# LAB 1

1. qemu安装，all warnings being treated as errors

## Part 1: PC Bootstrap

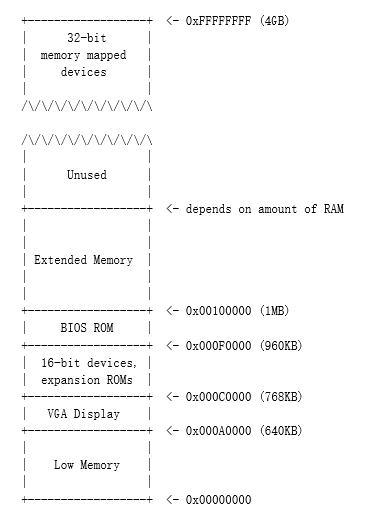
### Getting Started with x86 assembly

#### Exercise 1

熟悉x86汇编语言

### Simulating the x86

### The PC's Physical Address Space



这张图仅仅展示了内存空间的一部分。

　　第一代PC处理器是16位字长的Intel 8088处理器，这类处理器只能访问1MB的地址空间，即0x00000000~0x000FFFFF。但是这1MB也不是用户都能利用到的，只有低640KB(0x00000000~0x000A0000)的地址空间是用户程序可以使用的。如图所示。

　　而剩下的384KB的高地址空间则被保留用作其他的目的，比如(0x000A0000~0x000C0000)被用作屏幕显示内容缓冲区，其他的则被非易失性存储器(ROM)所使用，里面会存放一些固件，其中最重要的一部分就是BIOS，占据了0x000F0000~0x00100000的地址空间。BIOS负责进行一些基本的系统初始化任务，比如开启显卡，检测该系统的内存大小等等工作。在初始化完成后，BIOS就会从某个合适的地方加载操作系统。

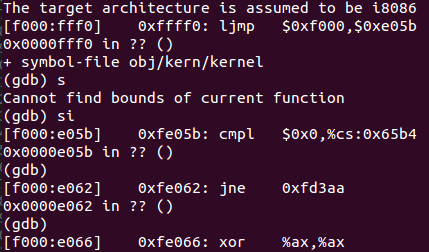
　　虽然Intel处理器突破了1MB内存空间，在80286和80386上已经实现了16MB，4GB的地址空间，但是PC的架构必须仍旧把原来的1MB的地址空间的结构保留下来，这样才能实现向后兼容性。所以现代计算机的地址 0x000A0000~0x00100000区间是一个空洞，不会被使用。因此这个空洞就把地址空间划分成了两个部分，第一部分就是从0x00000000~0x000A0000，叫做传统内存。剩下的不包括空洞的其他部分叫做扩展内存。而对于这种32位字长处理器通常把BIOS存放到整个存储空间的顶端处。

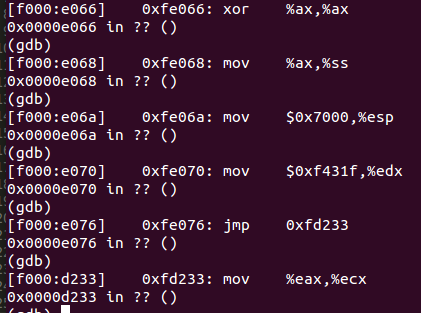
由于xv6操作系统设计的一些限制，它只利用256MB的物理地址空间，即它假设用户的主机只有256MB的内存。

The ROM BIOS

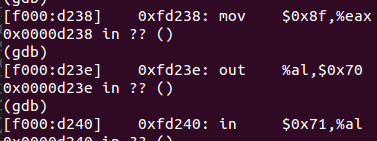
Exercise 2.

用si单步调试，揣摩每条指令的作用。





初始化寄存器的值



那么这三条命令就是要操作端口0x70，0x71，0x70端口和0x71端口是用于控制系统中一个叫做CMOS的设备，这个设备是一个低功耗的存储设备，它可以用于在计算机关闭时存储一些信息，它是由独立的电池供电的。

这个CMOS中可以控制跟PC相关的多个功能，其中最重要的就是时钟设备（Real Time Clock）的 ，它还可以控制是否响应不可屏蔽中断NMI(Non-Maskable Interrupt)。

