Lab1

**实验介绍:**

本次的实验分为三个部分，第一部分主要熟悉一下X86汇编语言，QEMU的X86模拟器和PC开机的引导过程。第二部分检查内核的引导加载程序，它位于实验源代码的boot文件夹中。最后，第三部分深入研究内核本身的初始化，相关代码在kernel目录中

**Part 1 PC引导程序**

**练习 1. 熟悉6.828 reference materials page面上提供的汇编语言的资料。你不需要马上阅读这部分的资料，但是实验的过程中你肯定会用这一部分资料。我们建议阅读一下 Brenna’s Guide to Inline Assembly 的 “The Sytax” 这一部分的内容。这一部分内容给出了我们将在JOS中与GUN汇编器一起使用的AT&T汇编语法的一个很好的（简洁的）描述。**

**答：AT&T 汇编和 Intel 汇编，是两种不同汇编语言格式，与具体 CPU 关系不大，只是 Intel 汇编格式基本只用在自家的 x86 系列 CPU 上，而 AT&T 汇编格式在多种 CPU 上都可以使用（x86，power，VAX 等等）。经过查阅网上资料，得知**

**AT&T 汇编语法大致有以下几条规则：**

**1.寄存器通过在名称前加百分号（%）前缀引用。example：为使用 eax 寄存**

**器，汇编代码中将使用%eax。（如果在 C 中内联汇编的话，C 代码必须指定两个百分号，才能在转给汇编器的输出中形成一个百分号）。**

**2.源寄存器总是在目的寄存器之前指定。example：在 mov 语句中，这意味着**

**mov a，b 将 寄存器 a 中的值 内容 copy 到寄存器 b 中。**

**3.操作数的长度由汇编语句的后缀指定。b 代表 byte，w 代表 word，l 代表**

**long。在 IA-32 上，将一个长整型从 eax 寄存器移动到 ebx 寄存器中，需要指定movl %eax，%ebx。**

**4.间接内存引用（指针反引用）需要将寄存器包含在括号中。example：movl**

**（%eax）,%ebx 将寄存器 eax 的值指向的内存地址中的长整型 copy 到 ebx 寄存器中。**

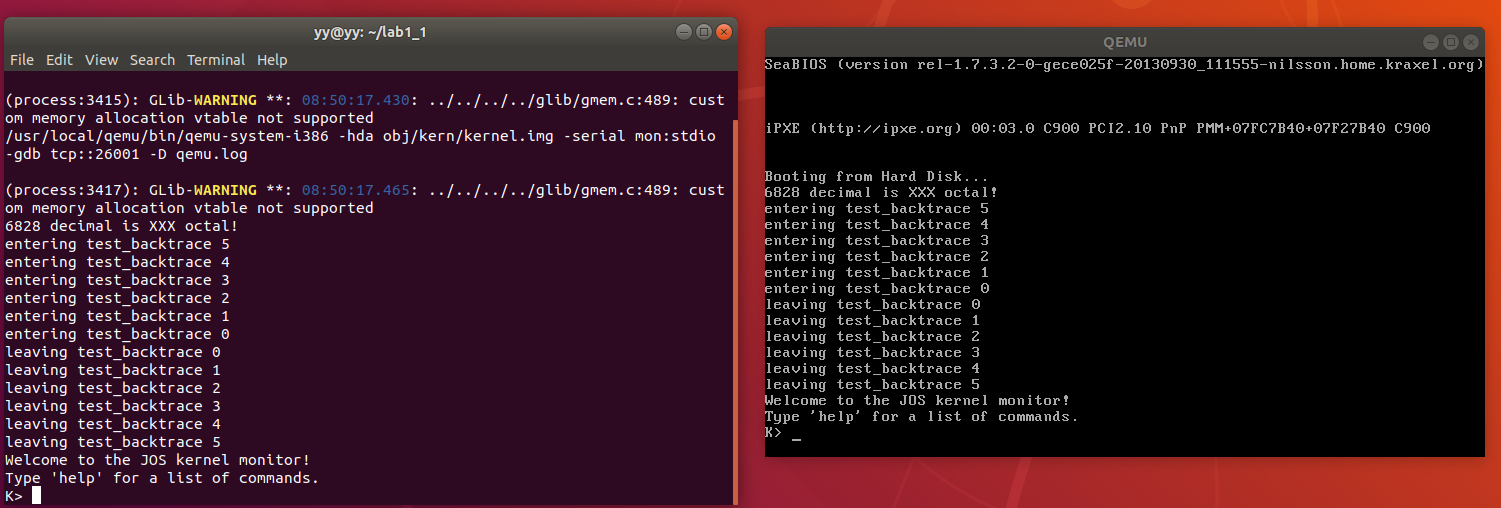
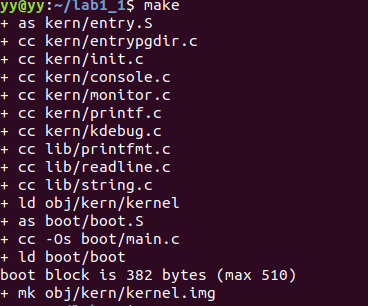
**5.offset（register）指定寄存器值与一个偏移量联用，将偏移量加到寄存**

