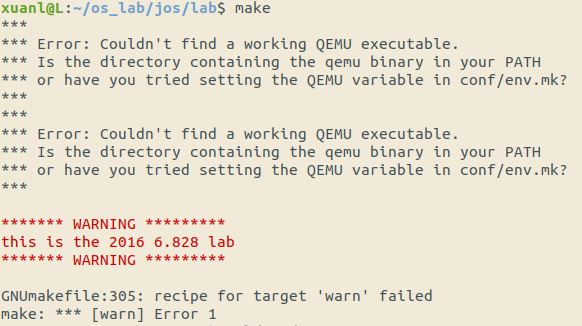
## Lab1

### Part1

注意.gitignore中忽略了/obj

Make失败，需要安装QEMU（是指Quick Emulator吗？）

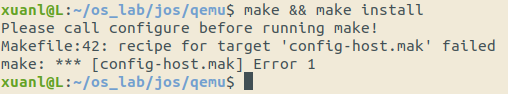


git clone http://web.mit.edu/ccutler/www/qemu.git -b 6.828-2.3.0

没有进行配置，可以用./configure配置安装路径

直接make && make install,将会安装到/usr/local

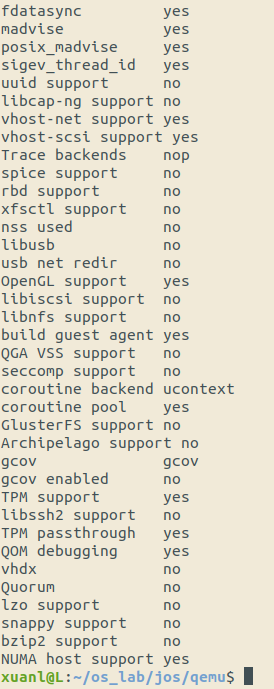
（注意：即使不配置安装路径，也要运行./configure）



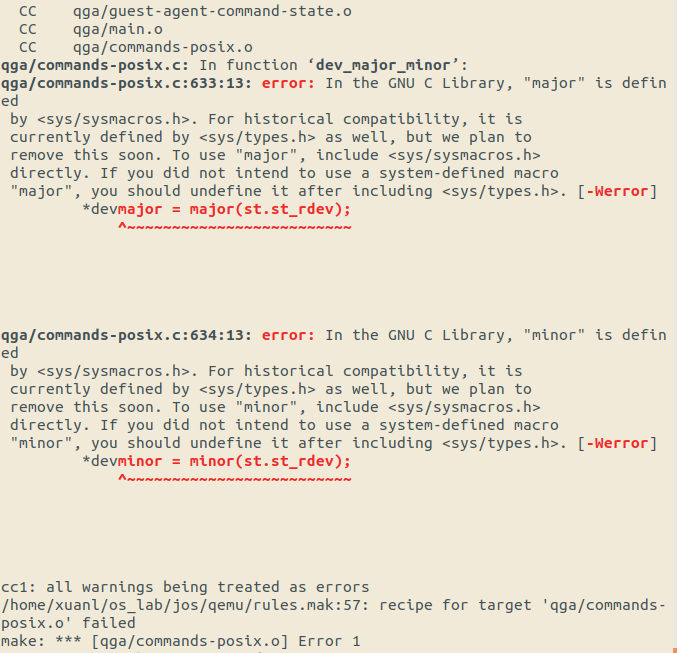
报错：

DTC (libfdt) not present.

不能直接./configure，要加上--disable-kvm



依然报错，感觉是版本兼容性问题。

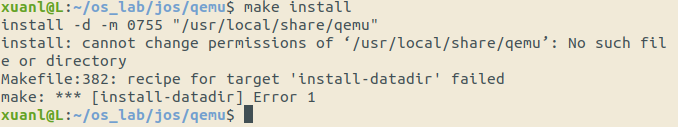


去看了18年的，发现configure的时候加上了--disable-werror，

确实忽略了警告，不会fail了

Make就整了好久

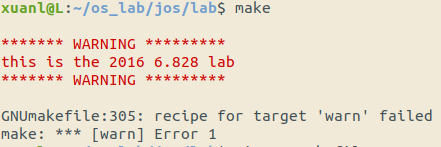
Make file又出问题



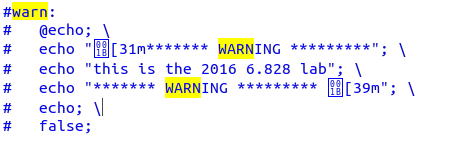
这是权限问题么？

切成root，就可以了。（所以上图中就是安装路径，顺便记下来了）

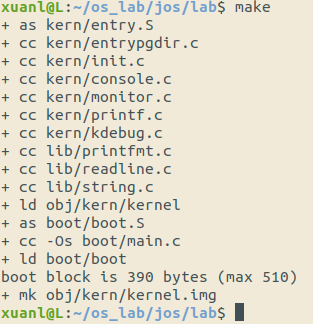
Lab1的make还是有问题，居然因为时2016年的就不让用了



手动修改makefile文件，把所有warn的部分注释掉

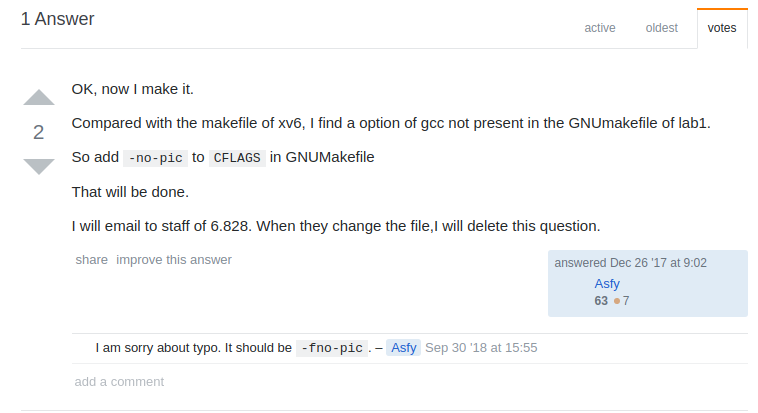
、

然后make，终于成功了！



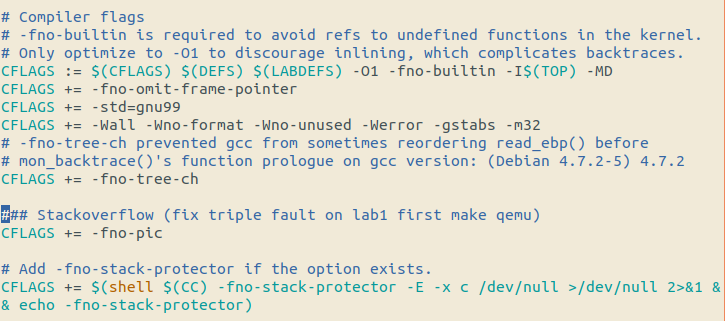
但是，继续make qemu遇到了大问题，出现了Triple fault，然后一直卡死了。

这个问题实在没有任何头绪，最后在stackoverflow上看到类似的情况，解决方法还是修改makefile文件。

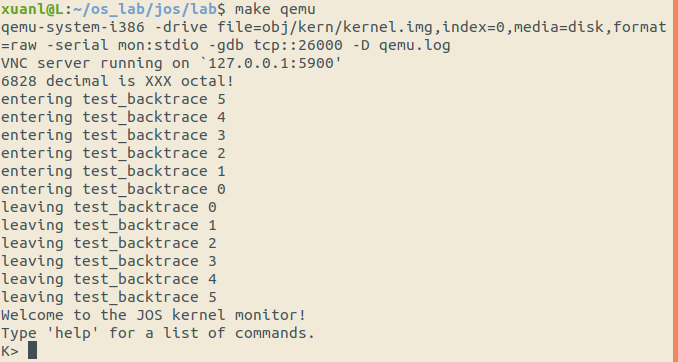


在makefile中的CFLAGS加上一行，CLAFGS += -fno-pic

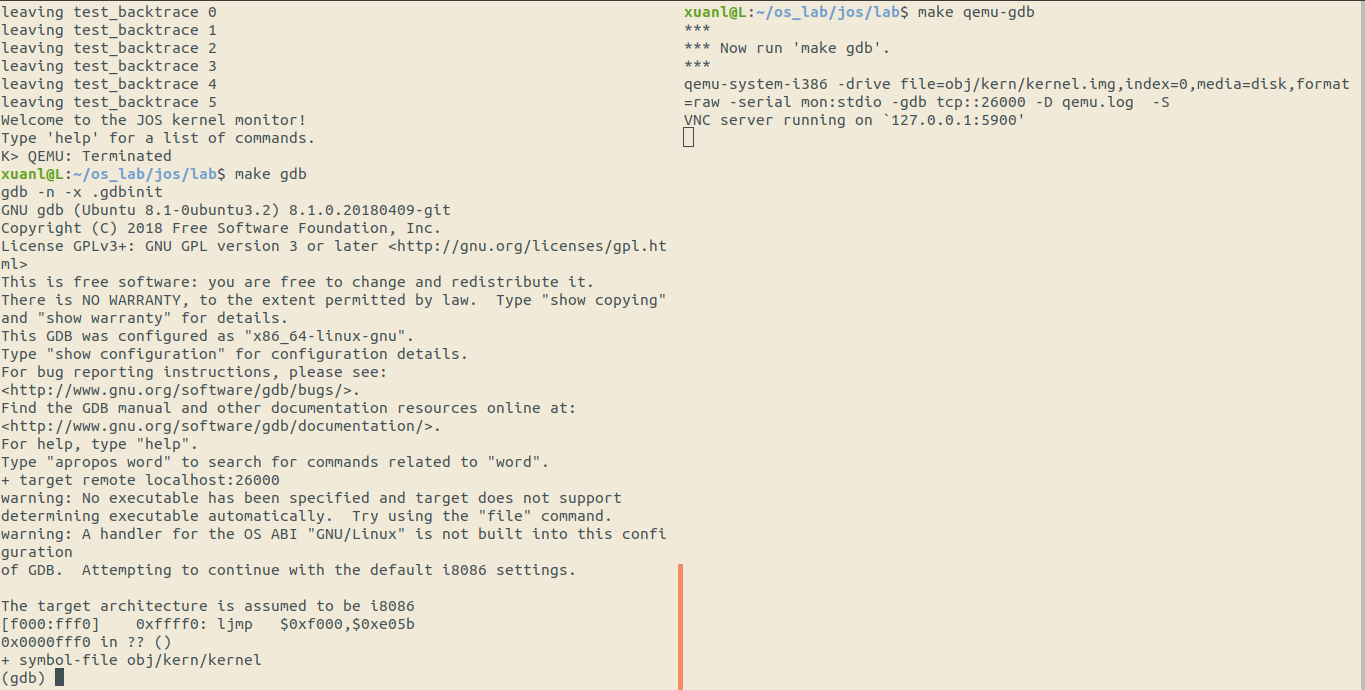
这个地方的pic表示position-independent code位置无关代码，用于生成动态链接的代码，那这里用-fno-pic，应该就是生成的代码里使用绝对地址而不是相对地址。但为什么必须要这样呢？