操作CMOS存储器中的内容需要两个端口，一个是0x70另一个就是0x71。其中0x70可以叫做索引寄存器，这个8位寄存器的最高位是不可屏蔽中断(NMI)使能位。如果你把这个位置1，则NMI不会被响应。低7位用于指定CMOS存储器中的存储单元地址，所以如果你想访问第1号存储单元，并且在访问时，我要使能NMI，那么你就应该向端口0x70里面送入0b10000001 = 0x81。

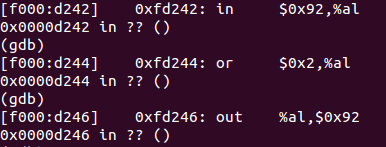
　 即mov $0x81, %al

out %al, 0x70

然后对于这个地址单元的操作，比如读或者写就可以由0x71端口完成，比如你现在想从1号存储单元里面读出它的值，在完成上面的两条指令后，就可以输入这条指令

in $0x71, %al

再回到我们的系统，这三条指令可以看出，它首先关闭了NMI中断，并且要访问存储单元0xF的值，并且把值读到al中，但是在后面我们发现这个值并没有被利用，所以可以认为这三条指令是用来关闭NMI中断的。

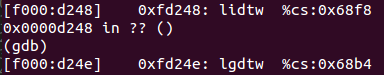


这三步操作又是在控制端口，此时被控制的端口号为0x92，通过上面那个链接 http://bochs.sourceforge.net/techspec/PORTS.LST

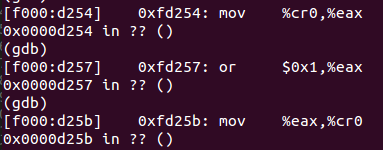
我们可以查看到，它控制的是 PS/2系统控制端口A，该操作明显是在把这个端口的1号bit置为1。这个端口的bit1的功能是

bit 1= 1 indicates A20 active

即A20位，即第21个地址线被使能，了解实模式和保护模式的同学肯定清楚，如果A20地址线被激活，那么系统工作在保护模式下。但是在之后的boot loader程序中，计算机首先要工作在实模式下啊。所以这里的这个操作，根据网上 http://kernelx.weebly.com/a20-address-line.html 所说应该是去测试可用内存空间。在boot loader之前，它肯定还会转换回实模式。



初始化idtr和gdtr



计算机中包含CR0~CR3四个控制寄存器，用来控制和确定处理器的操作模式。其中这三个语句的操作明显是要把CR0寄存器的最低位(0bit)置1。CR0寄存器的0bit是PE位，启动保护位，当该位被置1，代表开启了保护模式。

综上，我们可以看到BIOS的操作就是在控制，初始化，检测各种底层的设备，比如时钟，GDTR寄存器。以及设置中断向量表。这都和Lab 1 Part 1.2最后两段说的一样。但是作为PC启动后运行的第一段程序，它最重要的功能是把操作系统从磁盘中导入内存，然后再把控制权转交给操作系统。所 以BIOS在运行的最后会去检测可以从当前系统的哪个设备中找到操作系统，通常来说是我们的磁盘。也有可能是U盘等等。当BIOS确定了，操作系统位于磁 盘中，那么它就会把这个磁盘的第一个扇区，通常把它叫做启动区（boot sector）先加载到内存中，这个启动区中包括一个非常重要的程序--boot loader，它会负责完成整个操作系统从磁盘导入内存的工作，以及一些其他的非常重要的配置工作。最后操作系统才会开始运行。

可见PC启动后的运行顺序为 BIOS --> boot loader --> 操作系统内核

## Part 2: The Boot Loader

1. The boot loader consists of one assembly language source file, boot/boot.S, and one C source file, boot/main.c

#### Exercise 3

设置一个断点在地址0x7c00处，这是boot sector被加载的位置。然后让程序继续运行直到这个断点。跟踪/boot/boot.S文件的每一条指令，同时使用boot.S文件和系统为你反汇编出来的文件obj/boot/boot.asm。你也可以使用GDB的x/i指令来获取去任意一个机器指令的反汇编指令，把源文件boot.S文件和boot.asm文件以及在GDB反汇编出来的指令进行比较。

追踪到bootmain函数中，而且还要具体追踪到readsect()子函数里面。找出和readsect()c语言程序的每一条语句所对应的汇编指令，回到bootmain()，然后找出把内核文件从磁盘读取到内存的那个for循环所对应的汇编语句。找出当循环结束后会执行哪条语句，在那里设置断点，继续运行到断点，然后运行完所有的剩下的语句。