**器的实际值上。example：8（%eax）指定将 eax+8 用作一个操作数。该表示法主要用于内存访问，例如指定与栈指针或帧指针的偏移量，以访问某些局部变量。**

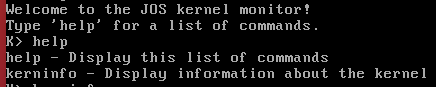
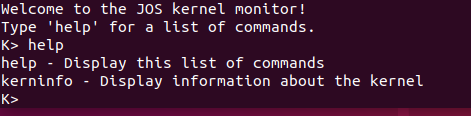
开始前，我们解压lab1 中的文件到你的主机里，接下来在lab文件夹下输入make（或者 在BSD的系统下输入gmake）去构建6.828最小的加载启动程序和内核。然后运行QEMU，创建 obj/kern.kernel.img 上面提供的文件作为仿真PC的“虚拟硬盘”的内容。这个硬盘镜像包含了我们需要的启动代码bootloader（obj/boot/boot） 以及我们使用的内核（obj/kernel）

Make qemu的结果如图，从“Booting from Hard Disk…”后所有内容都是从我们的JOS 内核打印出来的；

现在内核监控器只支持两种命令，help 和 kerninfo

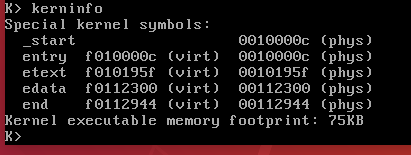
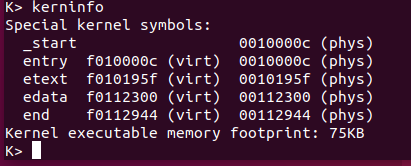


Help



Terminal qemu

Kerninfo



Terminal qemu

**PC的物理地址空间**

一个PC的物理地址空间具有以下大致的布局。

|  |
| --- |
| +------------------+  <- 0xFFFFFFFF (4GB)  |      32-位       |  |  内存映射        |  |     设备         |  |                  |  /\/\/\/\/\/\/\/\/\/\    /\/\/\/\/\/\/\/\/\/\  |                  |  |      未使用      |  |                  |  +------------------+  <- 取决于RAM的数量  |                  |  |                  |  |   扩展内存       |  |                  |  |                  |  +------------------+  <- 0x00100000 (1MB)  |     BIOS ROM     |  +------------------+  <- 0x000F0000 (960KB)  |  16-位 设备,     |  |  扩展ROM        |  +------------------+  <- 0x000C0000 (768KB)  |   VGA 显示器     |  +------------------+  <- 0x000A0000 (640KB)  |                  |  |    低地址空间    |  |                  |  +------------------+  <- 0x00000000 |
|  |

第一代PC是基于16位的Intel8088处理器设计的，只能处理1MB的物理内存。因此早期的PC的物理地址空间都是以0x00000000开始，但不是以现在0xFFFFFFFF结束的而是0x000FFFFF。前640KB区域被标记为“低地址空间”，是早期PC唯一可以使用的随机访问存储区（RAM），事实上，早期的PC只配有16KB,32KB,或者64KB的RAM!

从0x000A0000 到0x000FFFFF 的384KB区域被保留下来用作硬件的一些特殊用途，比如视频播放的缓冲区和保存固件的非易失性内存空间。这一部分保留区域中最重要的一部分是基本的输入输出系统（简称BIOS）,占据了从0x000F0000 到 0x000FFFFF这64KB的区域。早期的PC机上BIOS 是保存在只读内存区域，现在的PC机上BIOS 存储在新型的闪存存储器上。BIOS负责执行系统的基本初始化，例如激活视频卡或者是检查内存安装的总量。在执行完初始化操作后，BIOS 将从一些合适的地方加载操作系统，例如软盘，硬盘，光盘，或者网络上，然后将机器的控制权交给操作系统。

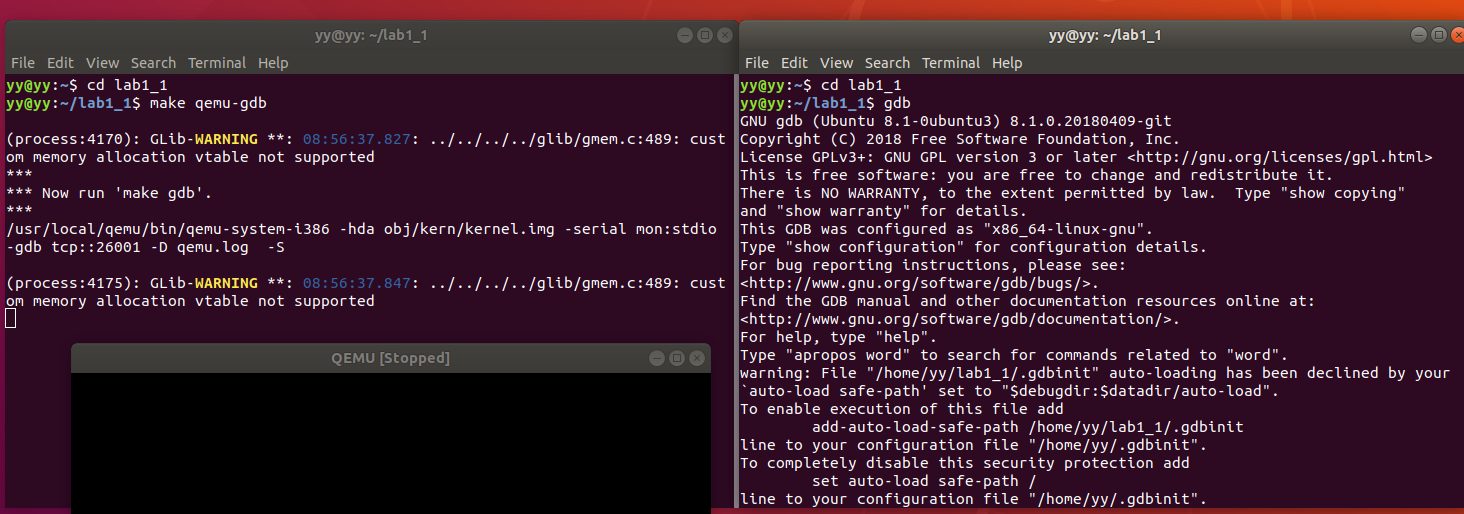
当英特尔的80286 和80386的处理器最终打破了“一兆字节的瓶颈”，可以支持16MB和4GB的物理地址空间，可是PC的架构师依然要保留原来1MB的低地址的物理空间布局以保证现有软件的向下兼容性。因此，现代PC有一个在0x000A0000到0x00100000 的物理地址上有一部分“空洞”，将RAM划分为两段，“低地址内存空间”（第一个640KB的内存空间）和“扩展内存空间”（其余的部分）。此外，一些在32位机器的物理地址空间（最重要的是物理RAM）最顶端的一些空间，通常被BIOS 保存用作32为的PCI的设备使用。