终于完成了PART1！快落泪了！



退出的话，似乎Ctrl+C不行，得Ctrl+A，X



调试的话，先在一个终端里make qemu-gdb，然后再在另一个终端里make gdb。

这里反汇编出来，[f000:fff0]处有一条指令，ljmp $0xf000,$0xe05b

课件BIOS的第一条指令是跳转指令，且这条指令的地址为0xffff0，跳转的目的地是0xfe05b

（回忆汇编课学的，CS和IP）

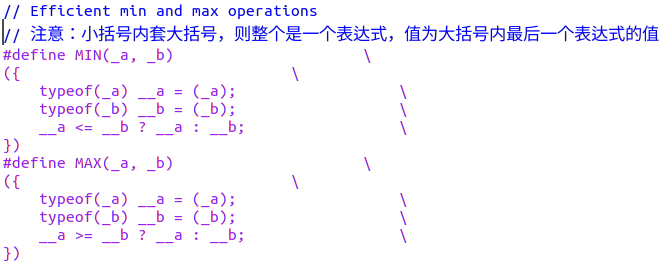
### Part2

阅读boot/boot.S和boot/main.c代码，前者是汇编代码，后者是对应的C源文件。

obj/boot/boot.asm是反汇编文件，可以看到在物理内存中的实际存放位置，在调试的时候非常有用。Obj目录下还有其他的反汇编文件。

直接看代码有点迷，还是得先看inc/目录下的头文件，把数据结构之类的搞清楚。

这个写法很妙啊，看了之后试了下，小括号不能不加，a=({a++;a++;});

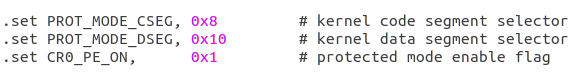


Elf.h里面主要是ELF头部数据结构，以及程序头表，和节头表，这三个数据结构。

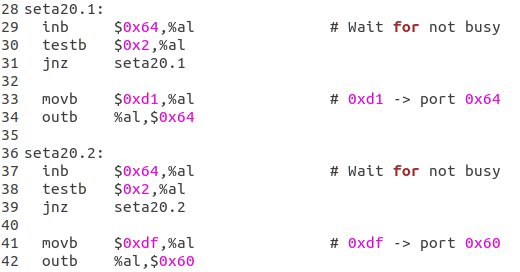
最后是一些常量宏定义

X86.h里面都是些内嵌汇编语句，读完要花不少时间，可以先看boot代码，遇到没见过的函数再去读

这里的.set应该就是设置常量，类似于.equ或者=之类的吧

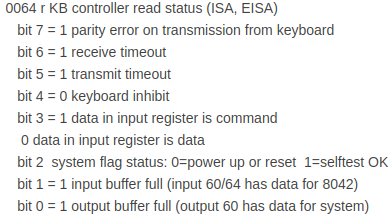


0x64号端口是什么呢？是键盘的状态寄存器。



这是网上找到的一个资料（<https://blog.csdn.net/haorenakuan/article/details/84622630>）

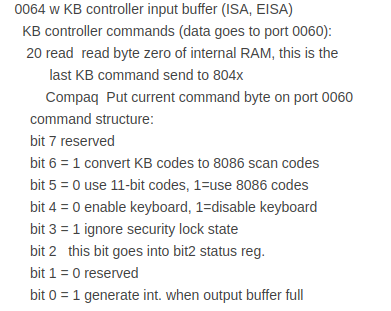
其实可以看bochs的资料，也可以（<http://bochs.sourceforge.net/techspec/PORTS.LST>）



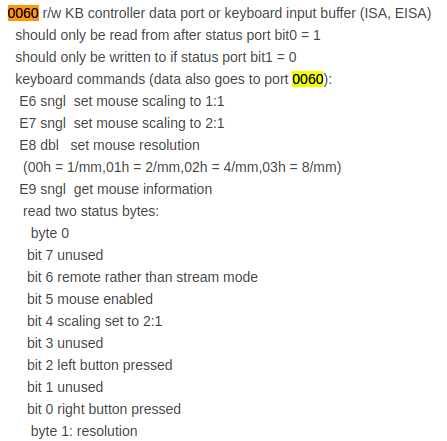
按照这个意思，如果testb $0x2, %al 之后标志位为0，说明input buffer为空，否则为满。那这一段代码就是等input buffer为空的时候继续。

之后把0xd1输入到0x64，目的是什么呢？不知道是不是应该看下面这个input buffer。

应该是同一个地址的端口，既可以写也可以读，然后含义是不同的？



至于0x60端口，因为是用来写入，所以应该看这部分吗？这里E7、E8啥的，都不知道是什么意思。

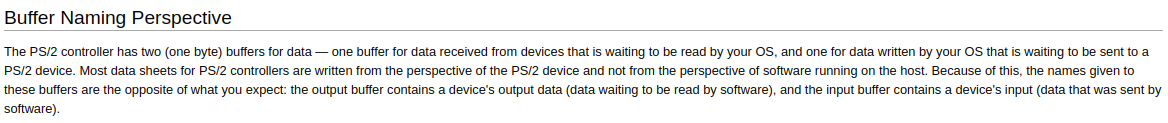


虽然还是不太明白，这个地方实在不知道要看什么了。

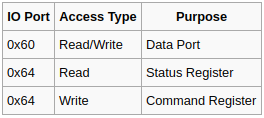
找到个这个，专门讲键盘鼠标部分的，不知道能不能看懂。（[https://wiki.osdev.org/%228042%22\_PS/2\_Controller](https://wiki.osdev.org/"8042"_PS/2_Controller)）

PS/2或称KBC：这个相对AT新一些，同时支持键盘和鼠标，但差不多。还有个XT接口标准，更老。

USB Legacy（遗产） Support：主板在把USB键盘和鼠标模拟为PS/2（硬件模拟），一般操作系统运行起来后会关闭这个模拟

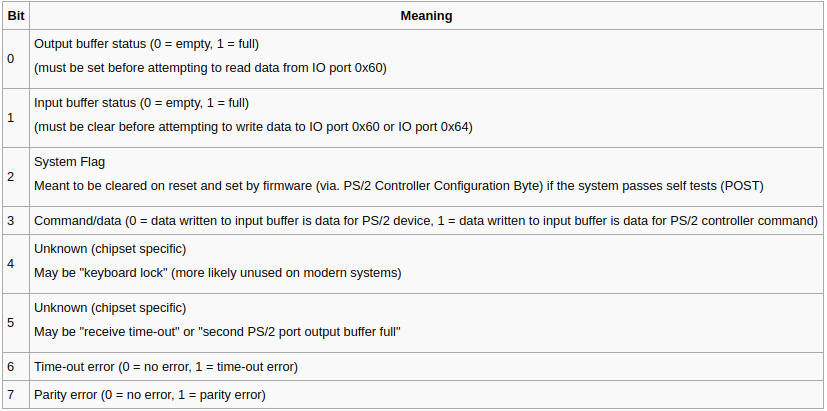


上面这一段提到了命名的问题。端口的命名是站在硬件的视角来的，因此output buffer其实是键盘输出的数据，对于软件来说需要用in指令来读取。反过来，input buffer则是软件输出，对于硬件是输入。



上面这个图很清楚。对于AT和PS/2协议来说，只用到了0x60和0x64两个端口。更老的PC-XT PPI才用到了0x61端口。

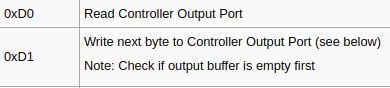
同一个端口地址的读取和写入，可能在实际的物理硬件内部对应的是不同的硬件寄存器。



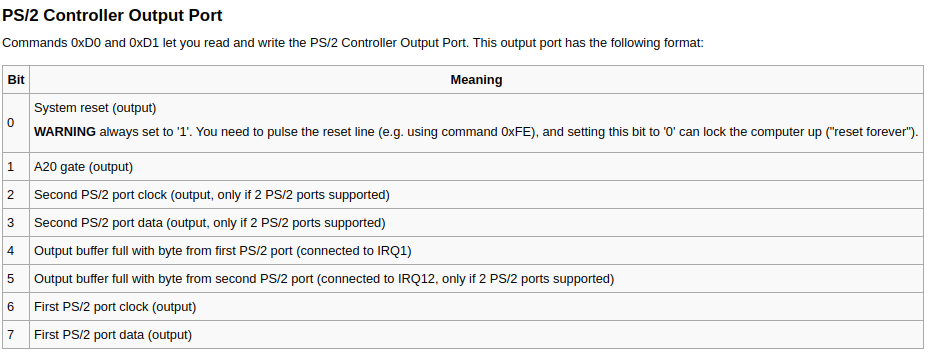
从这个表可以看出，首先，任何需要写数据的操作（写入0x60/0x64），都需要先等待0x64的1号比特位为0时才能写入（可见这一位为1，表示当前设备正在被其他什么东西写入，即写繁忙状态）。但之后写入0xd1到0x64寄存器，这一步感觉还是不能看这个表。继续往下看看。

