  <- depends on amount of RAM
|                  |
|                  |
| Extended Memory  |
|                  |
|                  |
+------------------+  <- 0x00100000 (1MB)
|     BIOS ROM     |
+------------------+  <- 0x000F0000 (960KB)
|  16-bit devices, |
|  expansion ROMs  |
+------------------+  <- 0x000C0000 (768KB)
|   VGA Display    |
+------------------+  <- 0x000A0000 (640KB)
|                  |
|    Low Memory    |
|                  |
+------------------+  <- 0x00000000
```

图 1.2 PC 物理地址空间

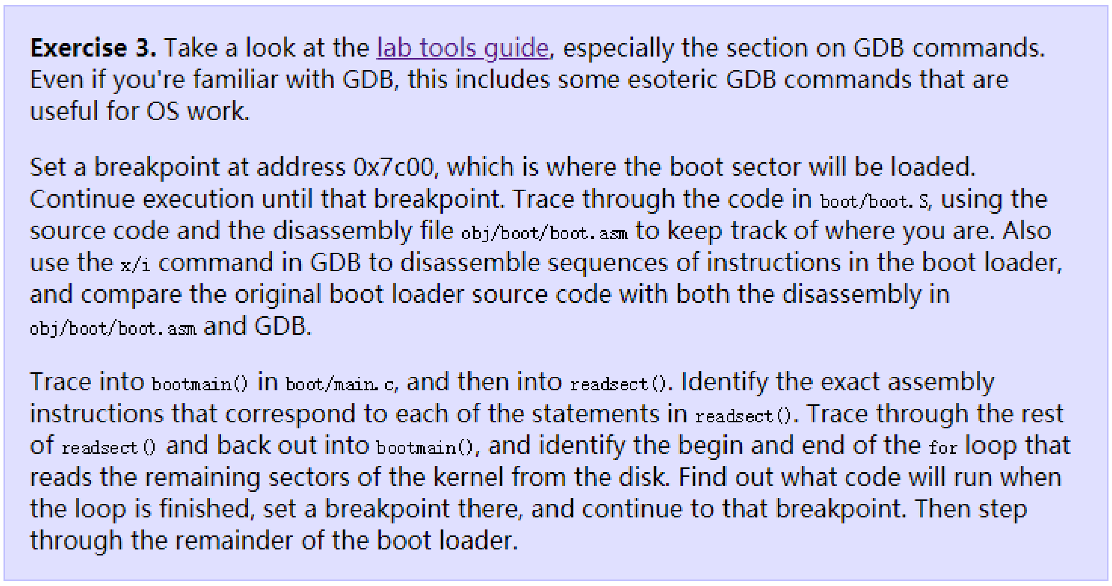
当处理器在上电启动的时候，BIOS 首先控制机器，这时候没有其他程序运行，BIOS 运行时设置一个中断描述符表(IDT)并初始化多个硬件设备。初始化 PCI 总线以及所有重要设备之后，BIOS 会搜索可引导设备(bootable device)。当找到可引导磁盘之后，BIOS 从磁盘读取引导加载程序(boot loader)，并将控制权转移至 boot loader。

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：

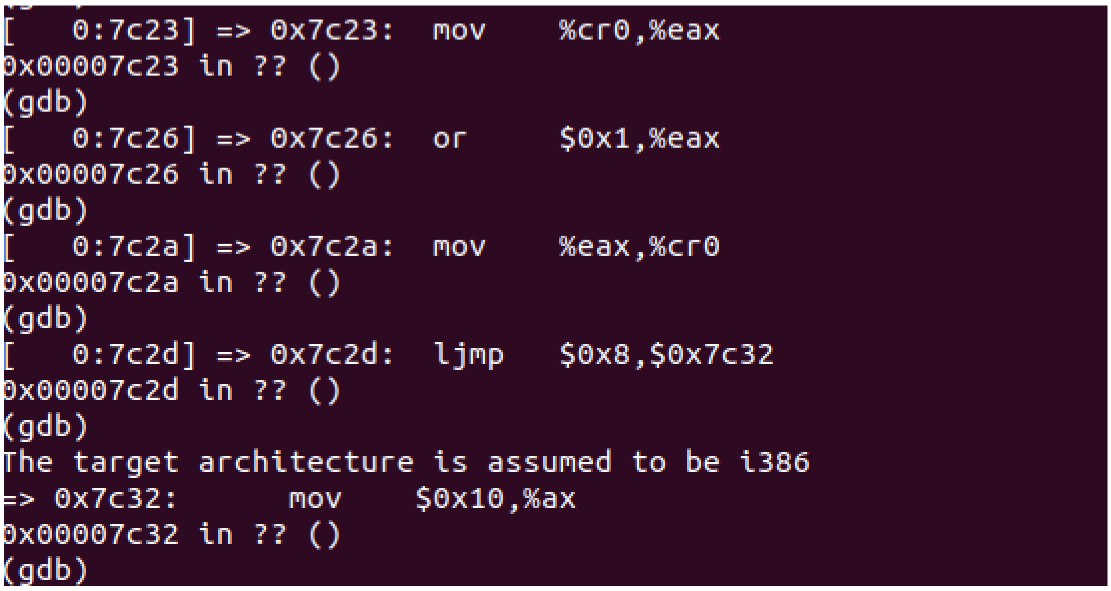


Exercise 3. Take a look at the lab tools guide, especially the section on GDB commands. Even if you're familiar with GDB, this includes some esoteric GDB commands that are useful for OS work.

Set a breakpoint at address 0x7c00, which is where the boot sector will be loaded. Continue execution until that breakpoint. Trace through the code in `boot/boot.S`, using the source code and the disassembly file `obj/boot/boot.asm` to keep track of where you are. Also use the `x/i` command in GDB to disassemble sequences of instructions in the boot loader, and compare the original boot loader source code with both the disassembly in `obj/boot/boot.asm` and GDB.

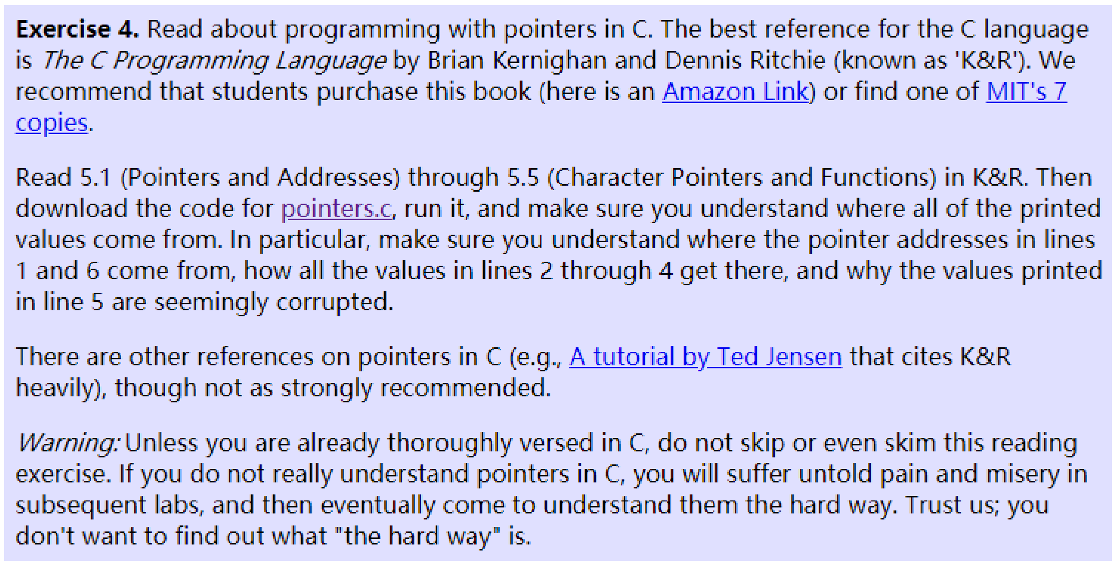
Trace into `bootmain()` in `boot/main.c`, and then into `readsect()`. Identify the exact assembly instructions that correspond to each of the statements in `readsect()`. Trace through the rest of `readsect()` and back out into `bootmain()`, and identify the begin and end of the `for` loop that reads the remaining sectors of the kernel from the disk. Find out what code will run when the loop is finished, set a breakpoint there, and continue to that breakpoint. Then step through the remainder of the boot loader.

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。



```
[   0:7c23] => 0x7c23:  mov    %cr0,%eax
0x00007c23 in ?? ()
(gdb)
[   0:7c26] => 0x7c26:  or     $0x1,%eax
0x00007c26 in ?? ()
(gdb)
[   0:7c2a] => 0x7c2a:  mov    %eax,%cr0
0x00007c2a in ?? ()
(gdb)
[   0:7c2d] => 0x7c2d:  ljmp   $0x8,$0x7c32
0x00007c2d in ?? ()
(gdb)
The target architecture is assumed to be i386
=> 0x7c32:      mov    $0x10,%ax
0x00007c32 in ?? ()
(gdb)
```

Exercise4：



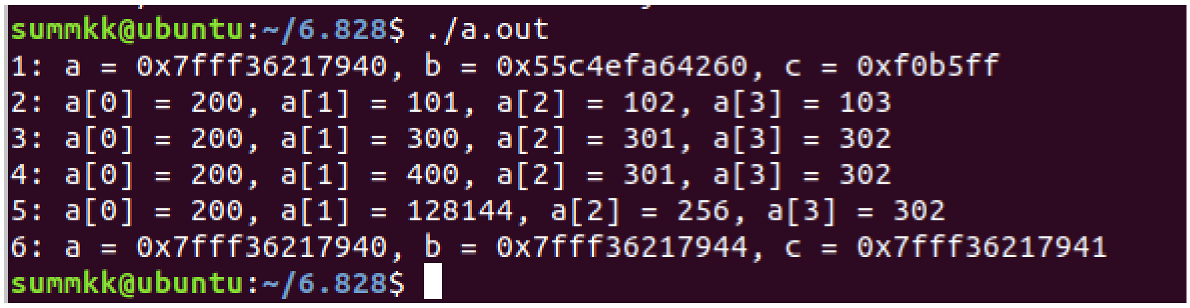
Exercise 4. Read about programming with pointers in C. The best reference for the C language is *The C Programming Language* by Brian Kernighan and Dennis Ritchie (known as 'K&R'). We recommend that students purchase this book (here is an Amazon Link) or find one of MIT's 7 copies.

Read 5.1 (Pointers and Addresses) through 5.5 (Character Pointers and Functions) in K&R. Then download the code for pointers.c, run it, and make sure you understand where all of the printed values come from. In particular, make sure you understand where the pointer addresses in lines 1 and 6 come from, how all the values in lines 2 through 4 get there, and why the values printed in line 5 are seemingly corrupted.

There are other references on pointers in C (e.g., A tutorial by Ted Jensen that cites K&R heavily), though not as strongly recommended.

Warning: Unless you are already thoroughly versed in C, do not skip or even skim this reading exercise. If you do not really understand pointers in C, you will suffer untold pain and misery in subsequent labs, and then eventually come to understand them the hard way. Trust us; you don't want to find out what "the hard way" is.

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。



```
summkk@ubuntu:~/6.828$ ./a.out
1: a = 0x7fff36217940, b = 0x55c4efa64260, c = 0xf0b5ff
2: a[0] = 200, a[1] = 101, a[2] = 102, a[3] = 103
3: a[0] = 200, a[1] = 300, a[2] = 301, a[3] = 302
4: a[0] = 200, a[1] = 400, a[2] = 301, a[3] = 302
5: a[0] = 200, a[1] = 128144, a[2] = 256, a[3] = 302
6: a = 0x7fff36217940, b = 0x7fff36217944, c = 0x7fff36217941
summkk@ubuntu:~/6.828$
```

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。）

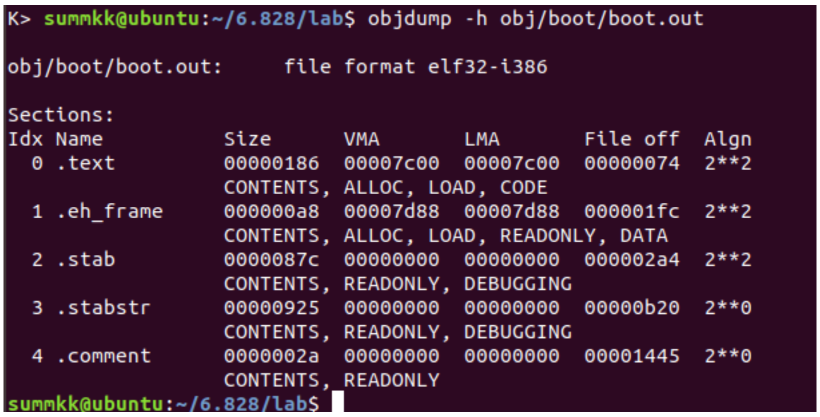
.data：数据部分保存程序的初始化数据，例如使用初始化程序（如 int x = 5;）声明的全局变量。

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：

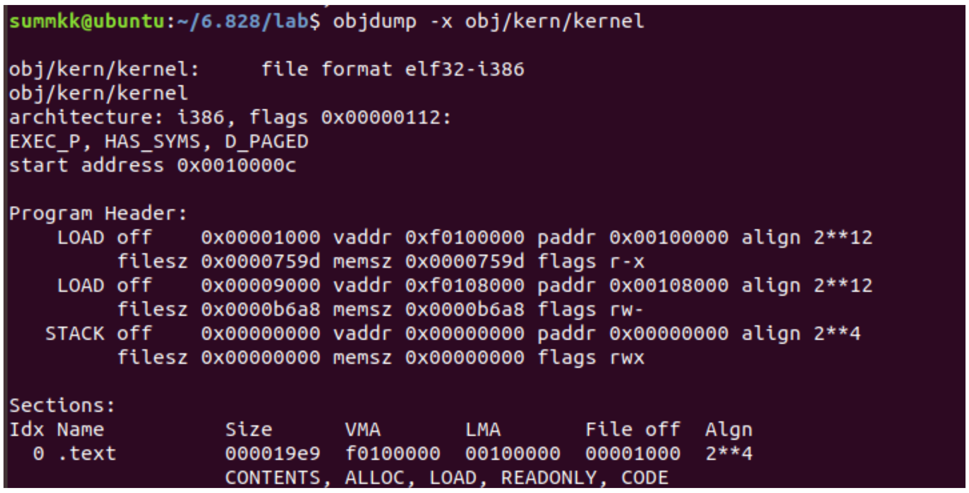
Exercise 5. Trace through the first few instructions of the boot loader again and identify the first instruction that would "break" or otherwise do the wrong thing if you were to get the boot loader's link address wrong. Then change the link address in `boot/Makefrag` to something wrong, run `make clean`, recompile the lab with `make`, and trace into the boot loader again to see what happens. Don't forget to change the link address back and `make clean` again afterward!

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。



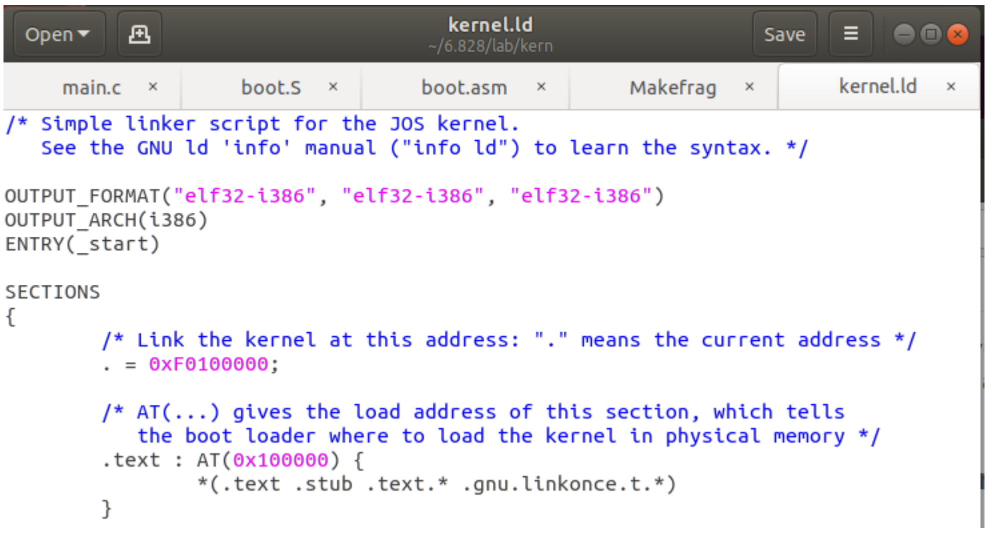
```
K> summkk@ubuntu:~/6.828/lab$ objdump -h obj/boot/boot.out

obj/boot/boot.out:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000186  00007c00  00007c00  00000074  2**2
                  CONTENTS, ALLOC, LOAD, CODE
  1 .eh_frame     000000a8  00007d88  00007d88  000001fc  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .stab         0000087c  00000000  00000000  000002a4  2**2
                  CONTENTS, READONLY, DEBUGGING
  3 .stabstr      00000925  00000000  00000000  00000b20  2**0
                  CONTENTS, READONLY, DEBUGGING
  4 .comment      0000002a  00000000  00000000  00001445  2**0
                  CONTENTS, READONLY
summkk@ubuntu:~/6.828/lab$
```

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。



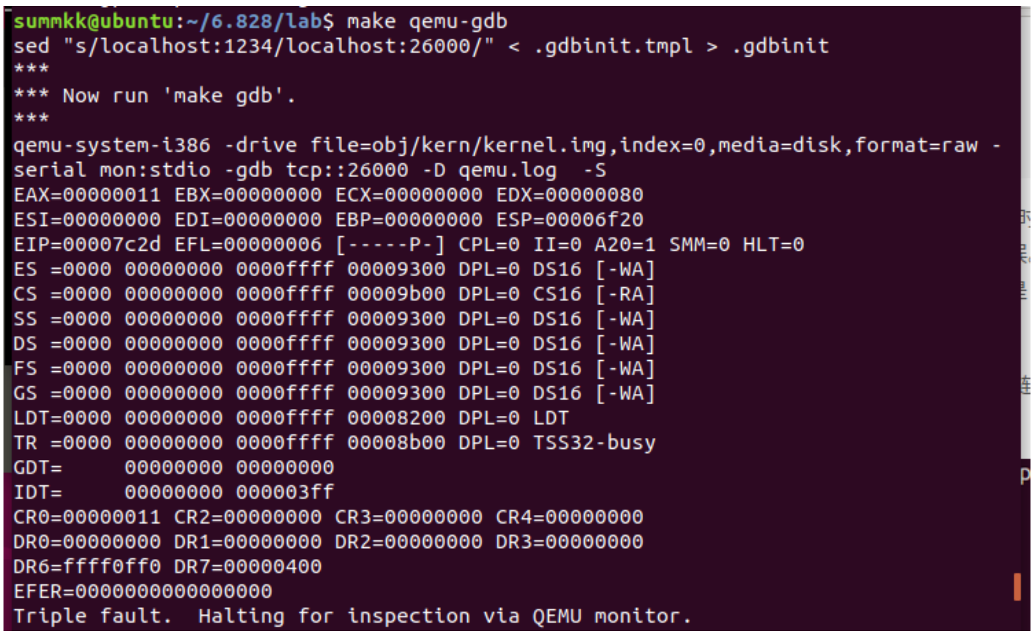
```
summkk@ubuntu:~/6.828/lab$ objdump -x obj/kern/kernel

obj/kern/kernel:     file format elf32-i386
obj/kern/kernel
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0010000c

Program Header:
    LOAD off    0x00001000 vaddr 0xf0100000 paddr 0x00100000 align 2**12
         filesz 0x0000759d memsz 0x0000759d flags r-x
    LOAD off    0x00009000 vaddr 0xf0108000 paddr 0x00108000 align 2**12
         filesz 0x0000b6a8 memsz 0x0000b6a8 flags rw-
   STACK off    0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**4
         filesz 0x00000000 memsz 0x00000000 flags rwx

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000019e9  f0100000  00100000  00001000  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
```

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。



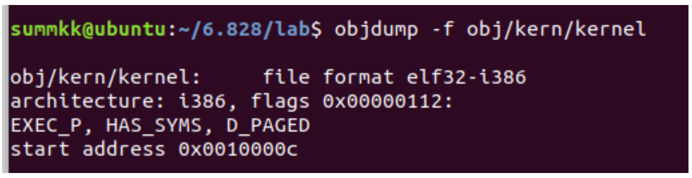
```
/* Simple linker script for the JOS kernel.
   See the GNU ld 'info' manual ("info ld") to learn the syntax. */

OUTPUT_FORMAT("elf32-i386", "elf32-i386", "elf32-i386")
OUTPUT_ARCH(i386)
ENTRY(_start)

SECTIONS
{
	/* Link the kernel at this address: "." means the current address */
	. = 0xF0100000;

	/* AT(...) gives the load address of this section, which tells
	   the boot loader where to load the kernel in physical memory */
	.text : AT(0x100000) {
		*(.text .stub .text.* .gnu.linkonce.t.*)
	}
```

Gdb 单步调试时无法继续，qemu 端报错如下



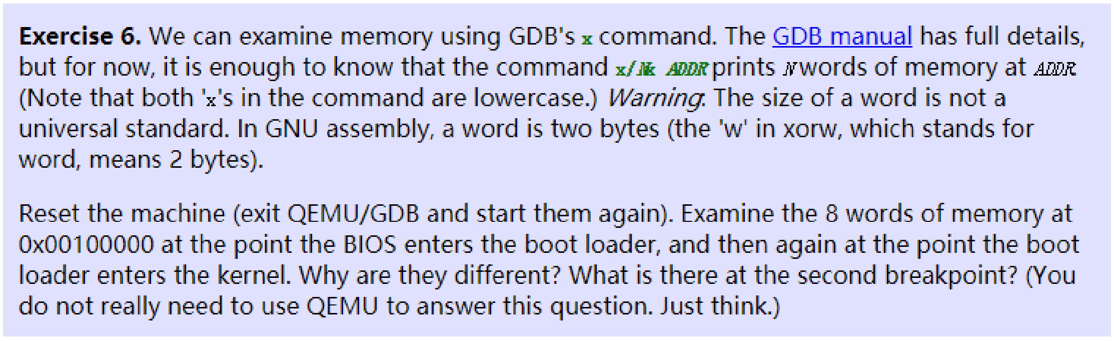
```
summkk@ubuntu:~/6.828/lab$ make qemu-gdb
sed "s/localhost:1234/localhost:26000/" < .gdbinit.tmpl > .gdbinit
***
*** Now run 'make gdb'.
***
qemu-system-i386 -drive file=obj/kern/kernel.img,index=0,media=disk,format=raw -
serial mon:stdio -gdb tcp::26000 -D qemu.log  -S
EAX=00000011 EBX=00000000 ECX=00000000 EDX=00000080
ESI=00000000 EDI=00000000 EBP=00000000 ESP=00006f20
EIP=00007c2d EFL=00000006 [-----P-] CPL=0 II=0 A20=1 SMM=0 HLT=0
ES =0000 00000000 0000ffff 00009300 DPL=0 DS16 [-WA]
CS =0000 00000000 0000ffff 00009b00 DPL=0 CS16 [-RA]
SS =0000 00000000 0000ffff 00009300 DPL=0 DS16 [-WA]
DS =0000 00000000 0000ffff 00009300 DPL=0 DS16 [-WA]
FS =0000 00000000 0000ffff 00009300 DPL=0 DS16 [-WA]
GS =0000 00000000 0000ffff 00009300 DPL=0 DS16 [-WA]
LDT=0000 00000000 0000ffff 00008200 DPL=0 LDT
TR =0000 00000000 0000ffff 00008b00 DPL=0 TSS32-busy
GDT=     00000000 00000000
IDT=     00000000 000003ff
CR0=00000011 CR2=00000000 CR3=00000000 CR4=00000000
DR0=00000000 DR1=00000000 DR2=00000000 DR3=00000000
DR6=ffff0ff0 DR7=00000400
EFER=0000000000000000
Triple fault.  Halting for inspection via QEMU monitor.
```

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。



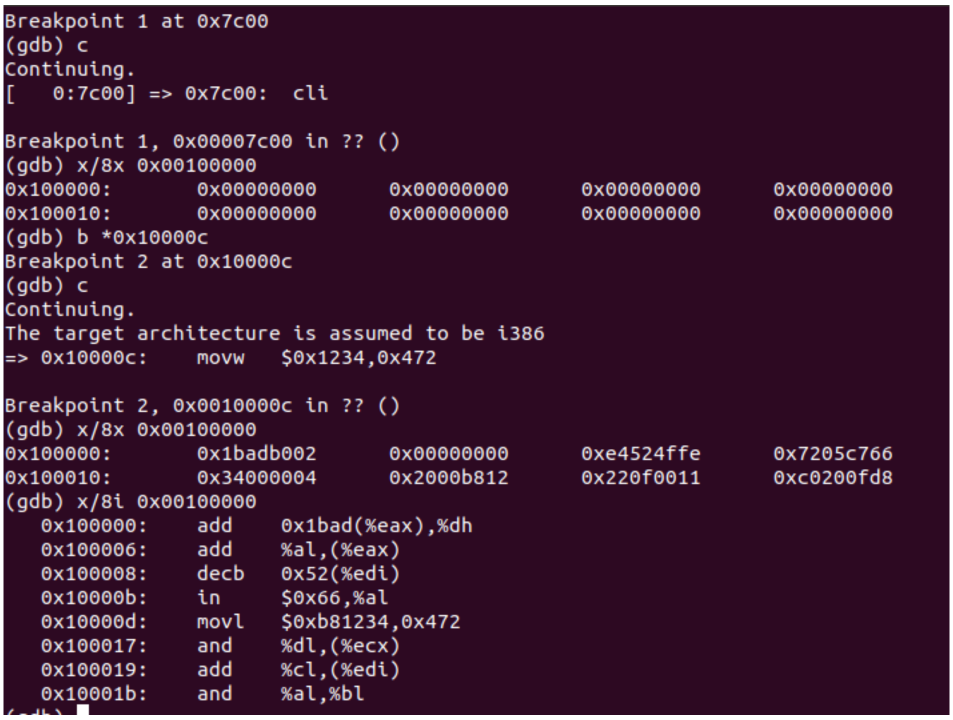
```
summkk@ubuntu:~/6.828/lab$ objdump -f obj/kern/kernel