最近X86的处理器能支持超过4GB的物理内存，因此RAM可以拓展到0xFFFFFFFF之上。在这种情况下BIOS 必须要在32位系统内存高层预留第二个“空洞”，用于为一些32位的设备预留映射空间。由于设计的限制，JOS只能使用前265MB的物理内存，现在我们假设电脑仅有一个连续的32位的物理地址空间。对于复杂的物理空间结构的管理和多种多样的已有设备的兼容，一直是OS不得不面对的一个问题，也是是未来发展中会一直存在的挑战。

**BIOS**

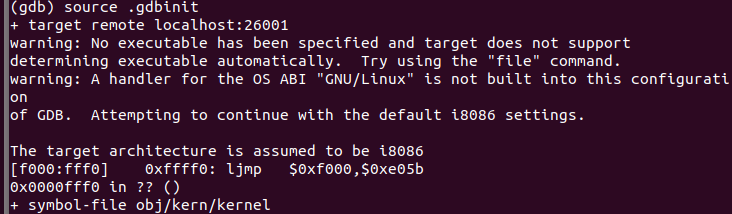
在这一部分实验中，将使用QEMU的调试功能来检查一个IA-32兼容的计算机的引导过程。

打开两个终端窗口，再一个里面输入make qemu-gdb(或者是make qemu-nox-gdb)。这个将启动QEMU，但是在QEMU将处理器执行第一个指令前停止，并等待来自GDB的调试连接。在第二个终端窗口，在运行make那个目录运行gdb，



出错：gdbinit脚本无法使用

解决方法：source .gdbinit



注意这一行:

[f000:fff0] 0xffff0: ljmp $0xf000,$0xe05b

这是GBD对要执行的第一条指令拆解。从这个输出我们可以总结出以下几件事情：

(1)IBM 的PC 是从物理地址是0x000ffff0的内存位置开始执行的，这是为ROM BIOS 保留的64KB的区域的最顶端。

PC开始执行时cs = 0xf000 和 IP = 0xfff0

(2)第一条被执行的指令是jmp 指令，跳转到分区地址是cs = 0xf000 和IP = 0x305b

为什么QEMU 是这样开始的？这是完全按照英特尔设计的8088处理器工作的，也就是模拟了IBM的PC机上的处理器工作模式。因为在PC中， BIOS 是通过 “硬件接线”连接到物理地址是0x000f0000-0x000fffff这段范围上，这种设计能确保BIOS在电脑后充电后开机或者是任何操作系统重启后立刻获取到机器的控制权。这十分的重要，因为重启后RAM中不会有任何的软件可供处理器执行。QEMU 模拟器有自己的BIOS，它位于处理器模拟的物理地址空间的位置上。在处理器复位的时候，（模拟）处理器进入实模式将CS设置到0xf000并且将IP设置到0xfff0 ,以便从（CS：IP）地址开始执行。分段机制地址0xf000:fff0如何转为物理地址的呢？

为了回答这个问题，我们必须了解一些实际模式寻址的知识。在实际模式（PC的启动模式），地址翻译是根据公式： 物理地址 = 16 \* 段 + 偏移。因此。当将PC 的CS设置为0xf000 IP设置为0xfff0,物理地址就是：

16 \* 0xf000 + 0xfff0 # 在16进制下乘以16

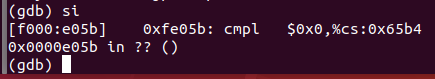
= 0xf0000 + 0xfff0 # 只追加一个0

= 0xffff0

0xffff0 是一个BIOS（0x100000）最后16个字节的地址。因此， BIOS做的第一件事是将jmp倒退到在BIOS较前的位置。毕竟在16字节中能完成的工作是很有限的

练习 2. 使用GDB的si (Step Instruction)指令来跟踪ROM BIOS 中的几个指令，并且尝试这些指令是要做什么的。你可能需要查看Phil Storrs I/O Ports Description,以及6.828 reference materials page 其他的资料。并不需要弄清楚所有的细节-仅需要先弄清楚BIOS工作的一个思路。

Si指令跟踪



当BIOS 运行起来，他建立了一个中断描述符表，并初始化例如VGA设备各种变量。这就是为什么会从QEMU窗口上看到“Starting SeaBIOS” 信息。

当初始化了PCI 总线和所有BIOS知道的重要设备后。他开始搜索启动设备例如软盘，硬盘，光盘等。最终，当找到了一个启动磁盘，BIOS就开始从磁盘里面读取启动加载程序，并将电脑的控制权转让给启动程序。

**Part2：引导加载程序**

PC的软盘和硬盘以512个字节为单位划分成区域，每个区域称为一个扇区。扇区是磁盘的最小访问单位：每个读、写操作的大小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，因为这是引导加载程序代码所在的位置。当BIOS找到可启动的软盘或硬盘时，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，将控制权传递给引导程序。引导程序又称为bootloader。像BIOS加载地址一样，这些地址是随意设定的 - 但对于PC它们是固定和标准化的。