同样是对0x64，out指令其实写入的是设备的Command Register，而in指令则是设备的Status Register，所以地址一样但功能完全不同。

如果命令是一个字节，则使用out指令输出到0x64地址即可。如果有两个字节，则第一个字节输出到0x64，第二个字节输出到0x60。要注意第二个字节输出前一定要读取Status Reg看input buffer是否空闲。



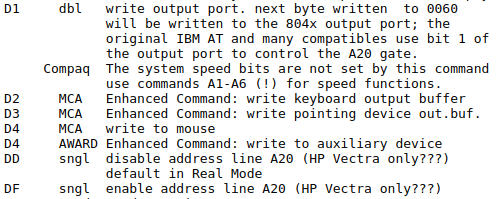
可见，0xd1送到0x64之后，外设此时会等待next byte送入0x60。



上图可见，0xd1的目的是写入Controller Output Port。写入的数据地址即0x60，根据汇编代码可知，数据为0xdf(1101 1111)。

但这里还是有些不太明白，最后为什么要设置成这样呢？上面每一位的含义具体是什么呢？

这个时候再回到bochs那篇文档，同样找命令，发现如下情况。可见，第一个命令0xd1确实是等待输入第二个byte命令，而这里出现了0xdf，其含义为enable address line A20（可见，这段代码的目的就是设置A20 GATE)，这和注释是一致的，也和上面的表是一致的（因为下面可以看到，0xdf和0xdd的区别就是是否开启A20）



最后就是A20 GATE的含义了，可以参考（<https://blog.csdn.net/zzobin/article/details/4276372>）这篇文章。

A20就是第21根地址总线（8086只有20根，后来的架构寻址能力增强了）。A20 GATE为0时，则第21根地址总线的值恒为0。当超过0x100000时，就取模了。在实模式下，打开A20 gate（即这一位设为1），可以访问所谓的“高端内存区”（地址100000h--10ffefh

），而在保护模式下，显然A20 GATE也必须打开，否则只能访问奇数M内存区域。

接下来遇到的问题是LGDT指令

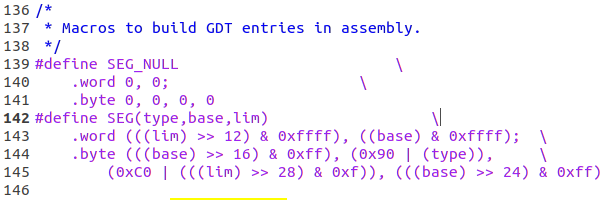
lgdt    gdtdesc

这个直接查也能查到，LGDT是load global descripter table的意思。这里的全局描述符表，主要是在虚拟内存那部分用到。Global这个是操作系统的虚拟地址映射用的，而local的那个每个进程都有，是用户进程用的。这个可以看《深入理解UNIX》上的讲解。

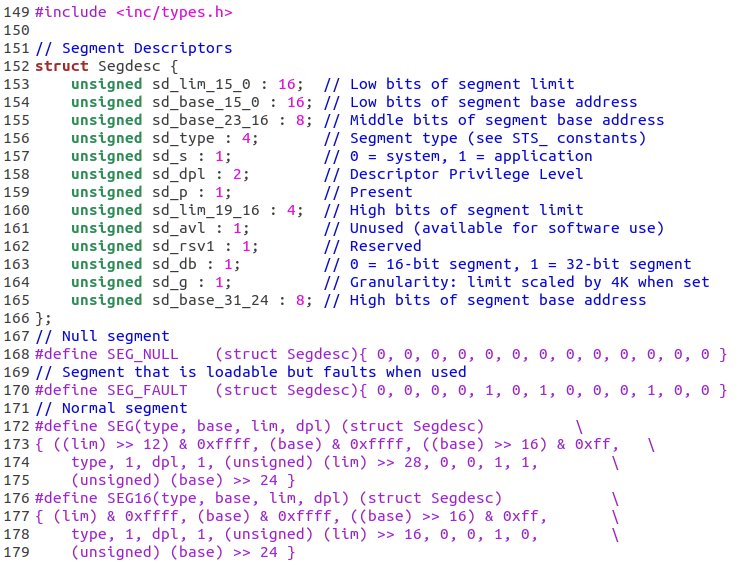
具体这个指令什么意思，可以参考（<https://www.felixcloutier.com/x86/index.html>）这个网站上的，上面各种x86指令都挺全的，而且比较新（19年5月更新）

在代码中查找gdtdesc，发现如下不明标识符。搜索后没有结果，因为这是第一次出现在这个代码中，而且明显不是汇编代码，因此应该是在别处定义的宏。查看代码第一行，果然发现了一个头文件，接下来要在这个头文件中寻找相关信息。

这个头文件中，和GDT相关的部分分为汇编和C语言两种写法。汇编如下



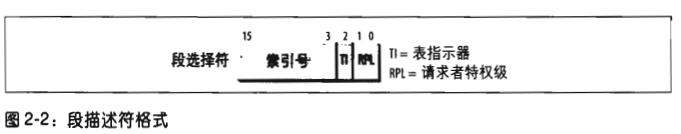
为了更好地理解汇编的写法，我觉得还是得参考下面的C语言的写法。注意这里用到了C语言位域的特性，即struct中成员名后面冒号加位数，可以节约空间。



对于上面这个Segdesc数据结构的细节，可以参考《深入理解Linux内核》的第二章

MMU包括分段单元（把逻辑地址转成线性地址）和分页单元（把线性地址转成物理地址）两部分。

段选择符（Segment Selector）16位，也就是段寄存器里存的内容。偏移则是32位。

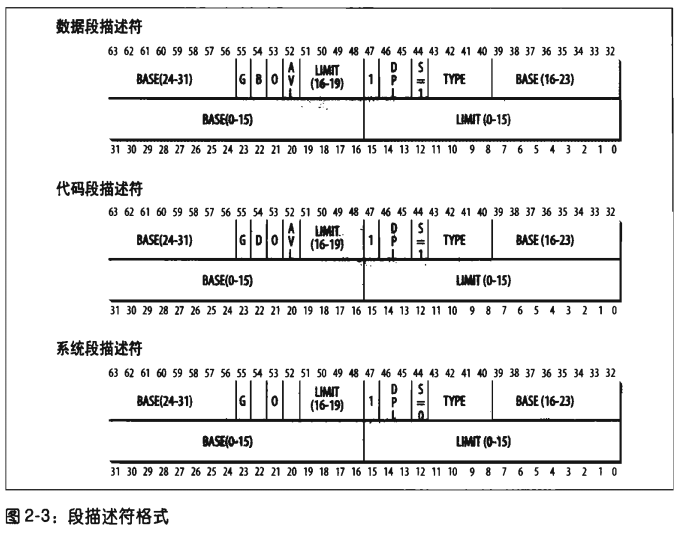


段选择符的高位是索引号，也就是“段描述符”在描述符表中的偏移，通过索引才能找到段描述符。

段描述符(Segment Descriptor)的大小为8字节，可以放在GDT或者LDT中，而GDT的基址存放在gdtr寄存器中，LDT则在gdtr寄存器中。

段描述符中的sd\_p注释为present，查阅资料可知，表示该段是否在内存中。Linux操作系统将这一位恒置1，因为linux从不把整段都交换到磁盘上。

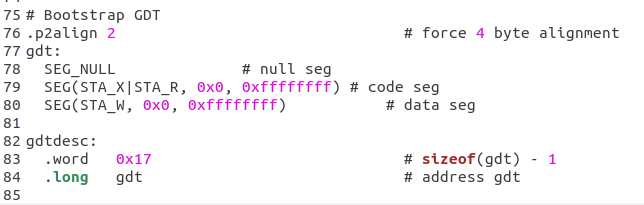
当sd\_g被置为1时，则limit的单位是页（即4K），所以此时段大小最大为4G。



使用上图可以和代码对照，就可以清楚地了解数据结构每一位的含义了。

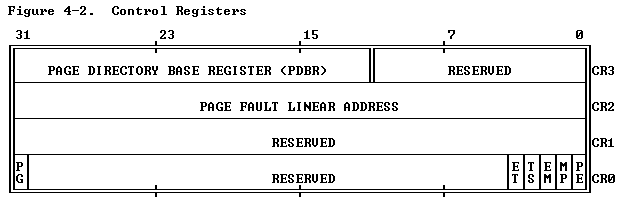
对于普通段，sd\_s=1,sd\_p=1,sd\_db=1,sd\_g=1，保留位都是0，其余需要根据传入的参数来设置。

注意和SEG16区别，SEG16是以字节为单位而不是页，因此不用于保护模式。

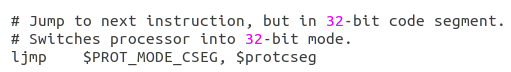


由上图可见，实际操作系统中的GDT里面，首先有一个空段（不清楚干啥的），然后是一个代码段，以及一个数据段，两个段的地址都是0，而且limit都是4G，除了保护作用外其实相当于没用到太多分段的功能，此时虚拟地址与物理地址实际上相等（见注释）。

