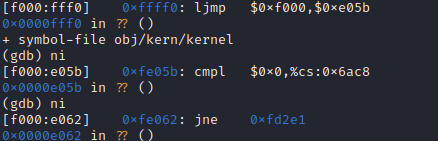
**LAB1**

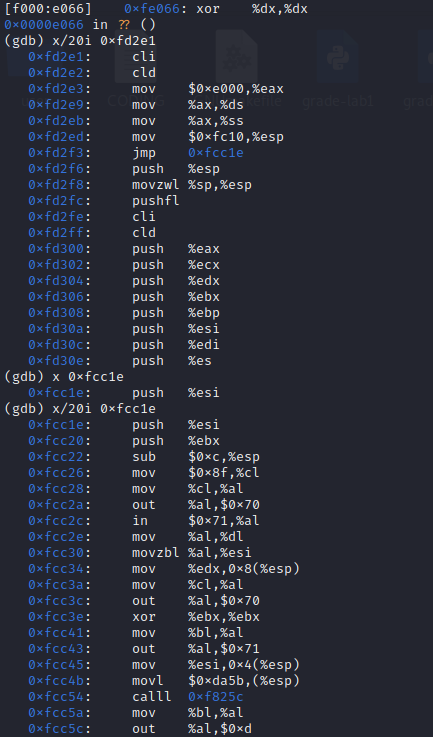
**Exercise2:**

不会的过程中参考的学习链接：https://www.cnblogs.com/fatsheep9146/p/5078179.html

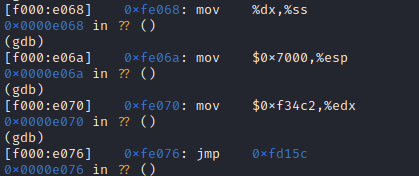
观察到前三条指令是这些



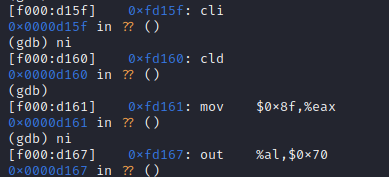
这个一开始的比较不知道干嘛用的。只知道如果是not-equal到达的指令是



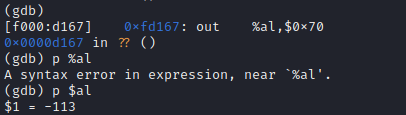
这里也没明白是什么意思。但是先不管分支跳转，跟着原程序走。



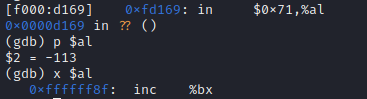
这几步初始化了一些内容。其中比较重要的是esp，接下来又是跳转，到如图位置。操作系统先关闭中断，之后cld决定操作系统内存增长方向。Out指令代表寄存器向端口输出数据。



这一句话向0x70端口写入%eax中的值，值为-113



接下来从71端口接受输入，用%al作为缓冲

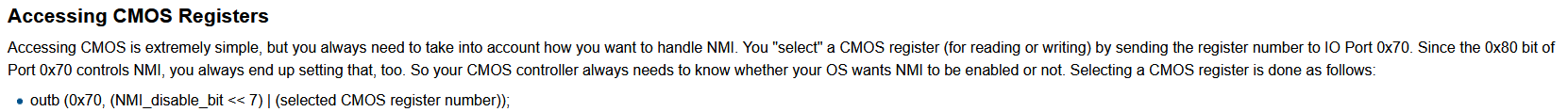


70和71端口是CMOS设备，查阅资料发现CMOS作用是储存启动信息，并且他作为一个单独供电设备。其中包含了时钟设备，以及可以控制non-maskable interrupts

The function of the CMOS memory is to store 50 (or 114) bytes of "Setup" information for the BIOS while the computer is turned off -- because there is a separate battery that keeps the Clock and the CMOS information active.——wiki

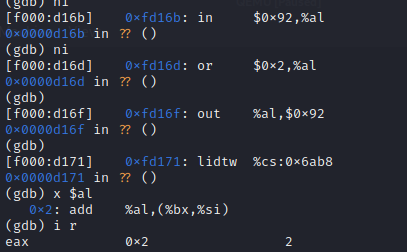
在%al寄存器中，最高位可以决定是否响应NMI，后面七位代表CMOS储存单元地址把这个位置置为1，则表示不响应。因此上图输入0X8F，就是读取10001111中地址为0001111的数据，并对于不可屏蔽中断也不响应。

