**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](https://web.archive.org/web/20040404164813/members.iweb.net.au/~pstorr/pcbook/book2/book2.htm) ，以及[6.828 reference materials page](http://oslab.mobisys.cc/pdos.csail.mit.edu/6.828/2014/reference.html) 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。

Answering: 先加载引导程序，然后从磁盘读取操作系统内核，进入系统。先运行boot.S->main.c->entry.S->init.c

**练习 3.**

* 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？

Answering:

（1）ljmp $PROT\_MODE\_CSEG, $protcseg 这一步是切换到保护模式。

（2）outb %al,$0x60

这条指令把CPU的60端口使能了。在这之前还要先向64端口输入一个值，代表下一个输入是控制命令。

Inb 和outb这样的指令是用来读取和控制CPU的引脚的命令，类似于之前接触过的Arduino等单片机上的控制引脚的函数。

* 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？

Answering:

Call \*0x10018; call entry.S

movw 0x1234, 0x472- to warm up

* 内核的第一个指令在哪里？

Answering:

0x10000c

* 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？

0x464c457f是0x7f，'E'，'L'，'F'(大端储存！！)

分析main.c可以得出总扇区数为8+(eph-ph)\* (ph->p\_memsz/512)，对这个式子的解释：

首先8是一开始读取的ELF所在的8个扇区，分析readseg可知，其第二个参数count代表调用一次该函数读取的字节数，在44行调用的readseg((uint32\_t) ELFHDR, SECTSIZE\*8, 0)，表示一开始读取的8个扇区，而在52-56行：

for (; ph < eph; ph++)

// p\_pa is the load address of this segment (as well

// as the physical address)

readseg(ph->p\_pa, ph->p\_memsz, ph->p\_offset);

程序完成了所有的读取操作，在这里readseg的count参数等于ph->p\_memsz，调用了eph-ph次readseg，容易得出，总扇区数为：

8+(eph-ph)\* (ph->p\_memsz/512)

困扰了很久的问题是，如何才能在反汇编代码中找到eph、ph和p\_memsz的值，关注以下几句：

//eph = ph + ELFHDR->e\_phnum;

7d3f: 0f b7 35 2c 00 01 00 movzwl 0x1002c,%esi

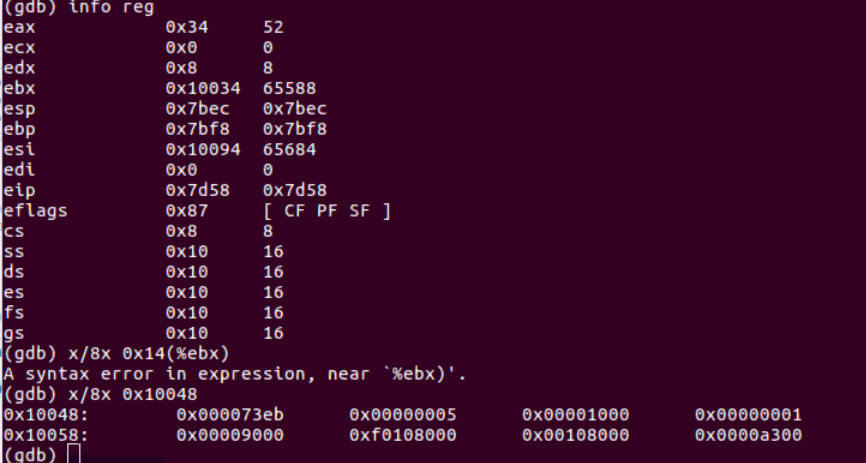
以上第一句表示%esi中存储了e\_phnum的值，而这个值就等于eph-ph，是关键的判断for循环走了几次的值，一开始我以为它是0x1002c也就是65580。。。后来发现这应该表示这个数据所在的地址。。也就是3

7d58: ff 73 14 pushl 0x14(%ebx)

这一句是找到p\_memsz值的位置，断点到这个位置，找到其值为0x73eb，也就是29675

因此磁盘需要读取的扇区数为：

8+(eph-ph)\* (ph->p\_memsz/512)=8+3\*58=182扇



**练习 4.**

下载 [pointers.c](http://video.mobisys.cc/materials/tools/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。

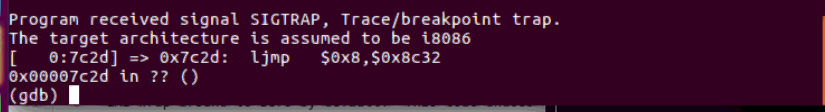
Int型数字在内存中的存储方式竟然是反的！！一开始非常疑惑为什么那个int数字的第一个字节打印出来是0。后来恍然大悟原来四个字节顺序是反的，查阅资料才知道原来还有大端和小端之分。

Answering:pointers.c里面的关键点其实就两个，一是int型指针+4，而char型+1，因此那个指针实际上指向了数组的第五个字节，读取的int数是第五到九字节。第二个点是，int型的存储方式是反的。