obj/kern/kernel:     file format elf32-i386
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0010000c
```

Exercise6：



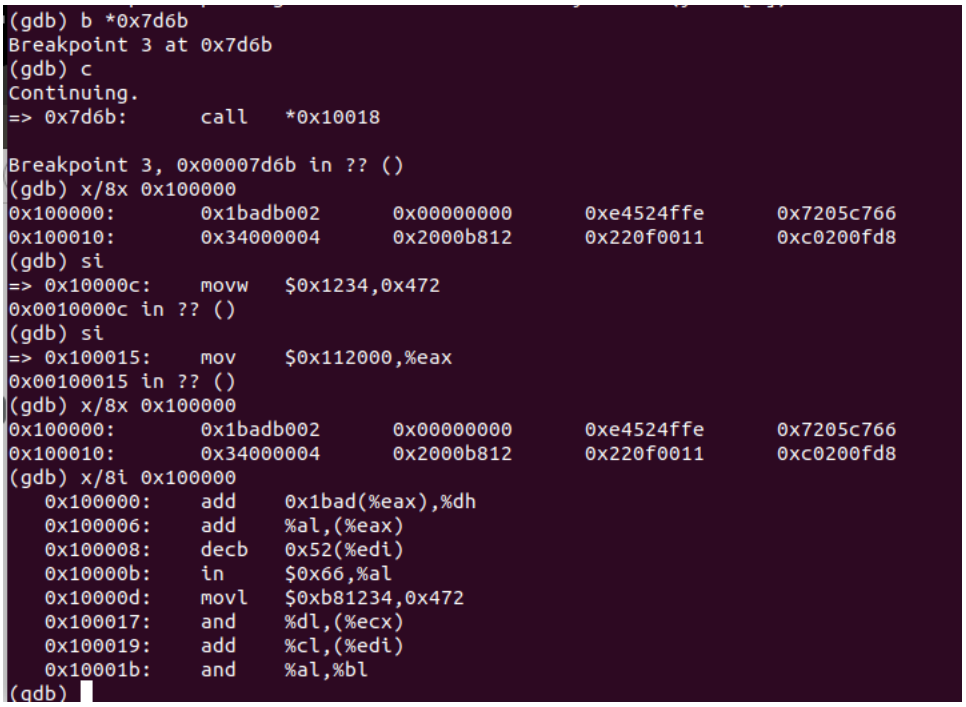
Exercise 6. We can examine memory using GDB's `x` command. The GDB manual has full details, but for now, it is enough to know that the command `x/Nx ADDR` prints *N* words of memory at *ADDR*. (Note that both 'x's in the command are lowercase.) *Warning*: The size of a word is not a universal standard. In GNU assembly, a word is two bytes (the 'w' in xorw, which stands for word, means 2 bytes).

Reset the machine (exit QEMU/GDB and start them again). Examine the 8 words of memory at 0x00100000 at the point the BIOS enters the boot loader, and then again at the point the boot loader enters the kernel. Why are they different? What is there at the second breakpoint? (You do not really need to use QEMU to answer this question. Just think.)



```
Breakpoint 1 at 0x7c00
(gdb) c
Continuing.
[   0:7c00] => 0x7c00:  cli

Breakpoint 1, 0x00007c00 in ?? ()
(gdb) x/8x 0x00100000
0x100000:       0x00000000      0x00000000      0x00000000      0x00000000
0x100010:       0x00000000      0x00000000      0x00000000      0x00000000
(gdb) b *0x10000c
Breakpoint 2 at 0x10000c
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x10000c:    movw   $0x1234,0x472

Breakpoint 2, 0x0010000c in ?? ()
(gdb) x/8x 0x00100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x7205c766
0x100010:       0x34000004      0x2000b812      0x220f0011      0xc0200fd8
(gdb) x/8i 0x00100000
   0x100000:    add    0x1bad(%eax),%dh
   0x100006:    add    %al,(%eax)
   0x100008:    decb   0x52(%edi)
   0x10000b:    in     $0x66,%al
   0x10000d:    movl   $0xb81234,0x472
   0x100017:    and    %dl,(%ecx)
   0x100019:    add    %cl,(%edi)
   0x10001b:    and    %al,%bl
```

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。



```
(gdb) b *0x7d6b
Breakpoint 3 at 0x7d6b
(gdb) c
Continuing.
=> 0x7d6b:      call   *0x10018

Breakpoint 3, 0x00007d6b in ?? ()
(gdb) x/8x 0x100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x7205c766
0x100010:       0x34000004      0x2000b812      0x220f0011      0xc0200fd8
(gdb) si
=> 0x10000c:    movw   $0x1234,0x472
0x0010000c in ?? ()
(gdb) si
=> 0x100015:    mov    $0x112000,%eax
0x00100015 in ?? ()
(gdb) x/8x 0x100000
0x100000:       0x1badb002      0x00000000      0xe4524ffe      0x7205c766
0x100010:       0x34000004      0x2000b812      0x220f0011      0xc0200fd8
(gdb) x/8i 0x100000
   0x100000:    add    0x1bad(%eax),%dh
   0x100006:    add    %al,(%eax)
   0x100008:    decb   0x52(%edi)
   0x10000b:    in     $0x66,%al
   0x10000d:    movl   $0xb81234,0x472
   0x100017:    and    %dl,(%ecx)
   0x100019:    add    %cl,(%edi)
   0x10001b:    and    %al,%bl
(gdb)
```

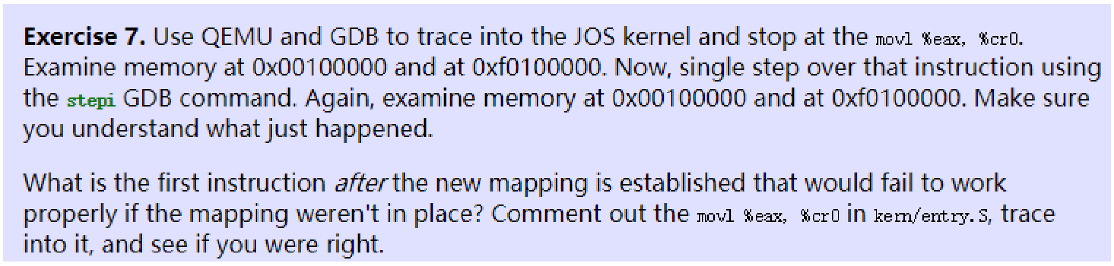
2.3 Part 3: The Kernel

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。

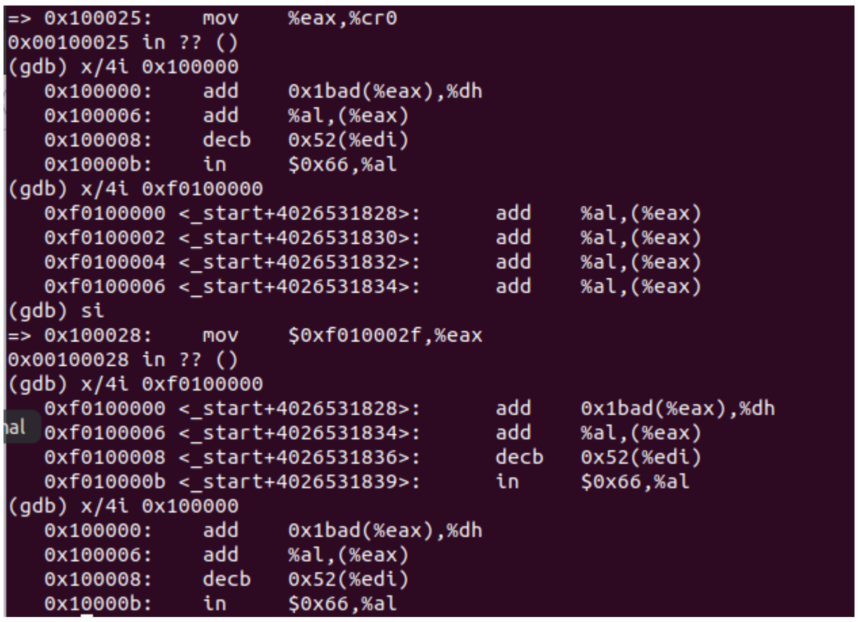
事实上，在下一个实验中，我们将把物理地址 0x00000000 到 0x0fffffff 的整个底部 256MB 的物理地址空间分别映射到虚拟地址 0xf0000000 到 0xffffffff。现在，我们只需映射前 4MB 的物理内存，这足以让我们启动并运行。我们使用 kern / entrypgdir.c 中手写的，静态初始化的页面目录和页表来完成此操作。直到 kern / entry.S 设置 CR0_PG 标志，内存引用被视为物理地址。一旦设置了 CR0_PG，内存引用就是虚拟内存硬件转换为物理地址的虚拟地址。我们建立一个页表将虚拟地址的[KERNBASE, KERNBASE+4MB)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）。

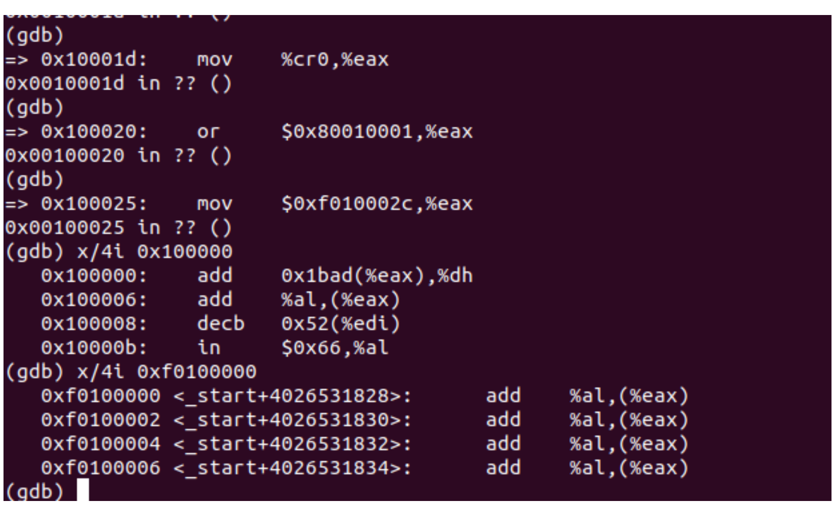
Exercise7：

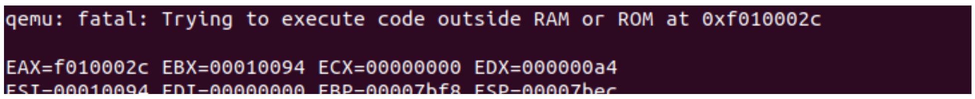

Exercise 7. Use QEMU and GDB to trace into the JOS kernel and stop at the `movl %eax, %cr0`. Examine memory at 0x00100000 and at 0xf0100000. Now, single step over that instruction using the `stepi` GDB command. Again, examine memory at 0x00100000 and at 0xf0100000. Make sure you understand what just happened.

What is the first instruction *after* the new mapping is established that would fail to work properly if the mapping weren't in place? Comment out the `movl %eax, %cr0` in `kern/entry.S`, trace into it, and see if you were right.

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。


```
=> 0x100025:    mov    %eax,%cr0
0x00100025 in ?? ()
(gdb) x/4i 0x100000
   0x100000:    add    0x1bad(%eax),%dh
   0x100006:    add    %al,(%eax)
   0x100008:    decb   0x52(%edi)
   0x10000b:    in     $0x66,%al
(gdb) x/4i 0xf0100000
   0xf0100000 <_start+4026531828>:      add    %al,(%eax)
   0xf0100002 <_start+4026531830>:      add    %al,(%eax)
   0xf0100004 <_start+4026531832>:      add    %al,(%eax)
   0xf0100006 <_start+4026531834>:      add    %al,(%eax)
(gdb) si
=> 0x100028:    mov    $0xf010002f,%eax
0x00100028 in ?? ()
(gdb) x/4i 0xf0100000
   0xf0100000 <_start+4026531828>:      add    0x1bad(%eax),%dh
   0xf0100006 <_start+4026531834>:      add    %al,(%eax)
   0xf0100008 <_start+4026531836>:      decb   0x52(%edi)
   0xf010000b <_start+4026531839>:      in     $0x66,%al
(gdb) x/4i 0x100000
   0x100000:    add    0x1bad(%eax),%dh
   0x100006:    add    %al,(%eax)
   0x100008:    decb   0x52(%edi)
   0x10000b:    in     $0x66,%al
```
注释掉汇编语句 mov %eax,%cr0，重新编译运行。


```
(gdb)
=> 0x10001d:    mov    %cr0,%eax
0x0010001d in ?? ()
(gdb)
=> 0x100020:    or     $0x80010001,%eax
0x00100020 in ?? ()
(gdb)
=> 0x100025:    mov    $0xf010002c,%eax
0x00100025 in ?? ()
(gdb) x/4i 0x100000
   0x100000:    add    0x1bad(%eax),%dh
   0x100006:    add    %al,(%eax)
   0x100008:    decb   0x52(%edi)
   0x10000b:    in     $0x66,%al
(gdb) x/4i 0xf0100000
   0xf0100000 <_start+4026531828>:      add    %al,(%eax)
   0xf0100002 <_start+4026531830>:      add    %al,(%eax)
   0xf0100004 <_start+4026531832>:      add    %al,(%eax)
   0xf0100006 <_start+4026531834>:      add    %al,(%eax)
(gdb)
```
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。

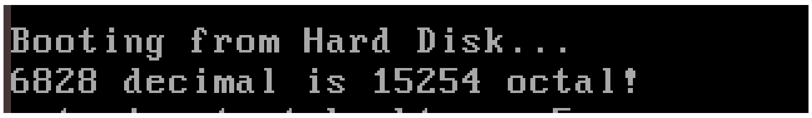

```
qemu: fatal: Trying to execute code outside RAM or ROM at 0xf010002c

EAX=f010002c EBX=00010094 ECX=00000000 EDX=000000a4
```

Exercise8:

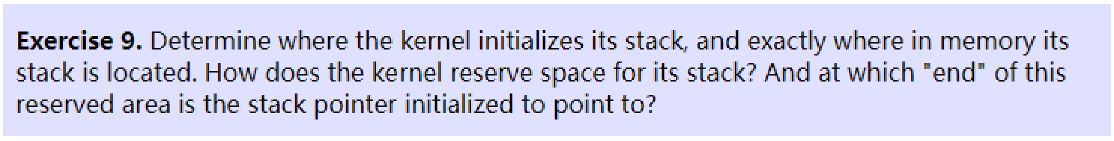
Exercise 8. We have omitted a small fragment of code - the code necessary to print octal numbers using patterns of the form "%o". Find and fill in this code fragment.

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。

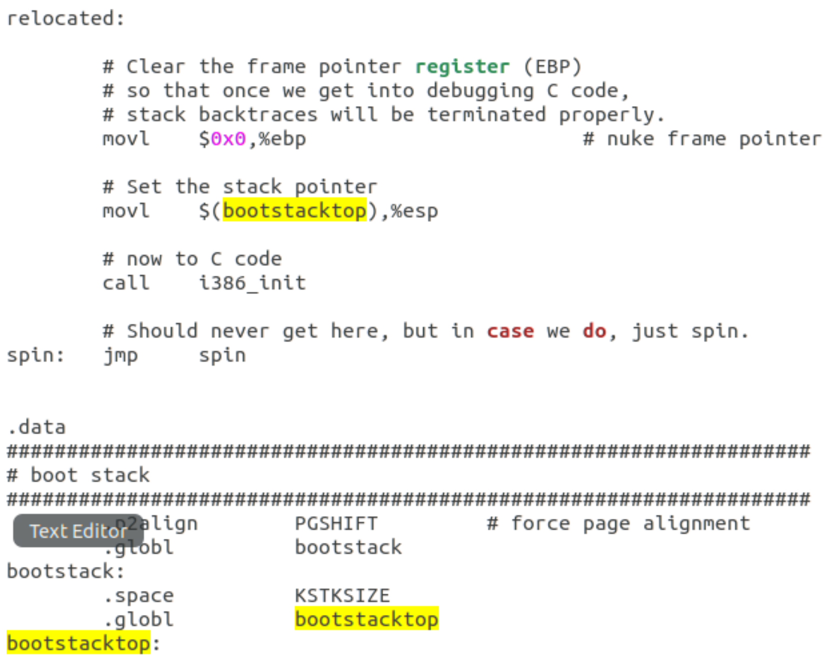
修改 printfmt 里的 vprintfmt 函数中 case’o’的部分，重新 make qemuzhih 输出改变。


```
Booting from Hard Disk...
6828 decimal is 15254 octal!
```

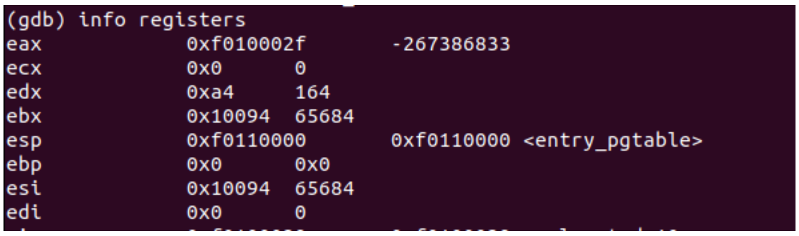
Exercise9：


Exercise 9. Determine where the kernel initializes its stack, and exactly where in memory its stack is located. How does the kernel reserve space for its stack? And at which "end" of this reserved area is the stack pointer initialized to point to?

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。


```
relocated:

        # Clear the frame pointer register (EBP)
        # so that once we get into debugging C code,
        # stack backtraces will be terminated properly.
        movl    $0x0,%ebp                       # nuke frame pointer

        # Set the stack pointer
        movl    $(bootstacktop),%esp

        # now to C code
        call    i386_init

        # Should never get here, but in case we do, just spin.
spin:   jmp     spin


.data
###################################################################
# boot stack
###################################################################
        .p2align        PGSHIFT         # force page alignment
        .globl          bootstack
bootstack:
        .space          KSTKSIZE
        .globl          bootstacktop
bootstacktop:
```

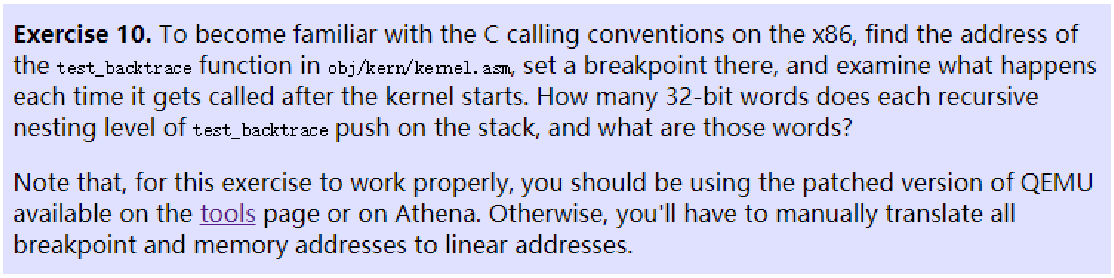
在 gdb 调试到对应位置 mov 语句后，查看 esp 和 ebp 的值。此时 esp 指向栈顶。


```
(gdb) info registers
eax            0xf010002f       -267386833
ecx            0x0      0
edx            0xa4     164
ebx            0x10094  65684
esp            0xf0110000       0xf0110000 <entry_pgtable>
ebp            0x0      0x0
esi            0x10094  65684
edi            0x0      0
```

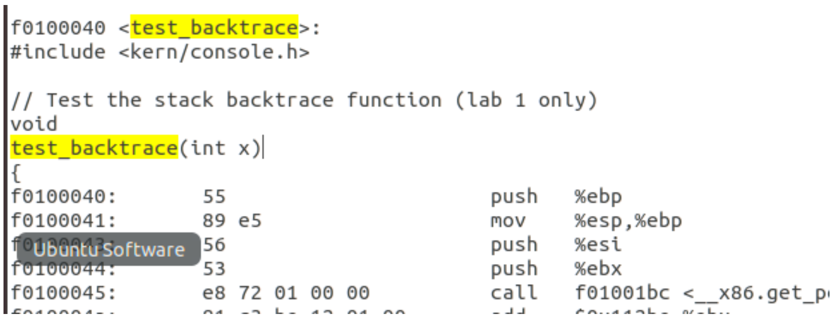
x86 堆栈指针（esp 寄存器）指向当前正在使用的堆栈上的最低位置。保留给堆栈的区域中该位置以下的所有内容都是空闲的。将值压入堆栈涉及减少堆栈指针，然后将值写入堆栈指针指向的位置。从堆栈中弹出一个值包括读取堆栈指针指向的值，然后增加堆栈指针。在 32 位模式下，堆栈只能保存 32 位值，esp 总是可以被 4 整除。

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。

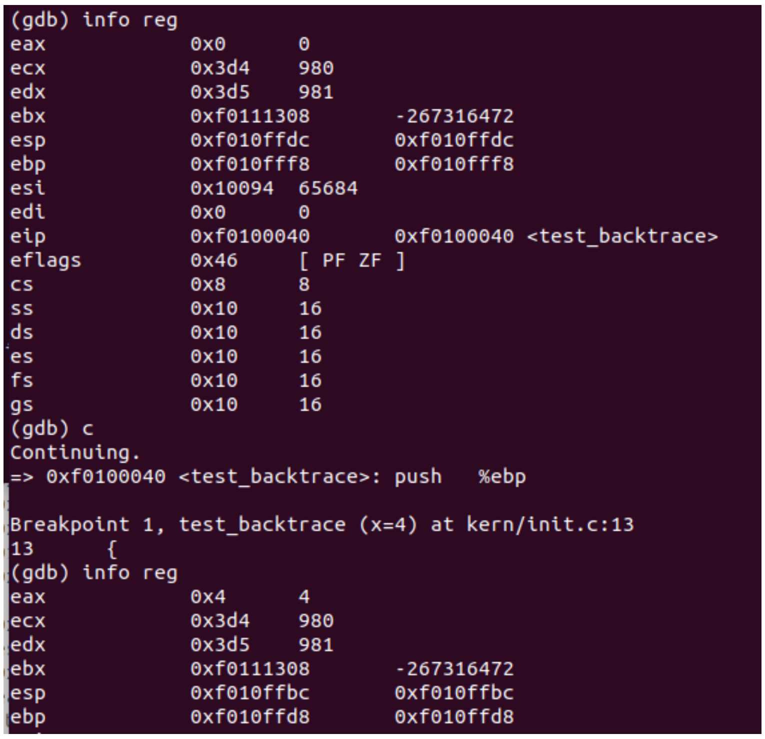
Exercise10：


Exercise 10. To become familiar with the C calling conventions on the x86, find the address of the `test_backtrace` function in `obj/kern/kernel.asm`, set a breakpoint there, and examine what happens each time it gets called after the kernel starts. How many 32-bit words does each recursive nesting level of `test_backtrace` push on the stack, and what are those words?

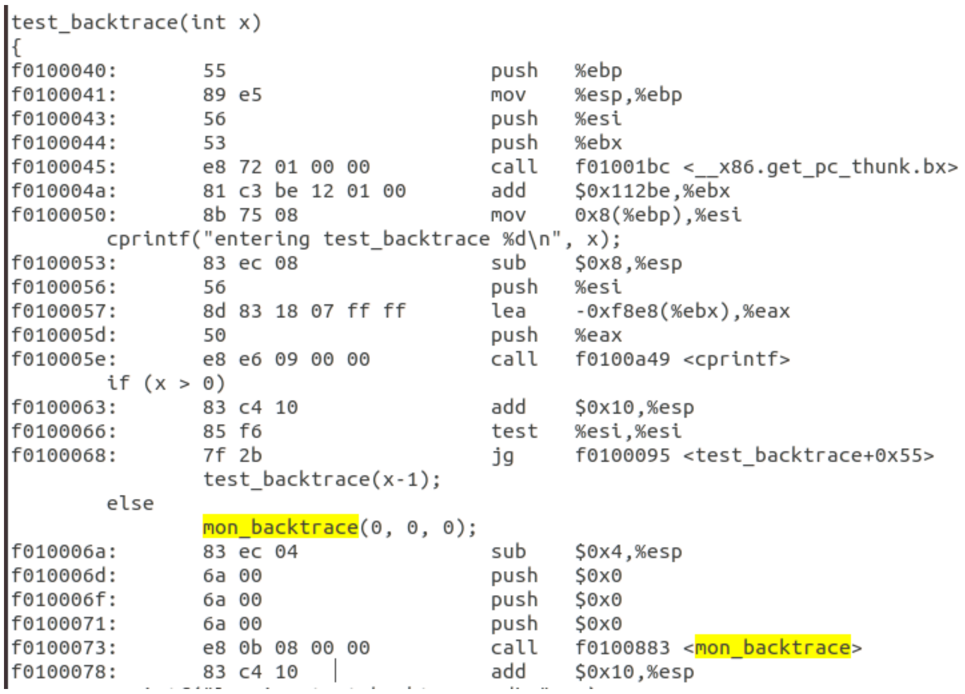
Note that, for this exercise to work properly, you should be using the patched version of QEMU available on the tools page or on Athena. Otherwise, you'll have to manually translate all breakpoint and memory addresses to linear addresses.