在PC的演进过程中，从CD-ROM启动的能力要晚很多，因此，PC架构师借此机会重新思考引导过程。因此，现代BIOS从CD-ROM启动的方式有点复杂（且更强大）。CD-ROM使用2048字节大小的扇区，而不是512字节，并且在将控制转移给它之前，BIOS可以将更大的启动映像从磁盘加载到内存（而不仅仅是一个扇区）中。有关详细信息，请参阅 "El Torito" Bootable CD-ROM Format Specification。但是，对于6.828，我们将使用传统的硬盘启动机制，这意味着我们的引导加载程序必须放进512字节的区域中。引导加载程序由两个文件组成，一个汇编语言源文件boot/boot.S和一个C语言源文件boot/main.c。仔细阅读这些源文件，并确保你了解发生了什么。引导加载程序必须执行两个主要功能：

首先，引导加载程序将处理器从实模式切换到32位保护模式，因为只有在该模式下，软件才能访问处理器物理地址空间中1MB以上的所有内存。保护模式在 PC Assembly Language第1.2.7和1.2.8节中有简要介绍，并在英特尔架构手册中有详细介绍。在这个阶段上，你只需要理解，在保护模式下，将分段地址（段地址：偏移量的组合）转换为物理地址的方式不同，而转换后的偏移量为32位而不是16位。

其次，引导加载程序通过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。如果你想更好地了解这里的特定I/O指令是什么意思，请查看the 6.828 reference page “IDE硬盘驱动器控制器(IDE hard drive controller)”部分。在本课程中, 你不需要了解编程特定设备的情况：编写设备驱动程序实际上是操作系统开发中非常重要的一部分，但从概念或架构的角度来看，它也是最无趣的一部分。

在了解了引导加载程序的源代码后，查看文件obj/boot/boot.asm。该文件是我们的GNUmakefile在编译引导加载程序之后创建的引导加载程序的反汇编代码。从这个反汇编文件中可以很容易地查看所有引导加载程序代码所在的物理内存的确切位置，并且可以更轻松地监视GDB中的引导加载程序中发生的情况。同样，obj/kern/kernel.asm中包含的是JOS内核的反汇编结果，这通常对调试很有用。

你可以使用b 命令在GDB中设置地址断点。例如，b \*0x7c00设置地址0x7C00的断点。一旦执行停在了断点处，可以使用继续执行c与 si命令：c使QEMU继续执行，直到下一个断点（或者直至你在GDB按下 Ctrl-C）；si N的意思是连续执行N条指令后停下来。

要检查内存中的指令（GDB会自动打印出马上要执行的下一个指令，但并不会显示其他指令），请使用该 x/i命令。该命令的语法是 x/Ni ADDR，其中N是要反汇编的连续指令的数量，ADDR是要开始反汇编的存储器地址。

【实模式和保护模式<https://www.cnblogs.com/chenwb89/p/operating_system_002.html>】

练习3. 查看lab tools guide，特别是GDB命令部分。即使您熟悉GDB，这包括一些对操作系统工作非常有用且深奥GDB命令。

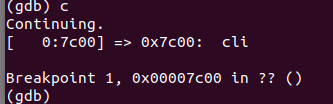
在地址0x7c00设置断点，这是引导扇区将被加载的位置。继续执行直到那个断点。跟踪boot/boot.S中的代码，使用源代码和反汇编文件 obj/boot/boot.asm来跟踪你在哪里。还可以使用GDB中的x/i命令来反汇编引导加载程序中的指令序列，并将原始引导加载程序源代码与obj/boot/boot.asm 和GDB中的反汇编进行比较。

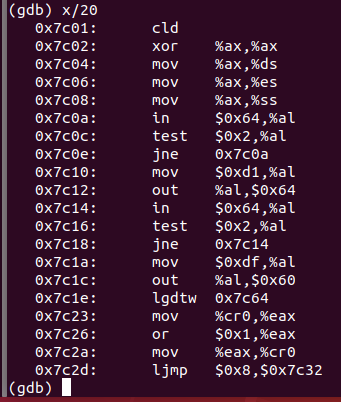
在boot/main.c中跟踪到bootmain（），然后进入readsect（）。识别与readsect（）中的每个语句相对应的确切的汇编指令。完成对readsect（）其余部分跟踪，并回到bootmain（），并找到用来从盘读取内核其余部分的for循环的开始和结束的位置。找出循环完成后运行的代码，在那里设置一个断点，并继续执行到该断点。然后再走完bootloader程序的其余部分

在0x7c00处设置断点 b \*地址



Continue ：c





从实模式切换到p保护模式，地址长度从16bits变为32bits

**下面回答以下问题：**

**处理器什么时候开始执行32位代码？如何完成的从16位到32位模式的切换？**

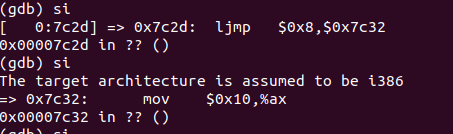
在上面的练习中，我们在地址 0x7c00 这一引导扇区将被加载的位置设

置了段点后，通过 si 单步执行时发现如图 8 处的代码。在执行了一个 ljmp 的指 令后，输出了一段提示“The target architecture is assumed to be i386”,

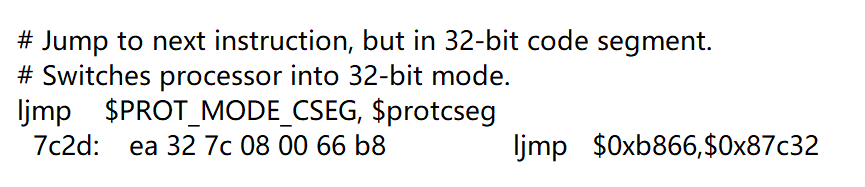
目标架设假设为 i386。而我们知道，i386 就是 intel 80386，是英特尔 32 位微

