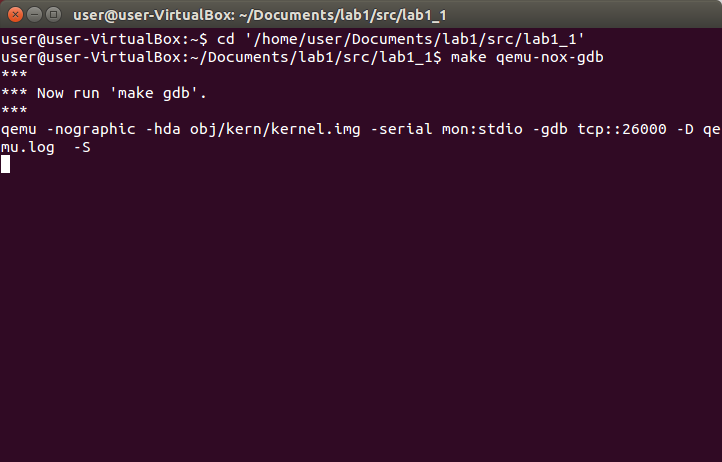
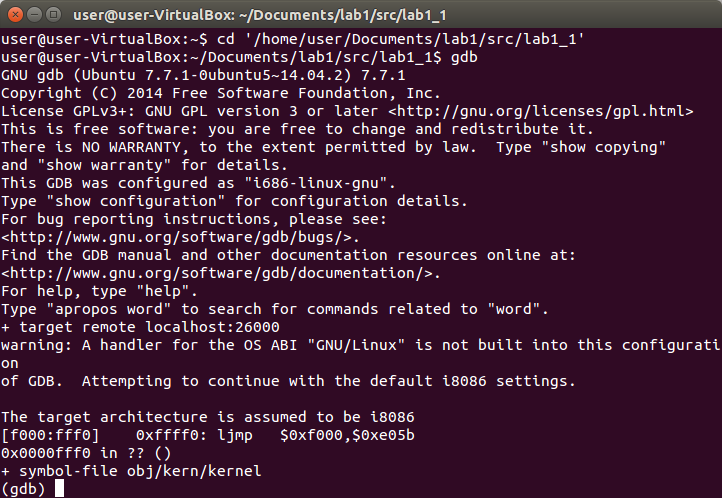
操作系统lab1

练习2

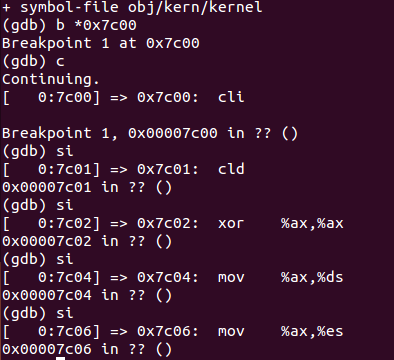
使用qemu的调试功能来检查一个IA-32兼容的计算机的引导过程



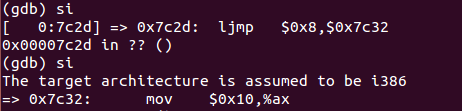
用gdb监听qemu，从[f000:fff0] 0xffff0: ljmp $0xf000,$0xe05b行得知IBM PC是从物理地址0x000ffff0开始执行的