CMOS链接：https://wiki.osdev.org/CMOS

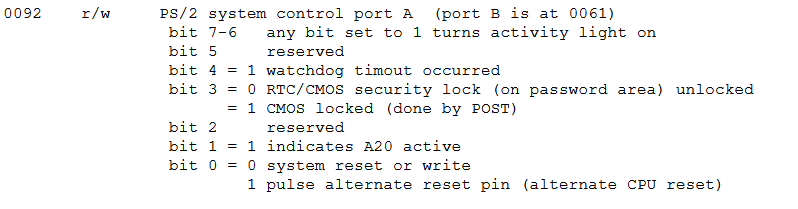


上述指令关闭NMI中断，并读取了CMOS中的一个数据。

接下来又在0x92端口上操作



查阅端口资料



这里看到bit1=1代表A20使能，也就是系统将工作在保护模式下。参考网站上的解释是测试内存总量（remains to be checked ）

接下来两条指令

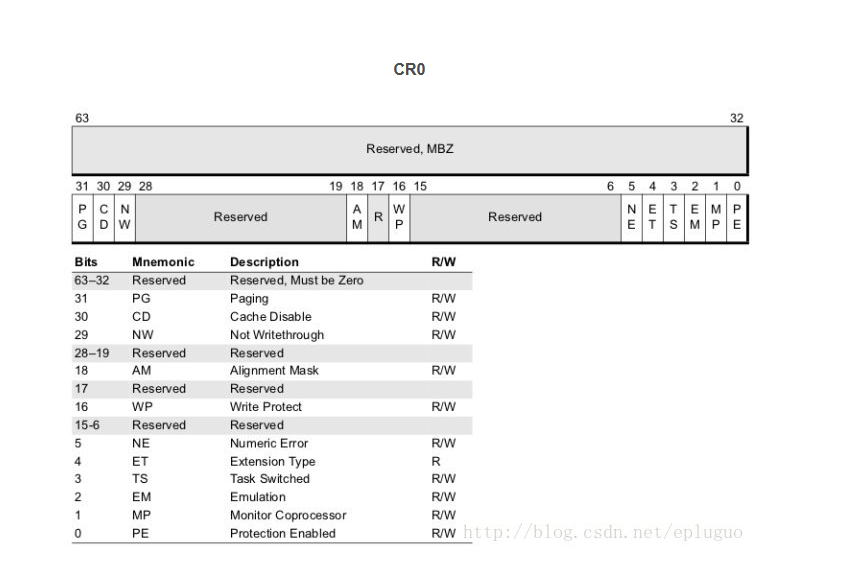




代表的是加载中断向量表和全局描述符表，从后面的地址中读入数据

接下来两步，设置CR0寄存器最低位为1，也就是启动保护模式

CRO寄存器含义如下图



接下来目标的处理器架构就突然变成了i386

**后面一些步骤无法理解**

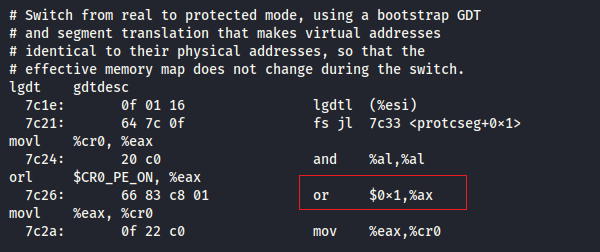
**Question2**

设置断点在\*0x7c00,continue执行，可以看到如下语句，判断得到是在boot.asm文件中

这个文件注释已经很详细了。接下来回答这几个问题

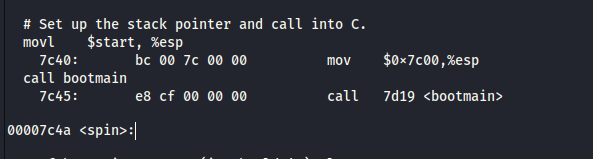
**At what point does the processor start executing 32-bit code? What exactly causes the switch from 16- to 32-bit mode?**

如下图



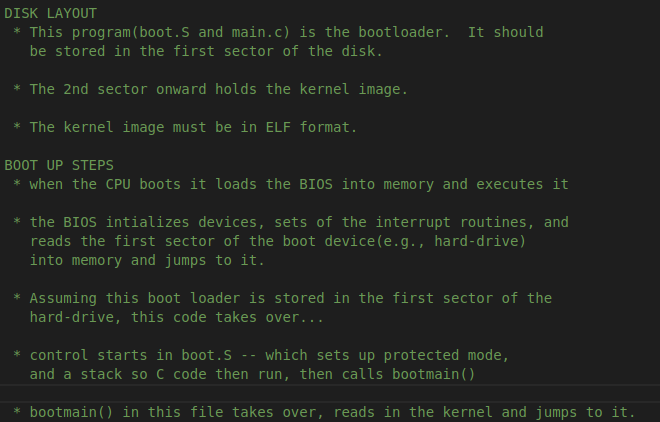
这里设置CR0位置是的操作系统转变为32为保护模式

**What is the *last* instruction of the boot loader executed, and what is the *first* instruction of the kernel it just loaded?**



Boot最后一句话是跳转到bootmain的c函数。这个函数在一个固定的物理空间的位置，并且跳转过后，就不会返回，跳转到的bootmain就是bootloder所在地方。

关于bootloder的位置，这里讲的很清楚



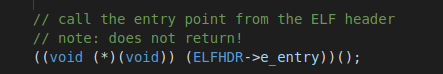
磁盘中第一个扇区是boot-loder（boot.s）,第二个是内核镜像，必须是ELF文件格式,接下来boot.s将跳转到bootmain()函数boot/main.c，这个文件读取kernel文件并且跳转到kernel部分。接下来看看这个main.c

Main.c调用了两个函数（其汇编内容在boot.asm中）

readseg(uint32\_t pa, uint32\_t count, uint32\_t offset)

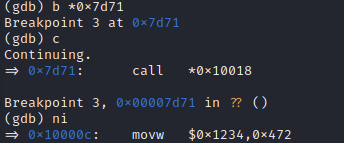
readsect(void \*dst, uint32\_t offset)

这三个函数起到的作用是读取扇区内容，大体来说:readseg控制读取扇区位置，readsect控制访问磁盘（等待空闲并读取数据）。Bootmain的最后一句指令是



这句话控制指令跳转到kernel部分

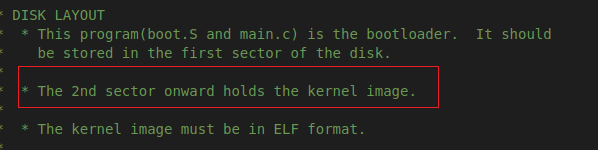
分析代码中的循环，找到跳转位置并下断点



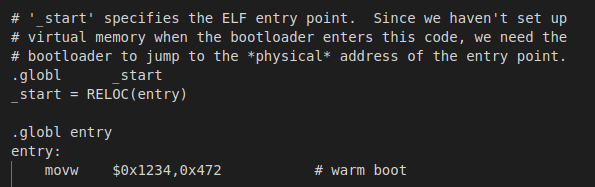
这就是kernel的第一条指令

e\_entry字段的含义是这个可执行文件的第一条指令的虚拟地址。

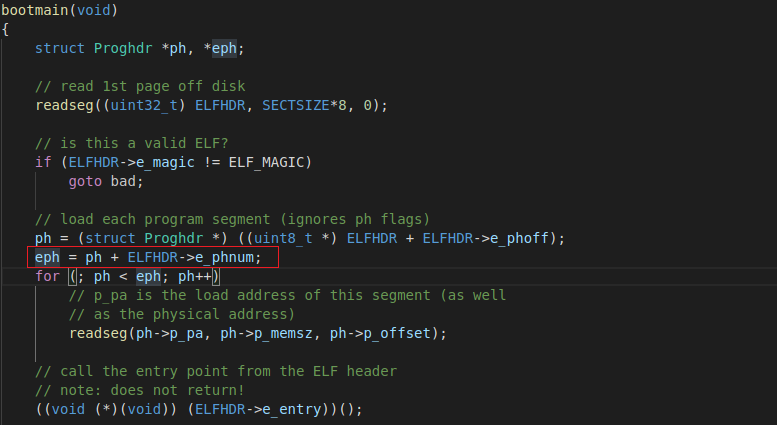
***Where* is the first instruction of the kernel?**



根据这里所说，第二个扇区位置就是kernel所在位置，以下是kernel第一句代码内容



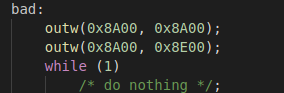
**How does the boot loader decide how many sectors it must read in order to fetch the entire kernel from disk? Where does it find this information?**

