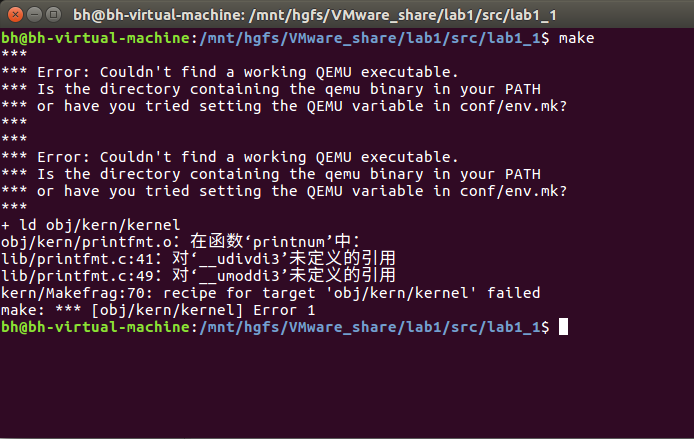
1.1x86汇编语言

Part1

1.2simulating the x86

Question1:当使用make system 编译bootloader和kernel时，出现了以下情况，所以当前解决“对‘\_\_udivdi3’未定义的引用…”

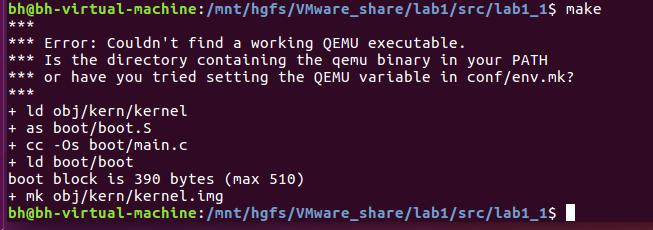


解决方案：运行



Question2:

解决Error之后的问题

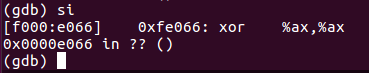




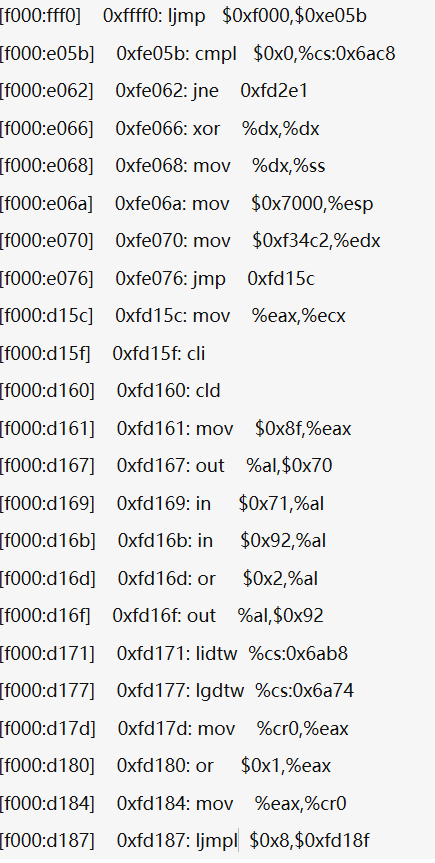
Question3：make qemu-gdb进入QEMU界面，然后通过关闭终端退出，再次make qemu-gdb时报错：“qemu-system-i386: -gdb tcp::25000: Failed to bind socket: Address already in use”

解决方法：发生这种问题是由于端口被程序绑定而没有释放造成。可以使用netstat -lp命令查询当前处于连接的程序以及对应的进程信息。然后用ps pid察看对应的进程，并使用kill pid关闭该进程即可。

【练习2】当BIOS运行时，它会设置一个中断描述符表，并初始化各种设备，比如VGA显示。这就是您在QEMU窗口中看的“Starting SeaBIOS”消息的来源。在初始化PCI总线和BIOS所知道的所有重要设备之后，它将搜索可引导的设备，如软盘、硬盘或CD-ROM。最终，当找到一个可引导的磁盘时，BIOS从磁盘中读取引导加载程序并将控制权转移到它.



上网查看资料，网友得到的结果如图：



附上相关知识：



基本功能：设置ss和esp寄存器的值，打开A20门（为了兼容老芯片而留下的历史包袱）、进入保护模式（需要设置cr0寄存器的PE标志）。

【练习3】

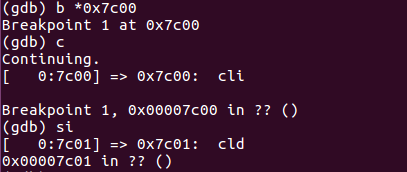
•资料补充：

1. 什么时GDT?