答：

1. 在什么时候处理器开始运行于32bit模式？到底是什么把CPU从16位切换为32位工作模式？

答：在boot.S文件中，计算机首先工作于实模式，此时是16bit工作模式。当运行完 " ljmp $PROT\_MODE\_CSEG, $protcseg " 语句后，正式进入32位工作模式。根本原因是此时CPU工作在保护模式下。

2. boot loader中执行的最后一条语句是什么？内核被加载到内存后执行的第一条语句又是什么？

答：boot loader执行的最后一条语句是bootmain子程序中的最后一条语句 " ((void (\*)(void)) (ELFHDR->e\_entry))(); "，即跳转到操作系统内核程序的起始指令处。

这个第一条指令位于/kern/entry.S文件中，第一句 movw $0x1234, 0x472

3. 内核的第一条指令在哪里？

答：上一个问题中已经回答过这个问题，第一条指令位于/kern/entry.S文件中。

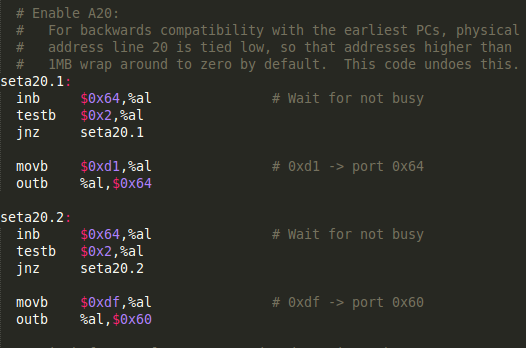
4. boot loader是如何知道它要读取多少个扇区才能把整个内核都送入内存的呢？在哪里找到这些信息？

答：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。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。所以如果我们能够找到这个表，那么就能够通过表项所提供的信息来确定内核占用多少个扇区。

那么关于这个表存放在哪里的信息，则是存放在操作系统内核映像文件的ELF头部信息中。

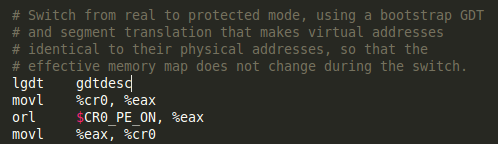
详解：

bios之后 控制权在boot.S



0x64端口属于键盘控制器804x，名称是控制器读取状态寄存器。

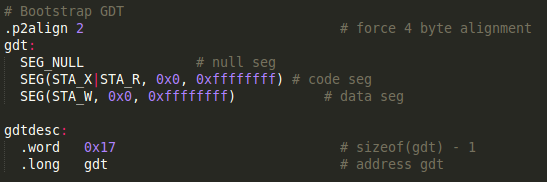
0xdf 使能A20，准备进入保护模式



指令 lgdt gdtdesc，是把gdtdesc这个标识符的值送入全局映射描述符表寄存器GDTR中。

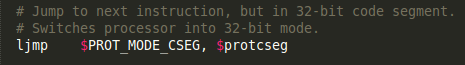
上面的操作是把CR0寄存器的bit0置1，CR0寄存器的bit0是保护模式启动位，把这一位值1代表保护模式启动。

这个GDT表是处理器工作于保护模式下一个非常重要的表。至于这条指令的功能就是把关于GDT表的一些重要信息存放到CPU的GDTR寄存器中，其中包括GDT表的内存起始地址，以及GDT表的长度。这个寄存器由48位组成，其中低16位表示该表长度，高32位表该表在内存中的起始地址。所以gdtdesc是一个标识符，标识着一个内存地址。从这个内存地址开始之后的6个字节中存放着GDT表的长度和起始地址。我们可以在这个文件的末尾看到gdtdesc，如下：

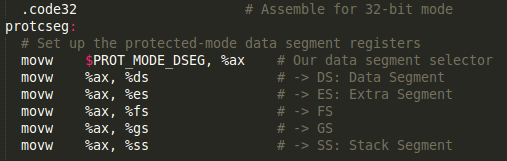


Gdt：后面是gdt表

调用SEG()子程序来构造GDT表项的。这个子函数定义在mmu.h中，函数需要3个参数，一是type即这个段的访问权限，二是base，这个段的起始地址，三是lim，即这个段的大小界限