**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！

Answering:

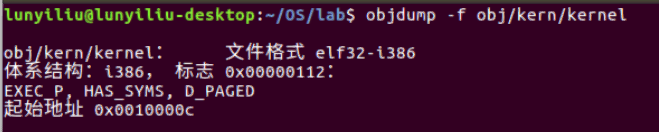


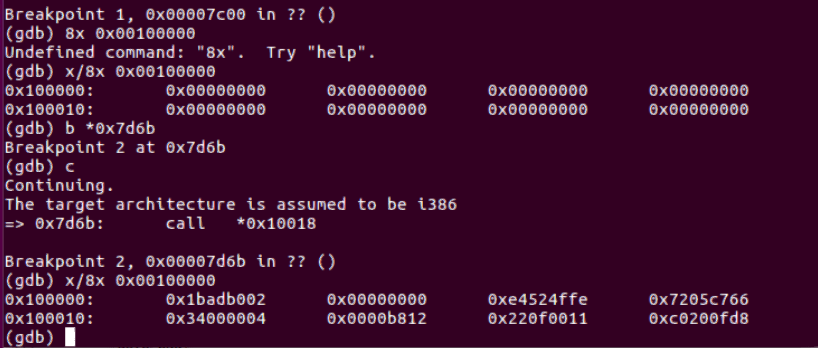
**练习 6.**

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）

Answering:

main.c把内核读取到了内存的0x10000c这个位置，这是由e\_entry决定的，因此在运行了main.c后，该处必然有值。这里面的值就是内核的执行段的数据。





**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。

Answering:

0x00100000 和 0xf0100000 里的值一样了，00100000被映射了过去。

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，看看你的猜测是否正确。

Answering:

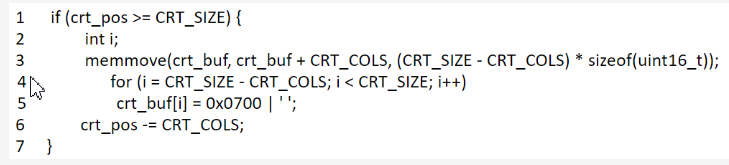
**0x100028: mov $0xf010002f,%eax 超限报错**

**练习8:**

* 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 是如何使用这些函数的？

Answering: printf.c使用了从console.c导出的 cputchar()，用于向屏幕上显示一个字符。 Printf把这个函数传入了vprintfmt().

* 解释 console.c 的以下内容：



Answering:如果当前字符输出位置超过了屏幕的范围，则执行以下操作：

1. 将缓冲区的内容整体向上移动一行（设有row行，先将2至row行的数据移动到1至row-1行，再将第row行清空）
2. 将字符输出位置定位到屏幕末行的起始位置

* 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？

Answering:fmt 是 格式字符串， ap是变量列表。

* + 按执行顺序列出每次调用 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。

Answering: vcprintf("x %d, y %x, z %d\n", 1, 3, 4)

|

cons\_putc(“x”)

|

va\_arg ()(调用之前指向1，调用之后指向3)

|

cons\_putc(1)

|

cons\_putc(“,”)

|

cons\_putc(“y”)

|

va\_arg ()(调用之前指向3，调用之后指向4)

|

cons\_putc(3)

|

cons\_putc(“,”)

|

cons\_putc(“z”)

|

va\_arg ()(调用之前指向3，调用之后指向3所在地址空间+4的位置)

|

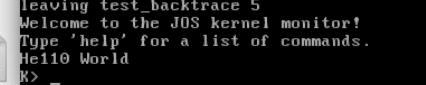
cons\_putc(4)

* 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。可以参考将字节映射到字符的 [ASCII表](http://web.cs.mun.ca/~michael/c/ascii-table.html)。

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？[这里](http://www.webopedia.com/TERM/b/big_endian.html)是一个关于大小端的描述，[这里](http://www.networksorcery.com/enp/ien/ien137.txt)是一个更加脑洞大开的说法。

Answering: （1）输出结果：



（2）原因：

对于57616，因为在格式字符串中%x代表16进制输出，因此应输出e110；

对于%s处的输出，与前面pointer.c中的情形类似，由于i是一个4字节的int型数，在内存中应当拆成四部分存放，每部分恰好是一个字节，即2个16进制表示中的位。由于x86的小端原则，实际上数是按先存低位，再存高位的顺序存放的，则在&i指向的内容中，前四个字节依次是0x72、0x6c、0x64、0x00。由于格式化字符串是按%s输出的，因此变量指针会一次前进一个字节，并把读到的内容转为char。因此将会依次打印出“r”,“l”，“d”.