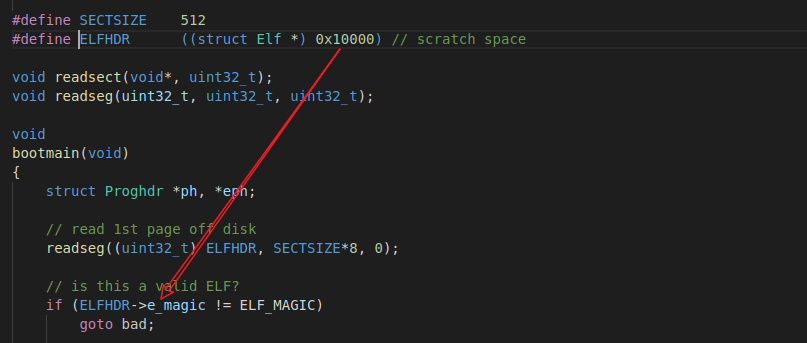


如图，在ELFHDR中存在eph字段指明了读取多少内容

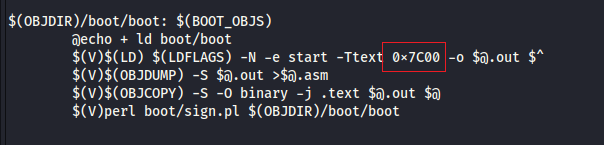
**Question5:**

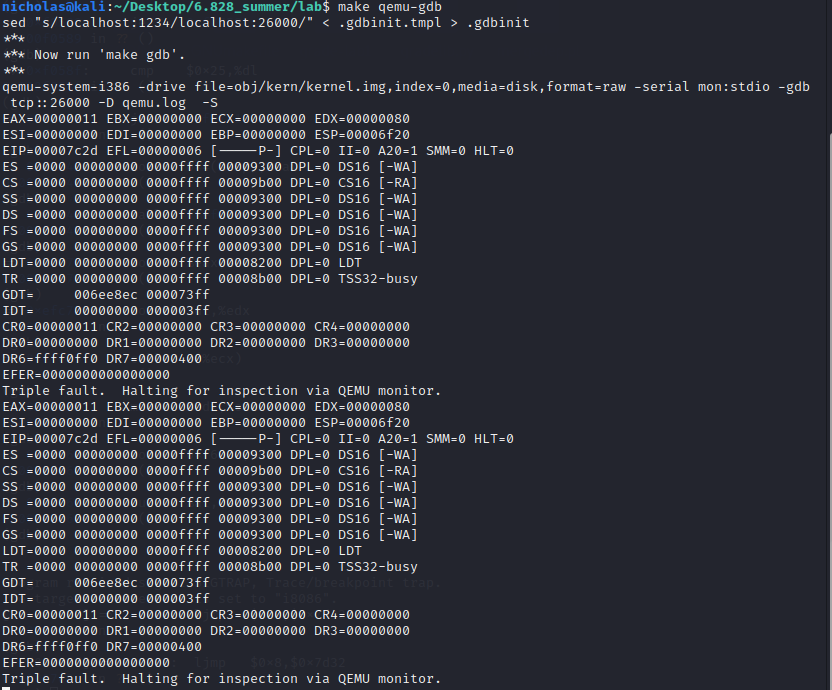
下图将会检测读入的elf文件是否符合格式，如果不符合，会到达bad里面去





接下来修改makefrag中链接地址，并观察程序,发现这里trace不进去，只能观察到报错

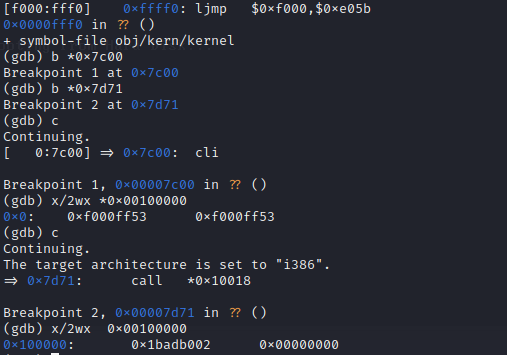




出现triple-fault

**Exercise6**

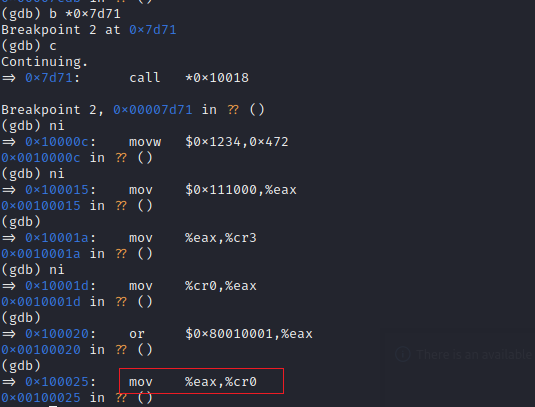
设置两个断点之后，发现内存值确实不一样，如下图



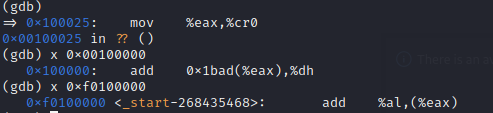
不一样的原因，是因为内核被加载进来了

**Kernel**

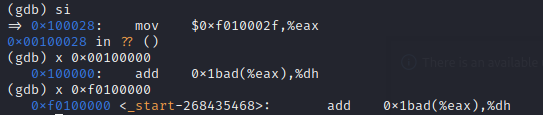
一上来说的加载位置和物理位置的区别就是kernel的物理地址(0x100000)和链接地址(0x7c00)的区别。后面讲了党CR0寄存器PG位被置位后就会开启虚拟地址模式。接下来我们trace进入刚开启虚拟地址模式的内核，看看会发生什么。下图为kernel执行的前几句语句。可以看到0x100025的位置上是mov %eax,%cr0