进入32位地址模式



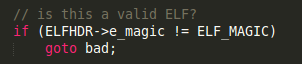
初始化保护模式下寄存器的值



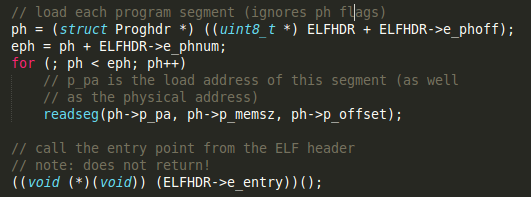
Main.c:



所以这条指令是把内核的第一个页(4MB = 4096 = SECTSIZE\*8 = 512\*8)的内容读取的内存地址ELFHDR(0x10000)处。其实完成这些后相当于把操作系统映像文件的elf头部读取出来放入内存中。



elf头部信息的magic字段是整个头部信息的开端。并且如果这个文件是格式是ELF格式的话，文件的elf->magic域应该是=ELF\_MAGIC的，所以这条语句就是判断这个输入文件是否是合法的elf可执行文件。



首先elf是表头起址，而phoff字段代表Program Header Table距离表头的偏移量。所以ph可以被指定为Program Header Table表头。eph指向该表末尾。

这个for循环就是在把操作系统内核的各个段从外存读入内存中。

e\_entry字段指向的是这个文件的执行入口地址。所以这里相当于开始运行这个文件。也就是内核文件。 自此就把控制权从boot loader转交给了操作系统的内核。

### Loading the Kernel

#### Exercise 4

熟悉C语言编程，尤其是有关指针。

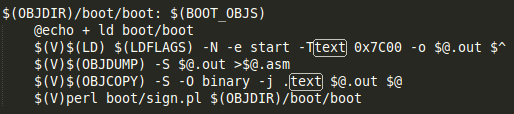
#### Exercise 5

再一次追踪一下boot loader的一开始的几句指令，找到第一条满足如下条件的指令处：

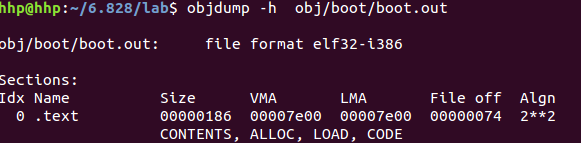
当我修改了boot loader的链接地址，这个指令就会出现错误。

找到这样的指令后，把boot loader的链接地址修改一下，我们要在boot/Makefrag文件中修改它的链接地址，修改完成后运行make clean， 然后通过make指令重新编译内核，再找到那条指令看看会发生什么。 最后别忘了改回来。

答：



打开这个boot/Makefrag文件，其中的-Ttext 0x7C00，就是指定链接地址，我们可以把它修改为0x7E00，然后保存退出。







把指令后面的值所指定内存地址处后6个字节的值输入全局描述符表寄存器GDTR，但是当前这条指令读取的内存地址是0x7e64，我们在图中也展示了一下这个地址处后面6个单元存放的值，发现是全部是0。这肯定是不对的，正确的应该是在0x7c64处存放的值，即图中最下面一样的值。可见，问题出在这里，GDTR表的值读取不正确，这是实现从实模式到保护模式转换的非常重要的一步。

#### Exercise 6.

用x/Nx ADDR打印内存信息。

## Part 3: The Kernel

### Using virtual memory to work around position dependence

在这个实验中，首先是采用分页管理的方法来实现上面所讲述的地址映射。但是设计者实现映射的方式并不是通常计算机所采用的分页管理机构，而是自己手写了一个程序lab\kern\entrygdir.c用于进行映射。既然是手写的，所以它的功能就很有限了，只能够把虚拟地址空间的地址范围：0xf0000000~0xf0400000，映射到物理地址范围：0x00000000~0x00400000上面。也可以把虚拟地址范围：0x00000000~0x00400000，同样映射到物理地址范围：0x00000000~0x00400000上面。任何不再这两个虚拟地址范围内的地址都会引起一个硬件异常。虽然只能映射这两块很小的空间，但是已经足够刚启动程序的时候来使用了。