```
f0100040 <test_backtrace>:
#include <kern/console.h>

// Test the stack backtrace function (lab 1 only)
void
test_backtrace(int x)
{
f0100040:       55                      push   %ebp
f0100041:       89 e5                   mov    %esp,%ebp
f0100043:       56                      push   %esi
f0100044:       53                      push   %ebx
f0100045:       e8 72 01 00 00          call   f01001bc <__x86.get_p
```

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。



```
(gdb) info reg
eax            0x0      0
ecx            0x3d4    980
edx            0x3d5    981
ebx            0xf0111308       -267316472
esp            0xf010ffdc       0xf010ffdc
ebp            0xf010fff8       0xf010fff8
esi            0x10094  65684
edi            0x0      0
eip            0xf0100040       0xf0100040 <test_backtrace>
eflags         0x46     [ PF ZF ]
cs             0x8      8
ss             0x10     16
ds             0x10     16
es             0x10     16
fs             0x10     16
gs             0x10     16
(gdb) c
Continuing.
=> 0xf0100040 <test_backtrace>: push   %ebp

Breakpoint 1, test_backtrace (x=4) at kern/init.c:13
13      {
(gdb) info reg
eax            0x4      4
ecx            0x3d4    980
edx            0x3d5    981
ebx            0xf0111308       -267316472
esp            0xf010ffbc       0xf010ffbc
ebp            0xf010ffd8       0xf010ffd8
```



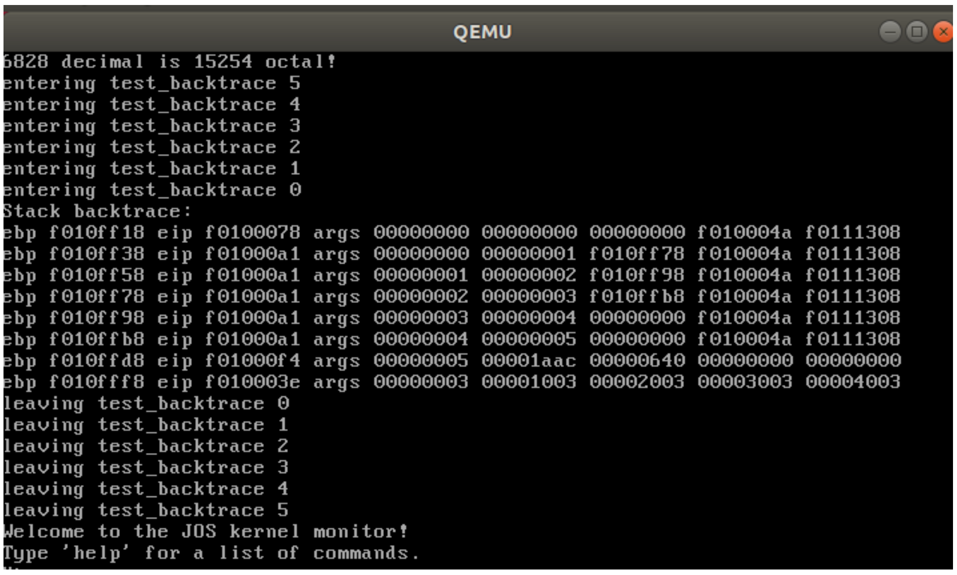
```
test_backtrace(int x)
{
f0100040:       55                      push   %ebp
f0100041:       89 e5                   mov    %esp,%ebp
f0100043:       56                      push   %esi
f0100044:       53                      push   %ebx
f0100045:       e8 72 01 00 00          call   f01001bc <__x86.get_pc_thunk.bx>
f010004a:       81 c3 be 12 01 00       add    $0x112be,%ebx
f0100050:       8b 75 08                mov    0x8(%ebp),%esi
        cprintf("entering test_backtrace %d\n", x);
f0100053:       83 ec 08                sub    $0x8,%esp
f0100056:       56                      push   %esi
f0100057:       8d 83 18 07 ff ff       lea    -0xf8e8(%ebx),%eax
f010005d:       50                      push   %eax
f010005e:       e8 e6 09 00 00          call   f0100a49 <cprintf>
        if (x > 0)
f0100063:       83 c4 10                add    $0x10,%esp
f0100066:       85 f6                   test   %esi,%esi
f0100068:       7f 2b                   jg     f0100095 <test_backtrace+0x55>
                test_backtrace(x-1);
        else
                mon_backtrace(0, 0, 0);
f010006a:       83 ec 04                sub    $0x4,%esp
f010006d:       6a 00                   push   $0x0
f010006f:       6a 00                   push   $0x0
f0100071:       6a 00                   push   $0x0
f0100073:       e8 0b 08 00 00          call   f0100883 <mon_backtrace>
f0100078:       83 c4 10                add    $0x10,%esp
```

Exercise11：

Exercise 11. Implement the backtrace function as specified above. Use the same format as in the example, since otherwise the grading script will be confused. When you think you have it working right, run `make grade` to see if its output conforms to what our grading script expects, and fix it if it doesn't. *After* you have handed in your Lab 1 code, you are welcome to change the output format of the backtrace function any way you like.

If you use `read_ebp()`, note that GCC may generate "optimized" code that calls `read_ebp()` *before* `mon_backtrace()`'s function prologue, which results in an incomplete stack trace (the stack frame of the most recent function call is missing). While we have tried to disable optimizations that cause this reordering, you may want to examine the assembly of `mon_backtrace()` and make sure the call to `read_ebp()` is happening after the function prologue.

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。



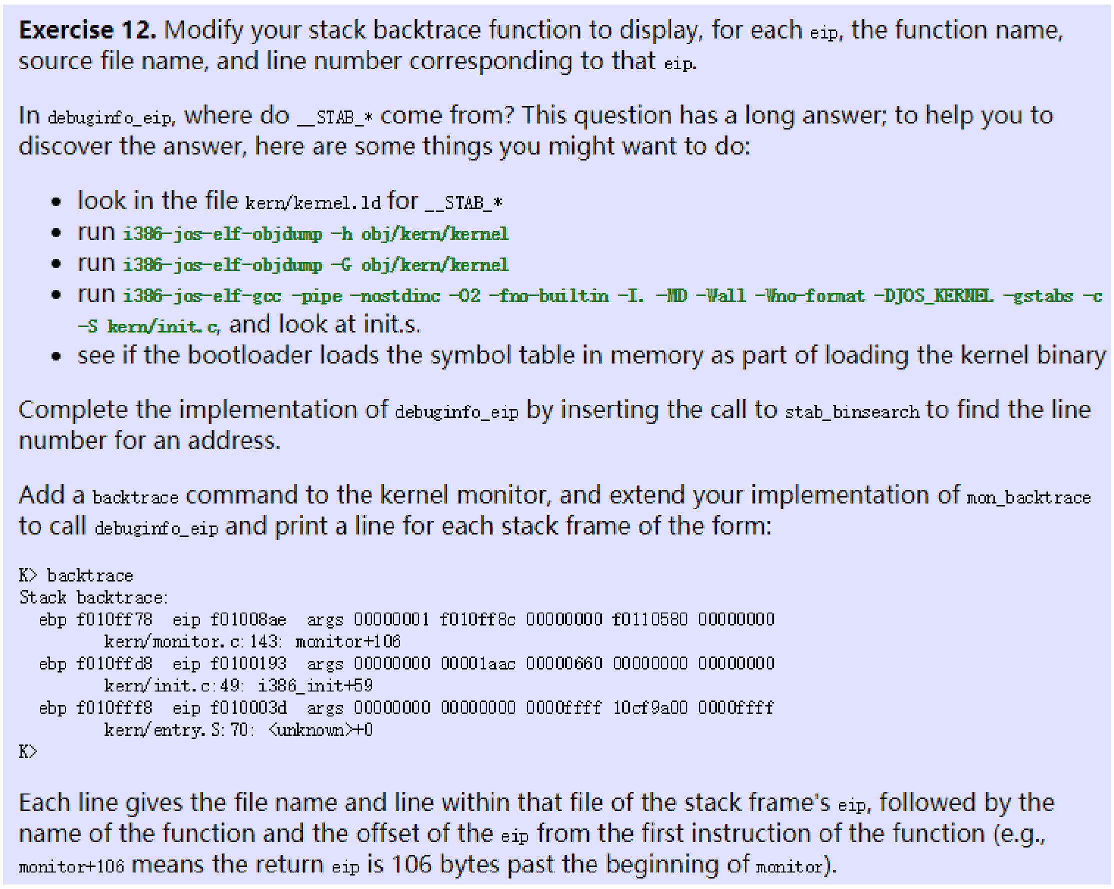
QEMU

```
6828 decimal is 15254 octal!
entering test_backtrace 5
entering test_backtrace 4
entering test_backtrace 3
entering test_backtrace 2
entering test_backtrace 1
entering test_backtrace 0
Stack backtrace:
ebp f010ff18 eip f0100078 args 00000000 00000000 00000000 f010004a f0111308
ebp f010ff38 eip f01000a1 args 00000000 00000001 f010ff78 f010004a f0111308
ebp f010ff58 eip f01000a1 args 00000001 00000002 f010ff98 f010004a f0111308
ebp f010ff78 eip f01000a1 args 00000002 00000003 f010ffb8 f010004a f0111308
ebp f010ff98 eip f01000a1 args 00000003 00000004 00000000 f010004a f0111308
ebp f010ffb8 eip f01000a1 args 00000004 00000005 00000000 f010004a f0111308
ebp f010ffd8 eip f01000f4 args 00000005 00001aac 00000640 00000000 00000000
ebp f010fff8 eip f010003e args 00000003 00001003 00002003 00003003 00004003
leaving test_backtrace 0
leaving test_backtrace 1
leaving test_backtrace 2
leaving test_backtrace 3
leaving test_backtrace 4
leaving test_backtrace 5
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
```

return when the function returns. The return instruction pointer typically points to the instruction after the `call` instruction (why?). Finally, the five hex values listed after `args` are the first five arguments to the function in question, which would have been pushed on the stack just before the function was called. If the function was called with fewer than five arguments, of course, then not all five of these values will be useful. (Why can't the backtrace code detect how many arguments there actually are? How could this limitation be fixed?)

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：



Exercise 12. Modify your stack backtrace function to display, for each `eip`, the function name, source file name, and line number corresponding to that `eip`.

In `debuginfo_eip`, where do `__STAB_*` come from? This question has a long answer; to help you to discover the answer, here are some things you might want to do:

- look in the file `kern/kernel.ld` for `__STAB_*`
- run `i386-jos-elf-objdump -h obj/kern/kernel`
- run `i386-jos-elf-objdump -G obj/kern/kernel`
- run `i386-jos-elf-gcc -pipe -nostdinc -O2 -fno-builtin -I. -MD -Wall -Wno-format -DJOS_KERNEL -gstabs -c -S kern/init.c`, and look at init.s.
- see if the bootloader loads the symbol table in memory as part of loading the kernel binary

Complete the implementation of `debuginfo_eip` by inserting the call to `stab_binsearch` to find the line number for an address.

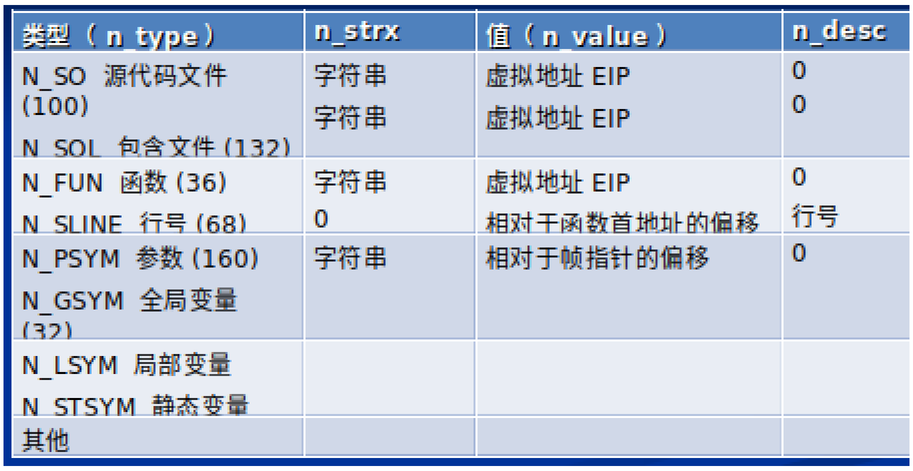
Add a `backtrace` command to the kernel monitor, and extend your implementation of `mon_backtrace` to call `debuginfo_eip` and print a line for each stack frame of the form:

```
K> backtrace
Stack backtrace:
  ebp f010ff78  eip f01008ae  args 00000001 f010ff8c 00000000 f0110580 00000000
         kern/monitor.c:143: monitor+106
  ebp f010ffd8  eip f0100193  args 00000000 00001aac 00000660 00000000 00000000
         kern/init.c:49: i386_init+59
  ebp f010fff8  eip f010003d  args 00000000 00000000 0000ffff 10cf9a00 0000ffff
         kern/entry.S:70: <unknown>+0
K>
```

Each line gives the file name and line within that file of the stack frame's `eip`, followed by the name of the function and the offset of the `eip` from the first instruction of the function (e.g., `monitor+106` means the return `eip` is 106 bytes past the beginning of `monitor`).

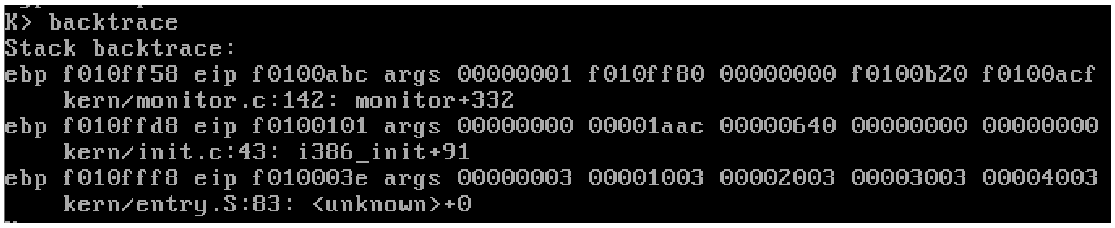
在 monitor.c 中添加 backtrace 命令，修改 kdebug.c 的 debuginfo_eip()和 monitor.c 中的 mon_backtrace()。

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。



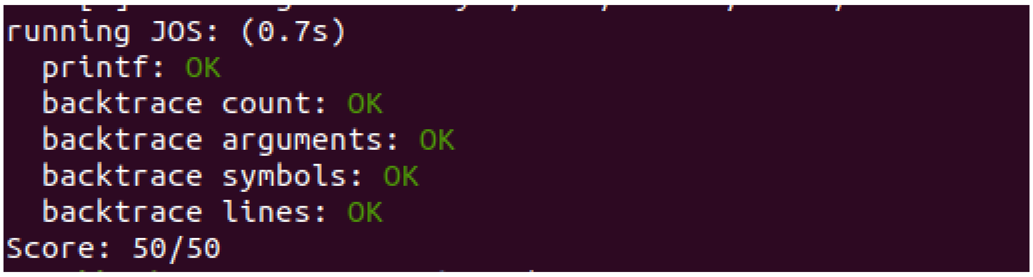
类型（n_type）	n_strx	值（n_value）	n_desc
N_SO 源代码文件 (100) N_SOL 包含文件 (132)	字符串 字符串	虚拟地址 EIP 虚拟地址 EIP	0 0
N_FUN 函数 (36) N_SLINE 行号 (68)	字符串 0	虚拟地址 EIP 相对于函数首地址的偏移	0 行号
N_PSYM 参数 (160) N_GSYM 全局变量 (32)	字符串	相对于帧指针的偏移	0
N_LSYM 局部变量 N_STSYM 静态变量			
其他			

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。



```
K> backtrace
Stack backtrace:
ebp f010ff58 eip f0100abc args 00000001 f010ff80 00000000 f0100b20 f0100acf
    kern/monitor.c:142: monitor+332
ebp f010ffd8 eip f0100101 args 00000000 00001aac 00000640 00000000 00000000
    kern/init.c:43: i386_init+91
ebp f010fff8 eip f010003e args 00000003 00001003 00002003 00003003 00004003
    kern/entry.S:83: <unknown>+0
```

运行 make grade 测试结果如图



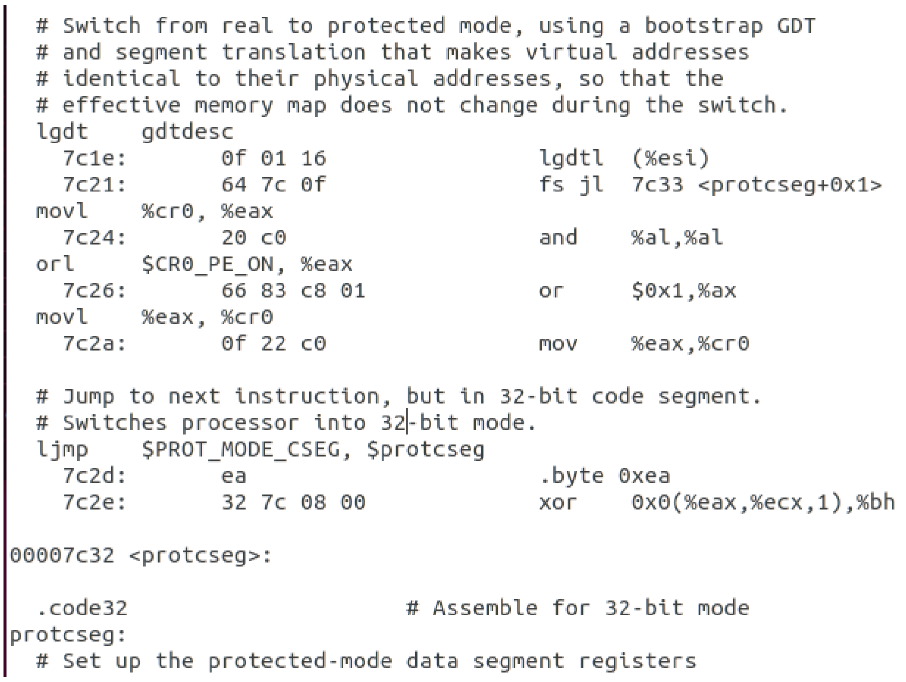
```
running JOS: (0.7s)
  printf: OK
  backtrace count: OK
  backtrace arguments: OK
  backtrace symbols: OK
  backtrace lines: OK
Score: 50/50
```

3 实验问题。

Exercise3 后的 question

Be able to answer the following questions:

- At what point does the processor start executing 32-bit code? What exactly causes the switch from 16- to 32-bit mode?
- What is the *last* instruction of the boot loader executed, and what is the *first* instruction of the kernel it just loaded?
- *Where* is the first instruction of the kernel?
- How does the boot loader decide how many sectors it must read in order to fetch the entire kernel from disk? Where does it find this information?



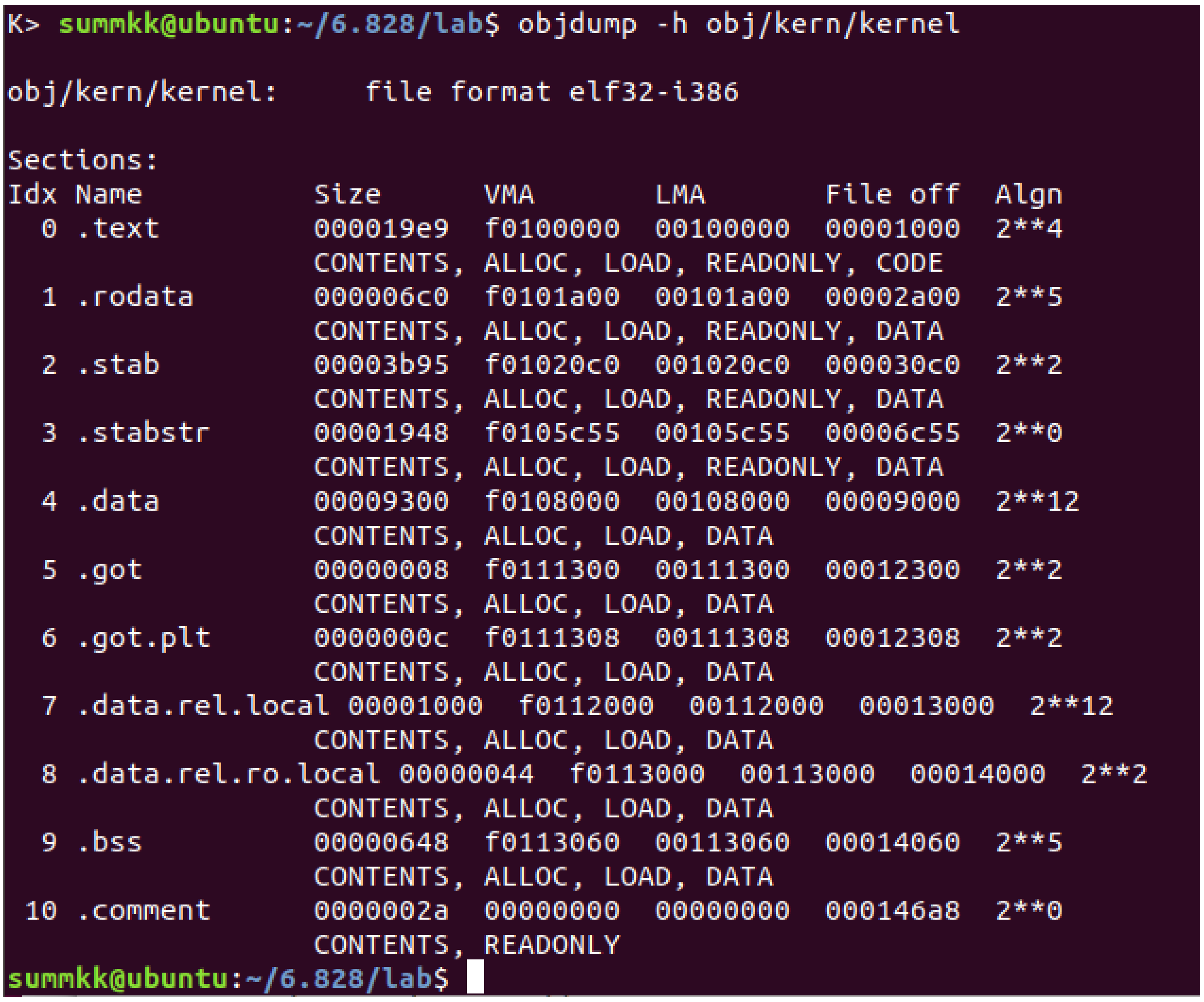
```
  # Switch from real to protected mode, using a bootstrap GDT
  # and segment translation that makes virtual addresses
  # identical to their physical addresses, so that the
  # effective memory map does not change during the switch.
  lgdt    gdtdesc
    7c1e:       0f 01 16                lgdtl  (%esi)
    7c21:       64 7c 0f                fs jl  7c33 <protcseg+0x1>
  movl    %cr0, %eax
    7c24:       20 c0                   and    %al,%al
  orl     $CR0_PE_ON, %eax
    7c26:       66 83 c8 01             or     $0x1,%ax
  movl    %eax, %cr0
    7c2a:       0f 22 c0                mov    %eax,%cr0