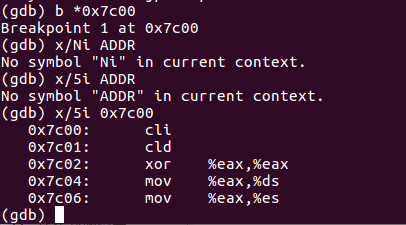
在Protected Mode下，一个重要的必不可少的数据结构就是GDT（Global Descriptor Table）。 为什么要有GDT？我们首先考虑一下在Real Mode下的编程模型： 在Real Mode下，我们对一个内存地址的访问是通过Segment:Offset的方式来进行的，其中Segment是一个段的Base Address，一个Segment的最大长度是64 KB，这是16-bit系统所能表示的最大长度。而Offset则是相对于此Segment Base Address的偏移量。Base Address+Offset就是一个内存绝对地址。由此，我们可以看出，一个段具备两个因素：Base Address和Limit（段的最大长度），而对一个内存地址的访问，则是需要指出：使用哪个段？以及相对于这个段Base Address的Offset，这个Offset应该小于此段的Limit。当然对于16-bit系统，Limit不要指定，默认为最大长度64KB，而 16-bit的Offset也永远不可能大于此Limit。我们在实际编程的时候，使用16-bit段寄存器CS（Code Segment），DS（Data Segment），SS（Stack Segment）来指定Segment，CPU将段积存器中的数值向左偏移4-bit，放到20-bit的地址线上就成为20-bit的Base Address。 到了Protected Mode，内存的管理模式分为两种，段模式和页模式，其中页模式也是基于段模式的。也就是说，Protected Mode的内存管理模式事实上是：纯段模式和段页式。进一步说，段模式是必不可少的，而页模式则是可选的——如果使用页模式，则是段页式；否则这是纯段模式。 既然是这样，我们就先不去考虑页模式。对于段模式来讲，访问一个内存地址仍然使用Segment:Offset的方式，这是很自然的。由于 Protected Mode运行在32-bit系统上，那么Segment的两个因素：Base Address和Limit也都是32位的。IA-32允许将一个段的Base Address设为32-bit所能表示的任何值（Limit则可以被设为32-bit所能表示的，以2^12为倍数的任何指），而不象Real Mode下，一个段的Base Address只能是16的倍数（因为其低4-bit是通过左移运算得来的，只能为0，从而达到使用16-bit段寄存器表示20-bit Base Address的目的），而一个段的Limit只能为固定值64 KB。另外，Protected Mode，顾名思义，又为段模式提供了保护机制，也就说一个段的描述符需要规定对自身的访问权限（Access）。所以，在Protected Mode下，对一个段的描述则包括3方面因素：[Base Address, Limit, Access]，它们加在一起被放在一个64-bit长的数据结构中，被称为段描述符。这种情况下，如果我们直接通过一个64-bit段描述符来引用一个段的时候，就必须使用一个64-bit长的段积存器装入这个段描述符。但Intel为了保持向后兼容，将段积存器仍然规定为16-bit（尽管每个段积存器事实上有一个64-bit长的不可见部分，但对于程序员来说，段积存器就是16-bit的），那么很明显，我们无法通过16-bit长度的段积存器来直接引用64-bit的段描述符。 怎么办？解决的方法就是把这些长度为64-bit的段描述符放入一个数组中，而将段寄存器中的值作为下标索引来间接引用（事实上，是将段寄存器中的高13 -bit的内容作为索引）。这个全局的数组就是GDT。事实上，在GDT中存放的不仅仅是段描述符，还有其它描述符，它们都是64-bit长，我们随后再讨论。 GDT可以被放在内存的任何位置，那么当程序员通过段寄存器来引用一个段描述符时，CPU必须知道GDT的入口，也就是基地址放在哪里，所以 Intel的设计者门提供了一个寄存器GDTR用来存放GDT的入口地址，程序员将GDT设定在内存中某个位置之后，可以通过LGDT指令将GDT的入口地址装入此积存器，从此以后，CPU就根据此积存器中的内容作为GDT的入口来访问GDT了。 GDT是Protected Mode所必须的数据结构，也是唯一的——不应该，也不可能有多个。另外，正象它的名字（Global Descriptor Table）所揭示的，它是全局可见的，对任何一个任务而言都是这样。 除了GDT之外，IA-32还允许程序员构建与GDT类似的数据结构，它们被称作LDT（Local Descriptor Table），但与GDT不同的是，LDT在系统中可以存在多个，并且从LDT的名字可以得知，LDT不是全局可见的，它们只对引用它们的任务可见，每个任务最多可以拥有一个LDT。另外，每一个LDT自身作为一个段存在，它们的段描述符被放在GDT中。 IA-32为LDT的入口地址也提供了一个寄存器LDTR，因为在任何时刻只能有一个任务在运行，所以LDT寄存器全局也只需要有一个。如果一个任务拥有自身的LDT，那么当它需要引用自身的LDT时，它需要通过LLDT将其LDT的段描述符装入此寄存器。LLDT指令与LGDT指令不同的时，LGDT指令的操作数是一个32-bit的内存地址，这个内存地址处存放的是一个32-bit GDT的入口地址，以及16-bit的GDT Limit。而LLDT指令的操作数是一个16-bit的选择子，这个选择子主要内容是：被装入的LDT的段描述符在GDT中的索引值——这一点和刚才所讨论的通过段积存器引用段的模式是一样的。

•根据文档提示进行gdb调试

按照文档正常设置断点



进行了x/Ni ADDR命令的操作，在这个期间我犯了非常2的错误，就是没能理解N,ADDR的含义，现在理解了，这两个是你要填写的参数。N是连续执行的指令的反汇编代码数目， ADDR是开始反汇编的起始内存地址。

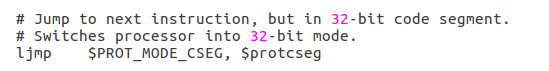


•问题回答

【1】



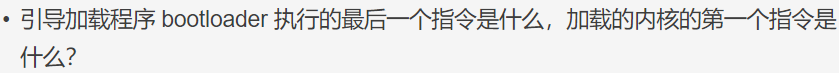
real model运行在16-bit系统上，protect model运行在32-bit系统上，所以当real model模式切换到protect model 是，就是16位到32位的切换。



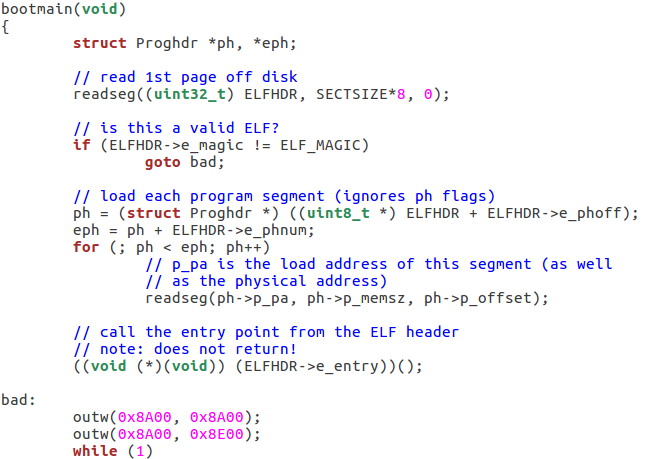
切换位置

长跳转指令 ljmp $PROT\_MODE\_CSEG, $protcseg 跳转到下一条代码，目的是跳过剩余的16位指令。此处我们看到新的GDT已经发挥作用，seg= PROT\_MODE\_CSEG, offset=protcseg，因为CSEG的基地址为0，则程序跳转到了代码段的protcseg偏移处，protcseg它就在下面，里面的指令重新写入了DS、ES等段寄存器的值，最后一句跳转到bootmain函数，开始读入内核，实模式跳转到保护模式完成。