接下来看到寄存器%cr0，不知道是啥。查了之后原来这属于控制寄存器。这里可以翻看Intel 80386手册的system register部分，对各种与操作系统相关的控制寄存器都有很好的描述。包括上面的LGDT指令等等，都有。



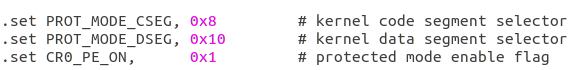
最低位的PE位表示Protection Enable，把这一位置1则将使CPU进入保护模式。



上面ljmp有两个操作数，其中第一个操作数为16位数，表示用这个值更新CS寄存器，后面的立即数为32位，表示地址。（因为protcseg是.code32下面的第一句，所以地址应该是0？）

这种jmp的用法在手册上也有，当然这里（<https://docs.oracle.com/cd/E19455-01/806-3773/instructionset-73/index.html>）

也更详细地描述了ljmp的用法。



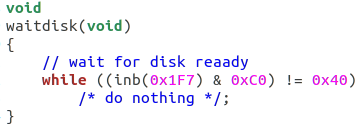
开头的这个常量设置现在也明白了，因为第一个段是SEG\_NULL，占8字节，所以代码段的段描述符在表中的偏移就是8了，然后代码段描述符为16，即0x10。



最后设置堆栈指针，然后调用bootmain，汇编部分就结束了。这里$start的值，不清楚是多少啊？要么是0，要么可能是0x7c00（因为注释提到BIOS将这一段代码从硬盘的第一个扇区读取出来之后，存放在物理地址的0x7c00处），这样压栈的话%esp减小，那栈空间其实非常有限？

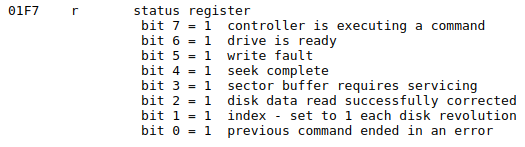
Main.c

这里面有一些声明在x86.h以及elf.h当中。

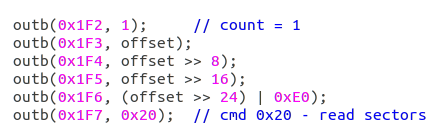


这里的inb(port)函数其实就是从0x1f7（port）端口读入数据作为返回值。之后和0xc0与。

查阅bochs手册之后，果然这个端口是硬盘的状态寄存器。



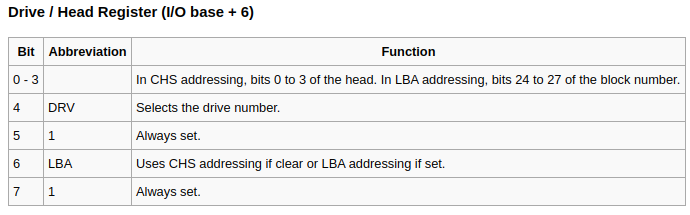
由上面可见，如果返回值和0xc0相与后等于0x40，说明两件事：其一是控制器不在执行指令（bit 7)，其二是设备准备就绪（bit 6)。



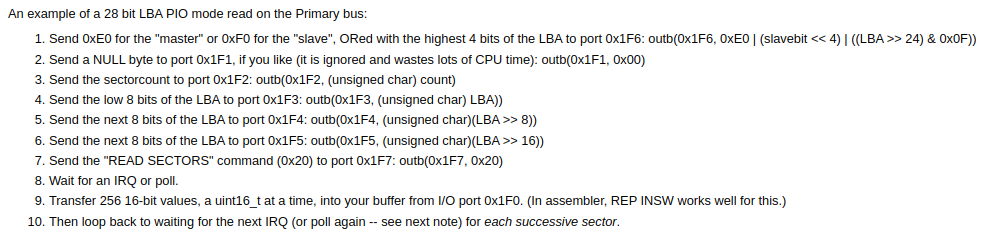
这个地方，0x1f2端口表示读取的扇区数量，0x1f3表示扇区号，0x1f4和0x1f5共同表示柱面号(cylinder low/high)，0x1f6端口有点没懂。查阅资料（[https://wiki.osdev.org/ATA\_PIO\_Mode#Writing\_28\_bit\_LBA](https://wiki.osdev.org/ATA_PIO_Mode" \l "Writing_28_bit_LBA)）可知，这里使用的是ATA硬盘规范。有且仅有一根线（一位）选择总线上的哪一个设备处于活动状态，也即总线上最多只有两个设备，被称为主设备（master）和从设备（slave）。

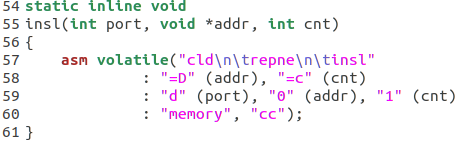
ATA规范有三种寻址模式，为28-bit LBA、48-bit LBA以及CHS。这里LBA即logical block address，而CHS则表示Cylinder-Head-Sector。

实验中使用的是LBA，在上面的网站中找到0x1f6的规范说明如下。



由这个图就可以知道0xE0 | (offset >> 24)的含义了。首先offset是LBA的线性偏移，右移24位后的高4位表示块号。而0xE0中相当于上图中的LBA位（6 bit）置1，即表示采用LBA寻址。而DRV位置0（4 bit），即选择第0号设备（即主设备）。这在下文中的28 bit PIO示例中也可以佐证。





Insl()函数是一个比较复杂的内嵌汇编。这里可以参考实验提供的行内汇编参考资料，但我感觉介绍的不是那么详细和完整。《汇编语言程序设计》这本书上用中文介绍，详细看扩展asm格式。这个地方，”memory”表示这段汇编使用了没有在输入值和输出值中定义的任何内存位置（因为写入了一段内存），而cc表示改变了标志寄存器（cld改变了direction）

内联汇编的一整段在debug的时候会被当做一条C语句，可以用stepi进入单步执行每一条汇编。

上面这一段汇编，把addr变量的值（存放读入数据的内存地址）存入寄存器%edi，cnt的值存入%ecx，而port的值存入%edx。Cld改变方向（clear direction)，repne表示重复下面的字符串指令直到%ecx == 0或者ZF==0。Insl指令从端口%dx中读取数据，然后存入%edi寄存器指向的内存空间，之后%edi = %edi + 4，而后%ecx = %ecx - 1，并循环。

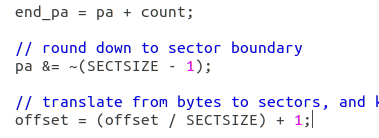
这一个函数，相当于读取port端口的内容，并存入addr地址中。

这里要注意，实际上addr指向的就是物理地址（因为段从0x0开始，limit为4G，所以逻辑地址就等于物理地址），因此这里不需要设置缓冲区什么的，这不是应用程序，这相当于是内核态，因此可以任意使用内存空间。

至于为什么要SECTSIZE/4，原因也很简单了，因为硬盘传输数据以双字为单位（insl的l表示long），双字这里是4个字节。



下面这几行，首先是让pa和段边界对齐。SECTSIZE==512，即0x00000200，减1后为0x000001ff，然后取反，得到0xfffffe00。



至于上面这个offset的变化，其实就是本来offset是以字节为偏移的，现在改成LBA的形式，所以是以块单位作偏移。最后+1，我觉得是因为，第0块是引导扇区，存放这个boot程序。然后第1块开始才是操作系统的代码。

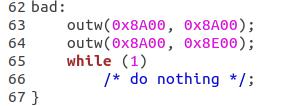


这里注意看注释，读的是第一页，而不是第一块，一页是4K，所以这里SECTSIZE\*8。

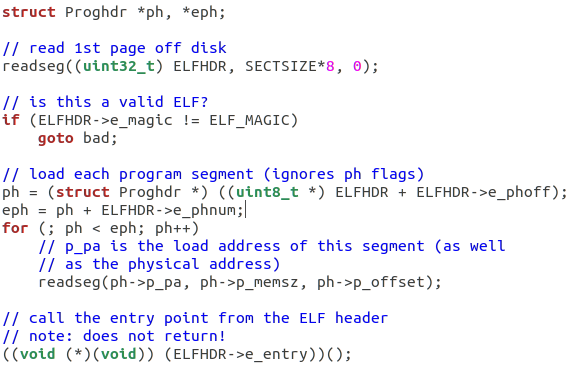
这里读进来的ELFHDR的类型为(struct Elf \*)，但是其实整个4K内存中只有开头部分是Elf首部，而不是说首部就占了4K。

0x8A00这个端口不知道干啥的，只能搜索了。

（<http://bochs.sourceforge.net/doc/docbook/development/debugger-advanced.html>）



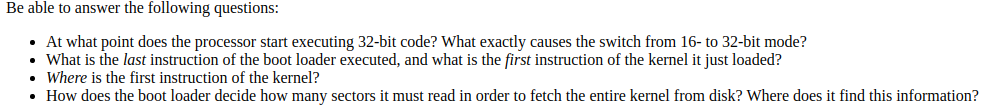
看起来，这是一个模拟器提供的端口，用于调试。首先发送的指令是0x8A00，表示激活调试设备。然后0x8E00应该是一个命令，但是在bochs的手册上没有查到这个命令，其他地方也实在查不到，这个可能是QEMU的命令？在bochs上有一个0x8AE0，效果是返回debuger的命令行，可能这两者效果类似？存疑。。。



通过上面这一段可以分析出，ph是程序头表（即段头表）的首地址，eph是程序头表的尾地址。

这里ph一开始指向的是第一个段头表项，也就是0号段。这一段的内容现在还在硬盘上（也可能已经加载进来了？），所以需要用readseg()函数，将这一段从硬盘上的位置（ph->p\_offset)加载到内存的p\_pa地址上，段大小为ph->p\_memsz。经过这个循环之后，所有需要加载到内存的段都已经从硬盘中加载到内存中了。