使用gdb的si指令来追踪ROM BIOS中的几个指令，使用b命令设置断点，并单步执行数个步骤，结果如图所示



指令ljmp $PROT\_MODE\_CSEG, $protcseg从16位切换至32位在.code32后开始执行32位代码



练习3

单步执行至call bootmain指令在gdb中为命令 call 0x7d0a，即bootmain的入口地址为0x7d0a



继续单步执行至进入readsect函数，该函数入口点为0x7c7c，在此之前先进入readseg函数，入口点为0x7cd1

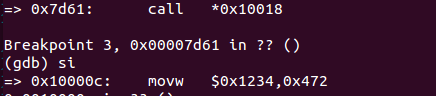


该指令为调用waitdisk函数

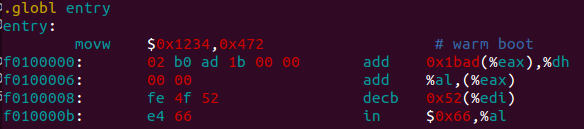


找到for循环跳出的地址为0x7d61,执行至断点，该指令为

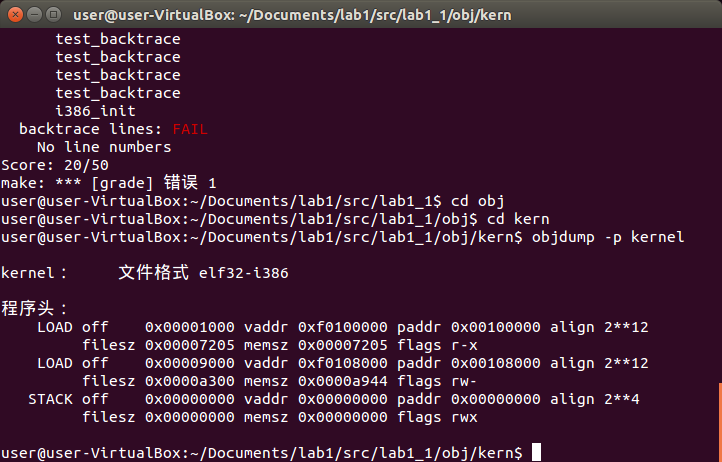
movw $0x1234,0x472 是kernel指令的第一条语句



与kernel.asm源代码一致

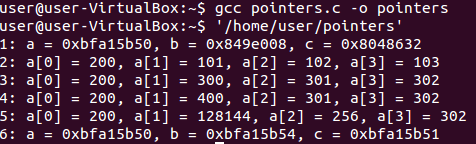


利用objdump指令查看内核，结果如下



练习4.

用vim创建一个名为pointers.c的文件，内含源代码，用gcc编译器生成可执行文件pointers,运行后得到如下输出



负责第一行输出的代码是printf("1: a = %p, b = %p, c = %p\n", a, b, c);输出的三个值分别为数组a的首元素地址，指针b的地址，指针c的地址；关于第二行输出值的相关代码为

c = a;//c指向a数组首地址

for (i = 0; i < 4; i++)

a[i] = 100 + i;//a[0]=100,a[1]=101,a[2]=102,a[3]=103

c[0] = 200;//a[0]=200

负责第三行输出的代码为

c[1] = 300;//a[1]=300

\*(c + 2) = 301;//a[2]=301

3[c] = 302;//a[3]=302

Ps:3[c] 与 c[3] 在编译时是等效的，都是\*(c + 3)

负责第四行输出的代码为

c = c + 1;//c指向a[1]

\*c = 400;//a[1]=400

负责第五行输出的代码为

c = (int \*) ((char \*) c + 1);//c指针向后移一个byte;

\*c = 500;//c指向的内存存值500

500的二进制表示为000111110100

400的二进制表示为000110010000‬

00011111010010010000//a[1]的值为400的前8位加上500左移8位；

000100000000//a[2]的值为301的后8位全置0

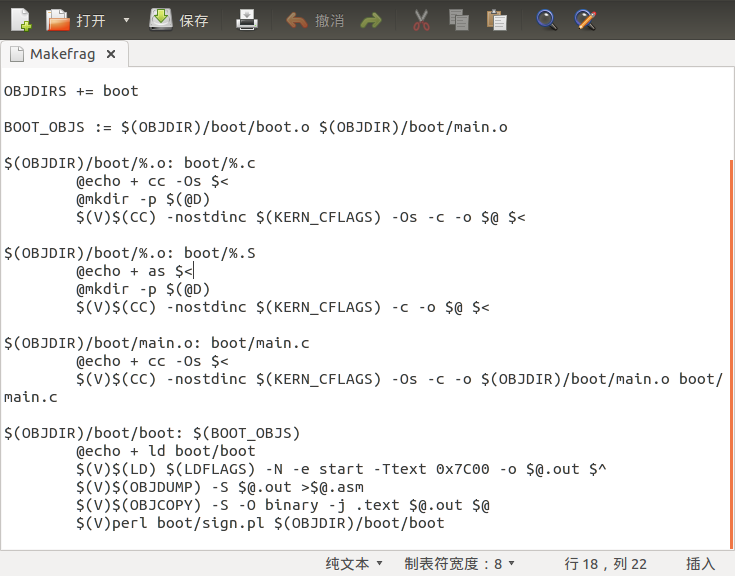
第六行的输出有关代码为；

b = (int \*) a + 1;//b指向a[1];b地址等于a地址加4

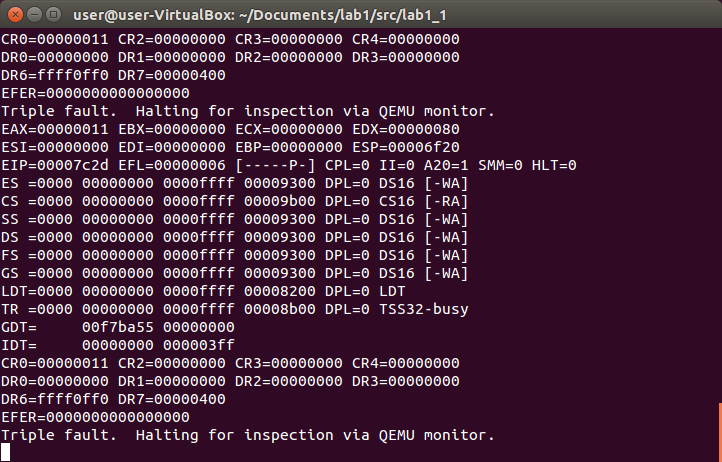
c = (int \*) ((char \*) a + 1);//c指向a[0]后移一个字节；c地址等于a地址加1