  # Jump to next instruction, but in 32-bit code segment.
  # Switches processor into 32-bit mode.
  ljmp    $PROT_MODE_CSEG, $protcseg
    7c2d:       ea                      .byte 0xea
    7c2e:       32 7c 08 00             xor    0x0(%eax,%ecx,1),%bh

00007c32 <protcseg>:

  .code32                     # Assemble for 32-bit mode
protcseg:
  # Set up the protected-mode data segment registers
```

- 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。


```
K> summkk@ubuntu:~/6.828/lab$ objdump -h obj/kern/kernel

obj/kern/kernel:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         000019e9  f0100000  00100000  00001000  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       000006c0  f0101a00  00101a00  00002a00  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .stab         00003b95  f01020c0  001020c0  000030c0  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .stabstr      00001948  f0105c55  00105c55  00006c55  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .data         00009300  f0108000  00108000  00009000  2**12
                  CONTENTS, ALLOC, LOAD, DATA
  5 .got          00000008  f0111300  00111300  00012300  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  6 .got.plt      0000000c  f0111308  00111308  00012308  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  7 .data.rel.local 00001000  f0112000  00112000  00013000  2**12
                  CONTENTS, ALLOC, LOAD, DATA
  8 .data.rel.ro.local 00000044  f0113000  00113000  00014000  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  9 .bss          00000648  f0113060  00113060  00014060  2**5
                  CONTENTS, ALLOC, LOAD, DATA
 10 .comment      0000002a  00000000  00000000  000146a8  2**0
                  CONTENTS, READONLY
summkk@ubuntu:~/6.828/lab$
```

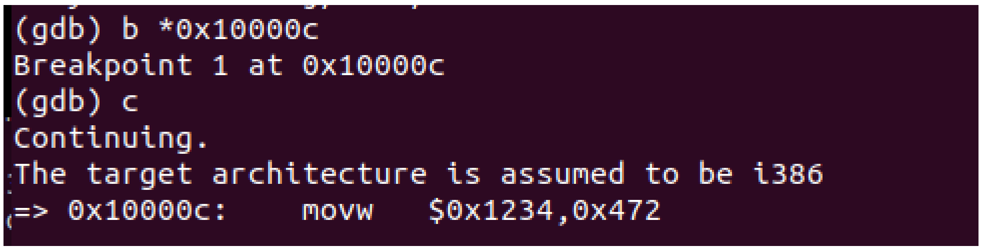
- 引导加载程序的最后一条指令是

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。

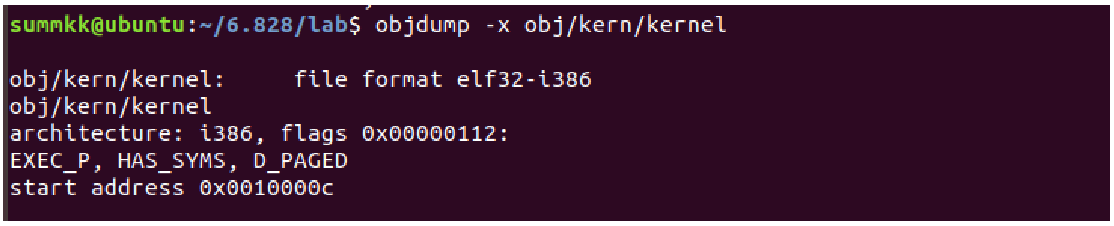
- 刚加载的内核的第一条指令是

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。


```
(gdb) b *0x10000c
Breakpoint 1 at 0x10000c
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0x10000c:	movw   $0x1234,0x472
```

- 内核的第一条指令在哪里？

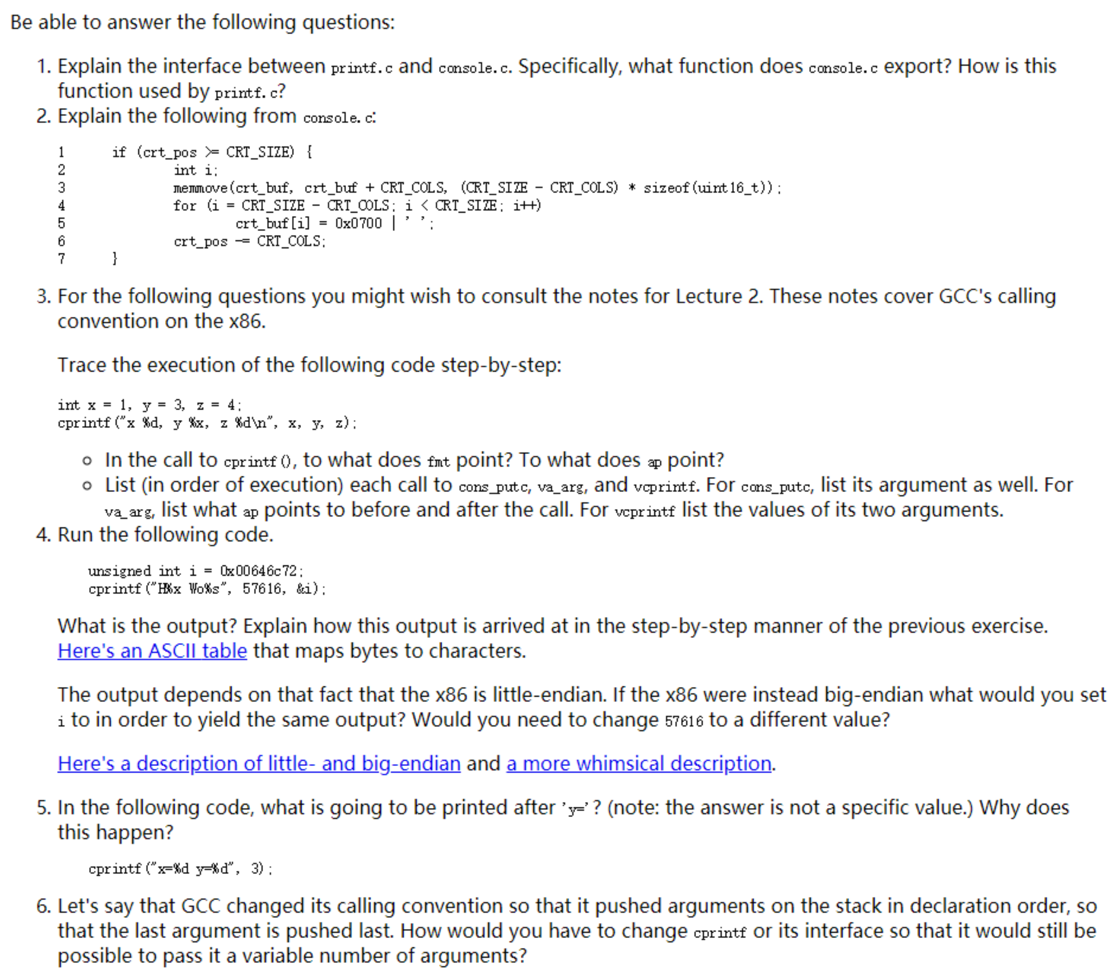

```
summkk@ubuntu:~/6.828/lab$ objdump -x obj/kern/kernel

obj/kern/kernel:     file format elf32-i386
obj/kern/kernel
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x0010000c
```

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



Be able to answer the following questions:

1. Explain the interface between `printf.c` and `console.c`. Specifically, what function does `console.c` export? How is this function used by `printf.c`?
2. Explain the following from `console.c`:

```
if (crt_pos >= CRT_SIZE) {
        int i;
        memmove(crt_buf, crt_buf + CRT_COLS, (CRT_SIZE - CRT_COLS) * sizeof(uint16_t));
        for (i = CRT_SIZE - CRT_COLS; i < CRT_SIZE; i++)
                crt_buf[i] = 0x0700 | ' ';
        crt_pos -= CRT_COLS;
}
```

3. For the following questions you might wish to consult the notes for Lecture 2. These notes cover GCC's calling convention on the x86.

   Trace the execution of the following code step-by-step:

```
int x = 1, y = 3, z = 4;
cprintf("x %d, y %x, z %d\n", x, y, z);
```

   - In the call to `cprintf()`, to what does `fmt` point? To what does `ap` point?
   - List (in order of execution) each call to `cons_putc`, `va_arg`, and `vcprintf`. For `cons_putc`, list its argument as well. For `va_arg`, list what `ap` points to before and after the call. For `vcprintf` list the values of its two arguments.
4. Run the following code.

```
    unsigned int i = 0x00646c72;
    cprintf("H%x Wo%s", 57616, &i);
```

   What is the output? Explain how this output is arrived at in the step-by-step manner of the previous exercise. Here's an ASCII table that maps bytes to characters.

   The output depends on that fact that the x86 is little-endian. If the x86 were instead big-endian what would you set `i` to in order to yield the same output? Would you need to change `57616` to a different value?

   Here's a description of little- and big-endian and a more whimsical description.

5. In the following code, what is going to be printed after `'y='`? (note: the answer is not a specific value.) Why does this happen?

```
    cprintf("x=%d y=%d", 3);
```

6. Let's say that GCC changed its calling convention so that it pushed arguments on the stack in declaration order, so that the last argument is pushed last. How would you have to change `cprintf` or its interface so that it would still be possible to pass it a variable number of arguments?

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。

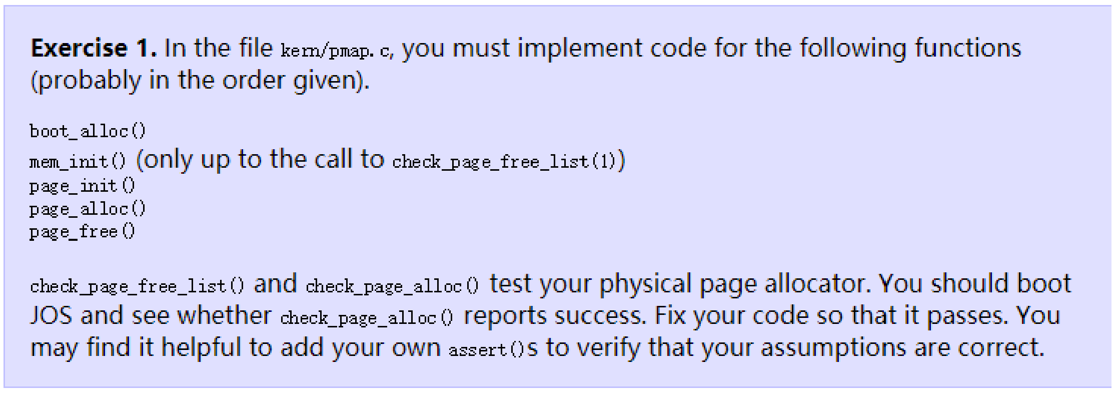
2 实验内容

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



Exercise 1. In the file `kern/pmap.c`, you must implement code for the following functions (probably in the order given).

`boot_alloc()`
`mem_init()` (only up to the call to `check_page_free_list(1)`)
`page_init()`
`page_alloc()`
`page_free()`

`check_page_free_list()` and `check_page_alloc()` test your physical page allocator. You should boot JOS and see whether `check_page_alloc()` reports success. Fix your code so that it passes. You may find it helpful to add your own `assert()`s to verify that your assumptions are correct.

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。

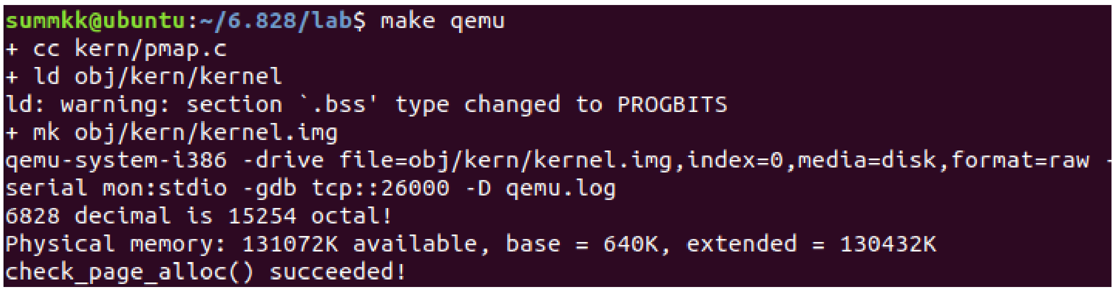
page_alloc()

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()。



```
summkk@ubuntu:~/6.828/lab$ make qemu
+ cc kern/pmap.c
+ ld obj/kern/kernel
ld: warning: section `.bss' type changed to PROGBITS
+ mk obj/kern/kernel.img
qemu-system-i386 -drive file=obj/kern/kernel.img,index=0,media=disk,format=raw -
serial mon:stdio -gdb tcp::26000 -D qemu.log
6828 decimal is 15254 octal!
Physical memory: 131072K available, base = 640K, extended = 130432K
check_page_alloc() succeeded!
```

图 2.1 exercise1 测试结果

2.2 Part 2: Virtual Memory

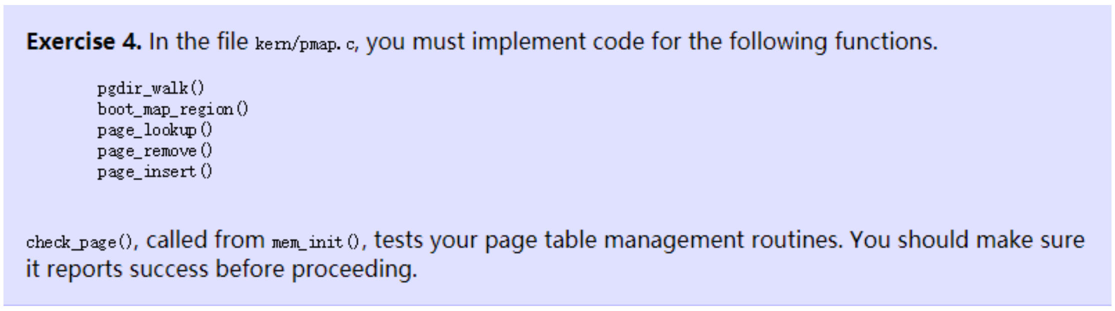
Exercise2

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



Exercise 4. In the file kern/pmap.c, you must implement code for the following functions.

```
pgdir_walk()
boot_map_region()
page_lookup()
page_remove()
page_insert()
```

check_page(), called from mem_init(), tests your page table management routines. You should make sure it reports success before proceeding.

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。

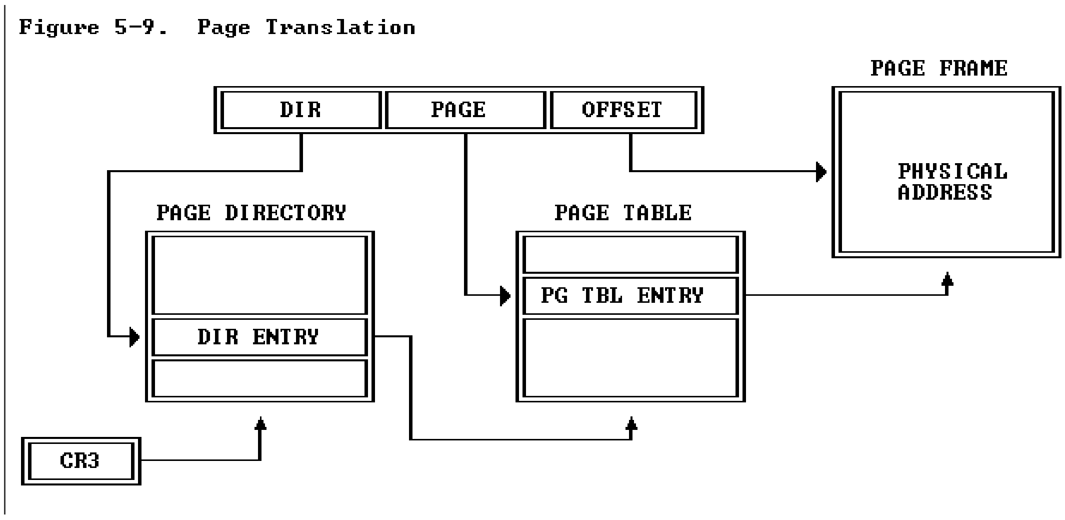


图 2.2 页表的寻址过程

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()

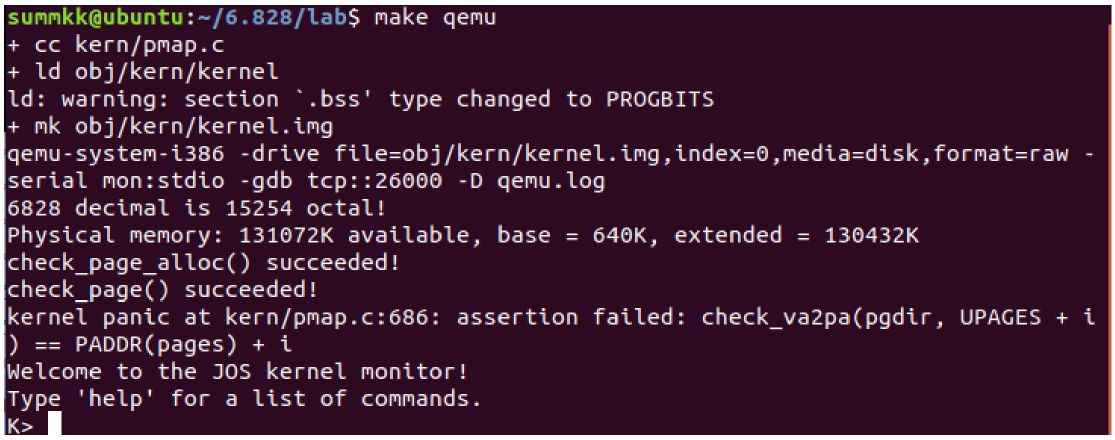

```
summkk@ubuntu:~/6.828/lab$ make qemu
+ cc kern/pmap.c
+ ld obj/kern/kernel
ld: warning: section `.bss' type changed to PROGBITS
+ mk obj/kern/kernel.img
qemu-system-i386 -drive file=obj/kern/kernel.img,index=0,media=disk,format=raw -
serial mon:stdio -gdb tcp::26000 -D qemu.log
6828 decimal is 15254 octal!
Physical memory: 131072K available, base = 640K, extended = 130432K
check_page_alloc() succeeded!
check_page() succeeded!
kernel panic at kern/pmap.c:686: assertion failed: check_va2pa(pgdir, UPAGES + i
) == PADDR(pages) + i
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

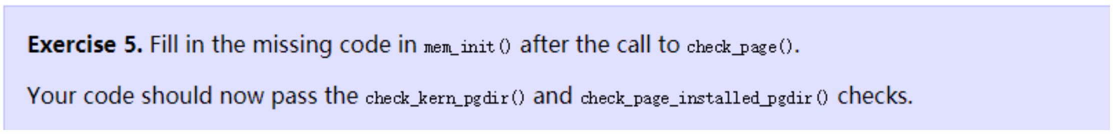
图 2.3 exercise4 测试结果

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：

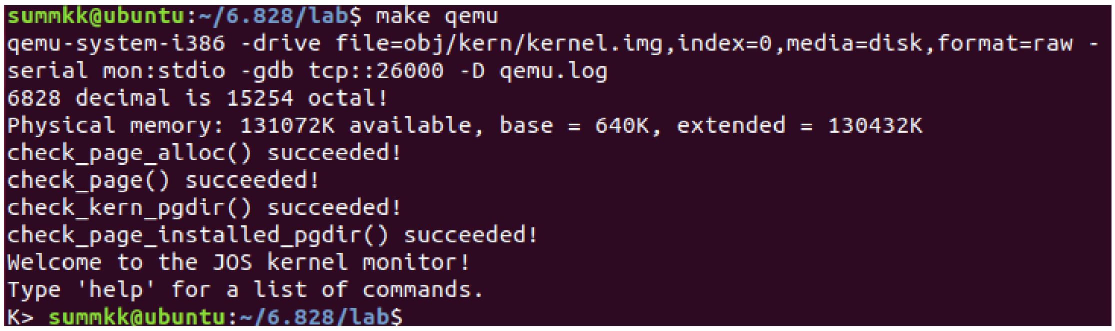

Exercise 5. Fill in the missing code in `mem_init()` after the call to `check_page()`.

Your code should now pass the `check_kern_pgdir()` and `check_page_installed_pgdir()` checks.

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。

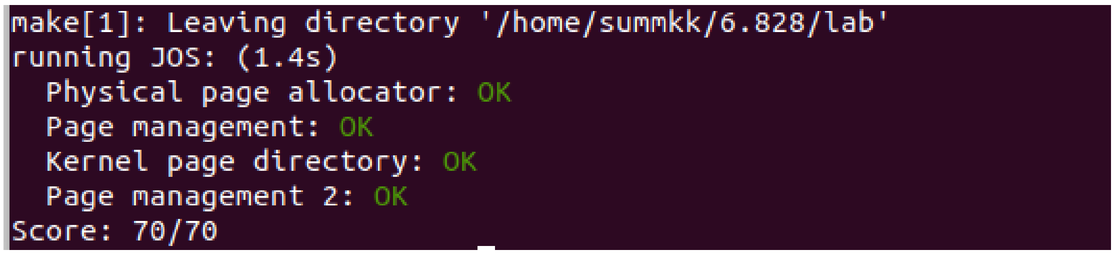
测试结果如图 2.4 所示



```
summkk@ubuntu:~/6.828/lab$ make qemu
qemu-system-i386 -drive file=obj/kern/kernel.img,index=0,media=disk,format=raw -
serial mon:stdio -gdb tcp::26000 -D qemu.log
6828 decimal is 15254 octal!
Physical memory: 131072K available, base = 640K, extended = 130432K
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_installed_pgdir() succeeded!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K> summkk@ubuntu:~/6.828/lab$
```

图 2.4 exercise5 测试

全部 exercise 完成后 make grade 如图 2.5 所示



```
make[1]: Leaving directory '/home/summkk/6.828/lab'
running JOS: (1.4s)
  Physical page allocator: OK
  Page management: OK
  Kernel page directory: OK
  Page management 2: OK
Score: 70/70
```

图 2.5 lab2 测试

3 实验问题

Question

1. Assuming that the following JOS kernel code is correct, what type should variable x have, uintptr_t or physaddr_t?

```
mystery_t x;
char* value = return_a_pointer();
*value = 10;
x = (mystery_t) value;
```

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。

2. What entries (rows) in the page directory have been filled in at this point? What addresses do they map and where do they point? In other words, fill out this table as much as possible:

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	

3. We have placed the kernel and user environment in the same address space. Why will user programs not be able to read or write the kernel's memory? What specific mechanisms protect the kernel memory?

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。

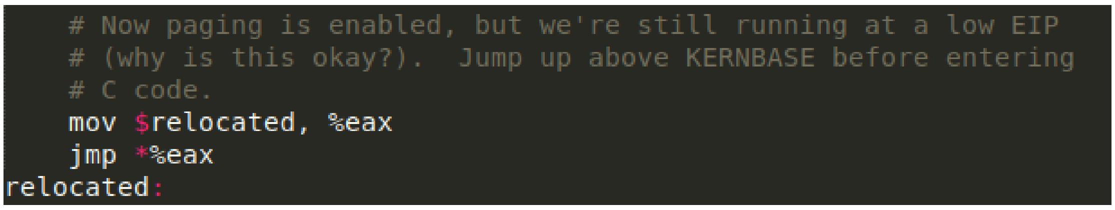
4. What is the maximum amount of physical memory that this operating system can support? Why?

最大可以支持的物理内存为(PTSIZE/sizeof(PageInfo))*2^{12} = 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 2^{12} 字节。

5. How much space overhead is there for managing memory, if we actually had the maximum amount of physical memory? How is this overhead broken down?