检查内存，发现为执行语句前，两处内存内容为



使用si指令

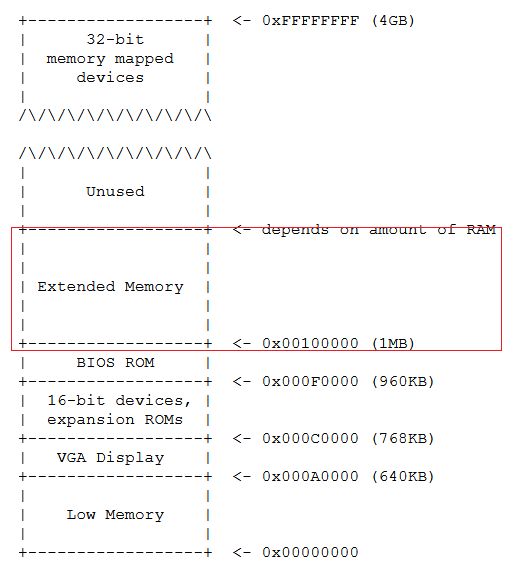


此时看到，原先0x100000处的内容就被映射到0xf0100000处。

为什么这么做，网站上写的很清楚

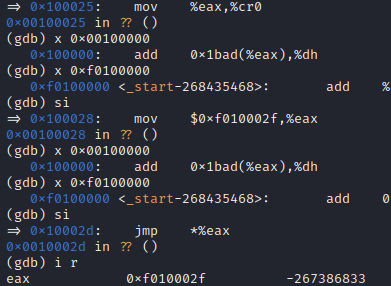
Many machines don't have any physical memory at address 0xf0100000, so we can't count on being able to store the kernel there. **Instead, we will use the processor's memory management hardware to map virtual address 0xf0100000 (the link address at which the kernel code *expects* to run) to physical address 0x00100000** (where the boot loader loaded the kernel into physical memory).

只有在计算机有0xf0100000的物理内存时，才会将kernel映射到这个位置。而这也是kernel预期执行的位置。也就是下图中红色框中的位置。

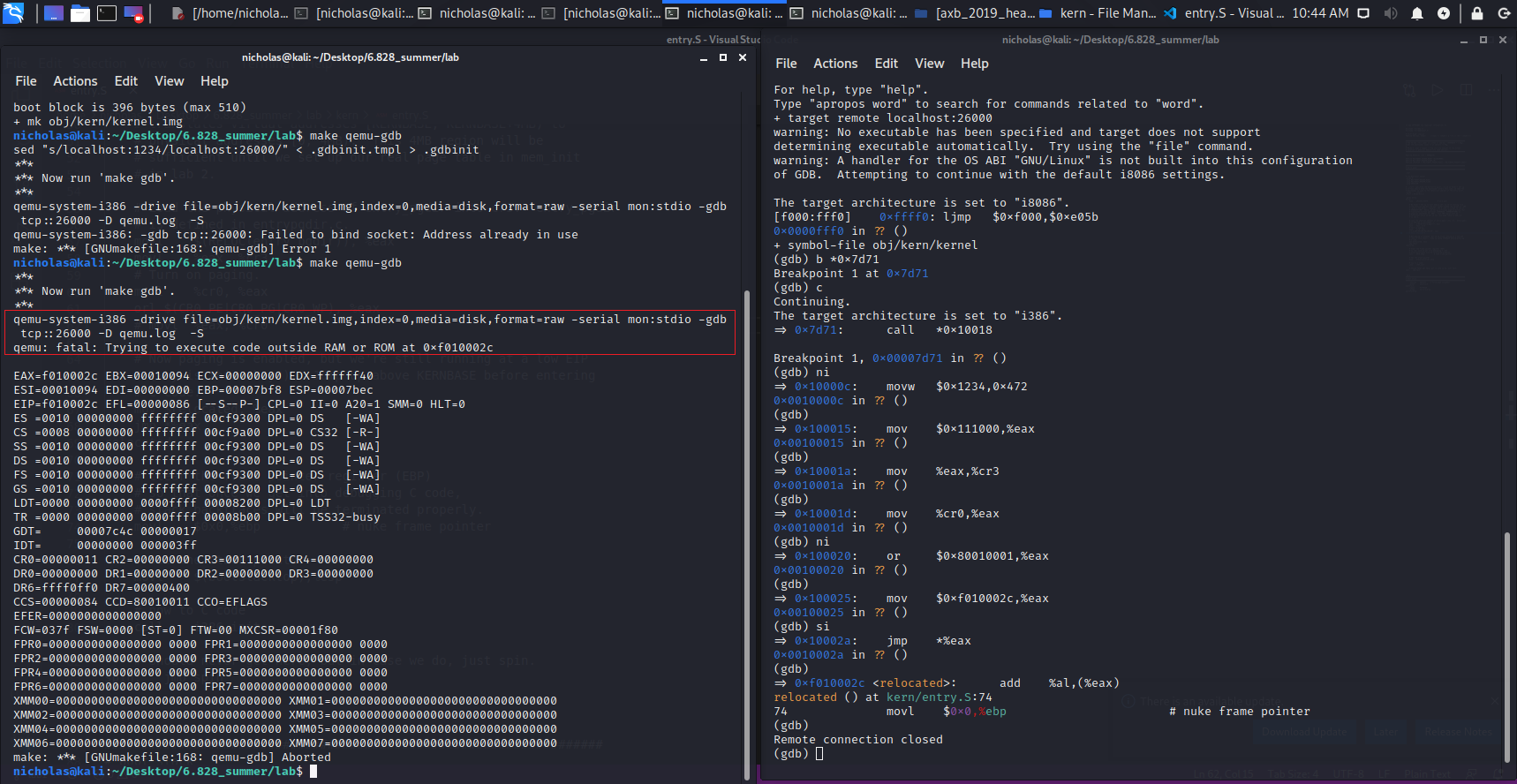


因此执行这句指令之后，kernel就会被映射到这里。

如果虚拟地址映射不正确，可以看到下一句jmp \*%eax就会有问题，因为%eax应该是内映射到的extended memory的地址。



以下是将mov %eax,%cr0注释之后重新运行的效果



如下图所示，程序将跳转到无法执行的位置导致崩溃

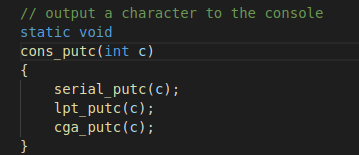
**Exercise8**

首先理解这三个程序之间的依赖关系

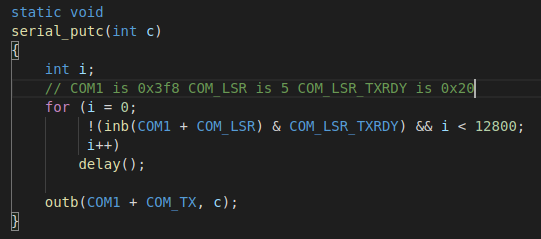
Printf.c包含了putch():输出单个字符以及cprintf()用来输出格式化字符串。Cprintf调用vcprintf以及vprintfmt,是真正实现输出格式化字符串的。

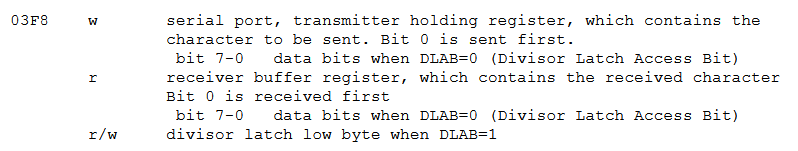
先看cputchar的实现。在console.c中，发现他调用cons\_putc