练习5

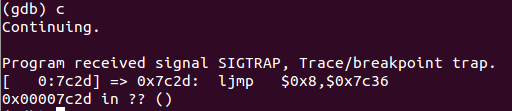
打开Makefrag文件，将其中的链接地址改为错误地址将0x7c00改为0x7c04



之后lab1\_1目录下make clean和makedistclean后重新make qemu

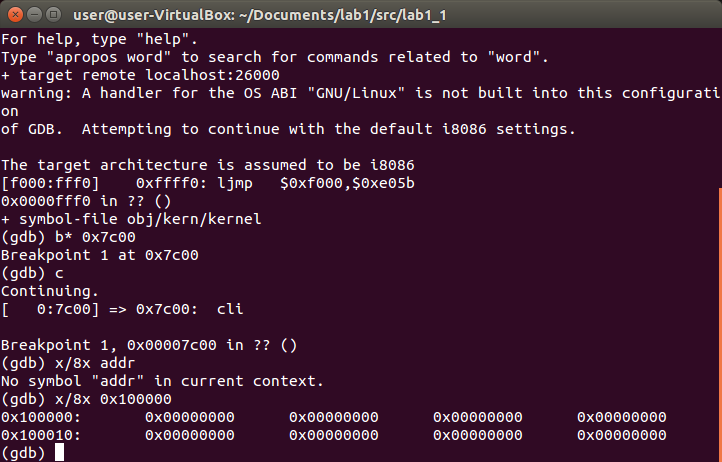


用gdb跟踪用c继续执行，发现在ljmp指令处程序停了下来,si单步执行后发现此后一直在重复ljmp指令

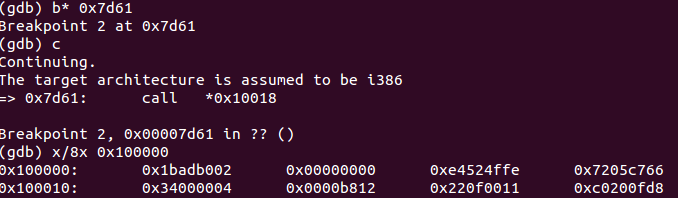


练习6

在0x7c00处设置断点，执行到断点后输入x/8x 0x100000显示8个存储器字

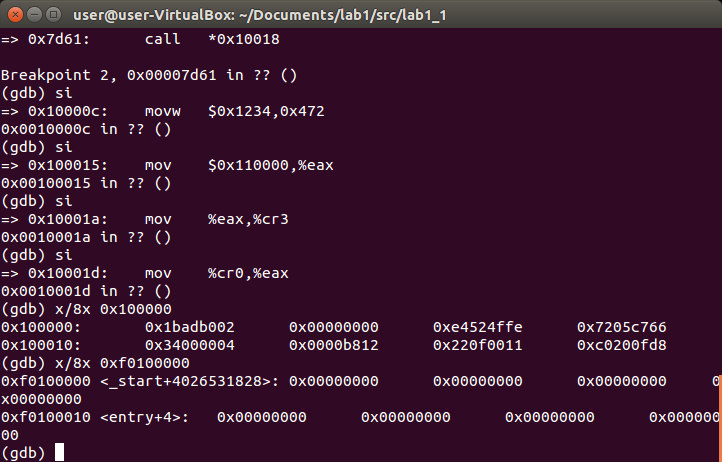


在bootloader进入内核处设置断点，即0x7d61处，然后再输入x/8x 0x100000显示8个存储器字

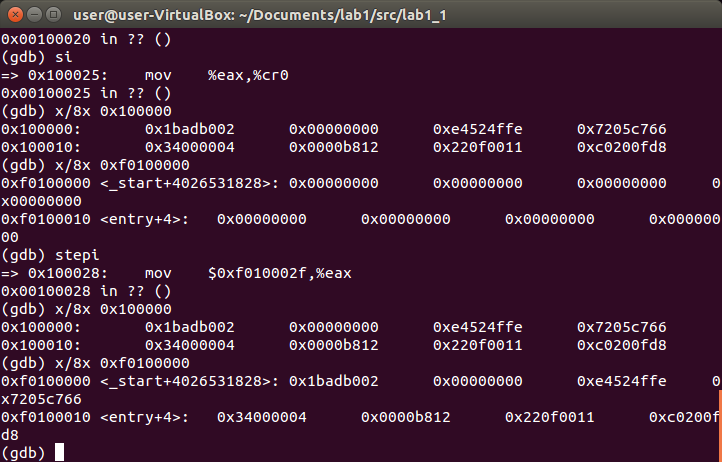


练习7

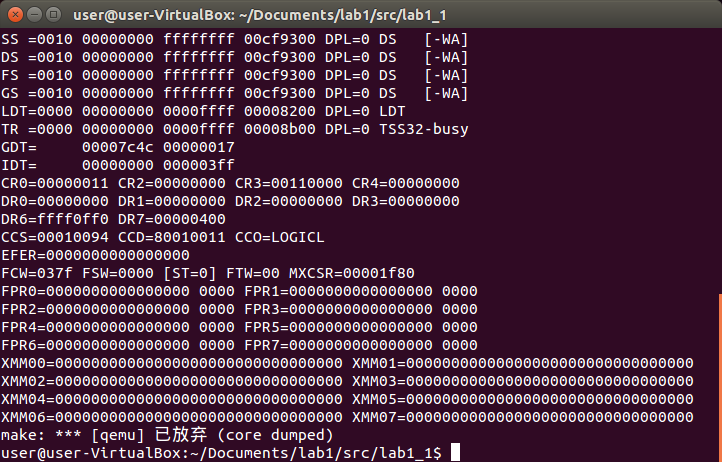
使用qemu和gdb跟踪到JOS内核并停止在movl％eax，％cr0。 查看内存中在地址0x00100000和0xf0100000处的内容。此处分别在两地址查看了8个存储字。



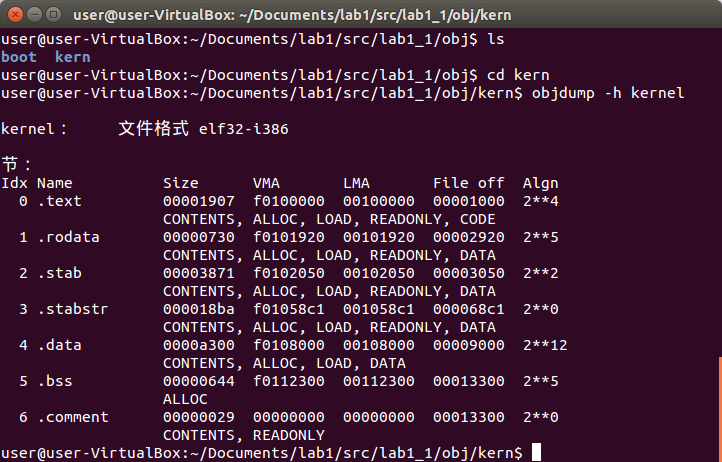