【2】



由于加载引导加载程序执行完boot/boot.S之后跳转到boot/boot.c执行bootmain函数，所以最后一个指令为如图：



【3】



跳转到JOS内核后，执行的第一个文件是kern/entry.s。

 movw   %ax,       %bx         #把32位的EAX寄存器值传送给32为的EBX寄存器值

【4】

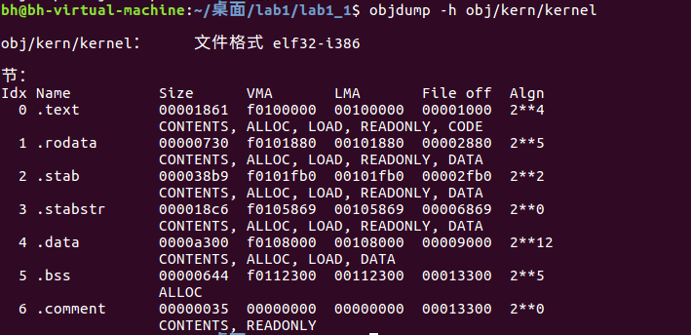


可以通过终端调试信息获取信息。

什么是ELF文件？当你在编译并且链接了像JOS内核这样的C语言程序之后，编译器会把C语言源文件(.c后缀)转换为目标文件(.o后缀)。目标文件中包含的是机器直接能够执行的机器指令。链接器在把所有的目标文件组合成一个单独的二进制映像(binary image)，比如obj/kern/kernel。这类二进制映像文件就是ELF格式的。

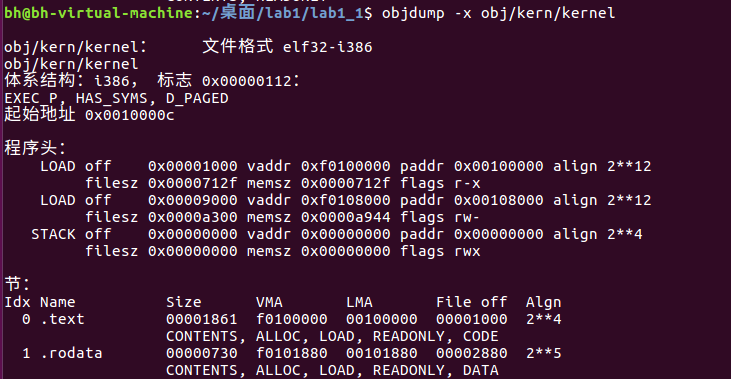
在图中我们会发现这个可执行文件的所有段的信息，其中不仅仅包括我们之前提到的那四个段，还有一些其他的，他们主要用于存放一些debug信息等等。

　　在每一个段中都有两个比较重要的字段，VMA(链接地址)，LMA(加载地址)。其中加载地址代表的就是这个段被加载到内存中后，它所在的物理地址。链接地址则指的是这个段希望被存放到的逻辑地址。



每一个ELF文件中都有一个Program Headers Table，用于指明ELF文件中哪些部分被加载到内存，以及被加载到内存中的地址。你可以通过输入下述指令来获取kernel的Program Headers Table的信息。其中Program Header中列出的是所有被加载到内存中的段的信息，这也是Program Headers Table的表项。每一个表项图中都把这个表项中涉及到的所有字段都列出来了。可见有一些段最后没有被加入到内存之中。在上图中，那些需要被加载到内存的段被标记为LOAD。

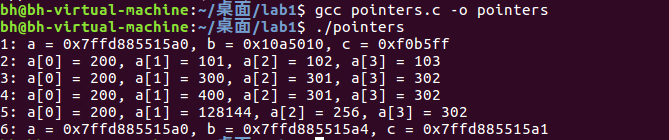
　　BIOS通常会把boot sector加载到内存地址0x7c00处，这是boot sector的加载地址，也是boot sector的链接地址。我们可以通过boot/Makefrag文件中的-Ttext 0x7c00语句设置boot sector的链接地址，并且这个链接地址后来会被链接器所使用，保证链接器产生正确的代码。



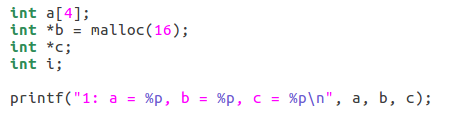
【练习4】

首先编译运行了pointers.c文件，得到如下输出

（使用命令gcc pointers.c -o pointers && ./pointers）



•第一句：



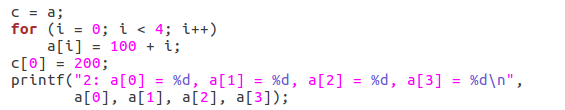
程序申明了一个int型变量i ，int型数组a[4]，两个int型指针b, c

第一行打印出a数组的起始位置，即a[0]的首地址；

打印b指针所指向的操作系统分配给它的起始空间；（malloc动态内存分配）

打印c所指向的位置；（c未定义）

•第二句：



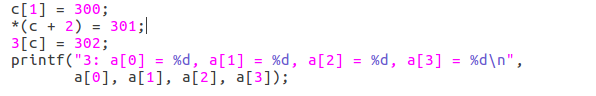
c = a让c和a指向统一内存地址，即a的地址

for循环使得a数组的数分别为a[0]=100; a[1]=101; a[2]=102; a[3]=103

c[0]=200,使得a[0]=c[0]=200;

所以打印结果a[0]才会变成200

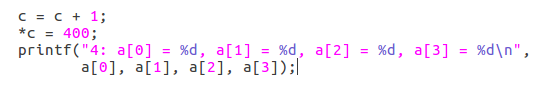
•第三句



将c[1],c[2],c[3]分别赋值为300，301，302；

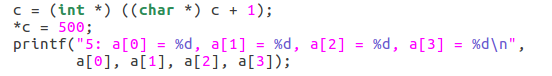
由于c ,a，指向统一内存空间，里面的值相同，所以打印出a的结果和修改过后的c相同