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

6. Revisit the page table setup in kern/entry.S and kern/entrypgdir.c. Immediately after we turn on paging, EIP is still a low number (a little over 1MB). At what point do we transition to running at an EIP above KERNBASE? What makes it possible for us to continue executing at a low EIP between when we enable paging and when we begin running at an EIP above KERNBASE? Why is this transition necessary?


```
	# Now paging is enabled, but we're still running at a low EIP
	# (why is this okay?).  Jump up above KERNBASE before entering
	# C code.
	mov	$relocated, %eax
	jmp	*%eax
relocated:
```

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

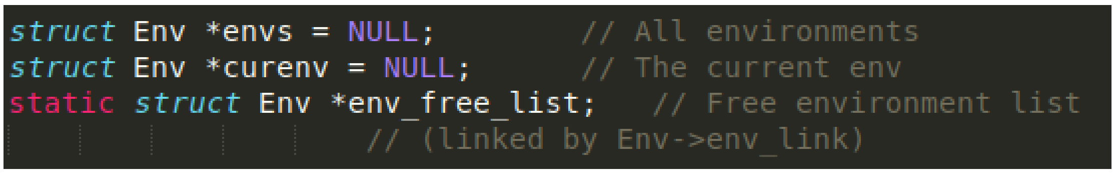
1 实验目的

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

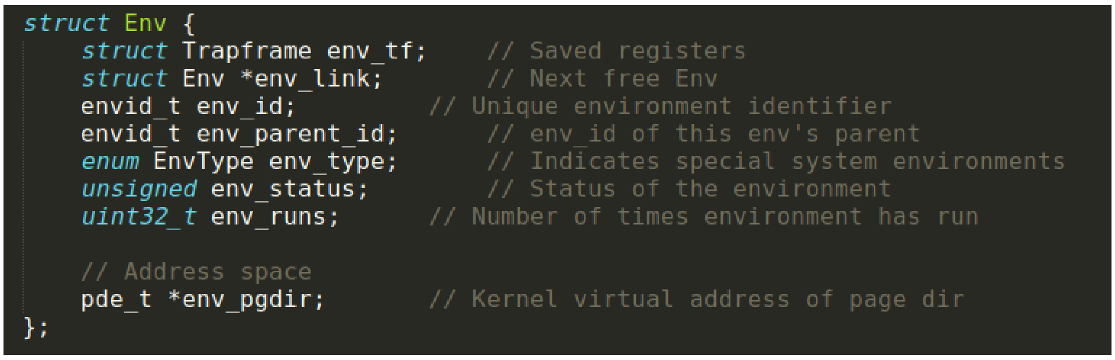
2.1 Part 1: User Environment and Exception Handling

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。



```
struct Env *envs = NULL;          // All environments
struct Env *curenv = NULL;        // The current env
static struct Env *env_free_list;    // Free environment list
                                  // (linked by Env->env_link)
```

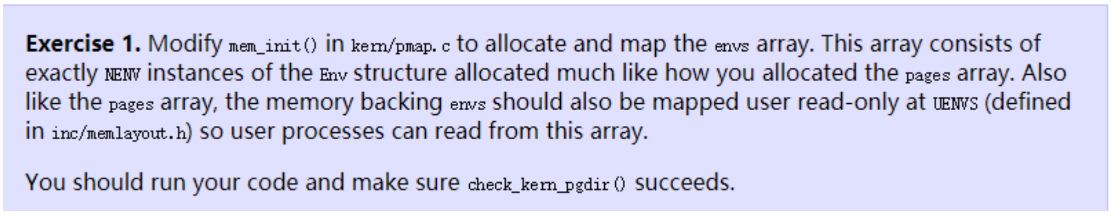
Env 结构以及里面各个字段的用途如图所示。



```
struct Env {
    struct Trapframe env_tf;     // Saved registers
    struct Env *env_link;        // Next free Env
    envid_t env_id;          // Unique environment identifier
    envid_t env_parent_id;       // env_id of this env's parent
    enum EnvType env_type;       // Indicates special system environments
    unsigned env_status;         // Status of the environment
    uint32_t env_runs;       // Number of times environment has run

    // Address space
    pde_t *env_pgdir;        // Kernel virtual address of page dir
};
```

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


Exercise 1. Modify `mem_init()` in `kern/pmap.c` to allocate and map the `envs` array. This array consists of exactly `NENV` instances of the `Env` structure allocated much like how you allocated the `pages` array. Also like the `pages` array, the memory backing `envs` should also be mapped user read-only at `UENVS` (defined in `inc/memlayout.h`) so user processes can read from this array.

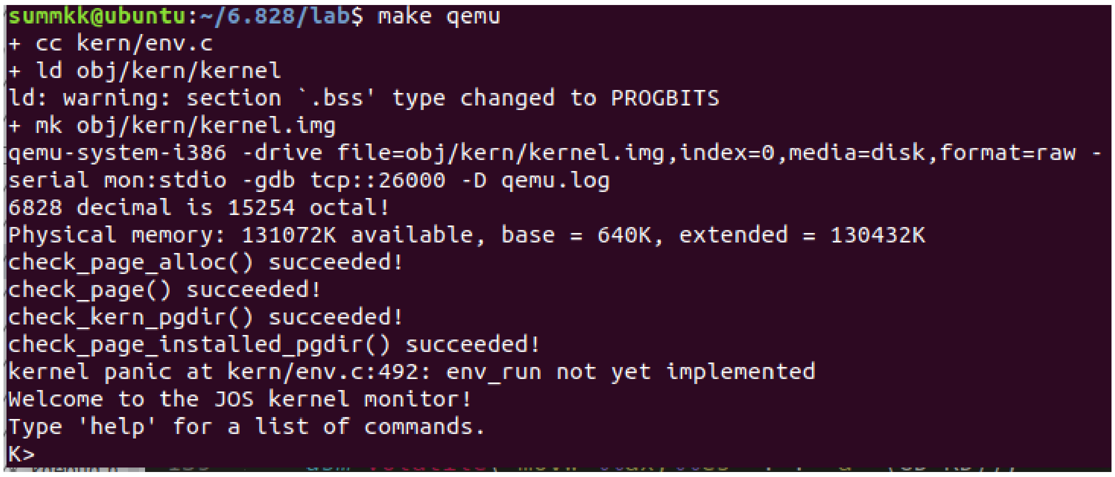
You should run your code and make sure `check_kern_pgdir()` succeeds.

练习 1 要求修改 mem_init 给 envs 数组分配空间。

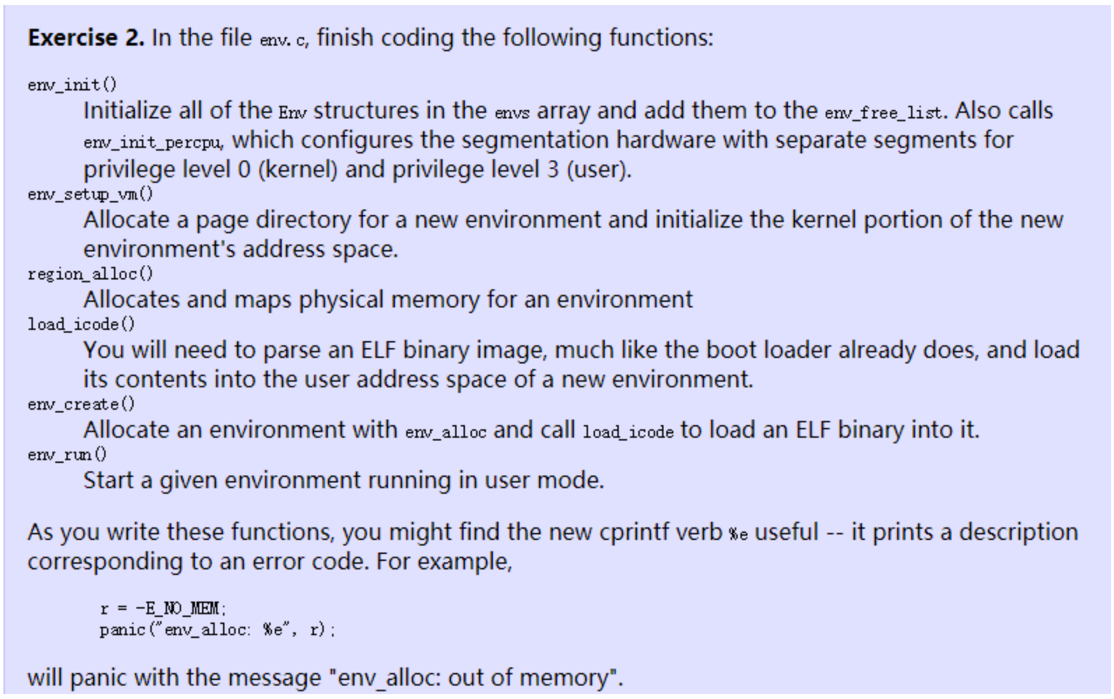
mem_init()

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


```
summkk@ubuntu:~/6.828/lab$ make qemu
+ cc kern/env.c
+ ld obj/kern/kernel
ld: warning: section `.bss' type changed to PROGBITS
+ mk obj/kern/kernel.img
qemu-system-i386 -drive file=obj/kern/kernel.img,index=0,media=disk,format=raw -
serial mon:stdio -gdb tcp::26000 -D qemu.log
6828 decimal is 15254 octal!
Physical memory: 131072K available, base = 640K, extended = 130432K
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_installed_pgdir() succeeded!
kernel panic at kern/env.c:492: env_run not yet implemented
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

Exercise2


Exercise 2. In the file env.c, finish coding the following functions:

env_init()
Initialize all of the Env structures in the envs array and add them to the env_free_list. Also calls env_init_percpu, which configures the segmentation hardware with separate segments for privilege level 0 (kernel) and privilege level 3 (user).

env_setup_vm()
Allocate a page directory for a new environment and initialize the kernel portion of the new environment's address space.

region_alloc()
Allocates and maps physical memory for an environment

load_icode()
You will need to parse an ELF binary image, much like the boot loader already does, and load its contents into the user address space of a new environment.

env_create()
Allocate an environment with env_alloc and call load_icode to load an ELF binary into it.

env_run()
Start a given environment running in user mode.

As you write these functions, you might find the new cprintf verb %e useful -- it prints a description corresponding to an error code. For example,

```
    r = -E_NO_MEM;
    panic("env_alloc: %e", r);
```

will panic with the message "env_alloc: out of memory".

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()

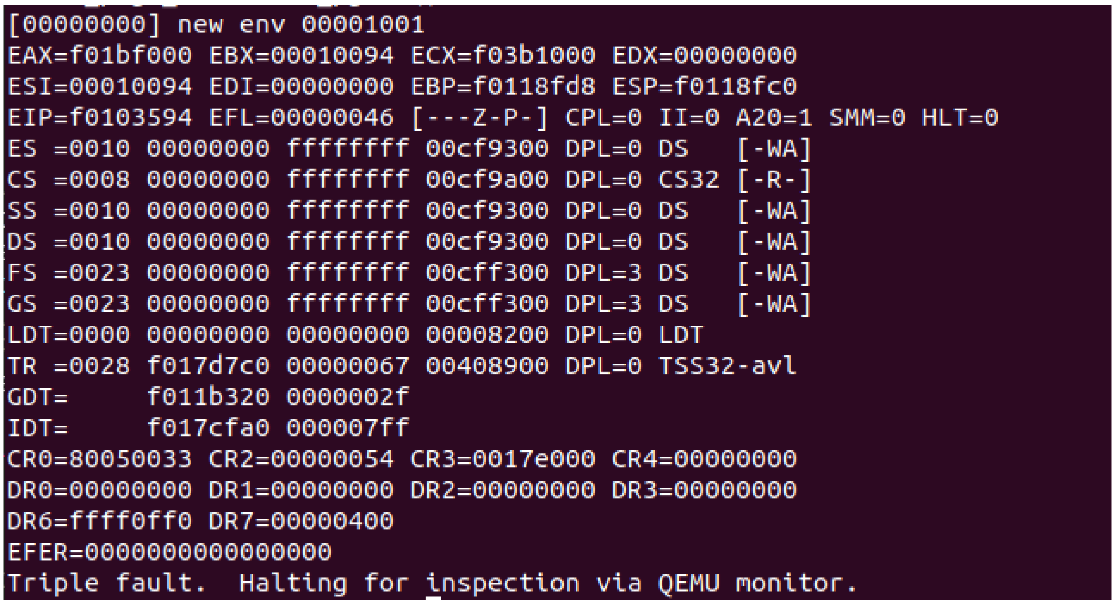
env_create()函数使用 env_alloc 分配环境并调用 load_icode 以将 ELF 二进制文件加载到其中。这个函数仅在 kernel 初始化的时候调用，在运行第一个用户程序之前。新的 env 的父 ID 设为 0.

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。

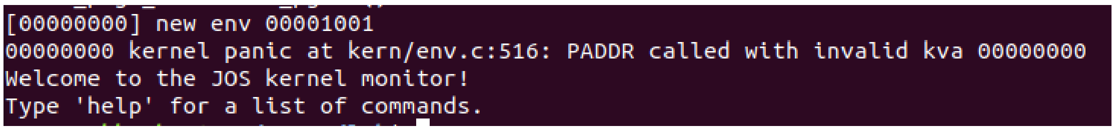
根据注释里的提示可以看到，首先需要设置当前的 curenv 的状态为 ENV_RUNABLE，将 e 设置为当前的 curenv，设置它的状态为 ENV_RUNNING,更新 env_runs 计数器然后调用 lcr3 函数切换地址空间。lcr3()这个函数，通过 lcr3 指令把页目录表的起始地址存入 CR3 寄存器。这里的拷贝到指定的虚地址处，是指用户空间的虚地址，而不是内核空间的虚地址，所以还需要用 lcr3 函数加载用户空间的页表目录才能将地址转换为用户空间地址。然后使用 env_pop_tf()恢复环境的寄存器并进入环境中的用户模式。

全部代码编写完成后执行会引发段错误如图



```
[00000000] new env 00001001
EAX=f01bf000 EBX=00010094 ECX=f03b1000 EDX=00000000
ESI=00010094 EDI=00000000 EBP=f0118fd8 ESP=f0118fc0
EIP=f0103594 EFL=00000046 [---Z-P-] CPL=0 II=0 A20=1 SMM=0 HLT=0
ES =0010 00000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
CS =0008 00000000 ffffffff 00cf9a00 DPL=0 CS32 [-R-]
SS =0010 00000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
DS =0010 00000000 ffffffff 00cf9300 DPL=0 DS   [-WA]
FS =0023 00000000 ffffffff 00cff300 DPL=3 DS   [-WA]
GS =0023 00000000 ffffffff 00cff300 DPL=3 DS   [-WA]
LDT=0000 00000000 00000000 00008200 DPL=0 LDT
TR =0028 f017d7c0 00000067 00408900 DPL=0 TSS32-avl
GDT=     f011b320 0000002f
IDT=     f017cfa0 000007ff
CR0=80050033 CR2=00000054 CR3=0017e000 CR4=00000000
DR0=00000000 DR1=00000000 DR2=00000000 DR3=00000000
DR6=ffff0ff0 DR7=00000400
EFER=0000000000000000
Triple fault.  Halting for inspection via QEMU monitor.
```

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。



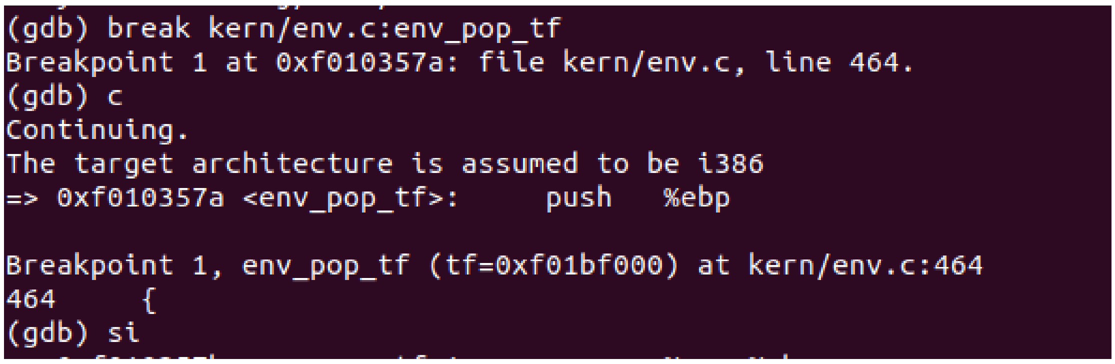
```
[00000000] new env 00001001
00000000 kernel panic at kern/env.c:516: PADDR called with invalid kva 00000000
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
```

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。

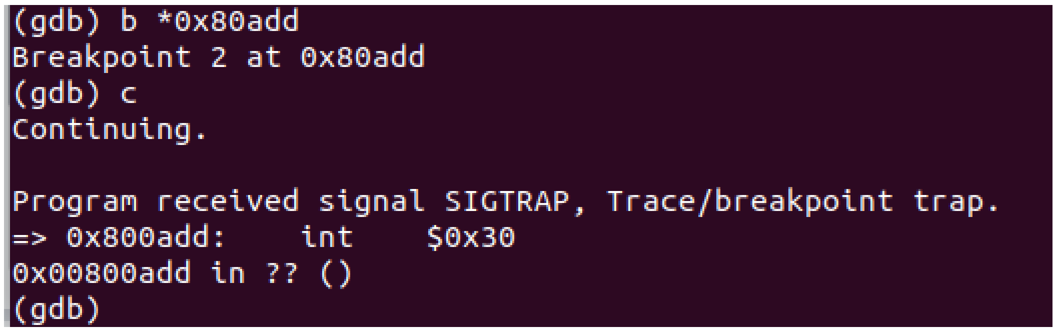
这个是一个 user code 在运行的时候的调用图，如果执行正确的话系统应该进入用户环境并执行 hello 的二进制文件。

- start (kern/entry.S)
- i386_init (kern/init.c)
  - cons_init
  - mem_init
  - env_init
  - trap_init (still incomplete at this point)
  - env_create
  - env_run
    - env_pop_tf

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。



```
(gdb) break kern/env.c:env_pop_tf
Breakpoint 1 at 0xf010357a: file kern/env.c, line 464.
(gdb) c
Continuing.
The target architecture is assumed to be i386
=> 0xf010357a <env_pop_tf>:      push   %ebp

Breakpoint 1, env_pop_tf (tf=0xf01bf000) at kern/env.c:464
464     {
(gdb) si
```



```
(gdb) b *0x80add
Breakpoint 2 at 0x80add
(gdb) c
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
=> 0x800add:    int    $0x30
0x00800add in ?? ()
(gdb)
```

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

Exercise 3. Read Chapter 9, Exceptions and Interrupts in the 80386 Programmer's Manual (or Chapter 5 of the IA-32 Developer's Manual), if you haven't already.

了解关于异常和中断的基本内容。

Exercise4

Exercise 4. Edit `trapentry.S` and `trap.c` and implement the features described above. The macros `TRAPHANDLER` and `TRAPHANDLER_NOEC` in `trapentry.S` should help you, as well as the T_* defines in `inc/trap.h`. You will need to add an entry point in `trapentry.S` (using those macros) for each trap defined in `inc/trap.h`, and you'll have to provide `_alltraps` which the `TRAPHANDLER` macros refer to. You will also need to modify `trap_init()` to initialize the `idt` to point to each of these entry points defined in `trapentry.S`; the `SETGATE` macro will be helpful here.

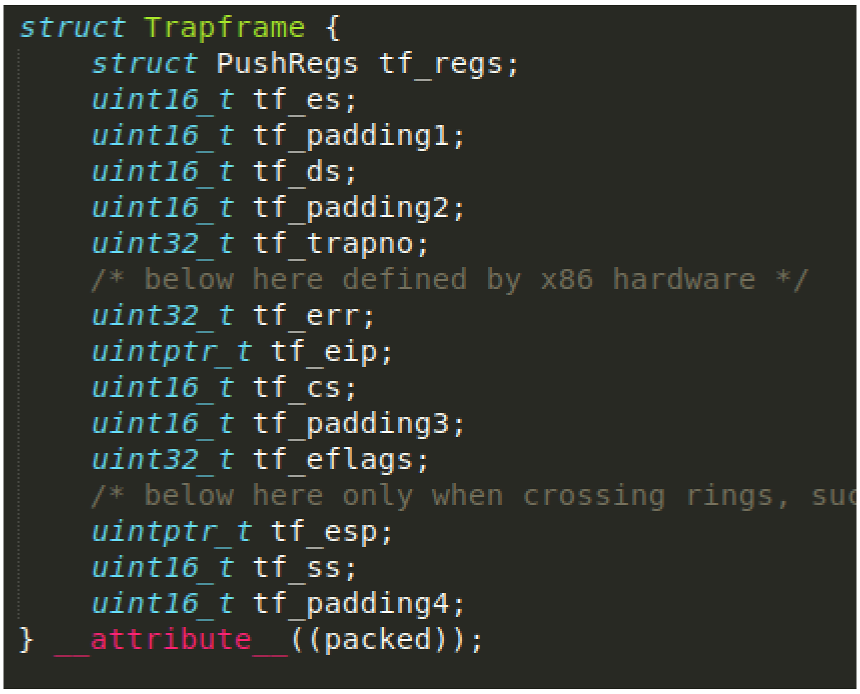
Your `_alltraps` should:

1. push values to make the stack look like a struct Trapframe
2. load `GD_KD` into `%ds` and `%es`
3. `pushl %esp` to pass a pointer to the Trapframe as an argument to trap()
4. `call trap` (can `trap` ever return?)

Consider using the `pushal` instruction; it fits nicely with the layout of the `struct Trapframe`.

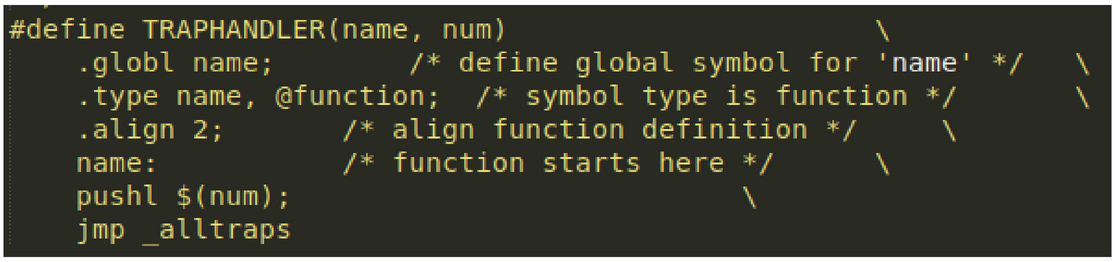
Test your trap handling code using some of the test programs in the `user` directory that cause exceptions before making any system calls, such as `user/divzero`. You should be able to get `make grade` to succeed on the `divzero`, `softint`, and `badsegment` tests at this point.

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。



```
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, suc
    uintptr_t tf_esp;
    uint16_t tf_ss;
    uint16_t tf_padding4;
} __attribute__((packed));
```

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。



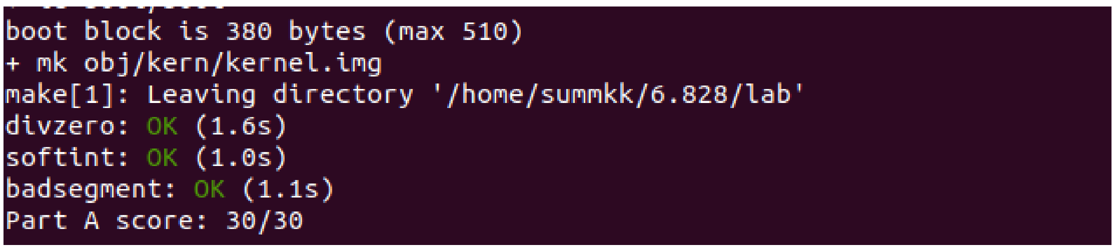
```
#define TRAPHANDLER(name, num)                                  \
    .globl name;          /* define global symbol for 'name' */   \
    .type name, @function;   /* symbol type is function */        \
    .align 2;         /* align function definition */       \
    name:             /* function starts here */       \
    pushl $(num);                                   \
    jmp _alltraps
```

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

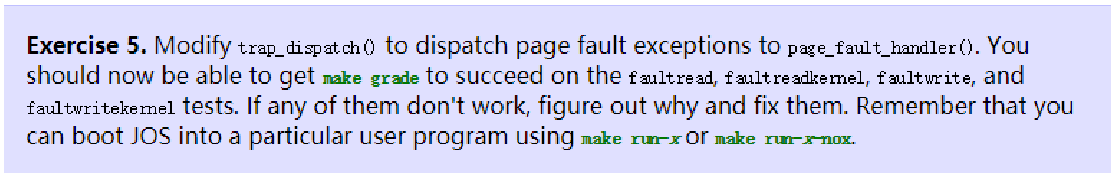