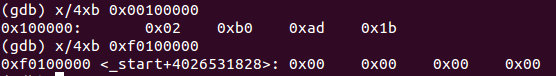
#### Exercise 7：

使用Qemu和GDB去追踪JOS内核文件，并且停止在movl %eax, %cr0指令前。此时看一下内存地址0x00100000以及0xf0100000处分别存放着什么。然后使用stepi命令执行完这条命令，再次检查这两个地址处的内容。确保你真的理解了发生了什么。

如果这条指令movl %eax, %cr0并没有执行，而是被跳过，那么第一个会出现问题的指令是什么？我们可以通过把entry.S的这条语句加上注释来验证一下。

答：

首先设置断点到0x10000C处，因为我们在之前的练习中已经知道了，0x10000C是内核文件的入口地址。 然后我们从这条指令开始一步步运行，直到碰到movl %eax, %cr0指令。在这条指令运行之前，地址0x00100000和地址0xf0100000两处存储的内容是：

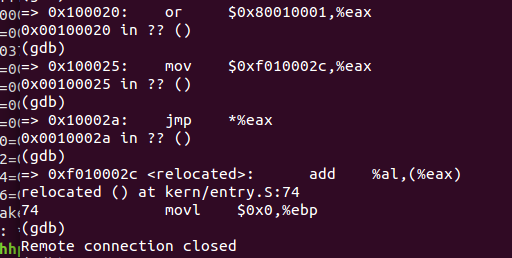


开始不一样

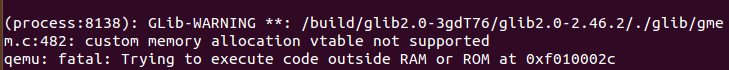


Si后 一样

第二问需要我们把entry.S文件中的%movl %eax, %cr0这句话注释掉，重新编译内核。我们需要先make clean，然后把%movl %eax, %cr0这句话注释掉，重新编译。 再次用qemu仿真，并且设置断点到0x10000C处，开始一步步执行。通过一步步查询发现了出现错误的一句。



其中在0x10002a处的jmp指令，要跳转的位置是0xf010002C，由于没有进行分页管理，此时不会进行虚拟地址到物理地址的转化。所以报出错误，下面是make qemu-gdb这个窗口中出现的信息。



已经超出内存

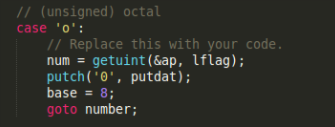
### Formatted Printing to the Console

#### Exercise 8

我们丢弃了一小部分代码---即当我们在printf中指定输出"%o"格式的字符串，即八进制格式的代码。尝试去完成这部分程序。

答：

补充vprintfmt函数中八进制输出功能。



在这个练习中我们首先要阅读以下三个源文件的代码，弄清楚他们三者之间的关系：

三个文件分别为 \kern\printf.c，\kern\console.c, \lib\printfmt.c

1.\kern\printf.c中的cprintf，vcprintf子程序调用了\lib\printfmt.c中的vprintfmt子程序。

2.\kern\printf.c中的putch子程序中调用了cputchar，这个程序是定义在\kern\console.c中的。

3.\lib\printfmt.c 中vprintfmt中的某些程序也依赖于cputchar子程序。

调用关系cprintf => vcprintf => vprintfmt

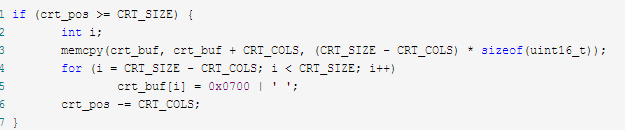
* putch => cputchar

Exercise 8 小问题

1. 解释一下printf.c和console.c两个之间的关系。console.c输出了哪些子函数？这些子函数是怎么被printf.c所利用的？

答：在Exercise 8的解答中我们已经很具体的分析了两个文件，在console.c中除了被static修饰符修饰的函数之外，都可以被外部所使用，其中被printf所使用的函数就是cputchar子函数。

2、解释一下console.c文件中，下面这段代码的含义：

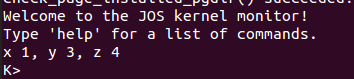


crt\_buf:这是一个字符数组缓冲区，里面存放着要显示到屏幕上的字符

crt\_pos:这个表示当前最后一个字符显示在屏幕上的位置

当crt\_pos >= CRT\_SIZE，其中CRT\_SIZE = 80\*25，由于我们知道crt\_pos取值范围是0~(80\*25-1)，那么这个条件如果成立则说明现在在屏幕上输出的内容已经超过了一页。所以此时要把页面向上滚动一行，即把原来的1~79号行放到现在的0~78行上，然后把79号行换成一行空格（当然并非完全都是空格，0号字符上要显示你输入的字符int c）。所以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。而紧接着的for循环则是把最后一行，79号行都变成空格。最后还要修改一下crt\_pos的值。

3、



4 . Run the following code. 12.