使用使用GDB命令stepi单步执行该指令。 指令执行后，再次检查0x00100000和0xf0100000的内存。注意到0x0010000处存储的内容不变，而0xf0100010处存储的值由全0变为了与地址0x00100000处存储的值相同



注释掉kern/entry.S中的movl％eax，％cr0，重新运行qemu出现如下信息，发现qemu不能正常工作



Dump出obj/kern/kernel文件，发现kernel的.text段被加载到了0x00100000处

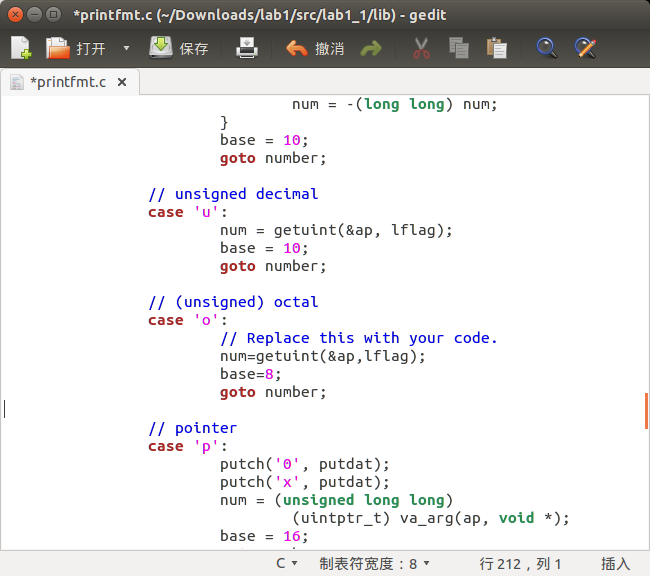


查看0x100000（内核的装载地址）和0xf0100000（内核的链接地址）处内存在启用页表机制之后的变化；停用页表机制，重新编译并运行JOS。启用页表机制之后，0x100000处的内存内容被“复制”到了0xf0100000处；停用页表机制之后，重新运行时在jmp处出错，因为这时候存放的是虚拟地址。

练习8

我们省略了一小段代码 - 使用“％o”形式的模式打印八进制数字所需的代码。查找并补全此代码片段。

在/lib/printfmt.c中找到如下代码，加入4行代码，首先输出0代表输出是8进制，然后输出8进制数，base=8决定输出为8进制数。



问1.printf.c中的第一个函数中调用了cputchar函数来实现在控制台上输出字符串，在console.c中找到该代码

可以看到该函数调用了cons\_putc函数

void

cputchar(int c)

{

cons\_putc(c);