```
boot block is 380 bytes (max 510)
+ mk obj/kern/kernel.img
make[1]: Leaving directory '/home/summkk/6.828/lab'
divzero: OK (1.6s)
softint: OK (1.0s)
badsegment: OK (1.1s)
Part A score: 30/30
```

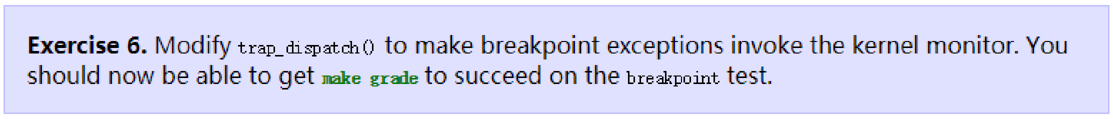
2.2 Part 2: Page Fault, Breakpoints Exception, and System Call

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:



Exercise 5. Modify `trap_dispatch()` to dispatch page fault exceptions to `page_fault_handler()`. You should now be able to get `make grade` to succeed on the `faultread`, `faultreadkernel`, `faultwrite`, and `faultwritekernel` tests. If any of them don't work, figure out why and fix them. Remember that you can boot JOS into a particular user program using `make run-x` or `make run-x-nox`.

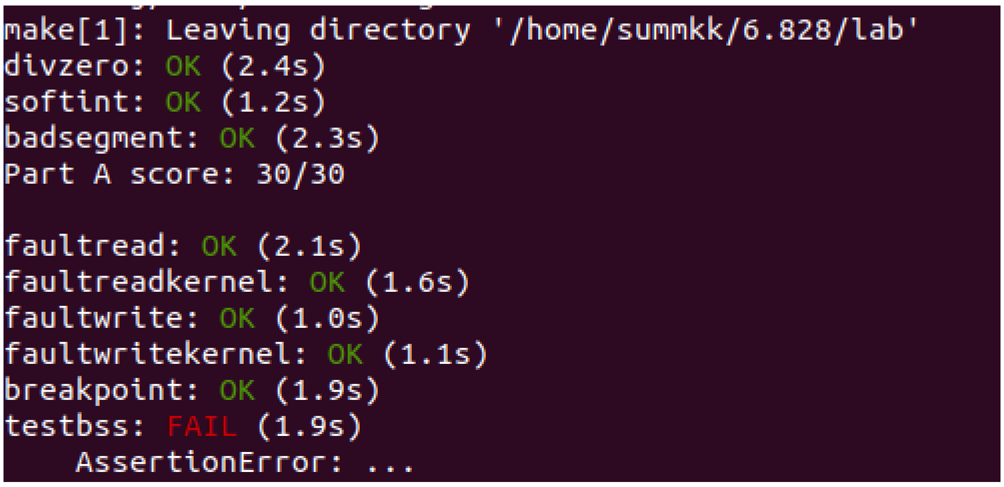


Exercise 6. Modify `trap_dispatch()` to make breakpoint exceptions invoke the kernel monitor. You should now be able to get `make grade` to succeed on the `breakpoint` test.

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。

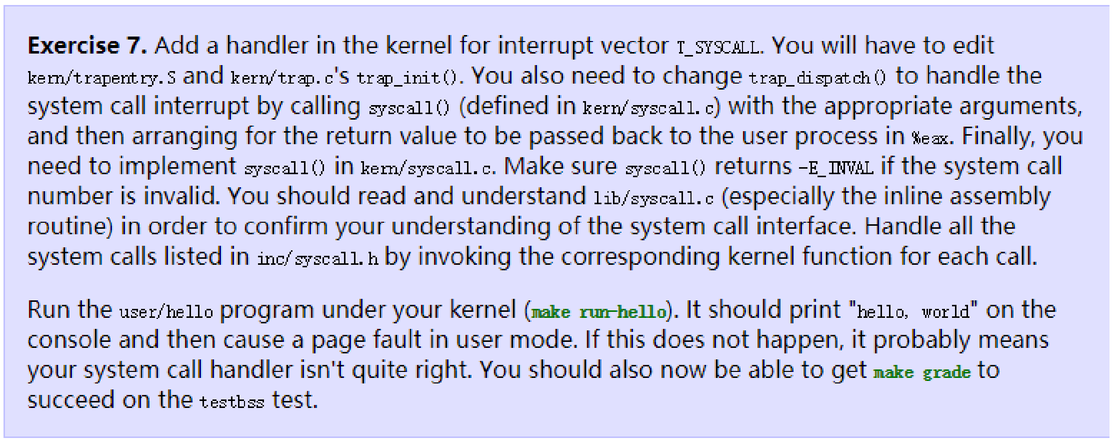
因为 exercise5 和 6 都是修改 trap_dispatch()函数调用对应的处理函数，处理方式差不多，放在一起写之后进行测试。Exercise6 需要注意断点异常的特权级为用户级，即 SETGATE 的最后一个参数为 3，否则会引发保护异常。

make grade 测试通过断点之前的测试，结果如图



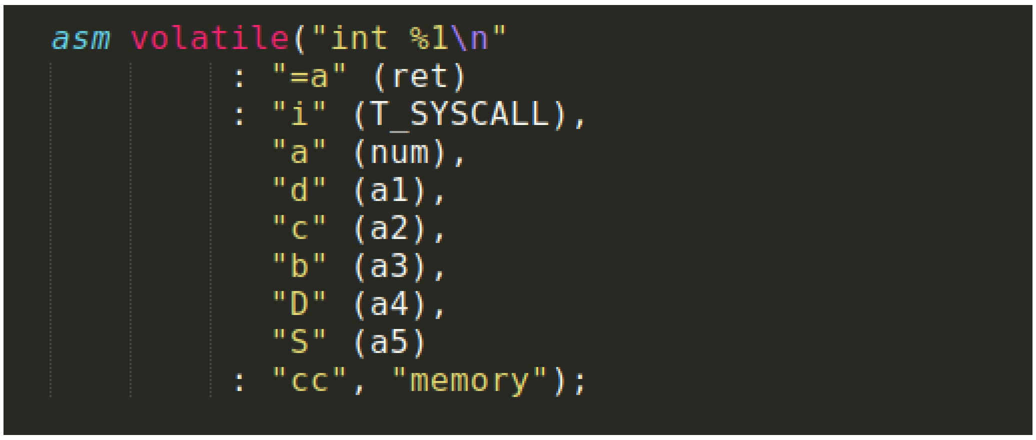
```
make[1]: Leaving directory '/home/summkk/6.828/lab'
divzero: OK (2.4s)
softint: OK (1.2s)
badsegment: OK (2.3s)
Part A score: 30/30

faultread: OK (2.1s)
faultreadkernel: OK (1.6s)
faultwrite: OK (1.0s)
faultwritekernel: OK (1.1s)
breakpoint: OK (1.9s)
testbss: FAIL (1.9s)
    AssertionError: ...
```

Exercise7：

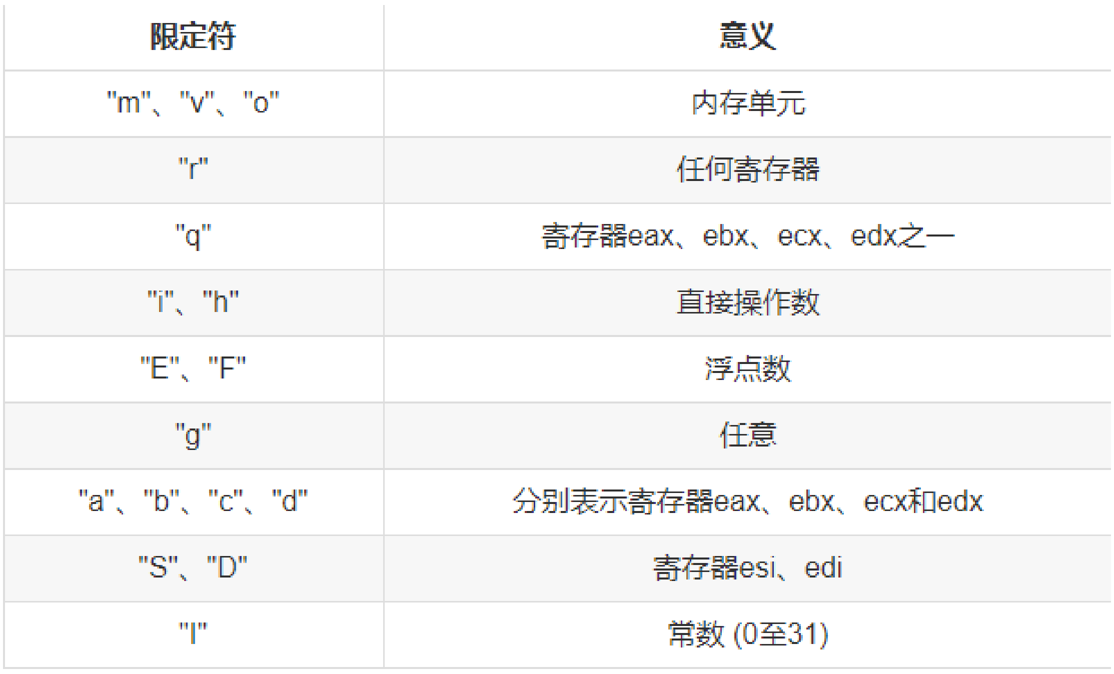

Exercise 7. Add a handler in the kernel for interrupt vector `T_SYSCALL`. You will have to edit `kern/trapentry.S` and `kern/trap.c`'s `trap_init()`. You also need to change `trap_dispatch()` to handle the system call interrupt by calling `syscall()` (defined in `kern/syscall.c`) with the appropriate arguments, and then arranging for the return value to be passed back to the user process in `%eax`. Finally, you need to implement `syscall()` in `kern/syscall.c`. Make sure `syscall()` returns `-E_INVAL` if the system call number is invalid. You should read and understand `lib/syscall.c` (especially the inline assembly routine) in order to confirm your understanding of the system call interface. Handle all the system calls listed in `inc/syscall.h` by invoking the corresponding kernel function for each call.

Run the `user/hello` program under your kernel (`make run-hello`). It should print "`hello, world`" on the console and then cause a page fault in user mode. If this does not happen, it probably means your system call handler isn't quite right. You should also now be able to get `make grade` to succeed on the `testbss` test.

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。


```
asm volatile("int %1\n"
		: "=a" (ret)
		: "i" (T_SYSCALL),
		  "a" (num),
		  "d" (a1),
		  "c" (a2),
		  "b" (a3),
		  "D" (a4),
		  "S" (a5)
		: "cc", "memory");
```

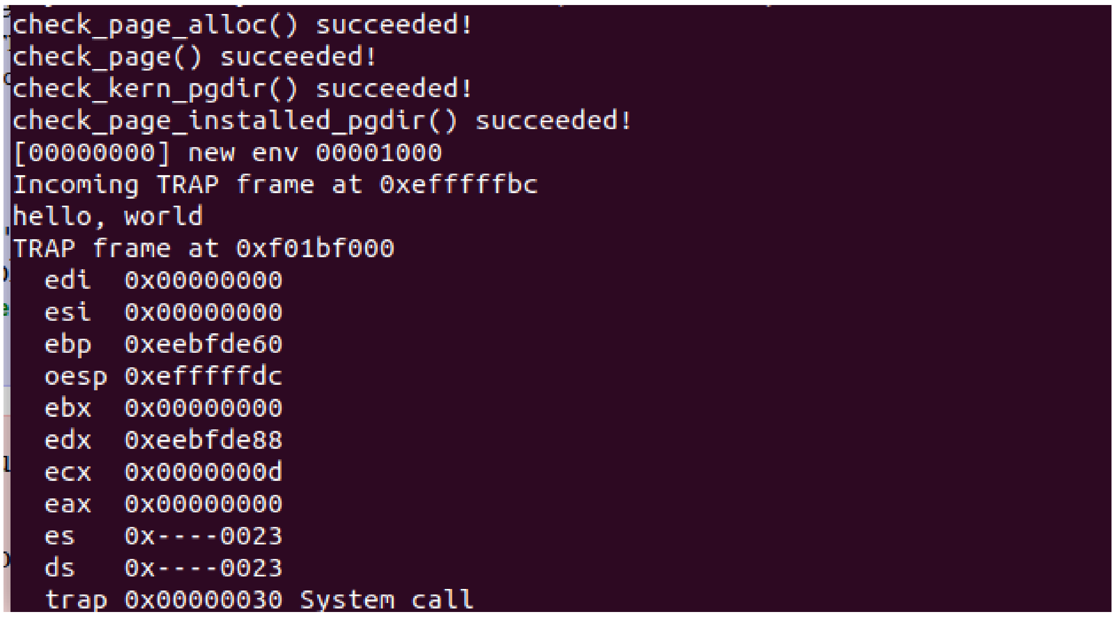
参考内联汇编的语法可以知道压入的参数然后在 trap_dispatch()函数中加入对应的调用。



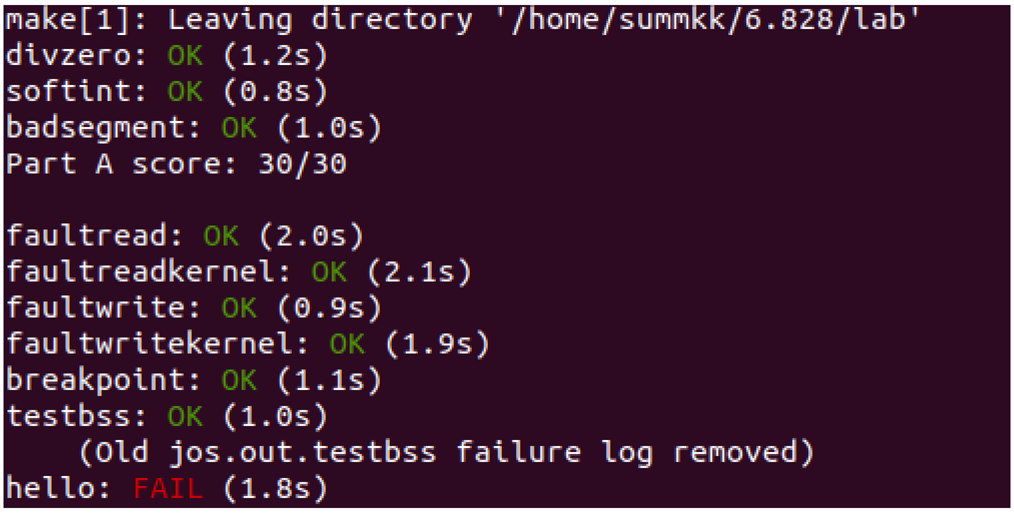
限定符	意义
"m"、"v"、"o"	内存单元
"r"	任何寄存器
"q"	寄存器eax、ebx、ecx、edx之一
"i"、"h"	直接操作数
"E"、"F"	浮点数
"g"	任意
"a"、"b"、"c"、"d"	分别表示寄存器eax、ebx、ecx和edx
"S"、"D"	寄存器esi、edi
"I"	常数 (0至31)

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

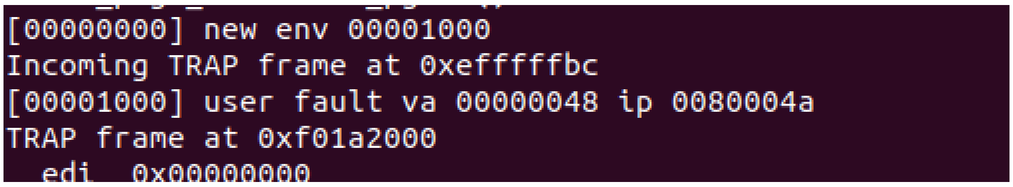


```
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_installed_pgdir() succeeded!
[00000000] new env 00001000
Incoming TRAP frame at 0xefffffbc
hello, world
TRAP frame at 0xf01bf000
  edi  0x00000000
  esi  0x00000000
  ebp  0xeebfde60
  oesp 0xefffffdc
  ebx  0x00000000
  edx  0xeebfde88
  ecx  0x0000000d
  eax  0x00000000
  es   0x----0023
  ds   0x----0023
  trap 0x00000030 System call
```



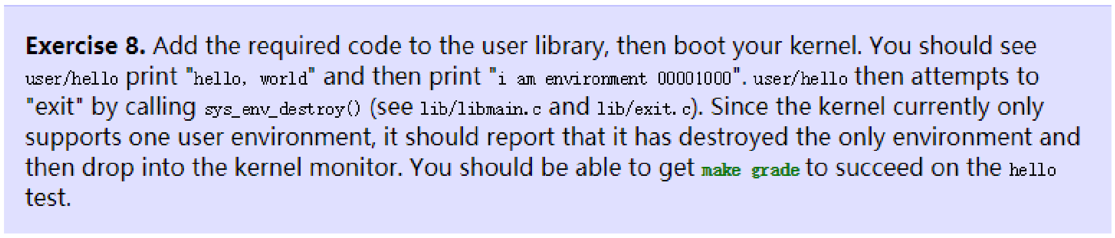
```
make[1]: Leaving directory '/home/summkk/6.828/lab'
divzero: OK (1.2s)
softint: OK (0.8s)
badsegment: OK (1.0s)
Part A score: 30/30

faultread: OK (2.0s)
faultreadkernel: OK (2.1s)
faultwrite: OK (0.9s)
faultwritekernel: OK (1.9s)
breakpoint: OK (1.1s)
testbss: OK (1.0s)
    (Old jos.out.testbss failure log removed)
hello: FAIL (1.8s)
```

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。

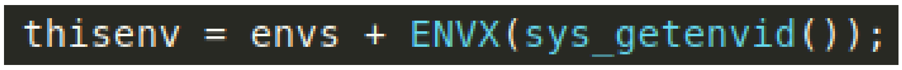


```
[00000000] new env 00001000
Incoming TRAP frame at 0xefffffbc
[00001000] user fault va 00000048 ip 0080004a
TRAP frame at 0xf01a2000
  edi  0x00000000
```

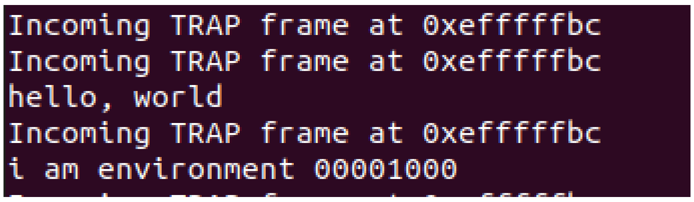
Exercise8：


Exercise 8. Add the required code to the user library, then boot your kernel. You should see `user/hello` print "`hello, world`" and then print "`i am environment 00001000`". `user/hello` then attempts to "exit" by calling `sys_env_destroy()` (see `lib/libmain.c` and `lib/exit.c`). Since the kernel currently only supports one user environment, it should report that it has destroyed the only environment and then drop into the kernel monitor. You should be able to get `make grade` to succeed on the `hello` test.

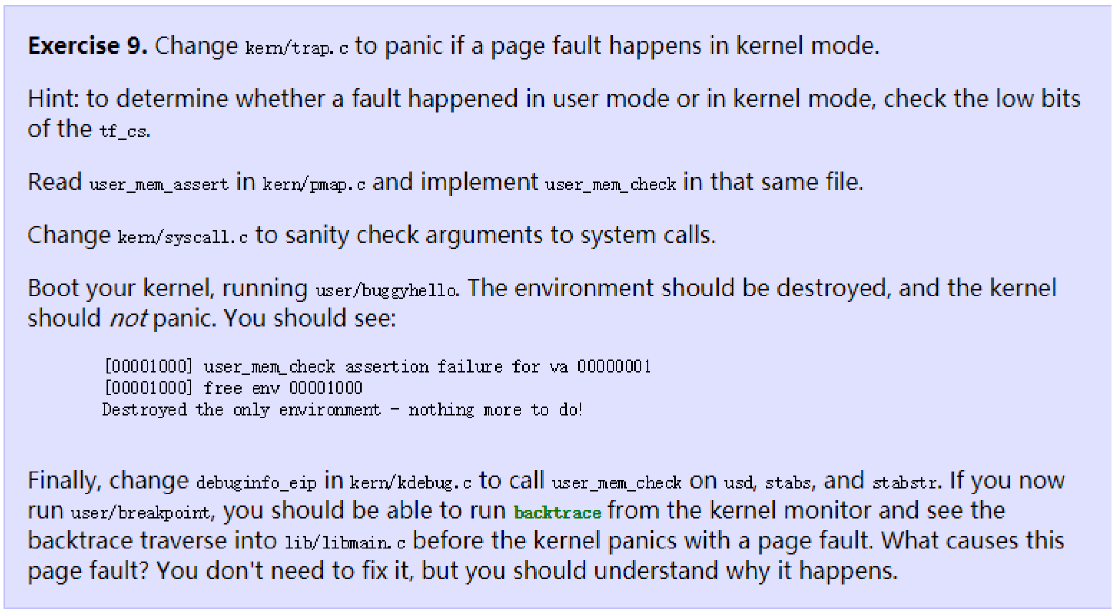
在 lic/libmain.c 中设置当前活动的进程即可


```
thisenv = envs + ENVX(sys_getenvid());
```

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


```
Incoming TRAP frame at 0xefffffbc
Incoming TRAP frame at 0xefffffbc
hello, world
Incoming TRAP frame at 0xefffffbc
i am environment 00001000
```

Exercise9：


Exercise 9. Change `kern/trap.c` to panic if a page fault happens in kernel mode.

Hint: to determine whether a fault happened in user mode or in kernel mode, check the low bits of the `tf_cs`.

Read `user_mem_assert` in `kern/pmap.c` and implement `user_mem_check` in that same file.

Change `kern/syscall.c` to sanity check arguments to system calls.

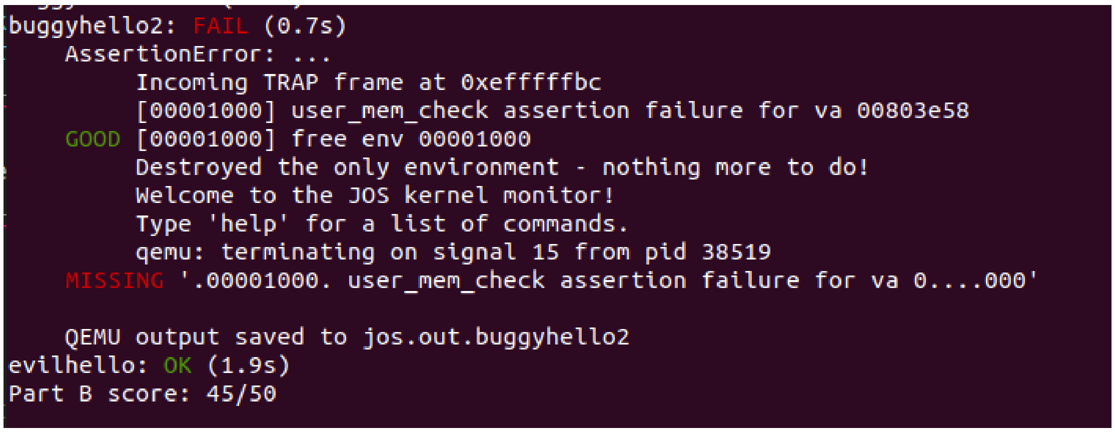
Boot your kernel, running `user/buggyhello`. The environment should be destroyed, and the kernel should *not* panic. You should see:

```
	[00001000] user_mem_check assertion failure for va 00000001
	[00001000] free env 00001000
	Destroyed the only environment - nothing more to do!
```

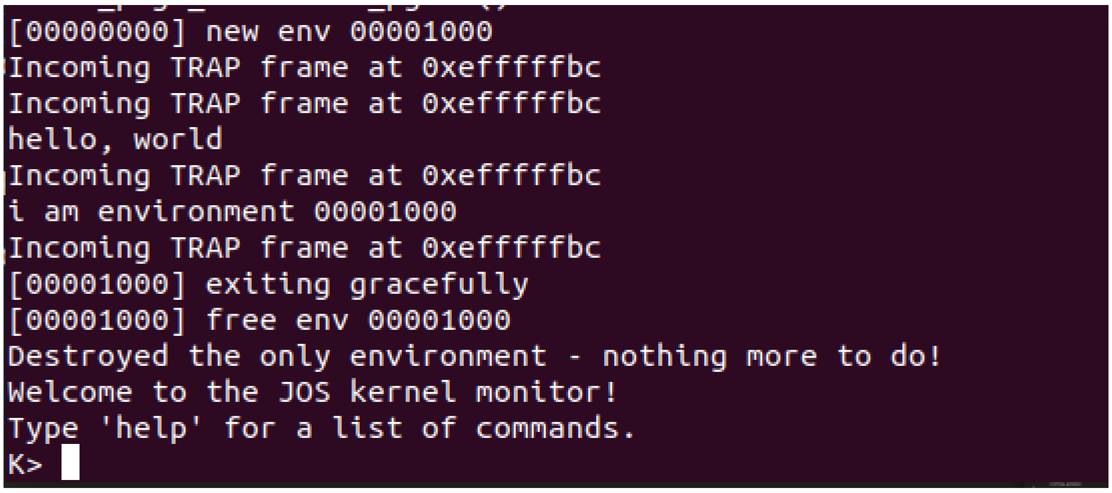
Finally, change `debuginfo_eip` in `kern/kdebug.c` to call `user_mem_check` on `usd`, `stabs`, and `stabstr`. If you now run `user/breakpoint`, you should be able to run `backtrace` from the kernel monitor and see the backtrace traverse into `lib/libmain.c` before the kernel panics with a page fault. What causes this page fault? You don't need to fix it, but you should understand why it happens.

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。

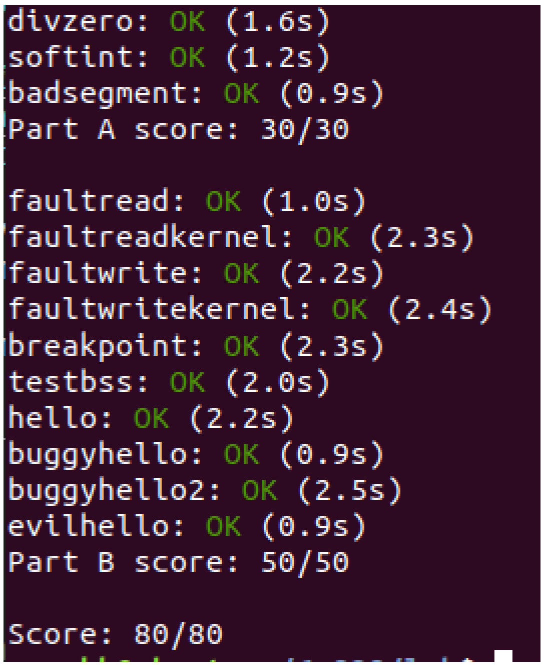
填写 pmap.c 里的内存检查函数。需要检查内存[va,va+len]是否有权限，va 和 len 不一定是页面对齐的。用户可以访问的虚拟地址要求地址低于 ULIM 并且在页表中有相应的权限。最开始用 page_lookup 因为地址对齐的问题导致无法通过测试如图。修改用 pgdir_walk 之后完成。