处理器的统称。所以我在这里猜测，这一指令就是使得处理器执行 32 位代码的

跳转位置



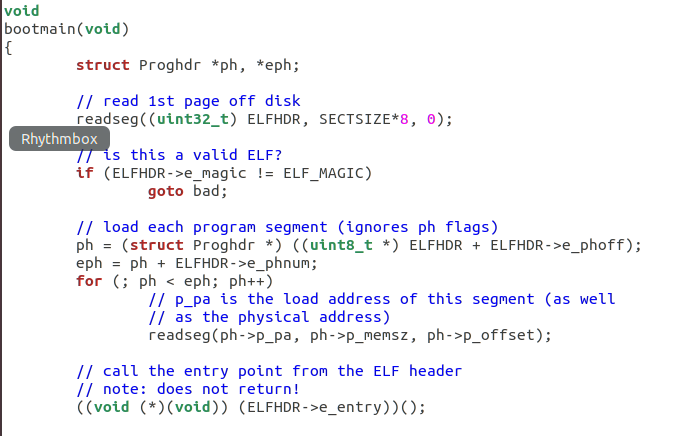
**Obj/boot/boot.asm**



根据注释我们可以看出来，此处，在地址为 0x7c2d 处，处理器转化为 32 位，

通过 ljmp 长跳转指令，从 16 位代码处跳转到 32 位代码处。之后开始执行 32 位代码。

**引导加载程序bootloader执行的最后一个指令是什么，加载的内核的第一个指令是什么？**



引导加载程序的最后一条指令是boot/main.c中bootmain函数最后的((void (\*)(void)) (ELFHDR->e\_entry))(); 这个第一条指令位于/kern/entry.S文件中，第一句 movw $0x1234, 0x472

Kern/entry.S



**内核的第一个指令在哪里？**

位于/kern/entry.S文件中

**引导加载程序如何决定为了从磁盘获取整个内核必须读取多少扇区？在哪里可以找到这些信息？**

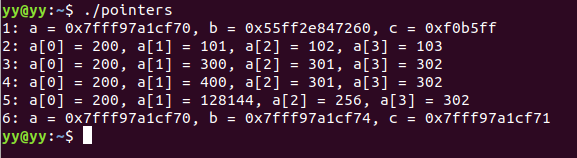
通过ELF program headers决定，他在操作系统内核映像文件的ELF头部信息里找到。

**加载内核**

**练习4. 阅读C中的指针编程.C语言的最佳参考是 Brian Kernighan和Dennis Ritchie（被称为“K＆R”）的C编程语言（中文翻译版）。**

**通读K＆R书中5.1（指针和地址）到5.5（字符指针和函数）的内容。然后下载pointers.c的代码，运行它，并确保你了解所有打印值的来源。特别是，请确保你理解第1行和第6行中指针指向哪里的地址，第2行到第4行的值是如何被写入的，以及为什么第5行中打印的值看起来像是错乱的。还有其他有关C指针的参考书（例如，A tutorial by Ted Jensen，其中大量引用了K＆R的内容）。**

编译运行pointers.c

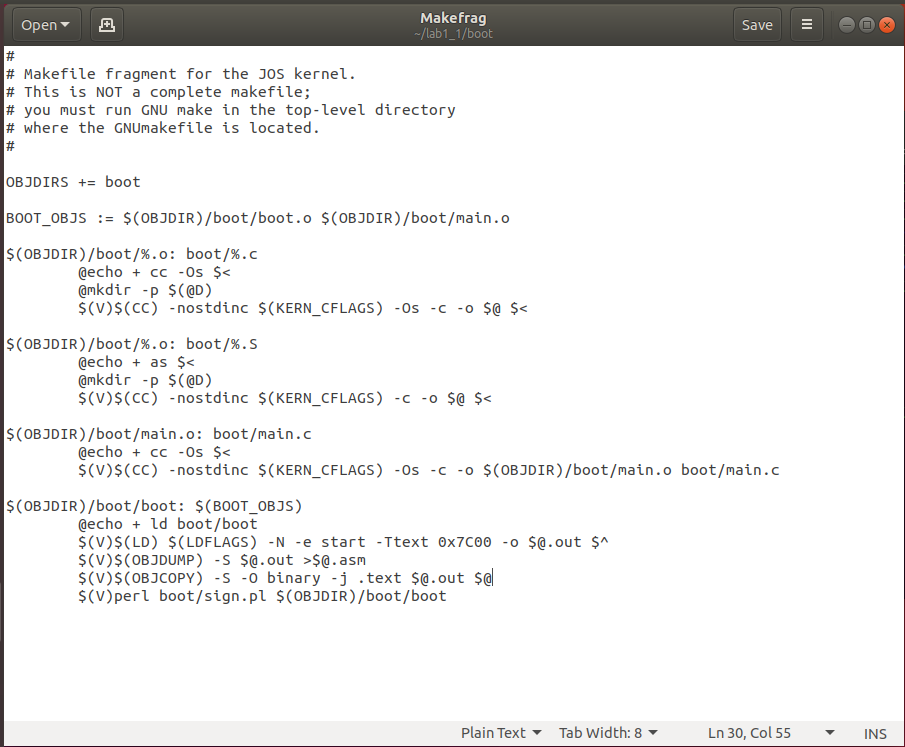
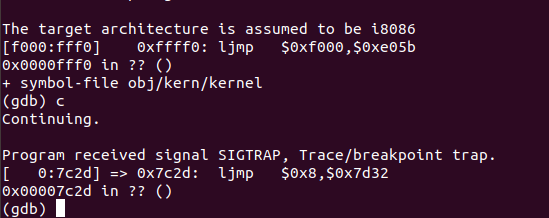
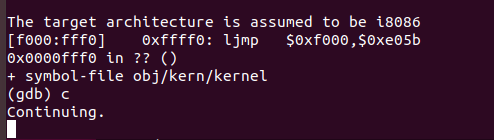