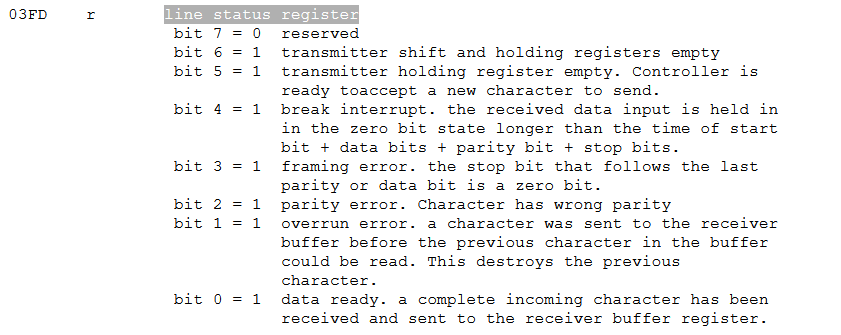
而cons\_puts调用以下三个函数



先看第一个函数serial\_putc

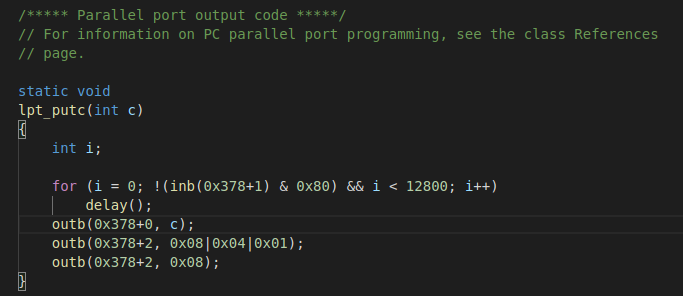




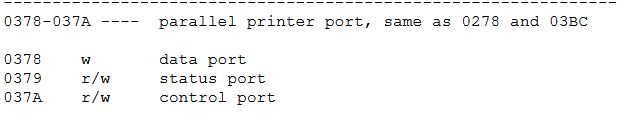


看代码循环体。Inb表示从端口03f8+0x5=03fd读一个字节与上0x20(注释中写明了这代表buffer目前可用)就把一个字符传输到0x3f8端口（串行端口）否则进行delay

接下来看下一个函数

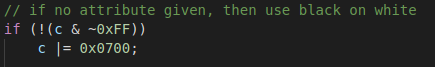


这是一个并行端口。从下面的数据猜测这里也是等待端口不繁忙时，向其data端口输出要打印的数据，控制端口输出。



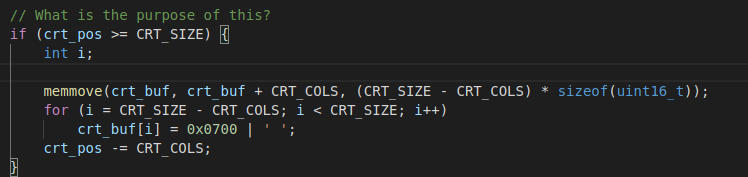
串行端口和并行端口的区别是：并行端口只能从硬盘传输信息，而串行端口可以在硬盘驱动器之间传输信息

接下来是cga\_put函数：图形化界面输出



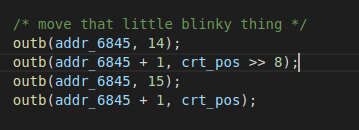
首先了解到，一个输出字符是由两个byte控制的。Byte[0]是字符的值，例如’a’,’d’这种，byte[1]是字符的颜色以及背景颜色。这里0x0700的意思就是如果用户没有默认设置颜色，就按照黑白输出

从代码中可以看到，这路主要对一些转义符号的具体实现做了解释，例如换行，tab,退回光标等。在代码中问我们一个部分是什么意思



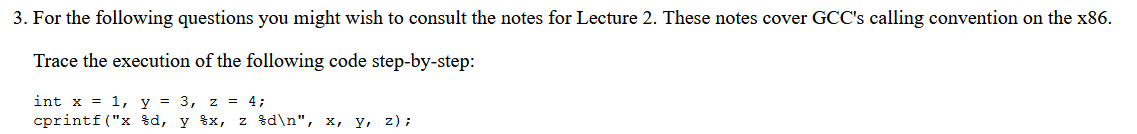
这里是指如果当前屏幕输出满了，就把第二行到最后一行复制到buffer中，从视觉上来看，就是滚轮向下移动屏幕上产生的行为。

接下来是移动光标，mit真是太可爱了w



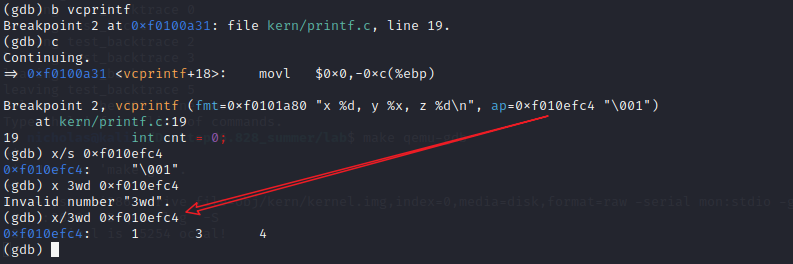
之后的vcprintfmt函数所做的，内容上和上述函数差不多。只是有更多的输出方法，浮点输出，控制位数输出等。基本上还是使用switch\_case方法。

接下来继续回答问题



在cprintf上下断点，si





可以看到ap的值就是1，3，4

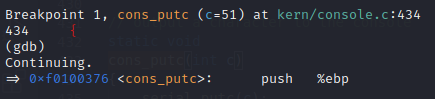
List (in order of execution) each call to cons\_putc, va\_arg, and vcprintf. For cons\_putc, list its argument as well. For va\_arg, list what ap points to before and after the call. For vcprintf list the values of its two arguments.

发现并没有va\_arg这个函数

执行顺序是

Vcprintf(参数1：格式化字符串，值为"x %d, y %x, z %d\n"，参数2：ap，值为顺序压入栈的三个参数1，3，4)

Cons\_putc(int c)参数就是输出参数的ascii码，可以看到括号里就是参数的ascii码



Run the following code.