}

找到cons\_putc的代码，使用该函数可以在控制台上输出一个字符

// output a character to the console

static void

cons\_putc(int c)

{

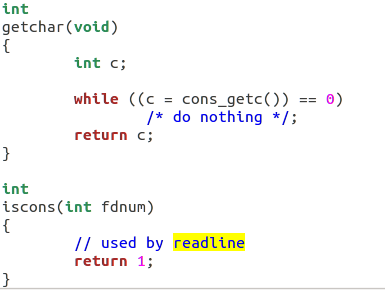
serial\_putc(c);

lpt\_putc(c);

cga\_putc(c);

}

Console.c中还提供了getchar以及iscons函数被readline函数调用



问2.解释Console.c中的以下内容

if (crt\_pos >= CRT\_SIZE) {//如果输出的数据超过一个屏幕

int i;

//memmove函数，向上卷一行

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;

}

如果输出的数据超过一个屏幕 80×25 大小 那么就会向上卷屏。

memmove作用是向上卷一行，那么下面自然是原来的最后一行，这样倒数第一行与倒数第二行重复，所以倒数第一行用空格填充并且把pos设置好。

问3.在下述代码中：

IMG_256

Q:在调用cprintf（）时，fmt是什么意思？ ap是什么意思？

A:fmt指向格式字符串；ap指向参数列表的第一个参数，即x

Q:列出（按执行顺序）以下每次调用cons\_putc，va\_arg和vcprintf这三段代码时的细节。 对于cons\_putc，列出其参数。 对于va\_arg，列出调用之前和之后的ap指针的指向。 对于vcprintf列出其两个参数的值。

A:调用顺序为vcprintf(fmt,ap), cons\_putc(‘x’), cons\_putc(‘ ’), va\_args(), cons\_putc(‘1’), cons\_putc(‘ ’), cons\_putc(‘y’), cons\_putc(‘ ’), va\_args(), cons\_putc(‘ ’), cons\_putc(‘z’), cons\_putc(‘ ’), va\_args(), cons\_putc(‘\n’)

问4.运行下面的代码

unsigned int i = 0x00646c72;

cprintf("H%x Wo%s", 57616, &i);

Q:输出是什么？如何按照上一个练习的执行步骤，说明为什么会显示这个输出信息。

A:输出是He110 World，e110是57616的16进制表示，x86是小端编址模式，而%s格式符是按字符输出，将16进制地址转为ascii码值为72=>r,6c=>l,64=>d

Q:由于x86是小端的，所以得到了上面的输出结果。如果x86是大端，那么为了产生相同的输出，你会设置什么？你需要将57616更改为不同的值吗？

A:i的值应为0x726c6400

问5. 在下面的代码中，将在“y =”之后打印什么？ （注意：答案不是一个固定的值。）为什么会发生这种情况？

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

cprintf输出的是参数栈上部4字节的内容，所以不是一个固定的值

问6.假设GCC更改了它的调用约定，以声明的顺序将参数压入栈中，这样会使最后一个参数最后被压入。 你将如何更改cprintf或其接口，以便仍然可以传递一个可变数量的参数？

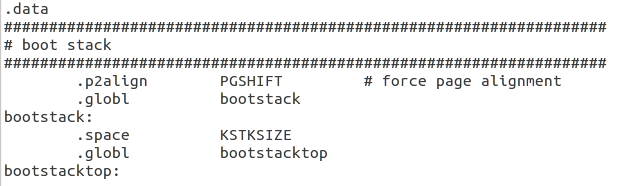
A:改变stdarg库当中va\_list宏的代码

练习9

确定内核在哪里完成了栈的初始化，以及栈所在内存的确切位置。内核如何为栈保留空间？栈指针初始化时指向的是保留区域的“哪一端”。

问1.内核在哪里完成了栈的初始化

进入entry.S在文件的最后部分看到其要调用i386\_init()函数，而该函数位于init.c文件中

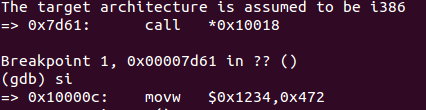


Init.c程序中已经开始对操作系统进行初始化工作，所以在运行i386\_init子函数时，内核的堆栈已经设置好了，设置堆栈的指令应该死entry.S中位于call i386\_init指令之前的两条指令



问2.栈所在内存的确切位置

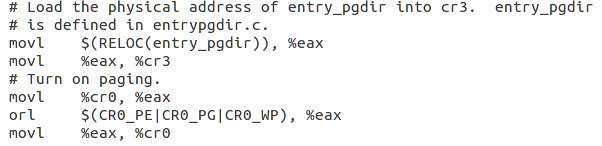
用gdb调试qemu在0x7d61处设置断点，执行到断点，然后单步执行，注意到此时movw指令的地址为0x10000c为kernel的第一条指令所在的物理地址