指针的类型标志着这个指针指向数据的类型，有两个作用：

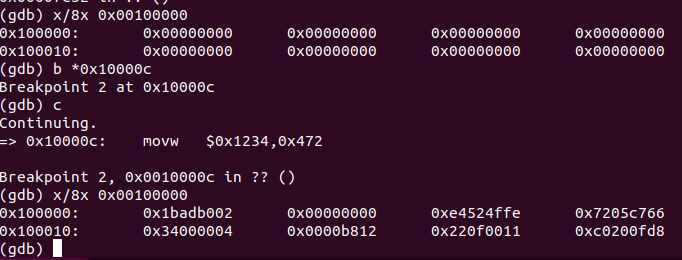
告诉了编译器需要从这个地址开始对多少字节（n）的数据进行操作, 以及操作模式

告诉编译器当对这个指针进行增减操作时，每加(减)一对应实际地址内存移动的字节数（n）

**练习5. 跟踪bootloader程序的前几个指令，找到开始使用链接地址的第一条指令，即，如果你使用了错误的链接地址，那么执行到这里的时候就必须要停下来，否则就会发生错误。然后将boot/Makefrag中的链接地址更改为一个错误的地址，运行make clean，用make命令重新编译实验，然后再次跟踪到引导加载程序，看看会发生什么。不要忘了改变链接地址后要再次执行make clean！**

练习6. 我们可以使用GDB的x命令检查内存。 GDB manual 中详细讲解了命令的用法，但就目前而言，知道命令x/Nx ADDR能够在ADDR处打印N个存储器字就足够了。（请注意，命令中的“ x ”均为小写。） 警告：字的大小不是通用标准。在GNU程序集中，一个字是两个字节（xorw中的'w'，代表字，表示2个字节）。复位机器（退出QEMU / GDB并再次启动）。在BIOS进入引导加载程序的那一刻停下来，检查内存中0x00100000地址开始的8个字的内容，然后再次运行，到bootloader进入内核的那一点再停下来，再次打印内存0x00100000的内容。为什么这8个字的内容会有所不同？第二次停下来的时候，打印出来的内容是什么？（你不需要使用QEMU来回答这个问题，思考即可）

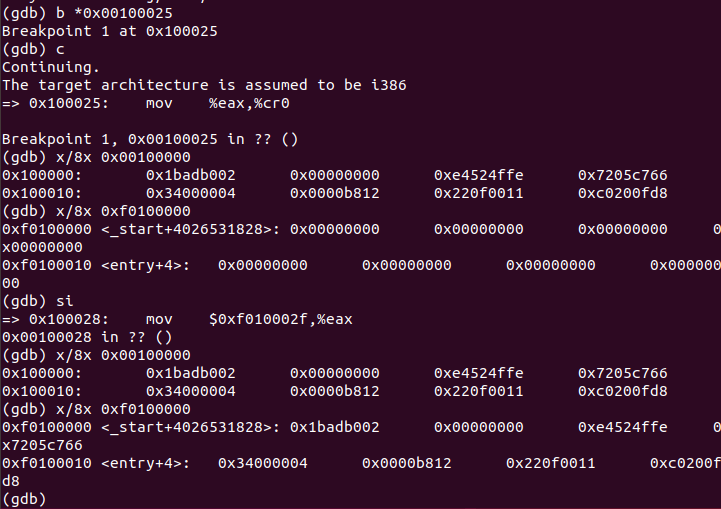


在 BIOS 进入引导加载程序的时候并没有

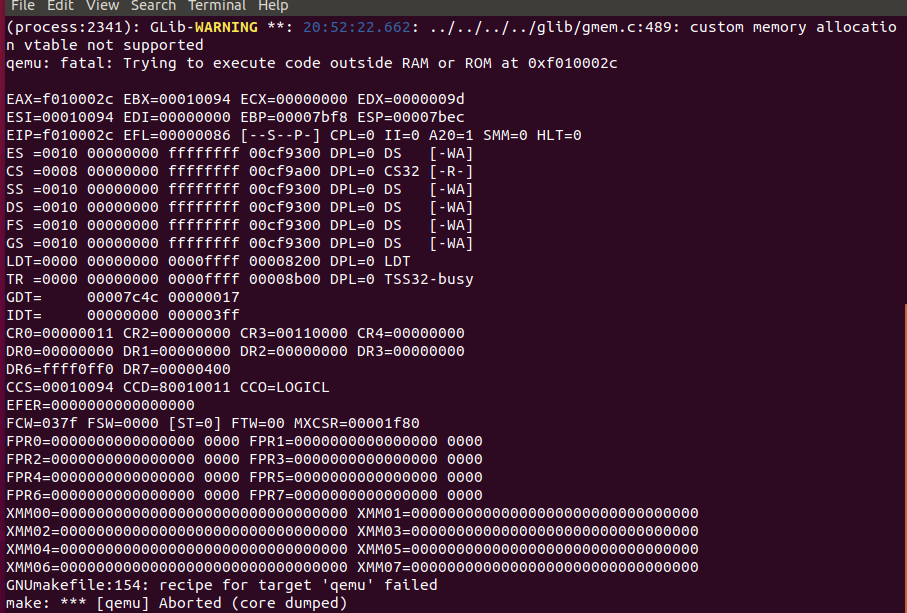
将内核写入内存，而在 bootloader 进入内核后，也就是第二次打印时候，内存