```
buggyhello2: FAIL (0.7s)
    AssertionError: ...
         Incoming TRAP frame at 0xefffffbc
         [00001000] user_mem_check assertion failure for va 00803e58
    GOOD [00001000] free env 00001000
         Destroyed the only environment - nothing more to do!
         Welcome to the JOS kernel monitor!
         Type 'help' for a list of commands.
         qemu: terminating on signal 15 from pid 38519
    MISSING '.00001000. user_mem_check assertion failure for va 0....000'

    QEMU output saved to jos.out.buggyhello2
evilhello: OK (1.9s)
Part B score: 45/50
```

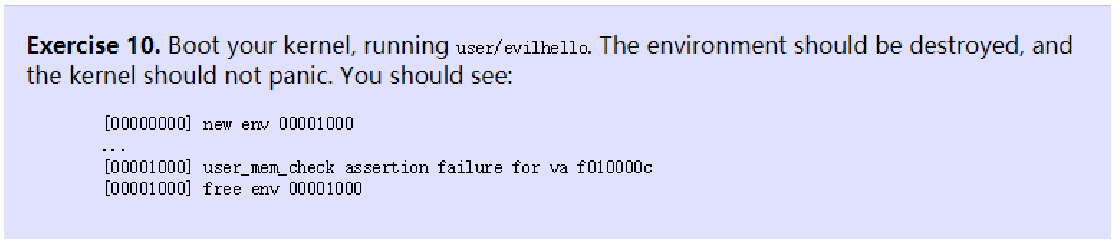
在 kernel/syscall.c 中的 sys_cputs()，kern/kdebug.c 中的 debuginfo_eip()中加入 user_mem_check()函数进行内存检查。完成后执行 make run-buggyhello 产生输出如图


```
[00000000] new env 00001000
Incoming TRAP frame at 0xefffffbc
Incoming TRAP frame at 0xefffffbc
hello, world
Incoming TRAP frame at 0xefffffbc
i am environment 00001000
Incoming TRAP frame at 0xefffffbc
[00001000] exiting gracefully
[00001000] free env 00001000
Destroyed the only environment - nothing more to do!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

make grade 测试结果如图


```
divzero: OK (1.6s)
softint: OK (1.2s)
badsegment: OK (0.9s)
Part A score: 30/30

faultread: OK (1.0s)
faultreadkernel: OK (2.3s)
faultwrite: OK (2.2s)
faultwritekernel: OK (2.4s)
breakpoint: OK (2.3s)
testbss: OK (2.0s)
hello: OK (2.2s)
buggyhello: OK (0.9s)
buggyhello2: OK (2.5s)
evilhello: OK (0.9s)
Part B score: 50/50

Score: 80/80
```

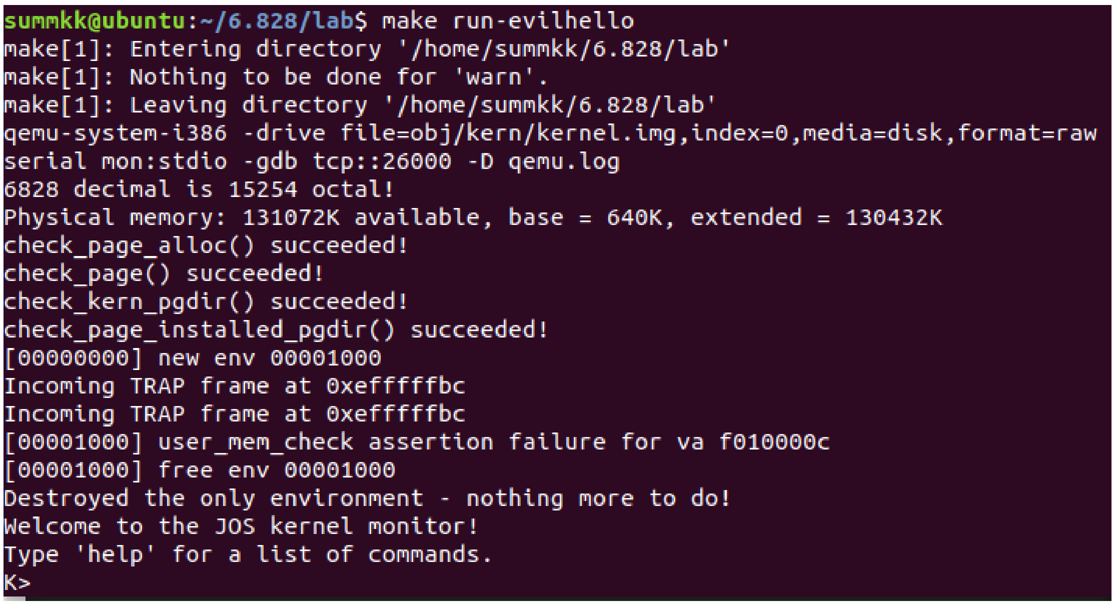
Exercise10：



Exercise 10. Boot your kernel, running `user/evilhello`. The environment should be destroyed, and the kernel should not panic. You should see:

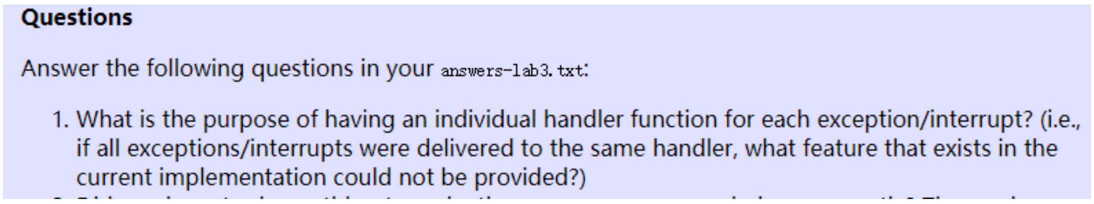
```
[00000000] new env 00001000
...
[00001000] user_mem_check assertion failure for va f010000c
[00001000] free env 00001000
```

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



```
summkk@ubuntu:~/6.828/lab$ make run-evilhello
make[1]: Entering directory '/home/summkk/6.828/lab'
make[1]: Nothing to be done for 'warn'.
make[1]: Leaving directory '/home/summkk/6.828/lab'
qemu-system-i386 -drive file=obj/kern/kernel.img,index=0,media=disk,format=raw
serial mon:stdio -gdb tcp::26000 -D qemu.log
6828 decimal is 15254 octal!
Physical memory: 131072K available, base = 640K, extended = 130432K
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_installed_pgdir() succeeded!
[00000000] new env 00001000
Incoming TRAP frame at 0xefffffbc
Incoming TRAP frame at 0xefffffbc
[00001000] user_mem_check assertion failure for va f010000c
[00001000] free env 00001000
Destroyed the only environment - nothing more to do!
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K>
```

3 实验问题

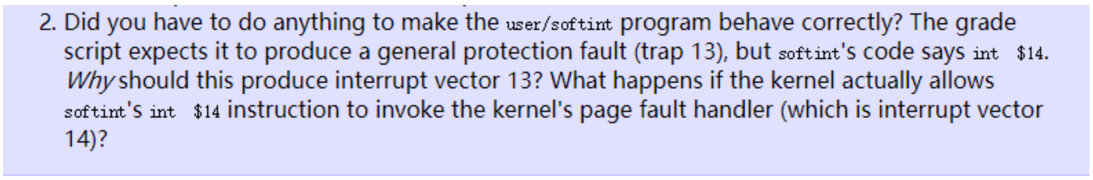


Questions

Answer the following questions in your `answers-lab3.txt`:

1. What is the purpose of having an individual handler function for each exception/interrupt? (i.e., if all exceptions/interrupts were delivered to the same handler, what feature that exists in the current implementation could not be provided?)

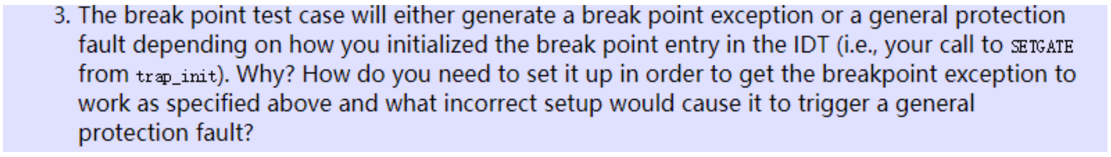
因为每个中断和异常的处理方式不同，特权级也不同，不能用同一个 handler 进行处理。



2. Did you have to do anything to make the `user/softint` program behave correctly? The grade script expects it to produce a general protection fault (trap 13), but `softint`'s code says `int $14`. *Why* should this produce interrupt vector 13? What happens if the kernel actually allows `softint`'s `int $14` instruction to invoke the kernel's page fault handler (which is interrupt vector 14)?

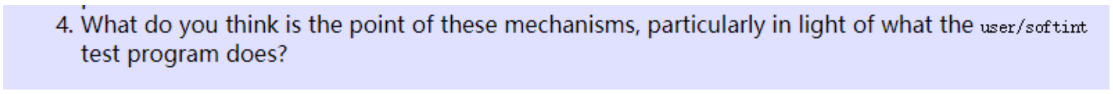
softint 期望产生一个缺页异常 14，但是实际产生的是通用保护异常 13。因为目前系统在用户态，权限为 3，而 int 系统调用的权限为 0。因此会产生通用并

保护异常。



3. The break point test case will either generate a break point exception or a general protection fault depending on how you initialized the break point entry in the IDT (i.e., your call to `SETGATE` from `trap_init`). Why? How do you need to set it up in order to get the breakpoint exception to work as specified above and what incorrect setup would cause it to trigger a general protection fault?

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。



4. What do you think is the point of these mechanisms, particularly in light of what the `user/softint` test program does?

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

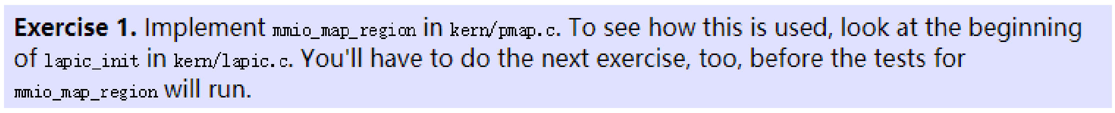
1 实验目的

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


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.

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。

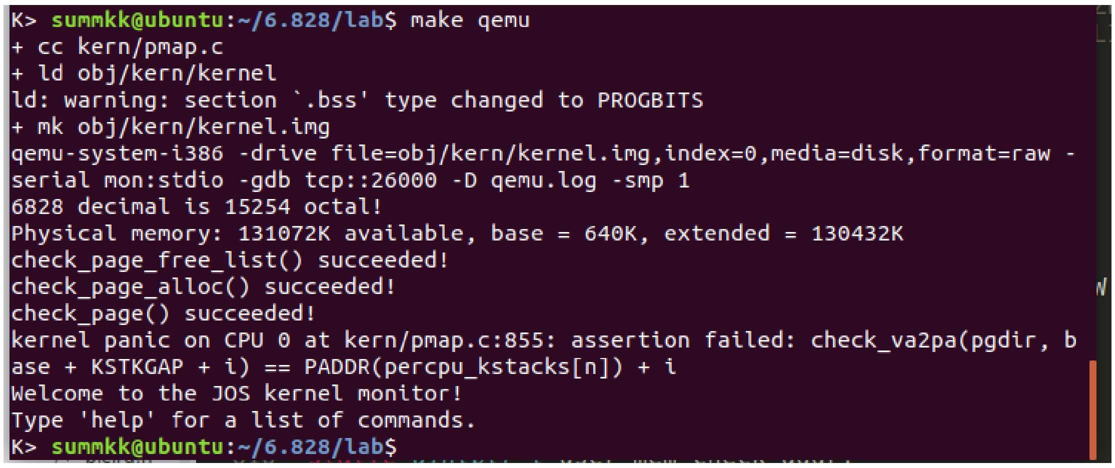
Exercise2

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).

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。



```
K> summkk@ubuntu:~/6.828/lab$ make qemu
+ cc kern/pmap.c
+ ld obj/kern/kernel
ld: warning: section `.bss' type changed to PROGBITS
+ mk obj/kern/kernel.img
qemu-system-i386 -drive file=obj/kern/kernel.img,index=0,media=disk,format=raw -
serial mon:stdio -gdb tcp::26000 -D qemu.log -smp 1
6828 decimal is 15254 octal!
Physical memory: 131072K available, base = 640K, extended = 130432K
check_page_free_list() succeeded!
check_page_alloc() succeeded!
check_page() succeeded!
kernel panic on CPU 0 at kern/pmap.c:855: assertion failed: check_va2pa(pgdir, b
ase + KSTKGAP + i) == PADDR(percpu_kstacks[n]) + i
Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K> summkk@ubuntu:~/6.828/lab$
```

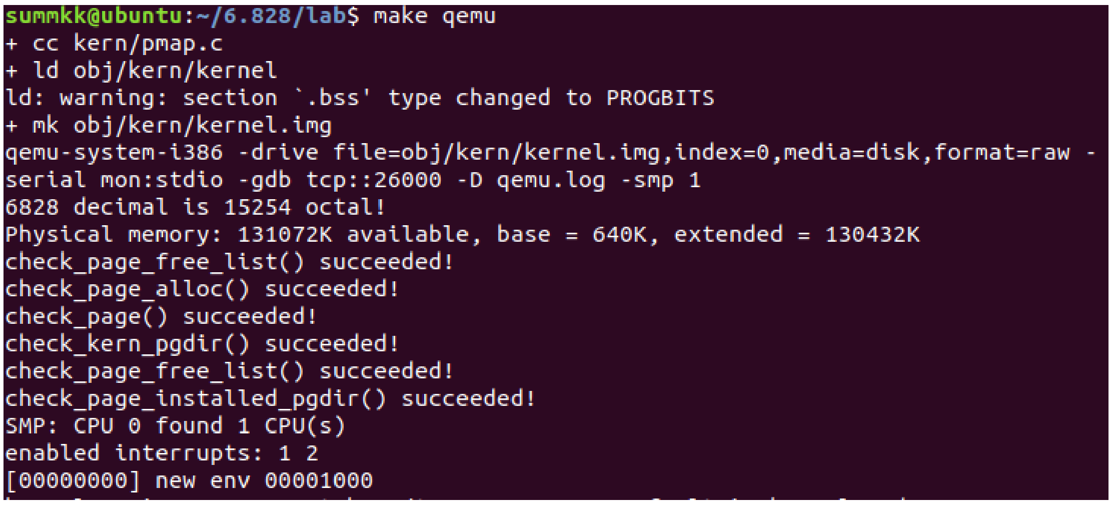
Exercise3：

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().

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。

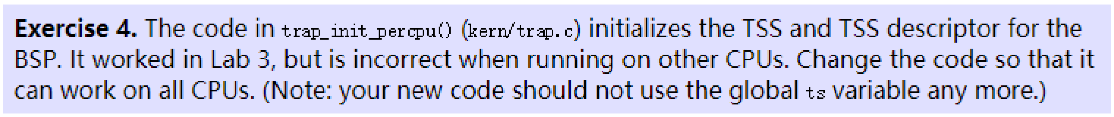
同样可以通过 boot_map_region()函数进行内存的映射，每次栈的地址向下增长即可。

完成 exercise3 之后 check_kern_pgdir()成功。



```
summkk@ubuntu:~/6.828/lab$ make qemu
+ cc kern/pmap.c
+ ld obj/kern/kernel
ld: warning: section `.bss' type changed to PROGBITS
+ mk obj/kern/kernel.img
qemu-system-i386 -drive file=obj/kern/kernel.img,index=0,media=disk,format=raw -
serial mon:stdio -gdb tcp::26000 -D qemu.log -smp 1
6828 decimal is 15254 octal!
Physical memory: 131072K available, base = 640K, extended = 130432K
check_page_free_list() succeeded!
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_free_list() succeeded!
check_page_installed_pgdir() succeeded!
SMP: CPU 0 found 1 CPU(s)
enabled interrupts: 1 2
[00000000] new env 00001000
```

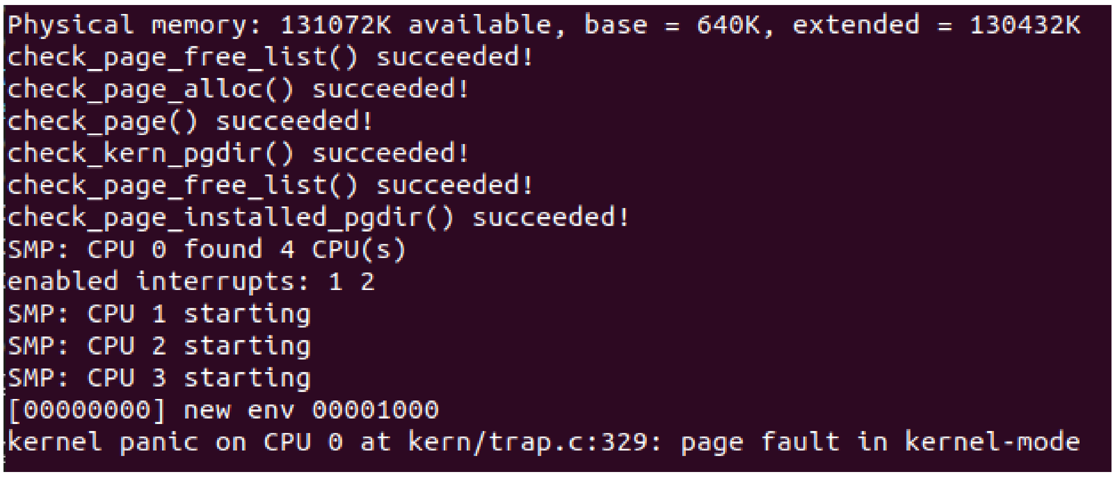
Exercise4：



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.)

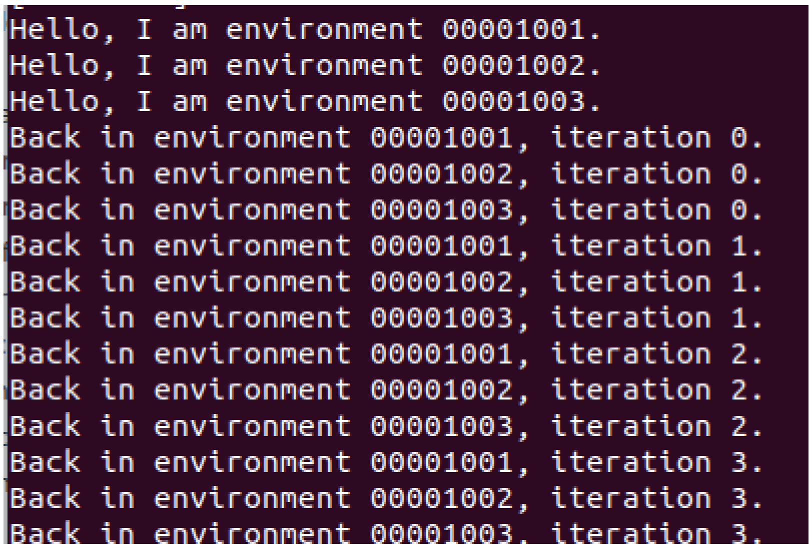
初始化每个 CPU 的进程，根据它已经给了的用 ts 初始化的过程进行修改。首先建立 TSS 然后进入 kernel，初始化 gdt 的 TSS 槽，加载 TSS selector，因为低位比较特殊，所以低位需要置为 0，最后加载 IDT。

编写完成后出现 page fault



```
Physical memory: 131072K available, base = 640K, extended = 130432K
check_page_free_list() succeeded!
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_free_list() succeeded!
check_page_installed_pgdir() succeeded!
SMP: CPU 0 found 4 CPU(s)
enabled interrupts: 1 2
SMP: CPU 1 starting
SMP: CPU 2 starting
SMP: CPU 3 starting
[00000000] new env 00001000
kernel panic on CPU 0 at kern/trap.c:329: page fault in kernel-mode
```

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



```
Hello, I am environment 00001001.
Hello, I am environment 00001002.
Hello, I am environment 00001003.
Back in environment 00001001, iteration 0.
Back in environment 00001002, iteration 0.
Back in environment 00001003, iteration 0.
Back in environment 00001001, iteration 1.
Back in environment 00001002, iteration 1.
Back in environment 00001003, iteration 1.
Back in environment 00001001, iteration 2.
Back in environment 00001002, iteration 2.
Back in environment 00001003, iteration 2.
Back in environment 00001001, iteration 3.
Back in environment 00001002, iteration 3.
Back in environment 00001003, iteration 3.
```

Exercise5

Exercise 5. Apply the big kernel lock as described above, by calling `lock_kernel()` and `unlock_kernel()` at the proper locations.

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。

- In `i386_init()`, acquire the lock before the BSP wakes up the other CPUs.
- In `mp_main()`, acquire the lock after initializing the AP, and then call `sched_yield()` to start running environments on this AP.
- In `trap()`, acquire the lock when trapped from user mode. To determine whether a trap happened in user mode or in kernel mode, check the low bits of the `tf_cs`.
- In `env_run()`, release the lock *right before* switching to user mode. Do not do that too early or too late, otherwise you will experience races or deadlocks.

Exercise6

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`.

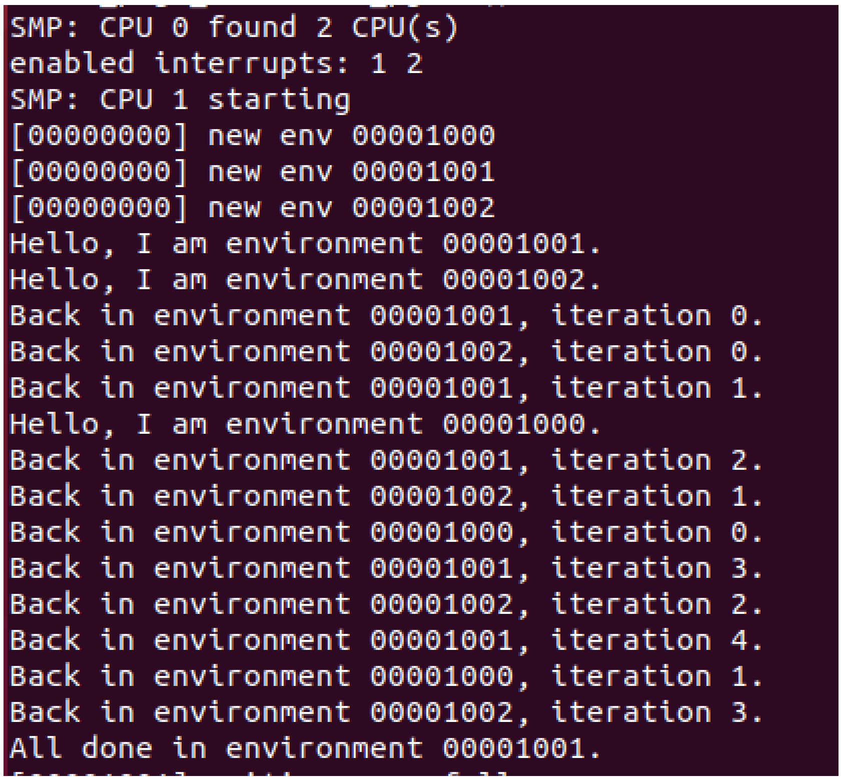
Modify `kern/init.c` to create three (or more!) environments that all run the program `user/yield.c`.