继续向后单步执行到jmp \*%eax后，指令的地址发生了变化，movl指令的地址为0xf010002f，这是一个虚拟地址，而其真实地址为0x0010002f，所有的内核代码都存放在这个内存区域中。程序员编程时能用的是虚拟地址空间的低地址空间。



在entry.S中找到如下代码，该代码实现了一个c语言的页表，entey\_pgdir是个页表，它把[0xf0000000-0xf0400000]这4MB的虚拟地址空间映射为[0x00000000-0x00400000]的物理地址空间



movl $(RELOC(entry\_pgdir)), %eax//把entry\_pgdir这个页表的起始物理地址传给eax,RELOC宏的功能是计算输入参数的物理地址

movl %eax, %cr3//把entry\_pgdir这个页表的起始地址传送给寄存器%cr3

控制寄存器cr2和cr3都是和分页机制相关的寄存器。其中cr3寄存器存放页表的物理起始地址。

后面3句修改cr0寄存器的值，把cr0的PE位，PG位, WP位都置位1。其中PE位是启用保护标识位，如果被置1代表将会运行在保护模式下。PG位是分页标识位，如果这一位被置1，则代表开启了分页机制。WP位是写保护标识，如果被置位为1，则处理器会禁止超级用户程序向用户级只读页面执行写操作。

movl %cr0, %eax

orl $(CR0\_PE|CR0\_PG|CR0\_WP), %eax

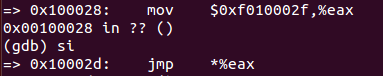
movl %eax, %cr0

在该段指令后，就开始工作在具有分页机制的模式之下了。接下来的指令就可以指定[0xf0000000-0xf0400000]范围的指令了。

然后通过以下两条指令将程序的地址空间变为0xf0000000-0xf0400000范围内



在实际的调试过程中这两句代码为，可见relocated的值为0xf010002f。此时分页系统会把这个虚拟地址，转换为真实的物理地址。



在接下来的三行代码中，分别设置了%ebp，%esp两个寄存器的值。其中ebp被修改为0。esp则被修改为bootstacktop的值。这个值为0xf0110000。另外在entry.S的末尾还定义了一个值，bootstack。

Ps:在数据段中定义栈顶bootstacktop之前，首先分配了KSTKSIZE这么多的存储空间，专门用于堆栈，这个KSTKSIZE = 8 \* PGSIZE = 8 \* 4096 = 32KB。所以用于堆栈的地址空间为 0xf0108000-0xf0110000，其中栈顶指针指向0xf0110000. 那么这个堆栈实际坐落在内存的 0x00108000-0x00110000物理地址空间中。



在movl $(bootstacktop),%esp指令后查看esp的值即为bootstacktop的值0xf0110000



问3.内核如何为栈保留空间？

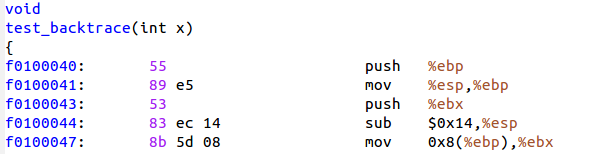
在上一问中，entry.S的数据段声明一块大小为32KB的空间作为堆栈使用。从而为内核保留了一块空间。

问4.栈指针初始化时指向的是保留区域的“哪一端”。

堆栈是向下生长的，栈指针初始化指向最高地址，在此处为bootstacktop的值。

练习10.要熟悉x86上C语言函数的调用约定，请在obj/kern/kernel.asm中找到test\_backtrace函数的地址，在其中设置一个断点，并检查在内核启动后每次这个函数被调用时会发生什么。 每一级的test\_backtrace在递归调用时，会在栈上压入多少个32位的字，这些字的内容是什么？

在kernel.asm中看到test\_backtrace子程序的地址为0xf0100040



test\_backtrace函数，其c语言形式为

void

test\_backtrace(int x)

{

cprintf("entering test\_backtrace %d\n", x);

if (x > 0)

test\_backtrace(x-1);

else

mon\_backtrace(0, 0, 0);

cprintf("leaving test\_backtrace %d\n", x);

}

看出该函数是一个循环调用，在每一次循环中先输出“entering test\_backtrace x”然后若x>0，则循环调用test\_backetrace，当循环调用完成后再输出“leaving test\_backtrace x”