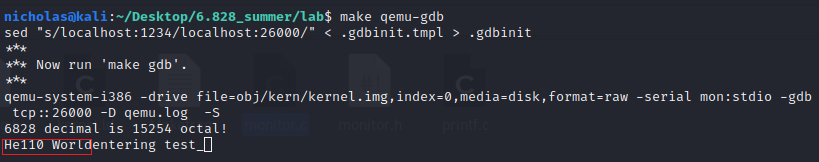
unsigned int i = 0x00646c72;

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

What is the output? Explain how this output is arrived at in the step-by-step manner of the previous exercise. [Here's an ASCII table](http://web.cs.mun.ca/~michael/c/ascii-table.html) 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?

这里是如何解析格式化字符串的问题。具体原因还是%x输出方式以及%s的解析方式，这两者定义在vcprintfmt函数中。最终的输出是



换成大端法就要写成0x726c6400

第五个问题

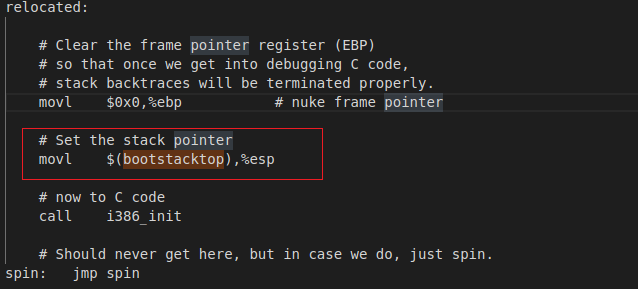
In the following code, what is going to be printed after 'y='? (note: the answer is not a specific value.) Why does this happen?

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

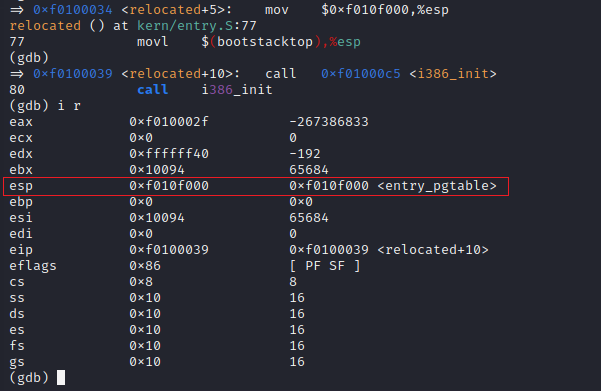
由于32位系统按照栈传递参数，因此这将输出栈上下一位参数。这也就是常说的格式化字符串漏洞产生的原因。