•第四句

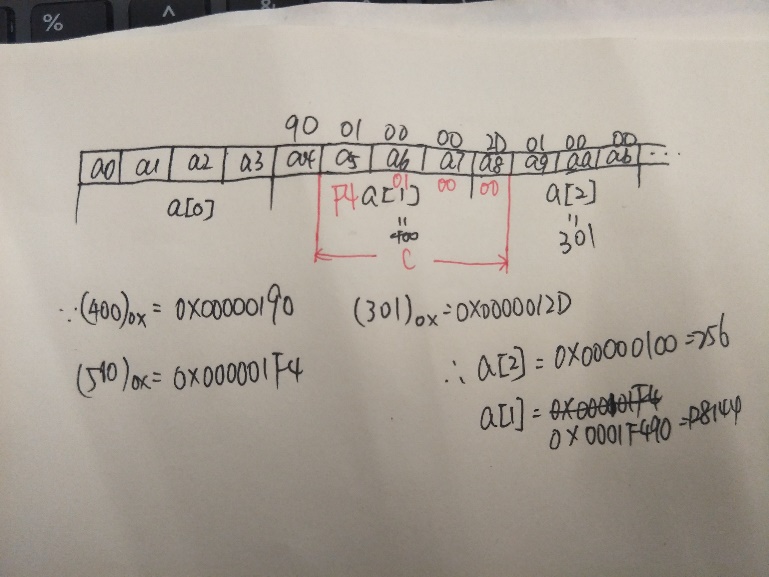


将c位置往后挪，指向a[1]所在位置并将a[1]值修改为400；

•第五句



第一句将c的地址往后移了一位，所以c现在指向的内存地址为0x7ffd885515a5;

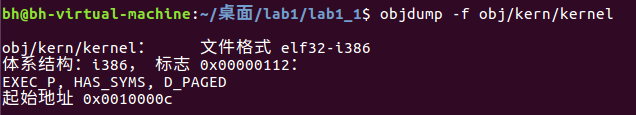


•第六句

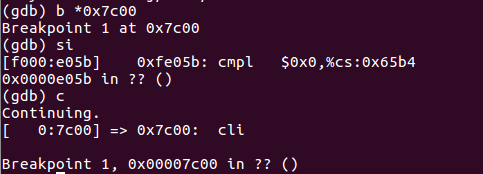


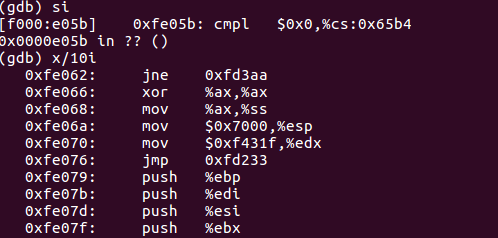
将a强制转换为int型加1的地址赋给b；将a强制转换成char型加1又强制转换成int型赋给c；打印a, b, c内存地址；

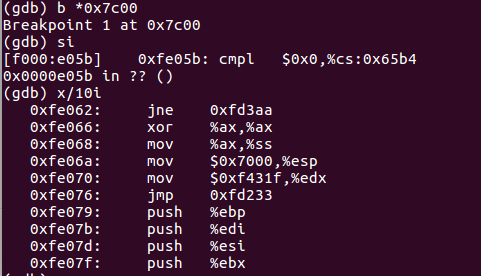
【练习5】



除了节信息，ELF头中还有一个对我们很重要的字段e\_entry。这个字段包含程序入口点的链接地址:程序text节的内存地址，程序应该在该地址开始执行。

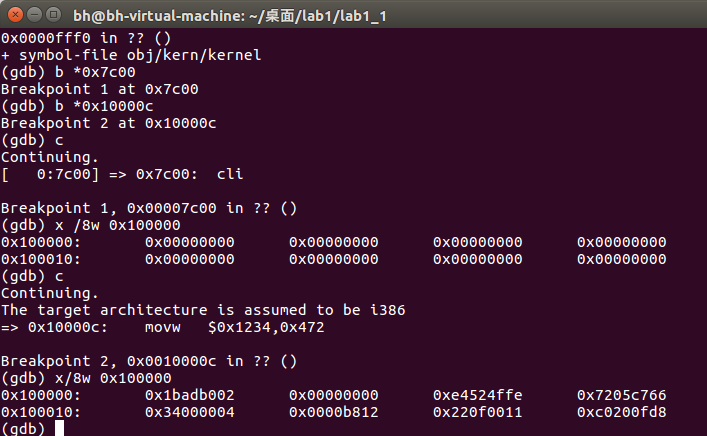




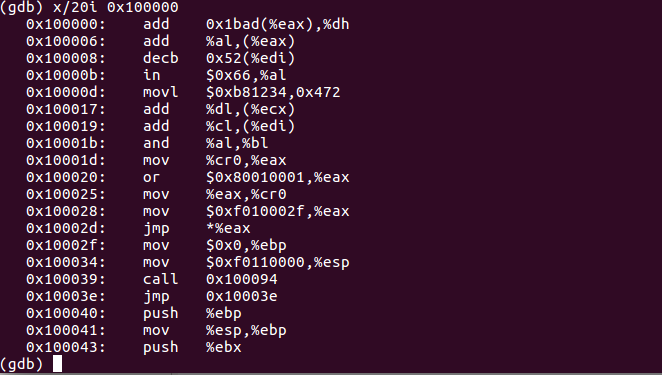
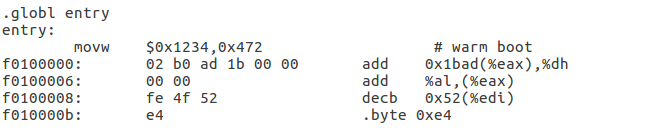


【练习6】

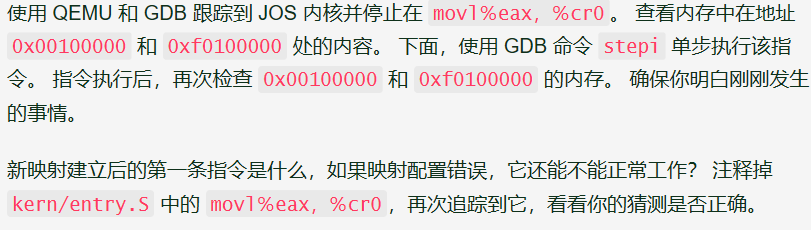
以下是根据练习要求运行出来的结果



通过gdb调试查看地址0x100000之后的20行代码，与obj/kern/kernel文件中的汇编代码比较（设置断点处），发现汇编代码相同，所以可以得知，内核代码是从0x10000c开始的，在进入内核的那一刻，内核程序加载到0x100000处，所以两次地址的内容才会不相同。