如果 x86 是大端,将i改为0x726c6400

* 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);

Answering:



Var\_arg中的指针将在完成输出3之后，指向原地址+4的位置，因为并不知道这个位置上的值是多少，所以会输出不确定的值。

* 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？

Answering:

每次调用var\_start()时，ap会指向&fmt+1的位置，即第一个可变参数所在的位置。每次调用var\_arg(),ap指针地址+1。

而现在参数入栈的顺序对调了。因此导致fmt的地址在所有参数中是最高的。有两种解决方法：

1. 重写var\_start()和var\_arg()，使ap每次地址-1
2. 不重写。只是在每次调用var\_start()和var\_arg()后把ap的地址值减去相应值。

**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？

Answering:

在entry.S里，第74行到77行。 栈位于0-1MB的位置。内核让EBP指向0x00，让栈指针初始化于1MB地址处，执行保留区域的末端。

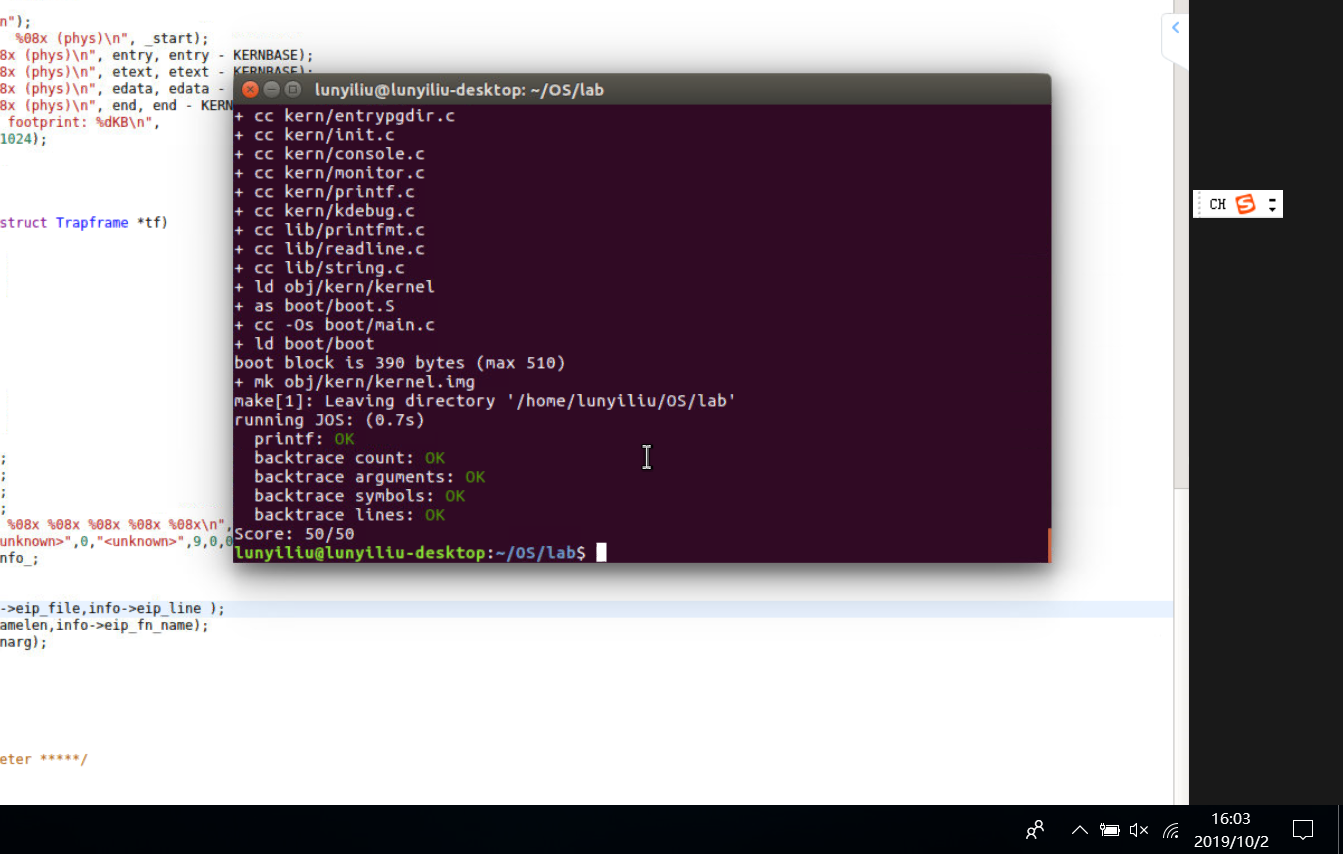
**练习 10.**

熟悉 x86 上 C 语言函数的调用约定，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？

Answering:

每一级test\_backtrace在递归调用时，分别依次压入%ebp、%eip、%eax。

%ebp是栈基指针、%eip是返回指令指针、 %eax是x



最后打分脚本期望的输出格式有一个空格没注意。网页上的格式是“文件名：行数： 函数+偏移”。在行数和函数之间有个空格。之前忘了加不知道为什么一会报错，后来查看python脚本的正则匹配才知道，在行数和函数之间有个“ +”的pattern，意味着要有一个空格。