**Exercise9**

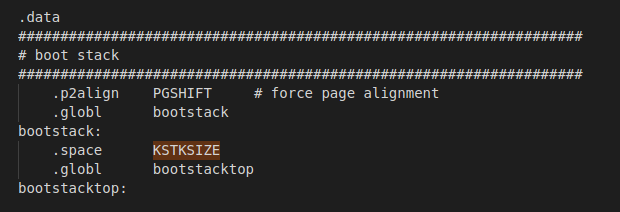
如图位置设置esp



调试可以看见，栈刚被初始化的时候的地址

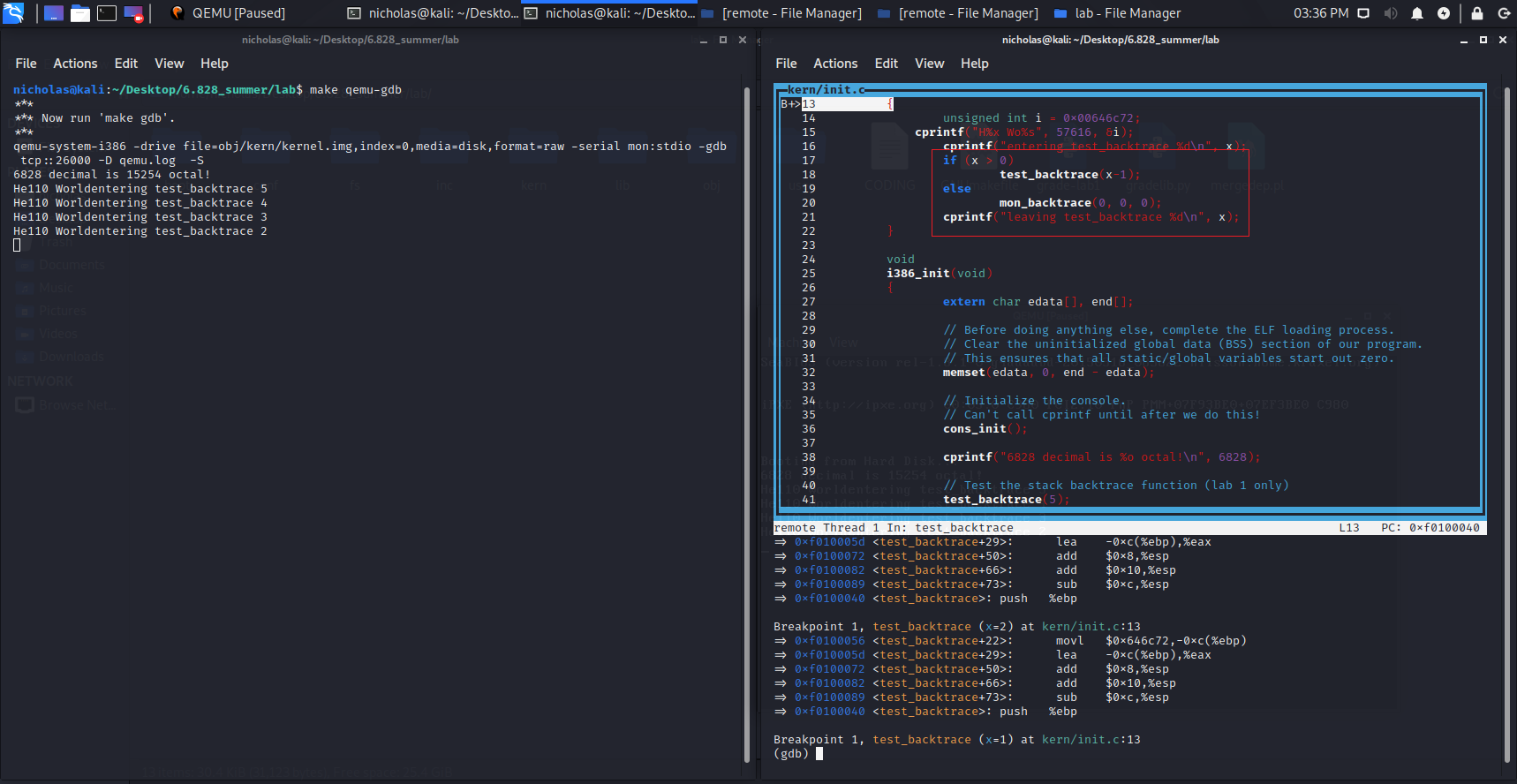


其大小没有在文件中找到，如下图KSTKSIZE没有在文件中搜索到

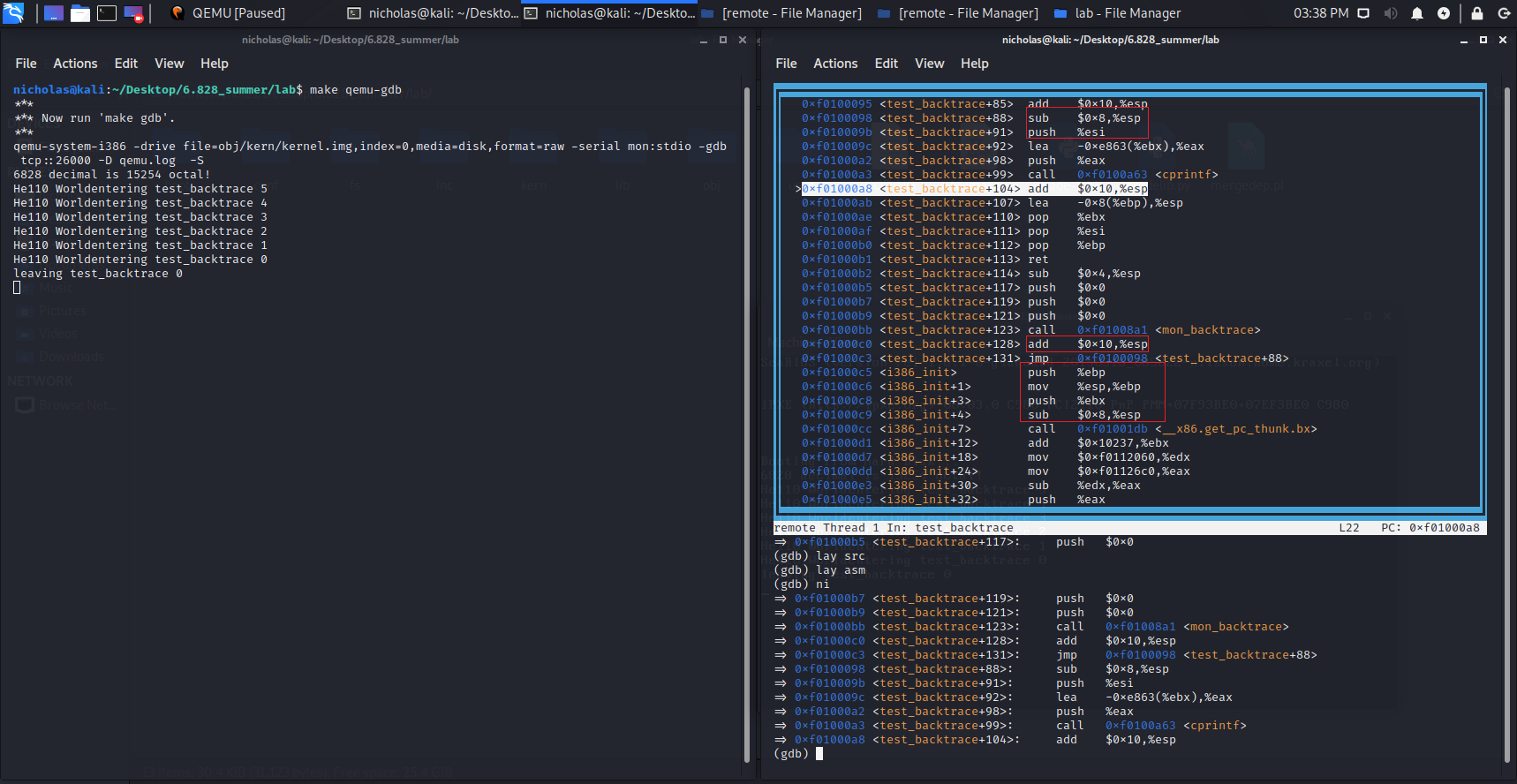


**Exercise10**

通过分析函数调用可知，test\_backtrace是一个递归调用，第一次push进去的是x，x代表递归调用栈的深度。

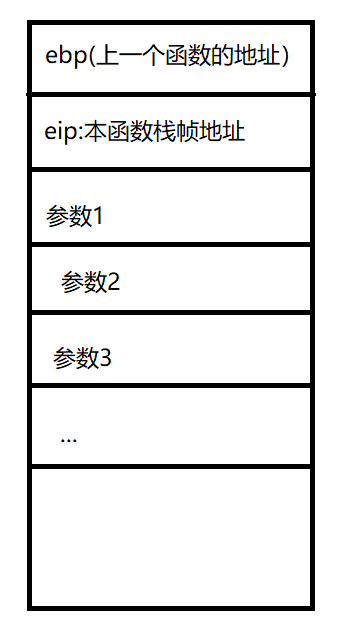


分析汇编代码可知，mon\_backtrace就是把esp加上0x10以回到上一个调用函数，而test\_backtrace就是保存当前ebp,并将esp减去0x8，为下一个函数调用栈留出空间。

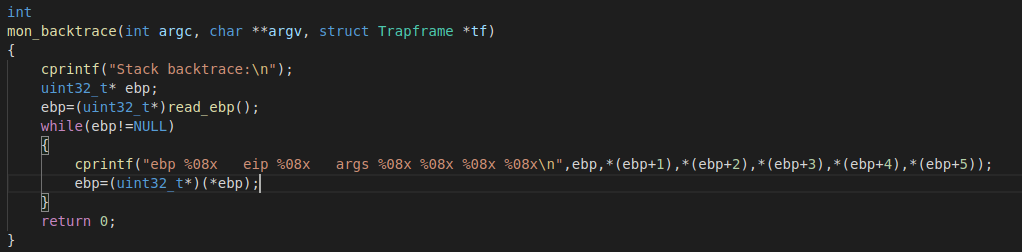


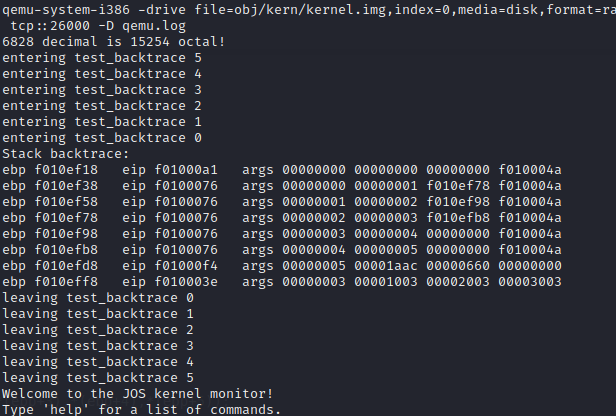
Test\_backtrace总共push10个参数到栈上。一次push两个，总共五次。

接下来一个问题需要对x86函数调用有深刻的理解。知道是怎么调用的，其实通过上一个exercise也可以获得理解。以下是x86函数调用栈栈帧.