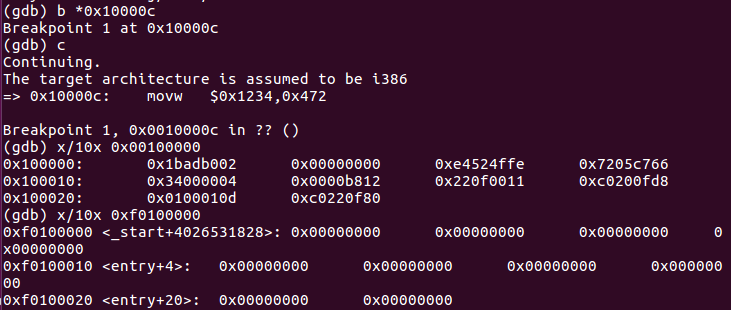
【练习7】



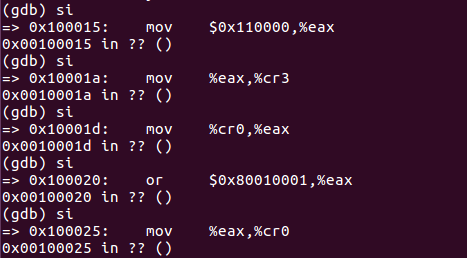
由于上面的实验，我们知道，内核代码的地址为0x10000c，所以在此位置设置断点，进行调试，可发现在没有执行之前，地址0xf0100000处的内容全为0，在单步调试至之后，两地址内容变成一样。原因：指令执行之后，其物理地址的内容被映射到虚拟地址中。（在这里查看地址内容可以使用两种命令：

（ （gdb）x/10x ADDR || (gdb) x/10xb ADDR ）

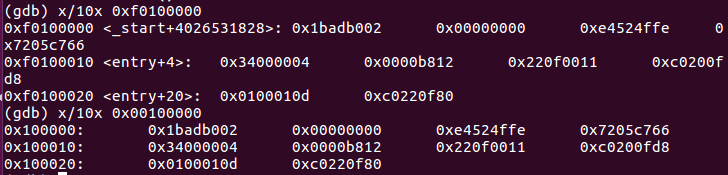
查看地址中的内容：



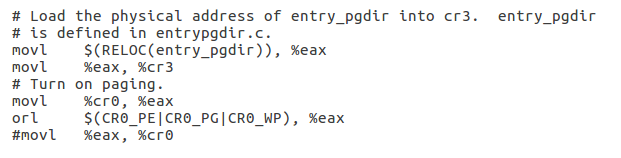
单步调试过程：

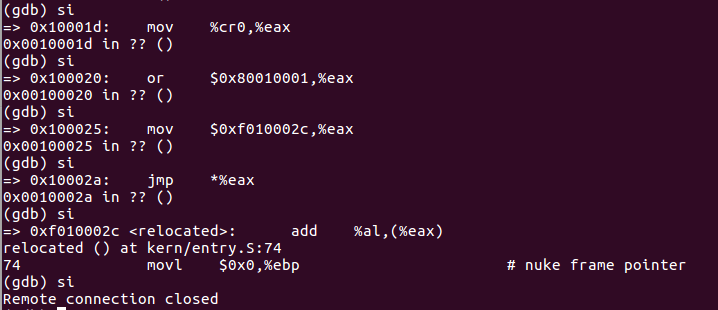


执行完指令之后的地址内容（变成一样了）：



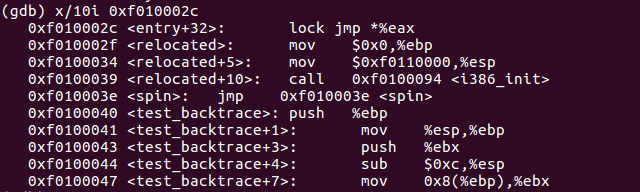
将注释掉之后，地址0xf010002c出现了问题，查看其汇编代码







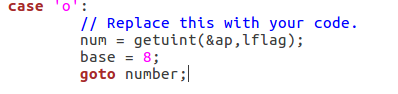
将注释删除修改为原来之后，代码恢复了正常



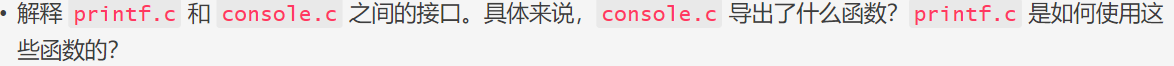
【练习8】



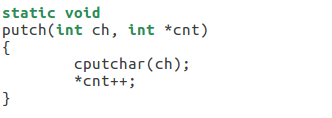
找到文件lab1\_1/lib/printfmt.c，仿照上面十进制的写法补充完整。



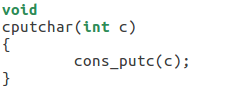
**•回答下面问题**

•

Printf.c:

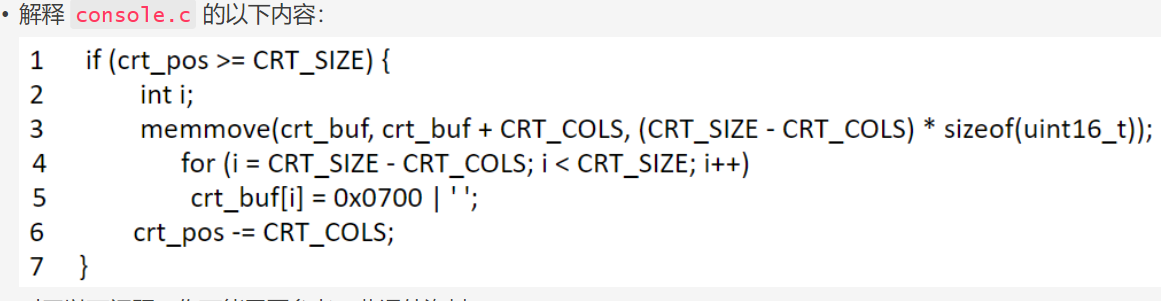


Console.c:



Console.c导出了函数cputchar(int c)；cputchar被printf.c通过Putch调用（资料：这里主要是说明所有的printf相关函数(JOS中),实质上都是“一层外壳”）

•



crt\_buf：一个字符数组缓冲区，里面存放着要显示到屏幕上的字符

crt\_pos：表示当前最后一个字符显示在屏幕上的位置

CRT\_SIZE：80\*25（c源程序中考虑的就是一种非常常见的文字模式，80x25文字模式）

CRT\_COLS：原来的长度

1：表示输入字符已经超过一页所显示的最大字符

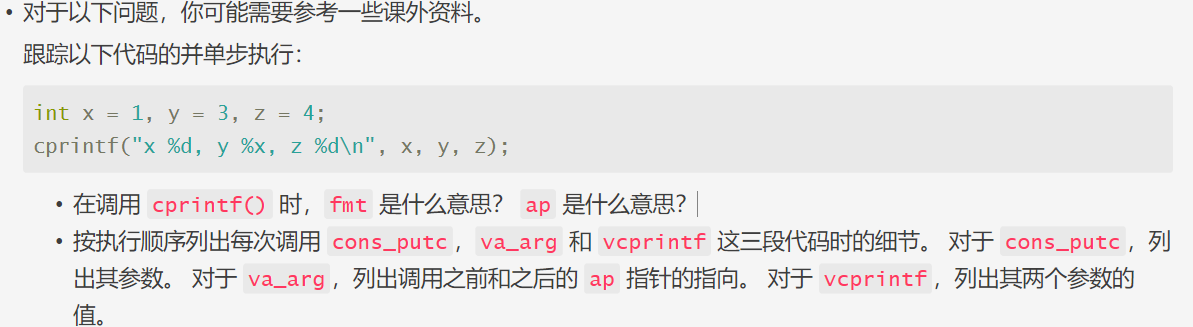
3：memcpy操作就是把crt\_buf字符数组中1~79号行的内容复制到0~78号行的位置上。

4：把最后一行，79号行都变成空格

6：修改crt\_pos的值

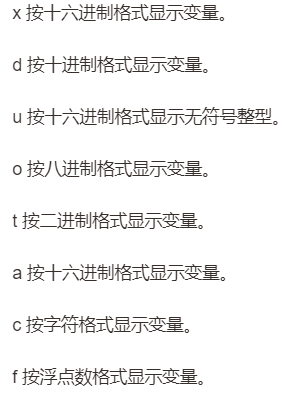
最终呈现效果：屏幕滚动，逐行上移，最后空出一行，光标放在最后一行行首

•

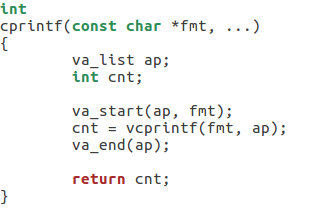


【补充资料】

查看内存gdb命令 --（gdb）x/N<x> addr



解答：这串代码位于文件kern/printf.c中

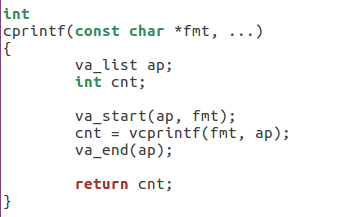


va\_start： 函数名称,读取可变参数的过程其实就是在堆栈中,使用指针,遍历堆栈段中的参数列表,从低地址到高地址一个一个地把参数内容读出来的过程

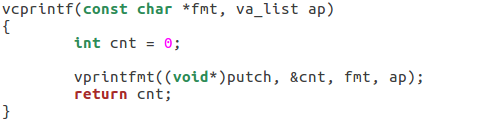
va\_end：清空va\_list可变参数列表

fmt指向的是显示信息的格式字符串，在这段代码中，它指向的就是"x %d, y %x, z %d\n"字符串；ap是va\_list类型的，ap指向所有输入参数的集合；

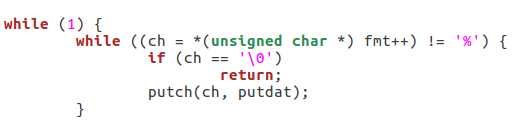
Printf.c

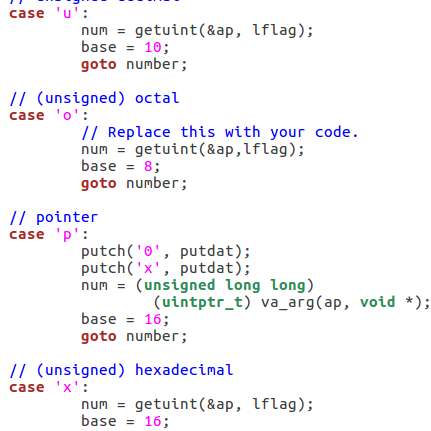


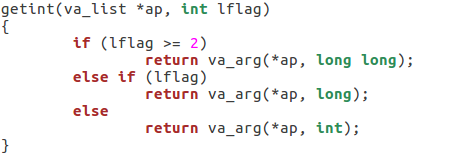
Kern/printf.c



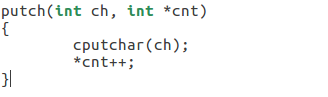
Lib/printmt.c



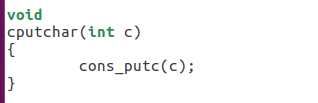




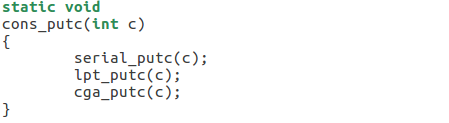
Kern/print.c



Kern/console.c



console.c

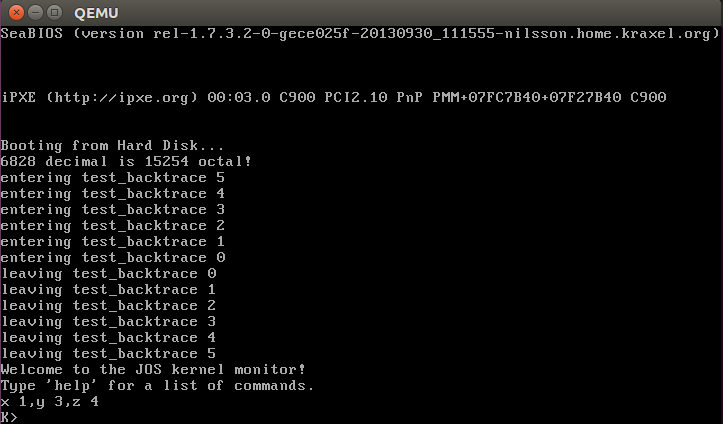


这里对va\_arg进行了一次调用，调用前ap中包括x，y，z三个参数的内容：1，3，4。调用完成后只剩下y，z的内容：3,4;