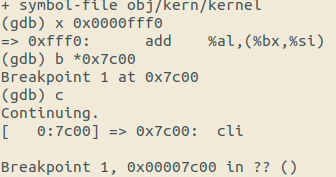
最后这个((void(\*)(void)) (ELFHDR->e\_entry))();是一个函数调用，感觉非常高级。这先是把这个地址的类型转成一个函数指针（参数为void，返回值void），然后用（）操作符作用于这个函数指针，则相当于call 了这个地址。



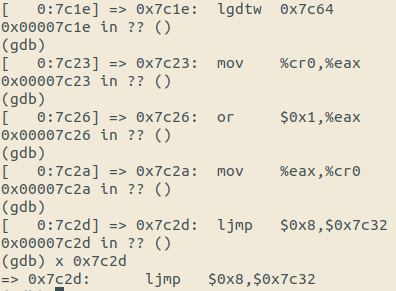
到此为止，代码分析得差不多了。可以回答一下实验的四个问题。

1. 在给处理器设置玩保护模式之后，就使用ljmp指令设置CS寄存器的值并跳转到32位代码段开始执行。将寄存器%cr0的最低位（PE位）设为1，即可开启保护模式。
2. 最后一条指令为call指令（call \*0x10018），而第一条内核指令则需要进入DGB之后调试才能知道。
3. 在0x10018内存空间内存放的指针所指向的地址位置，也需要用GDB。
4. 首先boot loader应该先读出ELF头部，因此先从1号扇区开始读取一页（4K）。然后，根据头部中的e\_phnum字段确定还有多少段头表项（即还有多少段需要从硬盘读取到内存中）

Gdb调试



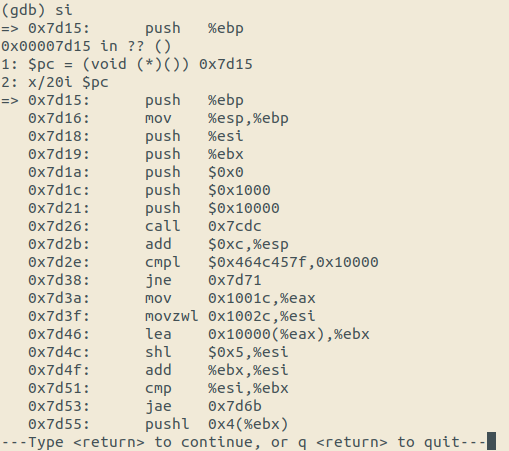
先用b \*0x7c00在boot程序的入口处设置断点，然后c继续执行到断点处。然后si单步执行机器指令。



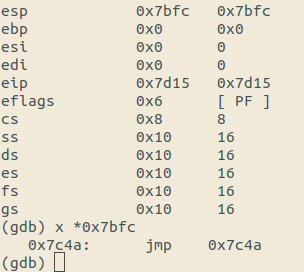
这里可以看到，对于lgdtw和ljmp指令都能够正常地反汇编，而使用ojbdump得到的反汇编文件中，这两条指令是错误的。一条指令被拆成了两条指令。



执行boos.S中的最后一个call指令之后，会提示要不要继续。这里已经跳转到了0x7d15，即进入了bootmain函数中。这下面显示的好像是曾经输入过的gdb的命令？



此时检查%esp寄存器指向的内存空间的内容，可见返回地址0x7c4a已经压入栈中。Push操作先将%esp -= 4，然后将%pc+1存入(%esp)中。即%esp栈顶永远是有效数据。



Boot loader的最后一条指令地址为0x7d6b，在这里设置断点。运行到这里之后，再输入si，就单步执行call \*0x10018，接下来就跳转到了0x10000c，进入kernel，内核的第一条指令是movw $0x1234, 0x472。



至此，就回答完了上面的四个问题。

#### Loading the Kernel

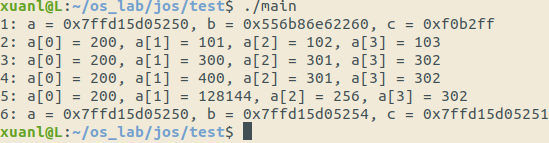
这里一开始的Excercise 4 要求阅读C语言教材的指针部分。本来感觉自己会了，但看这里说除非已经精通C语言，否则不要跳过这个阅读材料。Trust us; you don't want to find out what "the hard way" is.搞的挺吓人的，还是看一看吧。

书上也没有太多新的东西。然后看pointers.c，有些值得注意的地方。

首先是打印指针应用%p，这是按16进制打印。不能用%x，特别是在64位下结果可能不同。（操作系统实验里其实遇到了，lab3调试的时候）

其次这里面有个奇怪的写法：3[c]=302; 这条语句应该等价于c[3]=302，可见[]操作符的本质是把基址和变址部分相加，然后加上\*操作符。所以3[c]=\*(3+c)=\*(c+3)=c[3]。

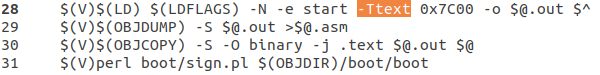
最后是c = (int \*) ((char \*) a + 1);这条语句，这样c指向的其实不是数组a的一个实际的元素了。但不清楚会不会有因为int类型而对齐之类的情况？



从实际运行的结果来看，并不会因为c是int \*而对其值进行四舍五入，也不会在\*c的时候进行对齐。所以，不用考虑对齐问题，照常分析即可。

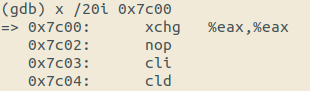
#### Exercise 5

接下来，根据Exercise 5的要求，需要修改这里的首地址0x7c00，观察会如何。我估计应该跳转到0x7c00之后，执行的都是些错误的指令？也不对，可能只有那些地址之类的有问题，但第一条指令还是在0x7c00？这里试试看改成0x7c01。



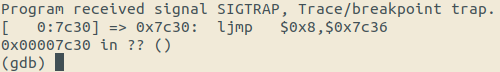
先make clean，然后obj目录整个就没了。之后再make，就又出来了。

从0x7c00开始，多了两条指令？实际的boot程序是从0x7c03开始的。前面这两个要么是随机生成的，不清楚是干啥用的。但从0x7c03开始应该还是可以正常执行。



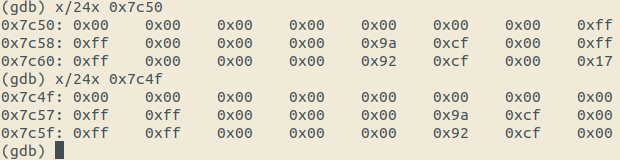
经过测试，可能前面的这些是用于后面.p2align 2对齐用的，在前面添加的一些无用指令？因为如果把0x7c00改成0x7c08，其实好像没有什么关系。

在用gdb调试的时候，用了display命令，这个其实是用来自动显示的，使用了之后每次中断都会把在list中的所有信息显示一遍。可以用delete display清空list。



在ljmp这个地方受到异常信号，与此同时另一边qemu模拟器发出Triple fault。

经过检查，发现是GDT出现了问题。反汇编指令表明，ldgt装载的是gdtdesc地址的内容到gdtr中，这没问题。关键是gdtdesc的高32位，地址为0x7c50，但是实际装载到内存中的位置却是0x7c4f。通过x/24x观察可以得出这个结论。



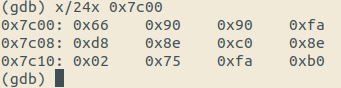
因此，在保护模式下通过GDT访问code段的信息就会出错。因为我修改了1位，所以这里相差了一个字节。那么是否改成8的话，就正好没事了呢？

这里可能得好好理解.p2align这个指令的作用？文档如下

（[https://sourceware.org/binutils/docs/as/P2align.html#P2align](https://sourceware.org/binutils/docs/as/P2align.html" \l "P2align)）

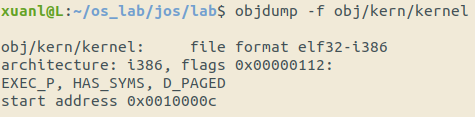
可见.p2align的作用就是将location counter向上对齐。

再检查0x7c00，发现这和反汇编文件中0x7c01位置开始的内容相同。由此可见，程序被装载到了内存的0x7c00位置（这是BIOS做的），但是程序里的各种偏移、地址等都是基于程序被装载到0x7c01位置这个前提来设置的。所以本来GDT是按照双字对齐的，装载完成之后就差了一个字节！

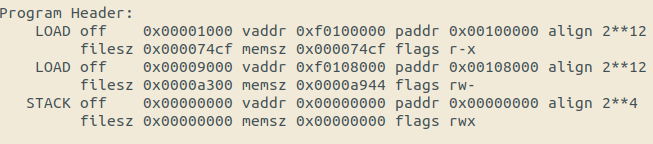




总之，对于这里来说，-Ttext后面接的就是所谓的VMA，也就是整个程序里各种地址都是相对于这个VMA的偏移。而实际装载到内存中的位置总是0x7c00，这就是LMA，这是BIOS完成的。对于这个程序来说，两者不等的话就会造成非常严重的后果！