已经写入了内容。

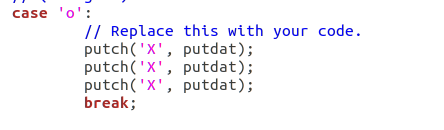
练习7.使用QEMU和GDB跟踪到JOS内核并停止在movl％eax，％cr0。 查看内存中在地址0x00100000和0xf0100000处的内容。 下面，使用GDB命令stepi单步执行该指令。 指令执行后，再次检查0x00100000和0xf0100000的内存。 确保你明白刚刚发生的事情。新映射建立后的第一条指令是什么，如果映射配置错误，它还能不能正常工作？ 注释掉kern/entry.S中的movl％eax，％cr0，再次追踪到它，看看你的理解是否正确。

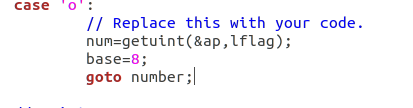


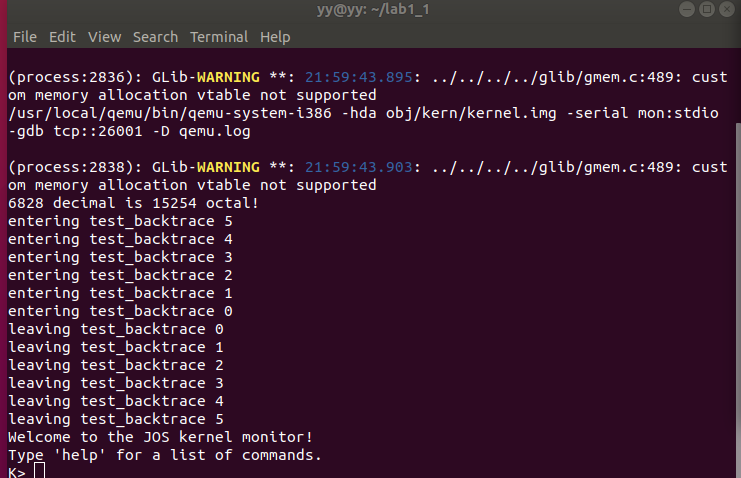
注释掉之后报错



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。







1.解释printf.c和console.c之间的接口。具体来说，console.c导出了什么函数？ printf.c如何使用这些函数？

console.c 导出 cputchar()函数,printf.c 中的函数 putch 调用了

cputchar()函数，将一个字符输出到显示屏。

2.从console.c解释以下内容：

1      if (crt\_pos >= CRT\_SIZE) {

2              int i;

3              memmove(crt\_buf, crt\_buf + CRT\_COLS, (CRT\_SIZE - CRT\_COLS) \* sizeof(uint16\_t));

4                      for (i = CRT\_SIZE - CRT\_COLS; i < CRT\_SIZE; i++)

5                     crt\_buf[i] = 0x0700 | ' ';

6             crt\_pos -= CRT\_COLS;

7     }

crt\_pos 当前字符将要输入的位置

CRT\_SIZE 能显示的最大字符数

该函数作用是检测输出位是否超过满屏，如果超过，则滚屏，将最后一行空出来全部置为空格，将光标置到最后一行行首

3.对于以下问题，你可能需要参考一些课外资料。

跟踪以下代码的并单步执行：

int x = 1, y = 3, z = 4;

cprintf("x %d, y %x, z %d\n", x, y, z);

在调用cprintf（）时，fmt是什么意思？ ap是什么意思？

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。

根据上一练习的实验，我们可以得知。fmt 指向的是我们编写的格式化

输出程序中所指定格式的字符串，也就是例如 printf 函数中双引号内的内容。

ap 指向的是对应 fmt 中的众多格式的多个输入参数。如例子中的 fmt 就是指向“x %d, y %x, z %d\n”的指针；而 ap 是指向第二个参数的地址。将代码写入

monitor.c 中进行编译，并且在 gdb 中下断点。可以发现，cons\_putc 的参数，

是 fmt 的 ascll 码形式，也就是：120，32，49，44，32，121，32，51，44，32，

122，32，52，10。对应的就是“x 1， y 3， z 4\n”。va\_arg 函数每次调用

ap 指向的分别是存放参数 x=1,y=3,z=4 的地址，并且每次调用，会使得 ap 的位置指向下一个变量位置。

4.运行下面的代码

unsigned int i = 0x00646c72;

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

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。

输出是 He110 World，因为 57616 的十进制是 e110，而且 Wo%s, &i 的

意义是把 i 作为字符串输出。其中，0x00 对应 '\0'，0x64 对应 'd'，0x6c

对应 'l'，0x72 对应 'r'

1. 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);

y的值没有给定，所以输出一个不确定的值

1. 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？

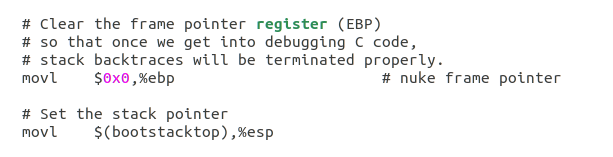
从上面可以看到, va arg 每次是以地址往后增长取出下一参数变量的地址的。而这个实现方式就默认假 设了编译器是以从右往左的顺序将参数入栈的. 因为栈是以从高往低的方向增长的。后压栈的参数放在了内存地址的低位置,所以如果要以从左到右的顺序依次取出每个变量,那么 编译器必须以相反的顺序即从右往左将参数压栈。 如果编译器更改了压栈的顺序,那么为了仍然能正确取出所有的参数, 那么需要修改上面代码中的 va\_start 和 va\_arg 两个宏,将其改成用减法 得到新地址即可