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。

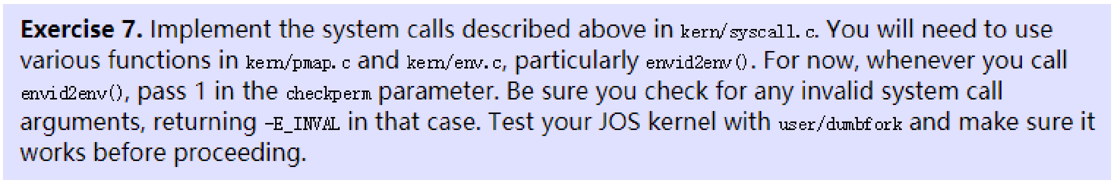
根据提示可以直接写代码，如果当前 curenv 为 null 的话，找到它的 env_id 作为遍历的开始，否则以 0 为开始进行遍历。遍历结束后没有找到可以运行的程序则继续运行当前的 curenv(如果存在)否则暂停。

测试结果如下



```
SMP: CPU 0 found 2 CPU(s)
enabled interrupts: 1 2
SMP: CPU 1 starting
[00000000] new env 00001000
[00000000] new env 00001001
[00000000] new env 00001002
Hello, I am environment 00001001.
Hello, I am environment 00001002.
Back in environment 00001001, iteration 0.
Back in environment 00001002, iteration 0.
Back in environment 00001001, iteration 1.
Hello, I am environment 00001000.
Back in environment 00001001, iteration 2.
Back in environment 00001002, iteration 1.
Back in environment 00001000, iteration 0.
Back in environment 00001001, iteration 3.
Back in environment 00001002, iteration 2.
Back in environment 00001001, iteration 4.
Back in environment 00001000, iteration 1.
Back in environment 00001002, iteration 3.
All done in environment 00001001.
```

Exercise7



Exercise 7. Implement the system calls described above in `kern/syscall.c`. You will need to use various functions in `kern/pmap.c` and `kern/env.c`, particularly `envid2env()`. For now, whenever you call `envid2env()`, pass 1 in the `checkperm` parameter. Be sure you check for any invalid system call arguments, returning `-E_INVAL` in that case. Test your JOS kernel with `user/dumbfork` and make sure it works before proceeding.

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()

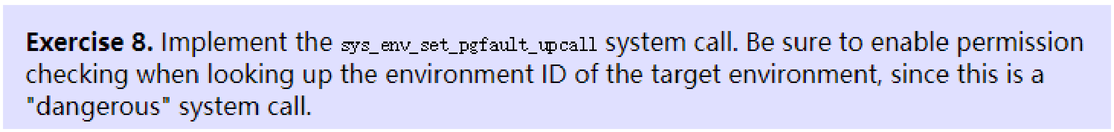
取消映射在给定环境中给定虚拟地址映射的页面。

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


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.

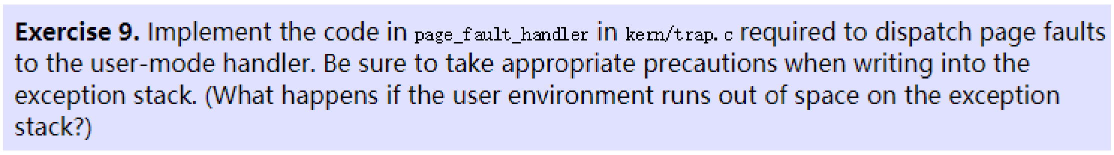
sys_env_set_pgfault_upcall()

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

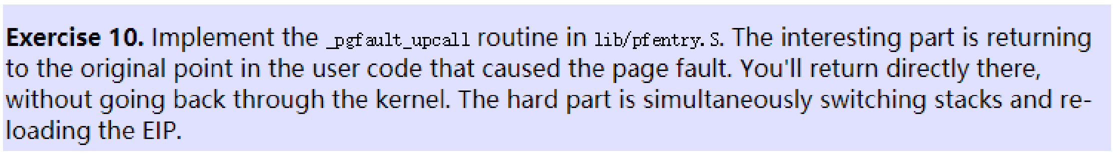

Exercise 9. Implement the code in `page_fault_handler` in `kern/trap.c` required to dispatch page faults to the user-mode handler. Be sure to take appropriate precautions when writing into the exception stack. (What happens if the user environment runs out of space on the exception stack?)

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。

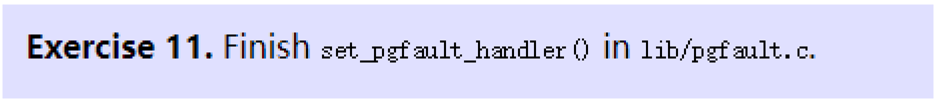
根据上面的情况编写代码即可。

Exercise10


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.

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

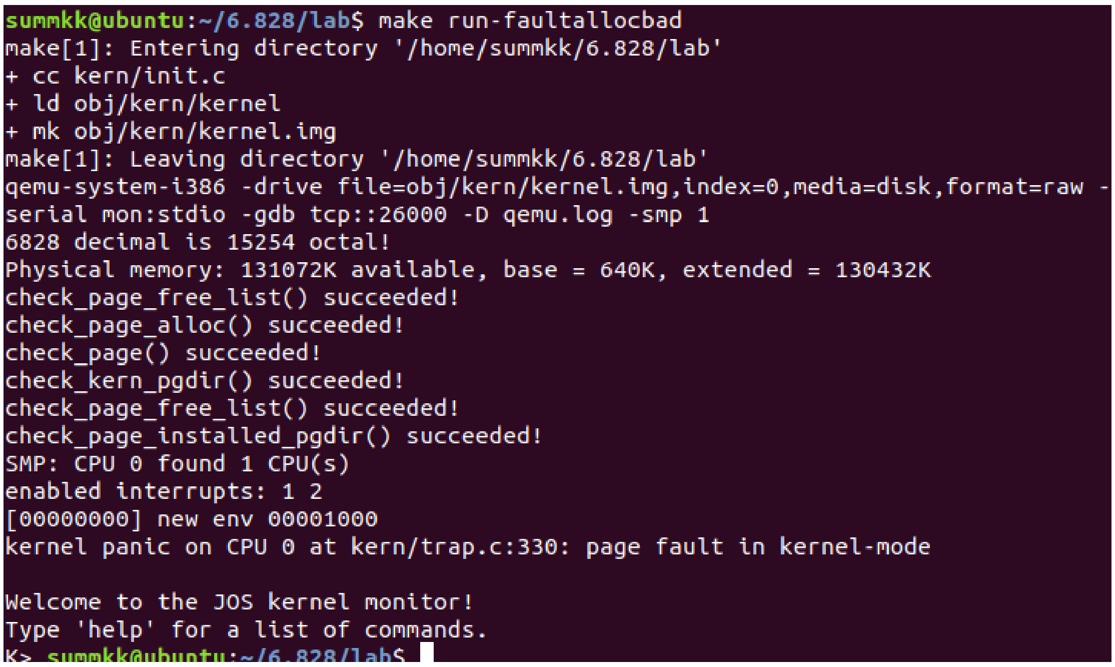


Exercise 11. Finish `set_pgfault_handler()` in `lib/pgfault.c`.

为用户异常栈分配页，注意边界条件的处理。

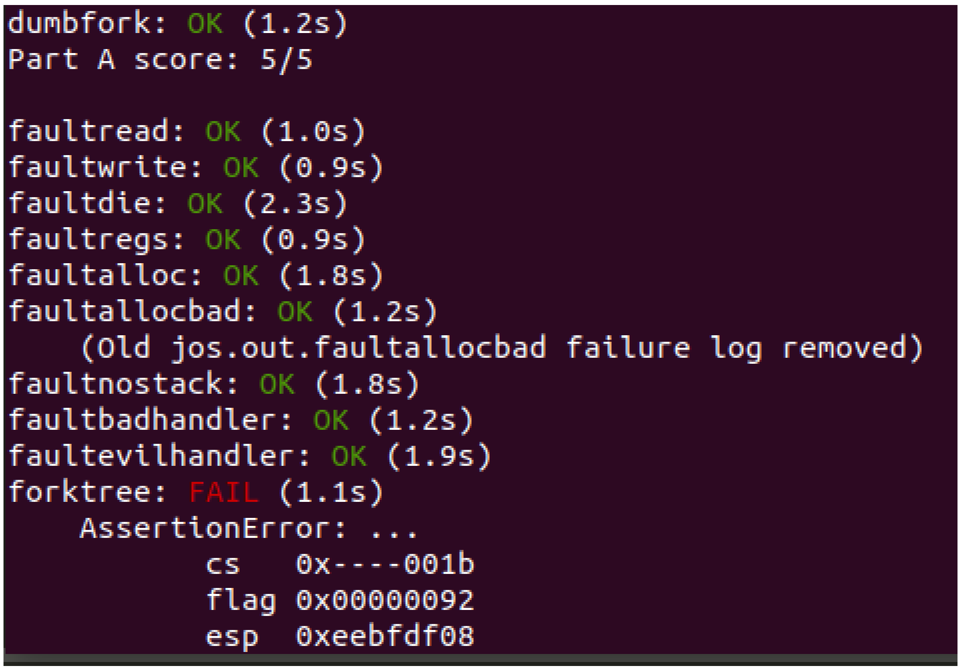
在完成 11 之后对它们进行测试。

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。



```
summkk@ubuntu:~/6.828/lab$ make run-faultallocbad
make[1]: Entering directory '/home/summkk/6.828/lab'
+ cc kern/init.c
+ ld obj/kern/kernel
+ mk obj/kern/kernel.img
make[1]: Leaving directory '/home/summkk/6.828/lab'
qemu-system-i386 -drive file=obj/kern/kernel.img,index=0,media=disk,format=raw -
serial mon:stdio -gdb tcp::26000 -D qemu.log -smp 1
6828 decimal is 15254 octal!
Physical memory: 131072K available, base = 640K, extended = 130432K
check_page_free_list() succeeded!
check_page_alloc() succeeded!
check_page() succeeded!
check_kern_pgdir() succeeded!
check_page_free_list() succeeded!
check_page_installed_pgdir() succeeded!
SMP: CPU 0 found 1 CPU(s)
enabled interrupts: 1 2
[00000000] new env 00001000
kernel panic on CPU 0 at kern/trap.c:330: page fault in kernel-mode

Welcome to the JOS kernel monitor!
Type 'help' for a list of commands.
K> summkk@ubuntu:~/6.828/lab$
```

修改后通过 part B 前面的检查



```
dumbfork: OK (1.2s)
Part A score: 5/5

faultread: OK (1.0s)
faultwrite: OK (0.9s)
faultdie: OK (2.3s)
faultregs: OK (0.9s)
faultalloc: OK (1.8s)
faultallocbad: OK (1.2s)
    (Old jos.out.faultallocbad failure log removed)
faultnostack: OK (1.8s)
faultbadhandler: OK (1.2s)
faultevilhandler: OK (1.9s)
forktree: FAIL (1.1s)
    AssertionError: ...
           cs   0x----001b
           flag 0x00000092
           esp  0xeebfdf08
```

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。

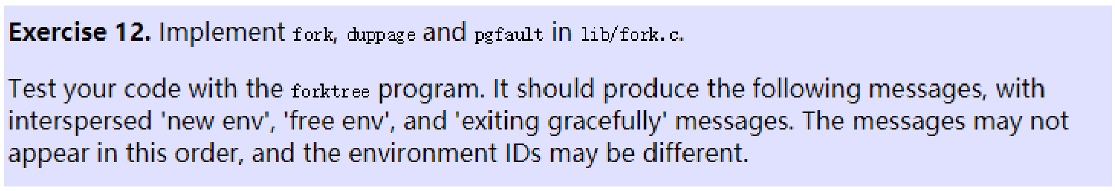
4.父级将子级的用户页面错误入口点设置为自己的。

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。

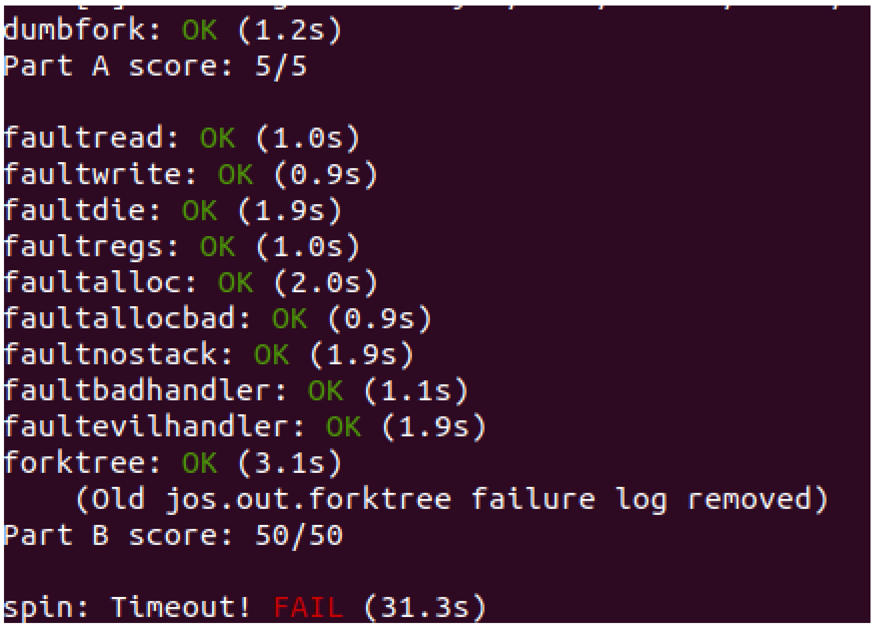
Exercise12



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

Test your code with the `forktree` program. It should produce the following messages, with interspersed 'new env', 'free env', and 'exiting gracefully' messages. The messages may not appear in this order, and the environment IDs may be different.

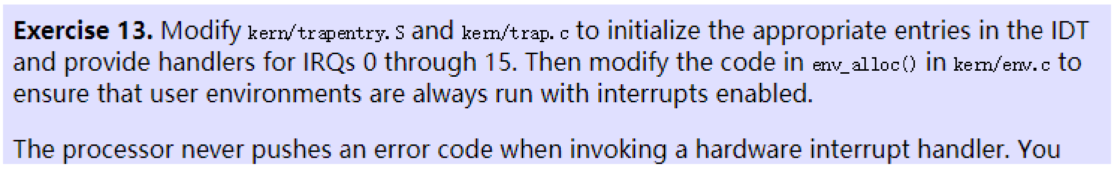
根据前面文档的讲述完成代码，测试结果如图，Part B 通过。



```
dumbfork: OK (1.2s)
Part A score: 5/5

faultread: OK (1.0s)
faultwrite: OK (0.9s)
faultdie: OK (1.9s)
faultregs: OK (1.0s)
faultalloc: OK (2.0s)
faultallocbad: OK (0.9s)
faultnostack: OK (1.9s)
faultbadhandler: OK (1.1s)
faultevilhandler: OK (1.9s)
forktree: OK (3.1s)
    (Old jos.out.forktree failure log removed)
Part B score: 50/50

spin: Timeout! FAIL (31.3s)
```

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

Exercises13



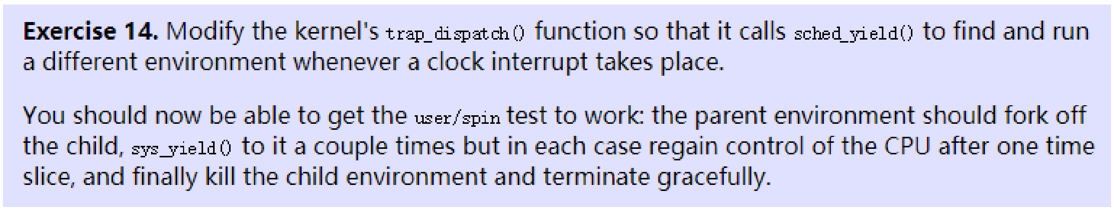
Exercise 13. Modify `kern/trapentry.S` and `kern/trap.c` to initialize the appropriate entries in the IDT and provide handlers for IRQs 0 through 15. Then modify the code in `env_alloc()` in `kern/env.c` to ensure that user environments are always run with interrupts enabled.

The processor never pushes an error code when invoking a hardware interrupt handler. You

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

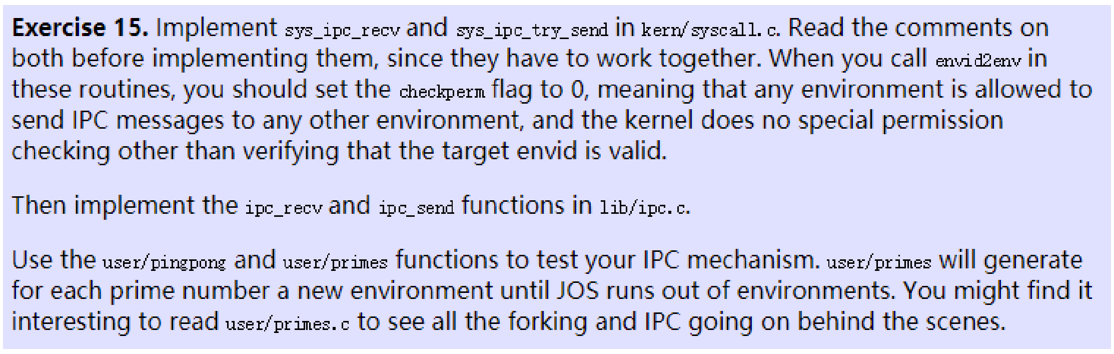


Exercise 14. Modify the kernel's `trap_dispatch()` function so that it calls `sched_yield()` to find and run a different environment whenever a clock interrupt takes place.

You should now be able to get the `user/spin` test to work: the parent environment should fork off the child, `sys_yield()` to it a couple times but in each case regain control of the CPU after one time slice, and finally kill the child environment and terminate gracefully.

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



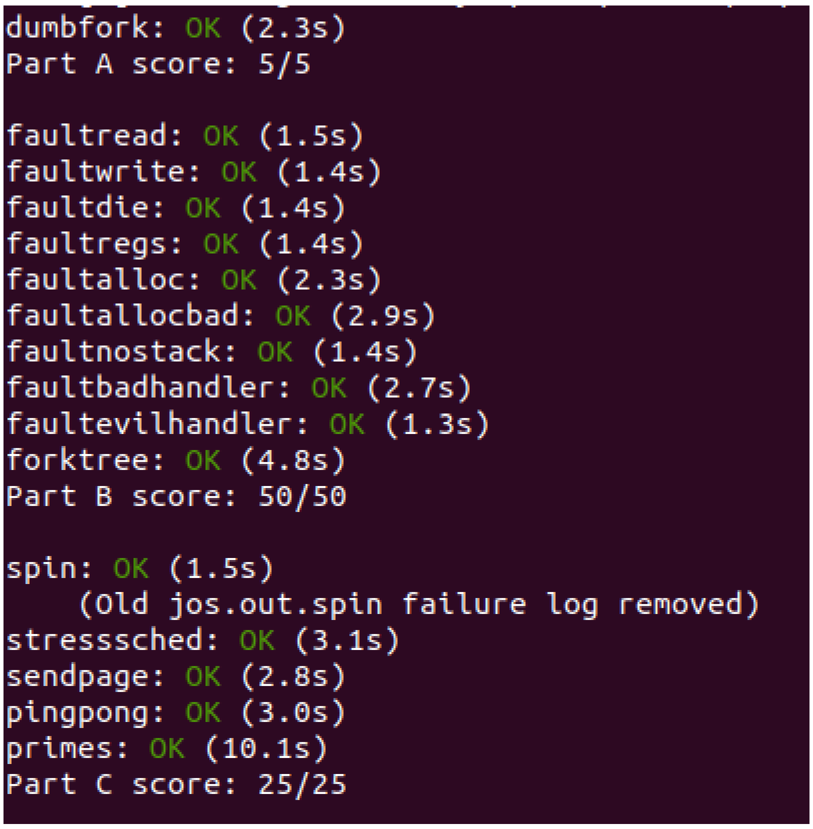
Exercise 15. Implement `sys_ipc_recv` and `sys_ipc_try_send` in `kern/syscall.c`. Read the comments on both before implementing them, since they have to work together. When you call `envid2env` in these routines, you should set the `checkperm` flag to 0, meaning that any environment is allowed to send 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`.

Use the `user/pingpong` and `user/primes` functions to test your IPC mechanism. `user/primes` will generate for each prime number a new environment until JOS runs out of environments. You might find it interesting to read `user/primes.c` to see all the forking and IPC going on behind the scenes.

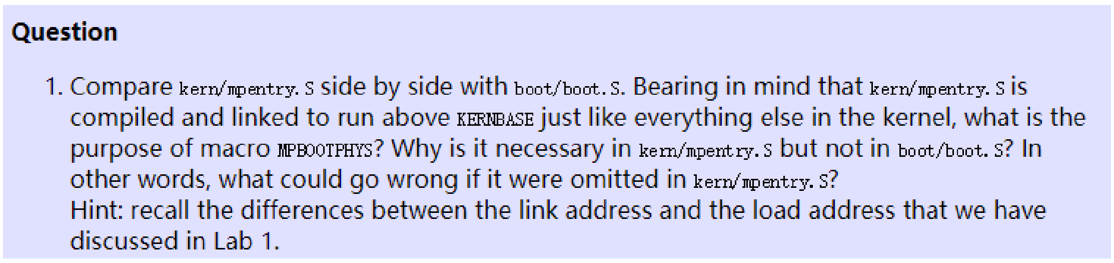
sys_ipc_recv()，等待接受信息，只需要改变进程状态，直接调度就会阻塞了。sys_ipc_try_send（）需要注意共享内存时，不能直接使用 sys_page_map，因为 sys_page_map 查找 env 时会检查权限。

make grade 通过检查。



```
dumbfork: OK (2.3s)
Part A score: 5/5

faultread: OK (1.5s)
faultwrite: OK (1.4s)
faultdie: OK (1.4s)
faultregs: OK (1.4s)
faultalloc: OK (2.3s)
faultallocbad: OK (2.9s)
faultnostack: OK (1.4s)
faultbadhandler: OK (2.7s)
faultevilhandler: OK (1.3s)
forktree: OK (4.8s)
Part B score: 50/50

spin: OK (1.5s)
    (Old jos.out.spin failure log removed)
stresssched: OK (3.1s)
sendpage: OK (2.8s)
pingpong: OK (3.0s)
primes: OK (10.1s)
Part C score: 25/25
```

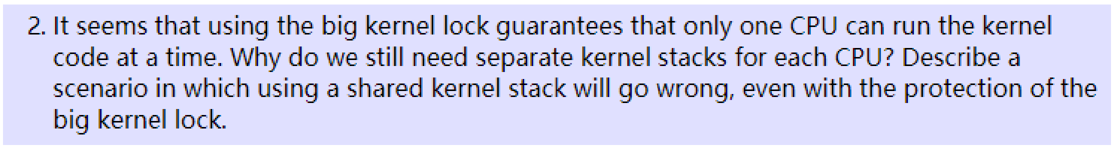
3 实验问题



Question

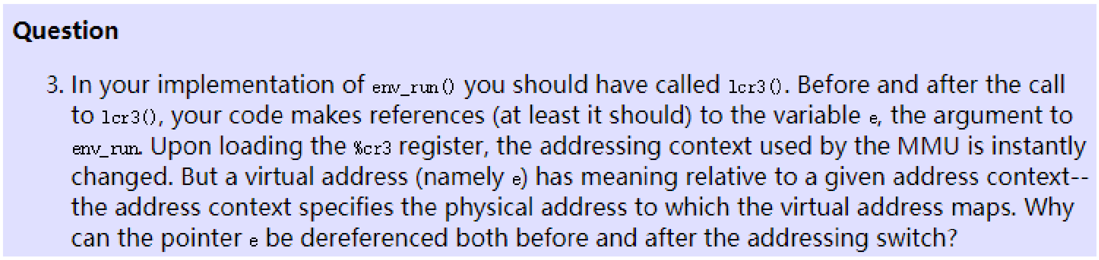
1. Compare `kern/mpentry.S` side by side with `boot/boot.S`. Bearing in mind that `kern/mpentry.S` is compiled and linked to run above `KERNBASE` just like everything else in the kernel, what is the purpose of macro `MPBOOTPHYS`? Why is it necessary in `kern/mpentry.S` but not in `boot/boot.S`? In other 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.

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。



2. 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.

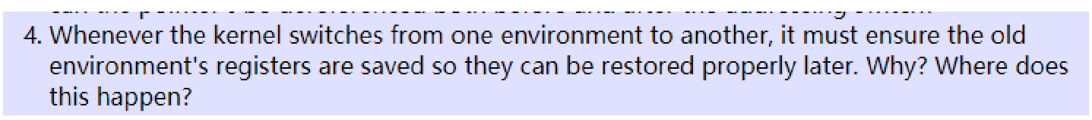
如果多个 CPU 同时进入中断而栈访问不互斥会导致压入的寄存器信息混乱出现错误。



Question

3. 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?

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。



4. 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?

Stack 中的 trapframe 保留了寄存器状态。