使用objdump -f 查看程序的入口地址，这和之前用gdb调试的结果是一致的。

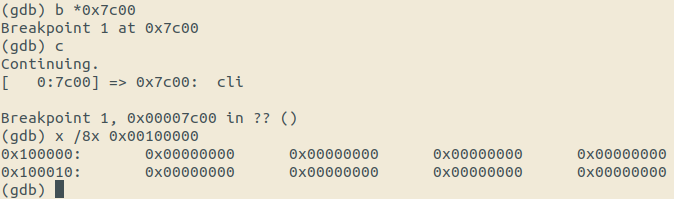


这里还能看见各个段装载的信息。

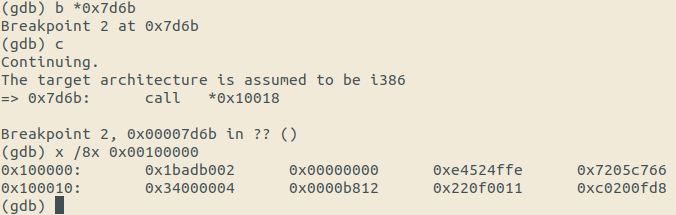
#### Exercise 6

要用gdb检查地址0x00100000的内容。前后肯定是不同的，但不是因为ELF头部装载在了这个地方，而是程序的数据段或是什么段装载在了这里。

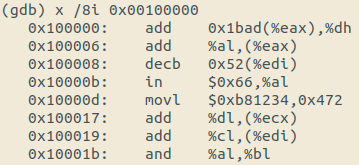
首先是刚离开BIOS，进入boot程序后这个位置的内容。



接下来是离开boot，进入kernel时这个位置的内容。



后面应该是从0x0010000c开始是kernel的代码，但前面这些不知道是干什么用的，直接反汇编似乎也不对（因为这样直接跳过了0x0010000c了）



### Part3

#### 开启虚拟地址

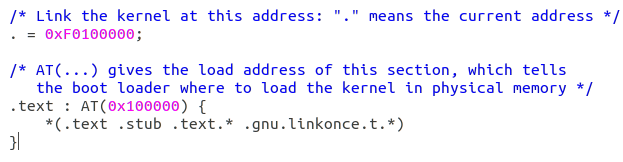
Kern/kernel.ld文件里是链接相关的脚本代码，要看懂的话，可以使用info ld查看ld相关的信息。也可以看官网（<https://sourceware.org/binutils/docs-2.33.1/ld/index.html>）

.表示location counter，即当前在内存中的地址。一般定义一个段的时候，就是把段的LMA设置为当前的位置计数器.，然后计数器自增，增加的大小为这一段的大小。

.text:{\*(.text)}表示，定义输出目标文件中有一个.text段，其内容为所有输入文件中的所有.text段。

Location counter是会自动对齐的。

第一条被执行的指令被称为入口点（entry point），使用ENTRY(symbol)设置。

根据注释来看，使用AT(...)就可以设置LMA，此时VMA依然是位置计数器所在的值，但LMA已经被修改了。

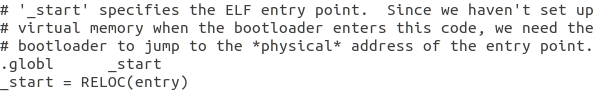
总的来说，这个脚本还是比较好看懂的。其中定义了一些变量，不知道是具体什么用途。

内核的虚拟地址是从0xf0100000开始的，但实际很多计算机没有这么大的内存，所以实际装载是从0x00100000开始。接下来，就需要一个虚拟内存到物理内存的转换了。

关于页表的配置，在kern/entrypgdir.c中。（别看1000多行，其实关键的就几行）

Kernel一开始先执行entry.S中的汇编代码，首先需要配置页表，建立映射关系。

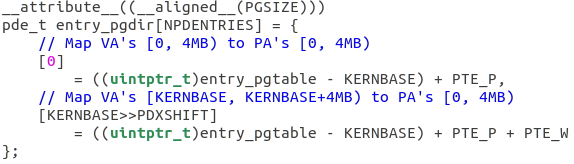
一些常量都定义在mmu.h和memlayout.h中，例如KERNBASE的值为0xf0000000。



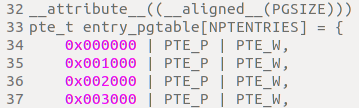
上面这个\_start表示的是kernel开始的物理地址，在kernel.ld文件中的入口地址就配置成了\_start而非\_entry，因为后者的值为0xf0100000。

这一部分还是比较好懂的，就是把页表装载到寄存器%cr3当中。但这个页表具体怎么配置的，还是得仔细把C文件和头文件看一遍，才能弄明白。

下面这个是页表目录（也就是一级页表），这里可以看到，它本身是一个数组，然后这种声明方式可以初始化特定的范围，而其余范围初始值都是0。这个地方，所有的一级页表项都指向同一个二级页表（entry\_pgtable - KERNBASE 就是这个唯一的二级页表的物理地址）。0xf0000000以上的4MB能够映射到从0开始的空间，其实主要是靠这个一级页表。

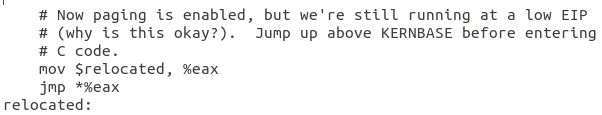


而二级页表也是一个数组，这里第一项表示虚拟地址的页号为0时，则其物理地址中的页号也为0。这个地方相当于没做变换，一模一样，所以说这里二级页表没有起作用。

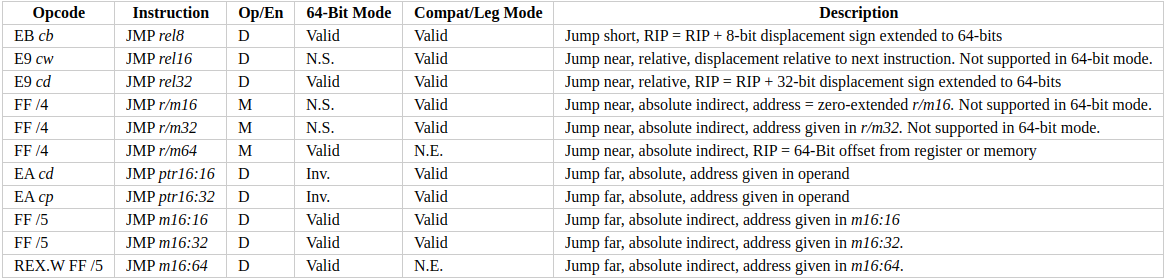


PTE\_P表示present，即在内存中，PTE\_W表示可以写。

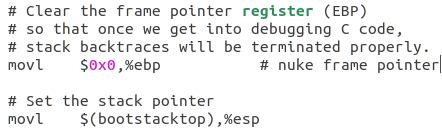
下面之所以配置完页表后还能在低地址运行，是因为低4MB也被映射到物理地址同样的位置了。



这个地方的jmp使用寄存器寻址而不是直接用立即数，因为立即数在汇编之后的机器指令中，是相对寻址方式，这可以参考手册上关于jmp的描述。

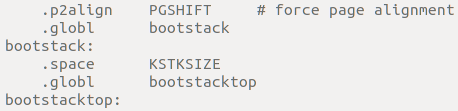


使用32位表示地址的话，相对寻址跳转范围是±2GB，但这里要从地址1MB跳转到和4GB只差4MB的地址，超过了立即数跳转的范围！所以这里只能使用寄存器来jmp。



这里配置了栈帧，其实栈帧在程序运行过程中不是必要的，但对于调试来说是必要的，通过栈帧可以获得函数递归调用的层级关系。

之后配置栈指针%esp。

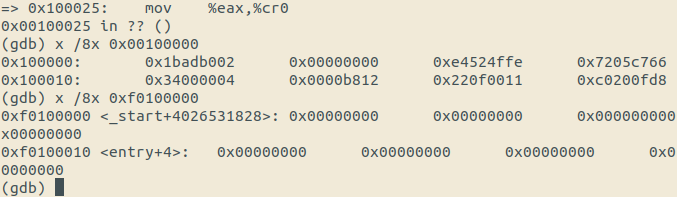


这里KSTKSIZE就是内核的栈空间大小，根据头文件可知，其大小为8×PGSIZE，所以内核栈大小为8页，即32KB。

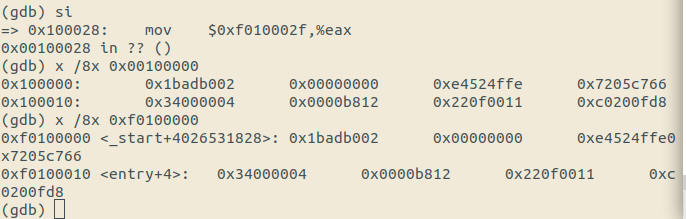
总之，entry.S的主要工作就是配置页表，建立虚拟地址到物理地址的映射关系。

Exercise 7

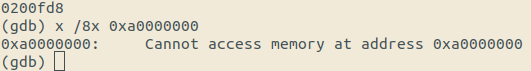
在mov %eax, %cr0指令前停下，查看内存数据。可见0x00100000是有数据的，但0xf0100000则都是0，也就是说这两个内存中的数据是不同的。



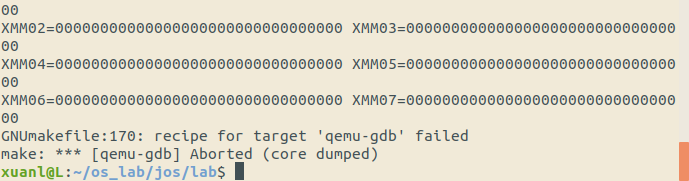
而执行完这条指令之后，PG位置1，开始使用页表。此时两个地址内的数据就完全一样了。这里开启页表后，使用gdb查看0xf0100000这个地址时，会进行虚拟地址到物理地址的映射，所以gdb本质上查看的是物理地址0x00100000的内容。