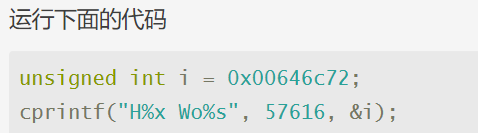
Vcprintf两个参数的值：fmt:”x %d, y %x, z %d\n”; ap:1,3,4;

Cons\_putc参数：每次要打印在屏幕上的字符；(x 1,y 3,z 4)

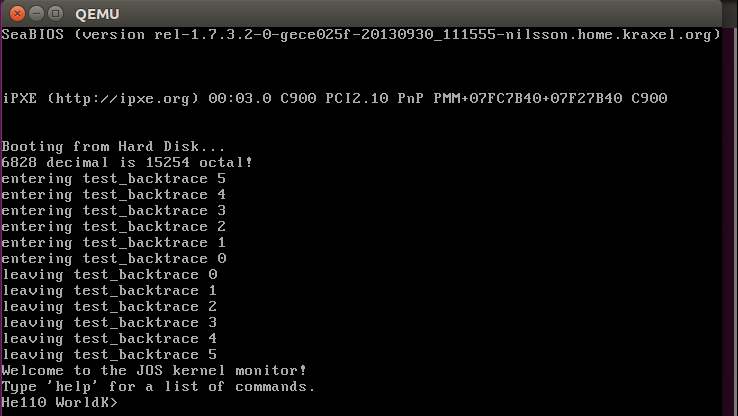
代码运行结果(在kern/monitor.c中添加代码，重新编译内核然后执行make qemu)：



••



执行显示：

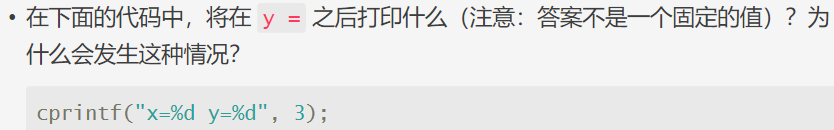


%x:按十六进制输出57616；（ello）

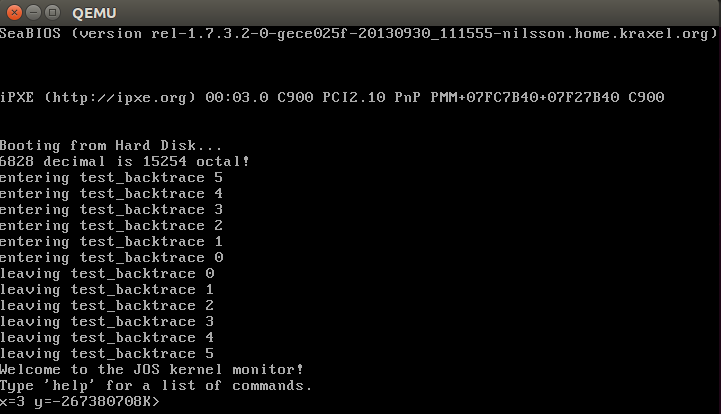
%s：按ascii码输出i处的内容；（存储为72 6c 64 00，转换为ascii码为：rld）

如果按照大端机器，则修改i的值为：0x726c6400;另一个值不用改。（小端模式，代表字的最高位字节存放在最高位字节地址上。大端模式相反）

•



打印结果：



Y并没有指定参数，会打印一个随意的值

•



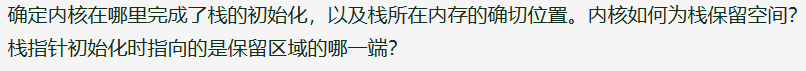
Va\_arg每次是将地址增长取出下一参数的地址的，所以可以看出编译器是从右往左顺序入栈的，如果要颠倒入栈顺序，则将增长地址改为用减法得到下一参数的地址，这样同样可以做到先前的效果。

**挑战**

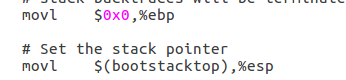




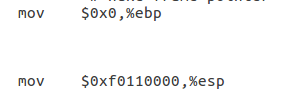
【练习9】



内核完成栈初始化的地方：kern/entry.s中完成初始化；

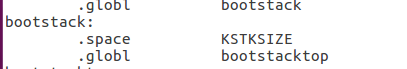


栈所在内存的确切位置：0xf0110000



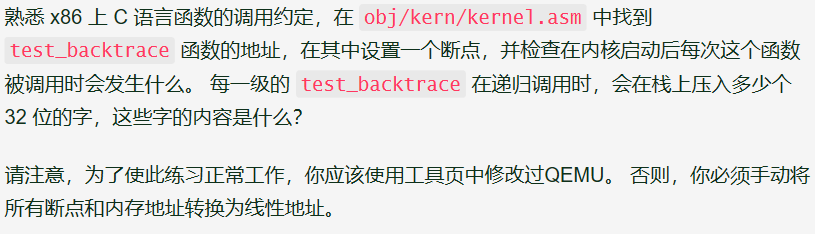
内核为栈保留空间：以下指令用于保留栈空间

entry.s



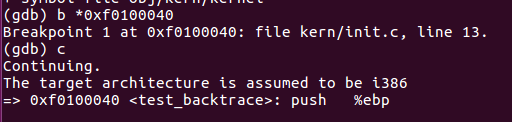
栈指针初始化时指向保留空间高位端

【练习10】



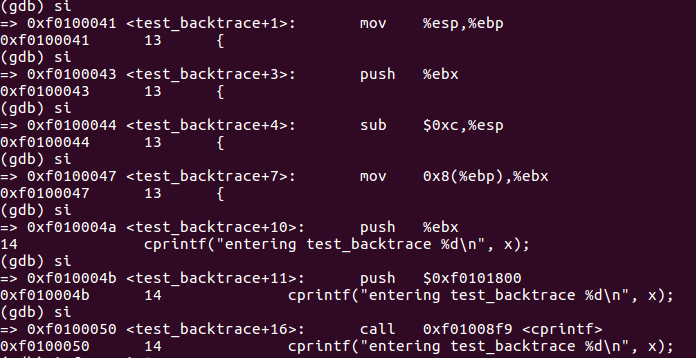
Obj/kern/kernel.asm中

得知<test\_backtrace>地址为0xf0100040





Test\_backtrace(5):