**栈**

**在本实验的最后一个练习中，我们将更详细地探讨C语言在x86上使用栈的方式，并编写一个新的内核监视器函数，该函数能够实现栈的回溯打印：一个保存引导到当前执行点的嵌套调用指令的指令指针（IP）值的列表。**

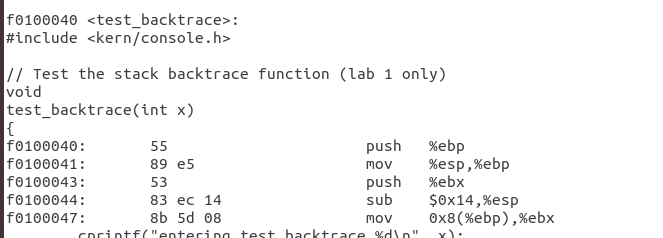
**练习9.确定内核在哪里完成了栈的初始化，以及栈所在内存的确切位置。内核如何为栈保留空间？栈指针初始化时指向的是保留区域的“哪一端”。**

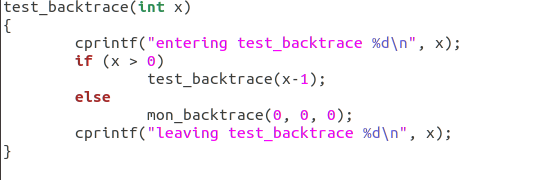
在entry.S文件可以找到这段代码，用以初始化堆栈的位置。这段完成了boot stack（0xf0108000~0xf0110000），预留大小是KSTKSIZE(32KB)栈顶是$(bootstacktop)，存在esp中。esp初始指向高位端 （由于堆栈是向下增长的，所以在运行init函数之前，esp寄存器的值就是0xf0110000，代表堆栈尚未使用）



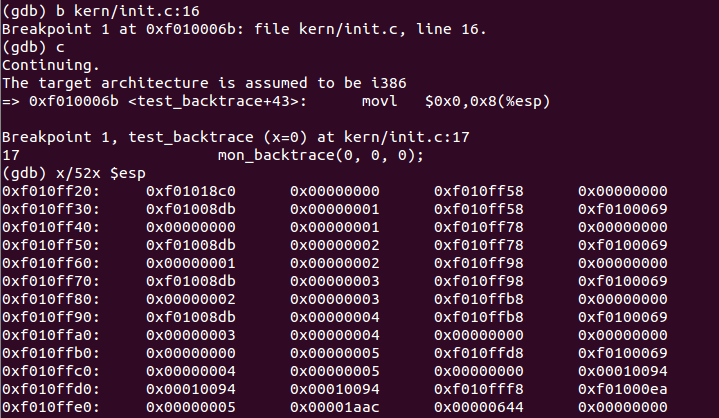
练习10.要熟悉x86上C语言函数的调用约定，请在obj/kern/kernel.asm中找到test\_backtrace函数的地址，在其中设置一个断点，并检查在内核启动后每次这个函数被调用时会发生什么。 每一级的test\_backtrace在递归调用时，会在栈上压入多少个32位的字，这些字的内容是什么？请注意，为了使此练习正常工作，你应该使用工具页中修改过QEMU。 否则，你必须手动将所有断点和内存地址转换为线性地址。

找到test\_backtrace 函数的地址

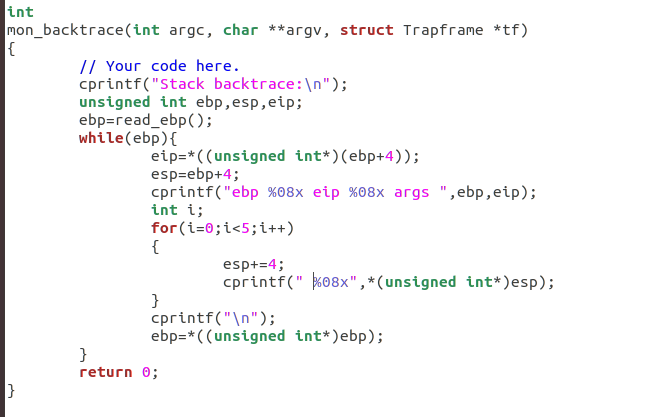




利用 GDB 加上断点，查看该函数。可见这个子程序是一个循环调用，在每一层循环中先打印信息 "entering test\_backtrace x"，在栈上压入 1 个 32 位的字。然后对 test\_backtrace 进行循环调用。当被循环调用完成，再打印信息 "leaving test\_backtrace x"。所以最先进入的程序会最后退出。因为对函数调用次数是五，所以会循环调用 5 次。



练习11.实现如上所述的回溯功能。 请使用与示例中相同的格式，否则打分脚本将会出错。当你认为你的工作正确的时候，运行make grade来看看它的输出是否符合我们的打分脚本所期待的，如果没有，修正发现的错误。在你成功提交实验1的作业后，欢迎你以任何你喜欢的方式更改回溯功能的输出格式。如果你使用read\_ebp（），请注意，GCC可能会生成优化后的代码，导致在mon\_backtrace（）的函数前导代码之前调用read\_ebp（），从而导致堆栈跟踪不完整（最近的函数调用的堆栈帧丢失）。我们可以尝试禁用优化以避免此重新排序的优化模式，你可能需要检查mon\_backtrace（）的汇编代码，并确保在函数前导代码之后调用的read\_ebp（）。



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。

int debuginfo\_eip(uintptr\_t addr, struct Eipdebuginfo \*info)

该函数输入eip，和一个Eipdebuginfo结构指针，执行完毕后，会将eip对应的信息填充到该结构中。接着完善mon\_backtrace函数：