这里可以尝试用gdb查看其他地方（0xa0000000）的数据，显示无法访问。可见gdb给出的是虚拟地址，当虚拟地址在页表项中没有配置时（全0），无法完成映射。



如果没有开启页表的话，那么之后的jmp指令应该会出问题。Jmp跳转到物理地址0xf0100000，这里其实没有任何指令，所以会出错。



执行完jmp之后，$pc的值变成了0xf010002c，而这个地址内的值为0x000000。接下来继续si一步，qemu报错退出。（此时终端回显消失，需要重启终端，因为似乎set -v on没用？)

#### 格式化输出printf的实现

这一部分首先要通读 kern/printf.c, lib/printfmt.c, 和 kern/console.c这三个源文件。

为了理解，必须要能够知道变参函数的使用，即va\_list一定要熟练掌握。

Kern/printf.c比较简单，给出的cprintf就是打印到终端上。

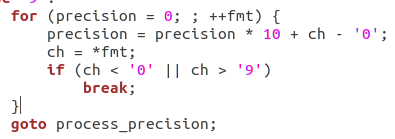
Kern/printfmt.c就比较复杂了，而且这里第一次需要填写代码了。

格式描述符（format specifier）范式如下： %[flags][width][.precision][length]specifier

这个网页上介绍地挺详细（<https://www.cnblogs.com/oddcat/articles/9773312.html>）

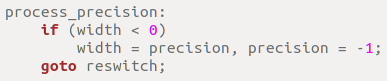
根据这上面的标准来看，对于flags字段，实验中只考虑了’-’（左对齐）和’0’（用0填充）这两种情况，还有’+’,(space)等情况没有考虑（‘#’放在后面考虑了，与altflag相关）。

对于width字段，考虑了所有的两种情况。其一是(number)，即设置宽度。其二是\*，即宽度的设置在之后的va\_list中。



这一段代码逐个读取数字并计算打印的宽度（精度），这里注意fmt指针指向的是下一个字符，所以大部分时候都领先于ch。当ch = \*fmt的时候，其实获取的是下一个字符。

其次要注意，这里最后是goto process\_precision，而不是goto reswitch。因为读入的数字既有可能是精度，也有可能是宽度，要根据width来判断。

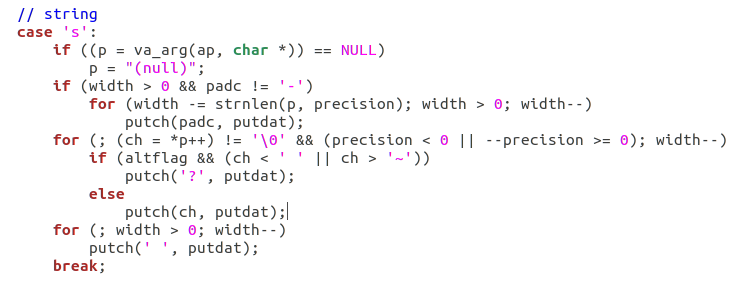


Lflag表示输出的位宽，即输出16位(short)还是32位(long)还是64位(long long)，具体怎么转换可以看getuint和getint，其实就是lflag>=2的时候都算long long。

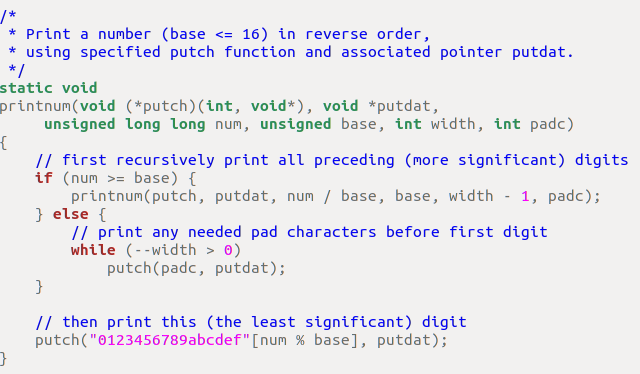
对于%c，直接用putch输出字符即可。

对于%e，这其实按照规范来说是输出科学计数法浮点数，但实验中用于输出错误信息。

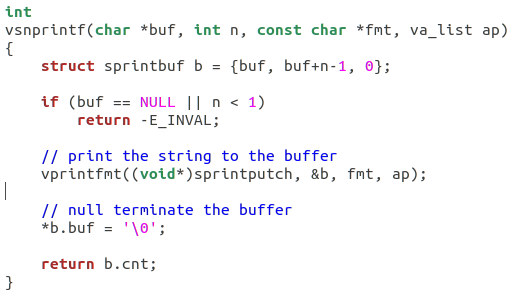
对于%s，有不少要注意的地方。首先，如果有’-’则左对齐，否则先根据width和字符串长度打印填充字符。接着，对于%w.ps这样的格式来说，w表示宽度，而p表示精度。具体来说，p表示最多能打印多少个字符，超过p的部分不打印。而w表示少于w的部分用字符填充。还有一点是，如果%后面有#的话，那么字符串中的不可打印字符都以’?’形式打印。不可显示字符为空格之前的字符和’~’之后的字符。



Printnum()函数上面的注释写的是reverse order，其实这个意思我感觉是“小端”的reverse，所以就是大端方式输出，因此是没有问题的。



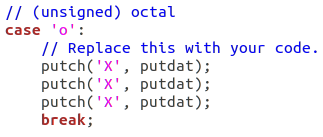
注意到上面这些函数的第一个参数都是一个与字符输出相关的函数，类型为void (\*putch)(int, void \*)。这个函数的目的就是为了满足不同输出目的的需要。虽然大多数情况的输出都是到终端，但例如输出到一个字符串缓冲区中这样的情况，就得修改putch了。这里putch的第一个参数其实就是输出的字符（虽然是int类型），而第二个参数对于输出到终端的情况仅仅是一个计数器（似乎没用），对于输出到字符串缓冲区的情况则提供了缓冲区的地址。



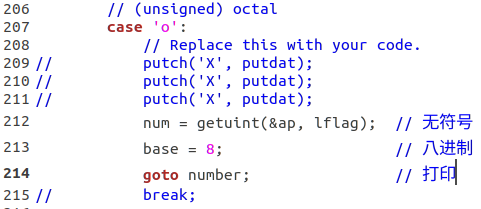
因此上面这段代码对于理解第一个参数和第二个参数的含义非常重要。

#### Exercise 8

Octal部分需要修改。



我感觉仿照hexadecimal的写法就行了？



接下来继续阅读kern/console.c，这个内容更加多。

这个里面主要分为4块，分别是Serial I/O、Parallel port output、CGA/VGA display、Keyboard input，最后有一个General divice-independent console部分。

这一部分可以和bochs的硬件端口说明一一对照着看，也可以直接看注释。

顺序上的话，感觉应该先看最后的general部分，因为这里的函数会被之前的函数调用。



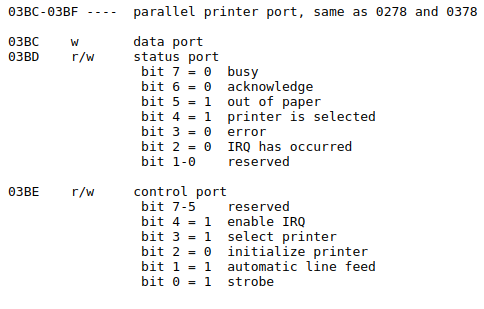
这是之前就看过的x86.h，现在知道为什么这里需要用w来修饰占位符”1”了，因为输入的port是int型，这里用w后就限制了”d”一定是%dx而非%edx，从而将int转为了unsigned short类型。

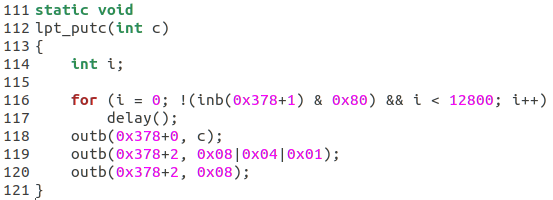
Serial port串行端口

这个输出设备现在好像用的不多了，主要用在操作系统debug中，还有networking中。

串行端口在实验的代码中注释非常详细，所以大概能看懂意思。但要知道每一句语句具体的含义，以及一系列专业术语的话，还是得参考（<https://wiki.osdev.org/Serial_ports>）的内容。

Parallel port 并行端口（0278、0378、03BC这三个并行端口协议是一样的）





对照上面的协议，inb(0x378+1) & (0x80) 是在检查最高位（busy），即设备是否忙。

0x08表示enable IRQ，0x04表示initialize printer，0x01表示automatic line feed。

详细可见（<https://wiki.osdev.org/Parallel_port>）

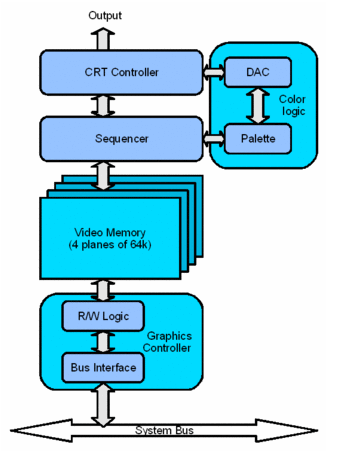
**CGA/VGA 显示**

这一部分是很关键的代码，而且需要增加能显示颜色的代码。

首先要参考相关的资料。（<http://www.osdever.net/FreeVGA/hardovr.htm>）上面提到了一些术语。

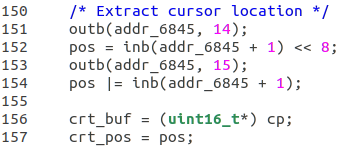
Frame Buffer帧缓冲，用于存放像素信息，是视频硬件设备的重要组成部分。这一块区域通常被映射到内存中的一块区域上。

CRT控制器，阴极射线管控制器。Sequencer从帧缓冲顺序读取数据，转换成像素和颜色信息。CRT控制器使用这些信息生成时序信号。Attribute控制器处理颜色信息，并交给DAC。 DAC（数模转换器）在颜色表中查找颜色，生成模拟信号。（下面这个图便于理解）