Test\_backtrace(4):



Test\_backtrace(3):



Test\_backtrace(2):



Test\_backtrace(1):



Test\_backtrace(0):



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

esp：0xf010ffe0 ebp : 0xf010fff8

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，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。

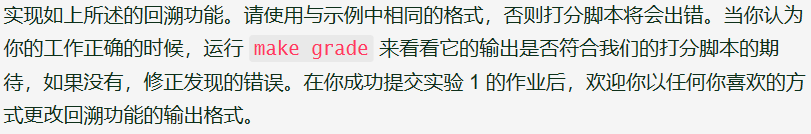
所以上述4条汇编指令，执行完成后，esp，ebp寄存器的值变化为

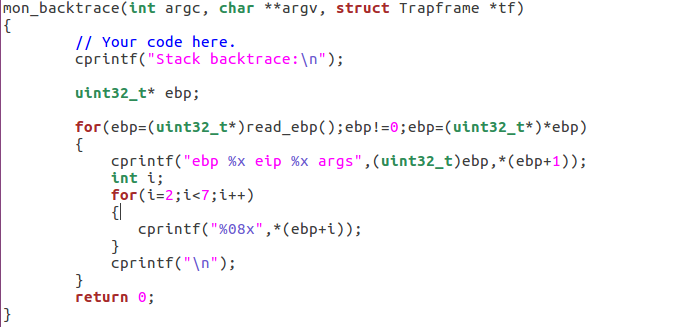
esp : 0xf010ffc0 ebp：0xf010ffd8

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。

每一级会压入3个32位字，内容如上。

【练习11】

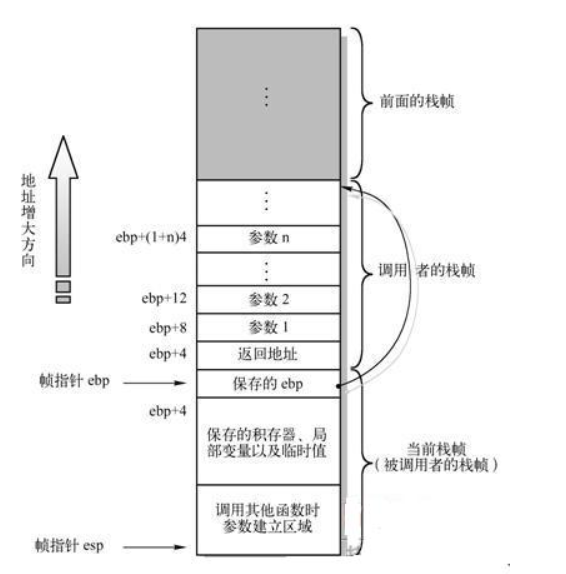




//打印args的5个参数

//打印ebp,eip值；ebp:当前栈帧ebp的值；eip:下一个要执行的指令地址

//循环栈帧

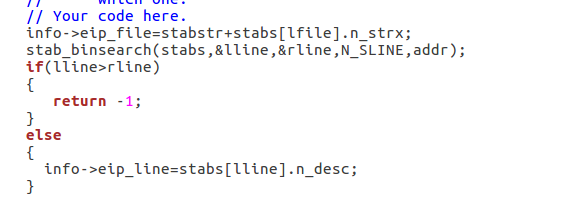


这个子程序的功能就是要显示当前正在执行的程序的栈帧信息。包括当前的ebp寄存器的值，这个寄存器的值代表该子程序的栈帧的最高地址。eip则指的是这个子程序执行完成之后要返回调用它的子程序时，下一个要执行的指令地址。后面的值就是这个子程序接受的来自调用它的子程序传递给它的输入参数。

【练习12】



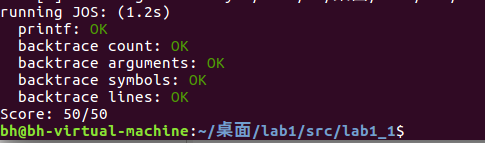
Kern/kdebug.c

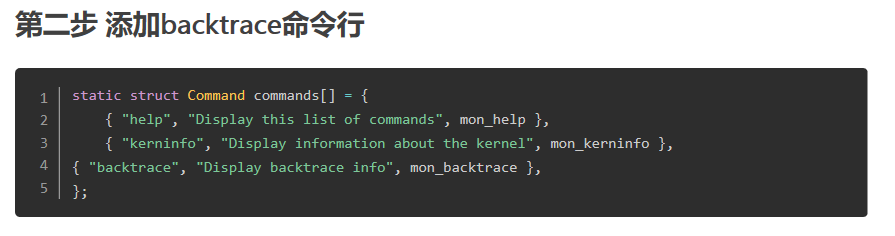


//二分法查找

//查看地址行号

结果：





【实验中遇到的问题】

1. 在这次实验中，做到练习7的时候，我设置断点在内核代码入口处时，打印练习中要求的内存地址信息时全部为0，于是我重新配置了qemu以及加载启动程序和内核。我将原先的文件删除并重新解压实验资源，发现make qemu无法执行，后来解决方法是将回收站原本的文件全部清空。
2. 解决练习7时起初我调试代码时设置了断点却没有运行它，导致查看的地址里的内存都为0。正确的做法是设置断点之后c在进行单步调试。
3. Linux gdb调试器用法全面解析：

<https://www.cnblogs.com/mingcaoyouxin/p/4225619.html>