What is the output? Explain how this output is arrived at in the stepby-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



首先看下第一个%x，指的是要按照16进制输出第一个参数，第一个参数的值是57616，它对应的16进制的表示形式为e110，所以前面就变成的He110。

然后看下一个%s，输出参数所指向的字符串。参数是&i，是变量i的地址，所以应该输出的是变量i所在地址处的字符串。

而在cprintf之前我们把i定义为一个int类型变量，所以现在我们要把它们进行拆分，按照一个字节一个字节来进行输出。

由于x86是小端模式，代表字的最高位字节存放在最高位字节地址上。假设i变量的地址为0x00，那么i的4个字节的值存放在0x00，0x01，0x02，0x03四处。由于是小端存储，所以0x00处存放0x72('r')，0x01处存放0x6c('l')，0x02处存放0x64('d')，0x03处存放0x00('\0').

所以在cprintf将会从i的地址开始一个字节一个字节遍历，正好输出 "World"

5、看下面的代码，在'y='后面会输出什么？为什么会这样？

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

答：

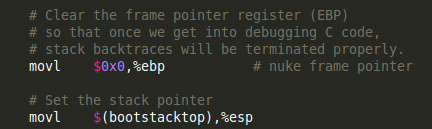
输出的结果如下，由于y并没有参数被指定，所以会输出一个不确定的值

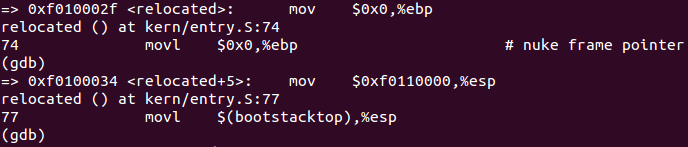


### The Stack

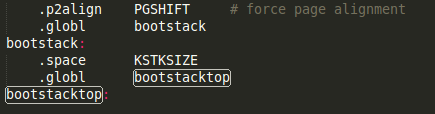
#### Exercise 9

1. 首先需要判断操作系统内核是从哪条指令开始初始化它的堆栈空间的。





在文件entry.S中，



这两个指令分别设置了%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。注意，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。

3. 内核是如何给它的堆栈保留一块内存空间的？

其实就是通过刚刚分析的，在entry.S中的数据段里面声明一块大小为32Kb的空间作为堆栈使用。从而为内核保留了一块空间。

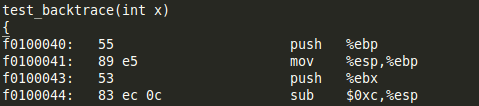
4. 堆栈指针又是指向这块被保留的区域的哪一端的呢？

堆栈由于是向下生长的，所以堆栈指针自然要指向最高地址了。最高地址就是我们之前看到的bootstacktop的值。所以将会把这个值赋给堆栈指针寄存器

#### Exercise 10

为了能够更好的了解在x86上的C程序调用过程的细节，我们首先找到在obj/kern/kern.asm中test\_backtrace子程序的地址， 设置断点，并且探讨一下在内核启动后，这个程序被调用时发生了什么。对于这个循环嵌套调用的程序test\_backtrace，它一共压入了多少信息到堆栈之中。并且它们都代表什么含义？

答：x=5





进入函数保存栈信息和压入参数之后栈帧情况

X=4 依次类推



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

#### Exercise 11

/kern/monitor.c文件中完成回溯函数mon\_backtrace()

#### Exercise 12

修改kern/kdebug.c文件中debuginfo\_eip()函数，以及上述mon\_backtrace()函数，实现输出对应eip的函数名，文件名以及行数信息。

？？？？？？？？？

Tip: printf format strings provide an easy, albeit obscure, way to print non-null-terminated strings like those in STABS tables. printf("%.\*s", length, string) prints at most length characters of string. Take a look at the printf man page to find out why this works.