Q:每一级的test\_backtrace在递归调用时，会在栈上压入多少个32位的字，这些字的内容是什么？

A:test\_backtrace的前4行代码为

push %ebp

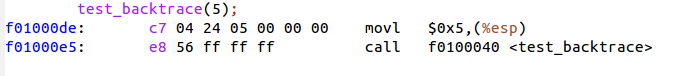
mov %esp,%ebp

push %ebx

sub $0x14,%esp

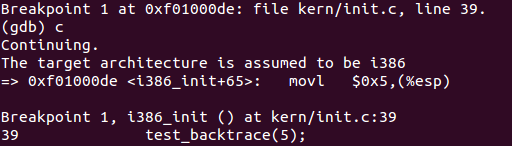
他们实现了保存调用该子程序的父程序的栈帧信息，以及为当前子程序分配新的栈帧

在i386\_init子程序中，找到调用test\_backtrace的指令，可以看到传递给test\_backtrace的参数为5



在 i386\_init 函数中，运行了子程序test\_backtrace(5)。

在0xf01000de处设置断点，执行到该断点



当运行test\_backtrace(5)之前，esp寄存器ebp寄存器的值分别为如下



0xf010ffe0~0xf010fff8就是当前i386\_init子程序的栈帧，当计算机要调用test\_backtrace(5)程序时，

单步执行进入test\_backtrace子程序，首先call指令把i386\_init的返回地址压入堆栈中，所以esp变为0xf010ffdc，然后进入test\_backtrace(5)子程序。

子程序中第一句push %ebp，把i386\_init的ebp寄存器的值压入堆栈中，即地址0xf010ffe0-4=0xf010ffd8处，此时esp的值变为0xf010ffd8。



然后 mov %esp, %ebp 把ebp的值更新为esp的值，0xf010ffd8。这个就是当前test子程序的ebp寄存器的值。即它的栈帧的高地址边界。

然后 push %ebx 把ebx寄存器的值压入堆栈，此时esp变为0xf010ffd4。因为%ebx寄存器可能被这个子程序所使用，所以必须把它之前的值保留。

然后 sub $0x14, %esp 把esp中的值减去0x14=20，esp的值变为0xf010ffc0。这就是给test子程序分配一个大小为20个存储单元的额外的栈帧空间，供它存储一些临时变量的值。

所以上述4条汇编指令，执行完成后，esp，ebp寄存器的值变化为



这就是test\_backtrace(5)子程序运行时的栈帧地址范围。而输入参数'5'的值存放在0xf010ffe0单元处。

　　紧接着就是调用test\_backtrace(4)，由于它和test\_backtrace(5)是一样的，只不过现在是在test\_backtrace(5)中调用test\_backtrace(4)，所以调用后内存中会同时存在二者的栈帧，test\_backtrace(4)的栈帧就在test\_backtrace(5)的栈帧的上面。test\_backtrace(4)中 esp， ebp的值也可以像我们分析test\_backtrace(5)一样被分析出来，如下：

　　esp：0xf010ffa0 ebp：0xf010ffb8

　　同理test\_backtrace(3)中：

　　esp：0xf010ff80 ebp：0xf010ff98

　　test\_backtrace(2)中

　　esp：0xf010ff60 ebp：0xf010ff78

　　test\_backtrace(1)中：

　　esp：0xf010ff40 ebp：0xf010ff58

test\_backtrace(0)中：

　　esp：0xf010ff20 ebp：0xf010ff38

对于任意一层调用，比如test\_backtrace(i)，它的esp和ebp的值假设分别为esp(i)和ebp(i)。那么在这个栈帧范围内主要存在这么几个重要的值：

　首先ebp(i)所指向的内存单元处存放着上一层程序的ebp寄存器的值，即ebp(i-1)。

　另外在esp(i)所指向的内存单元处存放着对下一层子程序调用时传入的参数，即i+1

内存中栈帧的分别如下；

stack top: +-------------------------------(low memory)

           +-----------last EBP------------(current SP)

           +------------EIP----------------(RET addr)

           +------------arg0---------------

           +------------arg1---------------

           +……----------------------------（high memory）

test\_backtrace(0)------------------(low memory)

test\_backtrace(1)

test\_backtrace(2)

test\_backtrace(3)

test\_backtrace(4)

test\_backtrace(5)

i386\_init----------------------------（high memory）

mon\_backtrace的代码实现如下：

int

mon\_backtrace(int argc, char \*\*argv, struct Trapframe \*tf)

{

// Your code here.

uint32\_t m\_ebp;

uint32\_t m\_eip;

uint32\_t arg0, arg1, arg2, arg3, arg4;

uint32\_t \*p\_bp;

m\_ebp = read\_ebp();

while (m\_ebp != 0)

{

p\_bp = (uint32\_t \*)m\_ebp;

m\_eip = \*(p\_bp + 1);

arg0 = \*(p\_bp + 2);

arg1 = \*(p\_bp + 3);

arg2 = \*(p\_bp + 4);

arg3 = \*(p\_bp + 5);

arg4 = \*(p\_bp + 6);

cprintf("ebp %08x eip %08x args %08x %08x %08x %08x %08x\n", m\_ebp, m\_eip, arg0, arg1, arg2, arg3, arg4);

m\_ebp = \*p\_bp;

}

return 0;

}

练习11实现如上所述的回溯功能。

在monitor.c文件中找到mon\_backtrace函数，插入如下代码

uint32\_t \*ebp = (uint32\_t \*) read\_ebp();//获取ebp值

cprintf("Stack backtrace:\n");//输出格式

while (ebp) {

cprintf(" ebp %08x eip %08x args ", ebp, ebp[1]);//输出ebp,eip，其中eip通过ebp[1]得到

int j=2;

while(j!=7) //输出args[i]