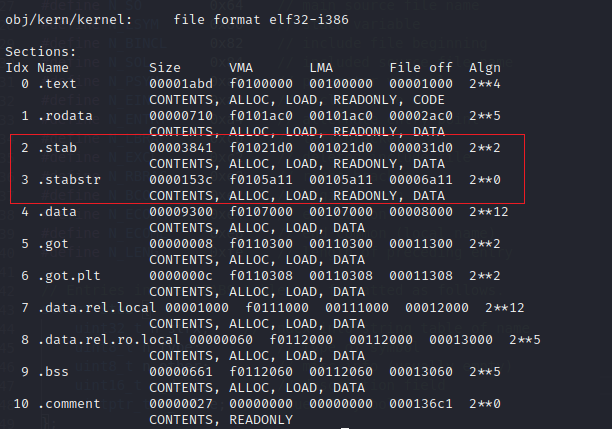
按照此栈帧结构，利用ebp对栈上内容读取，就可以输出题目想要我们得到的内容。



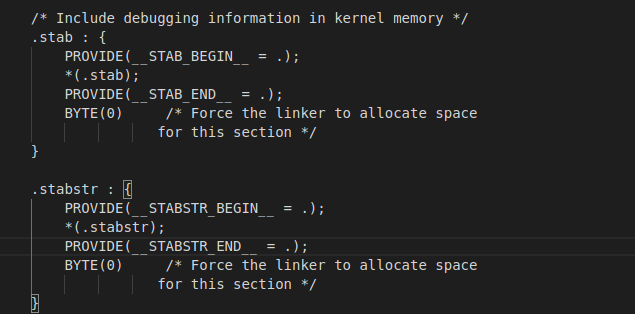


Exercise12要求输出程序文件名以及行号

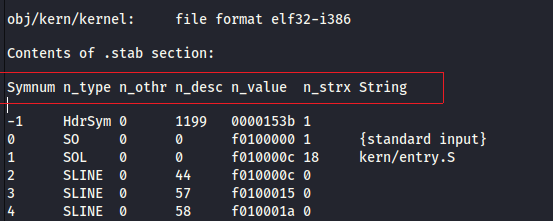
先了解一下这里stabs是什么。Lab上面让先看的四个文件真是一头雾水，直到看了/kern/kdebug.c才明白这个文件是干什么的。如下图所示。教程中的第一个命令是让我们知道，stab在elf文件头部部分是可以存在的



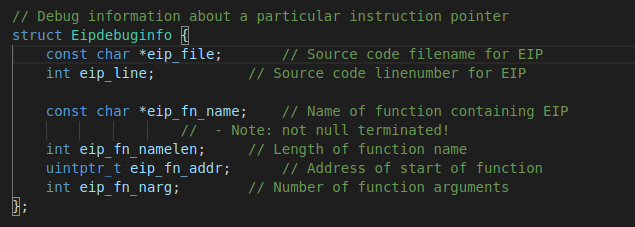
这恰恰是kernel.ld中部分代码给kernel留出的空闲位置



接下来objdump -G obj/kern/kernel命令则列出了所有stab的表项，如下图



这里的列项目，表示的含义可以在kdebug.h中找到

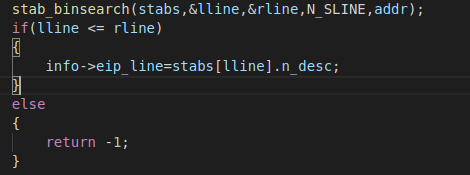


我们要做的debug函数所做的工作，就是把stabs的表项转换成一个eipdebuginfo的结构，便于debug

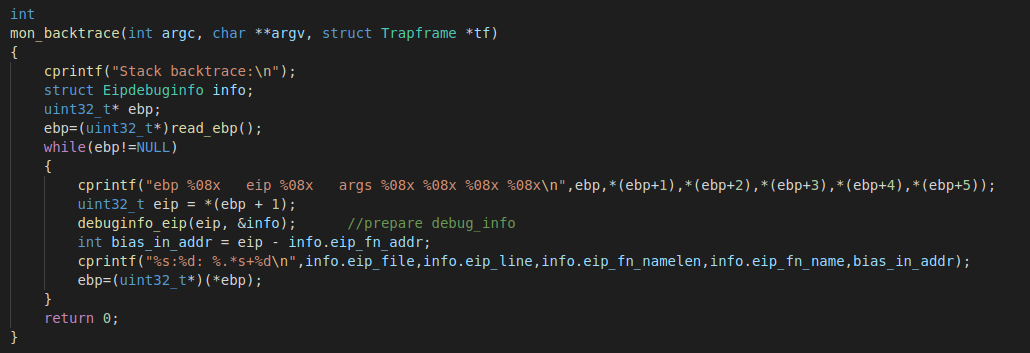
我们接下来要编写的，就是kdebug.c中寻找函数行号或者binary中指令偏移的函数



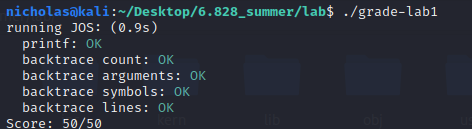
模仿此函数，很容易编写出自己的函数版本。



在最后，编写包含函数名称，行号的调试函数。在编写的时候注意函数名称限制长度输出那里，需要使用%.\*s限制输出长度，其中\*就是栈上的一个参数位置代表了输出长度。其余的基本按照题目想让输出的输出即可。



通过了grade测试



总结一下这个lab。还是学到了不少。首先是加点启动过程，从第一句语句的跳转到bios到从实模式到保护模式，再到bios搜索并加载完操作系统扇区，并将控制权跳转到操作系统主引导扇区。到达操作系统之后是操作系统的链接，跳转到操作系统代码部分，再到printf的详细实现原理（如何输出到屏幕——端口含义以及控制比特含义，背景颜色），格式化字符串的处理方式（原来竟然是switch-case）以及接下来的x86函数调用栈栈帧的具体实现，并根据栈帧编写自己的backtrace函数，最后在此基础上更进一步，了解了elf中stabs部分，将其加载到调试信息中并输出。基本完成了引导，交互和基本调试的步骤，为后续编写高级指令以及了解操作系统启动过程打下了坚实的基础。