{

cprintf(" %08x", ebp[j]);

j++;

}

ebp = (uint32\_t \*) (\*ebp);

}

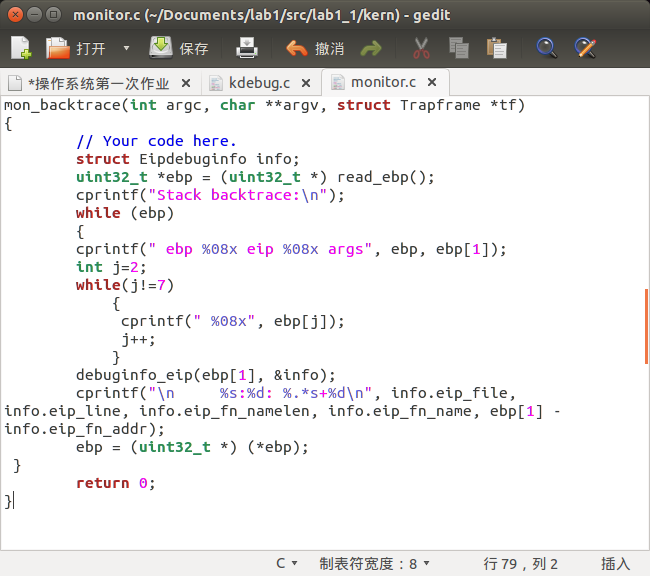
return 0;

练习12

修改你的堆栈回溯功能，为每个eip显示与该eip对应的函数名称，源文件名和行号。

继续完善monitor.c中的mon\_backtrace函数，加入输出函数名称，源文件名和行号的功能

完善后的代码如下

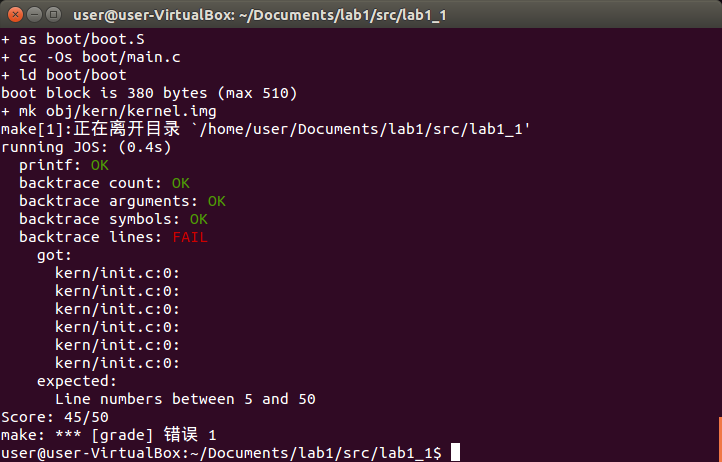


新增加的代码为

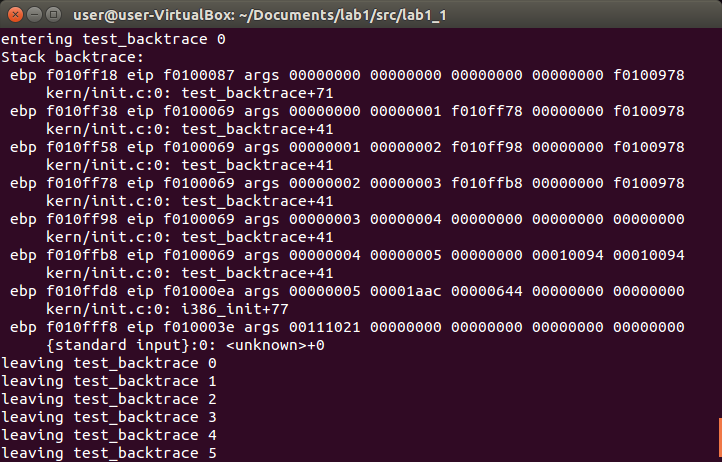
debuginfo\_eip(ebp[1], &info);

cprintf("\n %s:%d: %.\*s+%d\n", info.eip\_file, info.eip\_line, info.eip\_fn\_namelen, info.eip\_fn\_name, ebp[1] - info.eip\_fn\_addr);

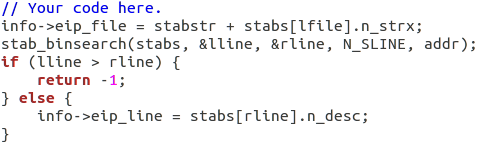
然而，在lab1\_1中make grade后分数为45/50，输出格式出现问题



make qemu后结果如下



在kdebug.c中的debuginfo\_eip函数中找到需要加入代码的部分，加入如下代码



再次在lab1\_1目录下make grade,可以看到问题被解决