VGA内部的寄存器数量超过300，但在I/O地址空间上只划分出了一小部分。因此很多寄存器都是先通过一个端口用索引进行访问，然后在另一个端口输入或读取数据。

实际中经常使用的端口有0x3c4，0x3ce，0x3d4等。实验中的CGA\_BASE就是0x0d4。一般这些端口用于索引，而这些端口号加1用于读写数据。所以使用中这一段就稍微能看懂了。这里先读取14号寄存器的值作为高8位，然后读取15号寄存器的值作为低8位。



这里还提到了控制颜色需要用端口0x3c8，其使用方法为用0x3c8选择索引，然后对0x3c9依次写入三个值（红、绿、蓝）。

内存布局的四种模式：16-color graphics modes、256-color graphics modes、text modes（实验中采用）、4-color modes。

Text mode中只有3个planes被使用。Plane0存放字符编码（ASCII码），plane1存放相应的属性，plane2存放字体数据。

在实验中，缓冲区的类型是uint16，所以低8位是ASCII码，而高8位是属性（颜色之类的）。

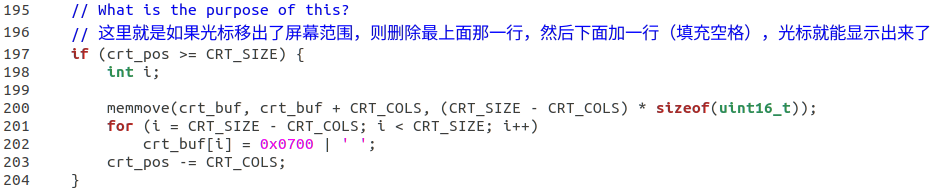
基本上cga\_putc()能看懂了，下面keyboard的部分恐怕和printf没太大关系，而且太过于复杂，可以先放一放。

下面先把几个问题回答一下，然后主要把修改颜色的部分完成。

1. Explain the interface between printf.c and console.c. Specifically, what function does console.c export? How is this function used by printf.c?

Printf.c中使用了console.c中的cputchar()。这个函数被用在printf.c中的putch()函数中。而putch()函数又在vcprintf()函数使用（被当成一个调用时的参数），而vcprintf()被cprintf()函数调用。最后cprintf()就是printf.c向外提供的接口（功能类似标准库的printf()函数）

1. Explain the following from console.c:



1. In the call to cprintf(), to what does fmt point? To what does ap point?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.

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

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

Cprintf()函数中，fmt指向一个常量字符串，即"x %d, y %x, z %d\n"。ap一开始指向的是变量x的首地址。

首先cprintf()会调用vcprintf()，这里传入的两个参数一个就是fmt，另一个是上面提到的ap变量。此时ap仍然指向的是变量x的首地址。

接着打印第一个x，因为就是普通的字符，所以会调用putch函数，而putch最终会调用cons\_putc函数，传入的参数就是x的ASCII码（因为是int型，所以高位补0）。同样，打印空格也会调用putch。函数

只有打印%d的时候，才会先调用va\_arg，此时ap指针一开始指向x的首地址，调用完后指向变量y的首地址（因为这里是%d，所以调用的是va\_arg(\*ap, int)）

1. Run the following code.

unsigned int i = 0x00646c72;

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

这里真有些不知道要做什么。看网上一些前人的做法是，在开机欢迎的代码那里添加这两行代码，然后观察运行结果，这确实是个办法。也不知道有没有更好的办法，因为如果直接写一个文件然后编译肯定不行，这里cprintf本质上最终依赖qemu模拟环境提供的接口。总之，暂时先修改开机处的代码好了。





结果是He110 World，这里其实前面不是Hello而是He110，因为数字1和字母l比较像。

前面的e110其实就是57616的16进制形式。而后面的rld则分别是0x72、0x6c、0x64，也就是变量i的从低位开始的三个字节的数据。因为x86按照小端存储数据，所以才能这么打印。最高位的0x00正好相当于字符串结尾的’\0’。

如果是大端存储的话，需要把变量i的值修改为0x726c6400，而57616则不需要修改。

1. 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);

确实，打印的x=3，而y=后面应该是一个随机数。当检查完第一个%d之后，ap的值就指向第二个参数之后的位置的地址了。这个地址的值归上一层函数栈帧使用，因此是一个任意的数。因为cprintf是变参的，所以编译阶段可能无法检测出来这里少了参数。

1. Let's say that GCC changed its calling convention so that it pushed arguments on the stack in declaration order, so that the last argument is pushed last. How would you have to change cprintf or its interface so that it would still be possible to pass it a variable number of arguments?

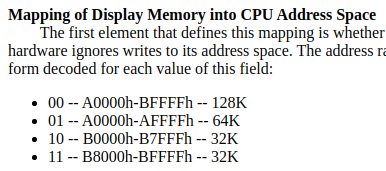
如果其他的接口，例如va\_list的功能没有变的话，那修改cprintf函数的接口可能是比较简单的方法（也没想出别的好办法）。可能得修改成 int cprintf((void \*) a, ...);

参数的顺序全部颠倒过来，最后是fmt？

例如cprintf(y, x, “x = %d, y = %d\n”); 大概是这样的形式。

Challenge

增强console的功能，能够用不同的颜色打印。参考（<http://rrbrandt.dee.ufcg.edu.br/en/docs/ansi/>）反正感觉不简单，首先就是不清楚这是需要修改哪个文件？感觉修改printf.c肯定不行，可能是修改printfmt.c，但console.c里面和颜色相关的内容有点没懂，主要是高8位和颜色怎么对应，这个资料不知道在哪里。

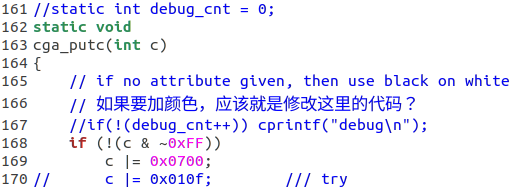


实验中的地址是从B8000开始的，这可能有什么关联。

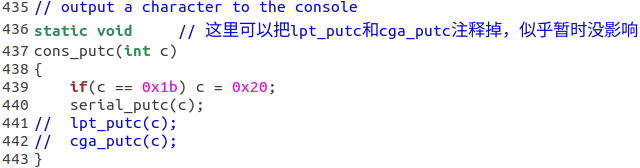
CPU访问VGA的display memory有四种模式，但一些细节还是不太清楚。

。。。

实在没有头绪，我尝试修改下面这一段的代码，发现怎么改都没有任何变化



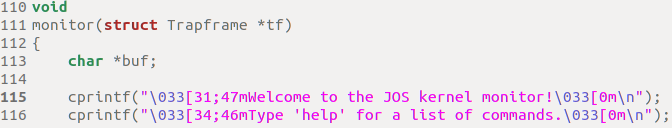
再回到上一层调用，我把CGA和并行接口都注释掉，发现结果依然没有变化！

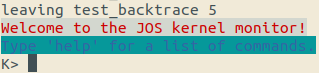


这里就有些不清楚后面两个函数具体发挥了怎样的作用了，但总之打印字符似乎仅需要serial\_putc函数就可以了。

本来以为这里需要自己写什么代码，最后实在不会，上网查了之后发现其实目前已经支持修改颜色了。只需要自己在monitor.c里面添加一些escape sequence就行了。

那这个challenge我实在有些摸不着头脑了，在这里浪费了很多时间，看了很多VGA的手册。其实它的意思好像是，如果想挑战自己的话，可以尝试把VGA改成graphic模式（现在是text模式），我感觉可能是这个意思。





总之，这个challenge其实不需要修改代码。但是，我发现网上的一些解释其实是错误的。

这里之所以能够解析颜色转义序列，并不是因为cga\_putc函数，而是因为serial\_putc函数。这是通过实验测试得出的结果。

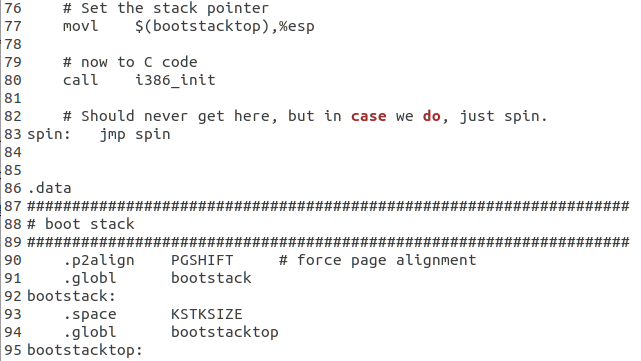
### Stack

终于到了lab1的最后一部分了。

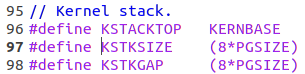
#### Exercise 9

Exercise 9. Determine where the kernel initializes its stack, and exactly where in memory its stack is located. How does the kernel reserve space for its stack? And at which "end" of this reserved area is the stack pointer initialized to point to?

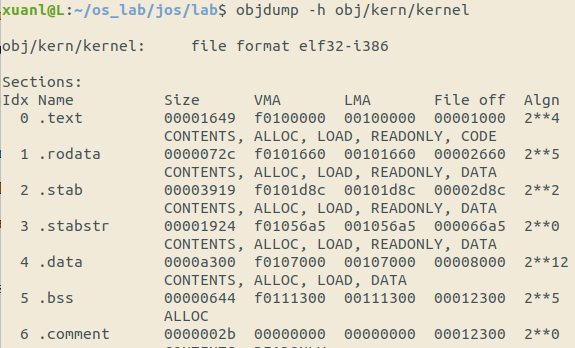
这个其实在之前已经有了，就是在entry.S中的最后部分。



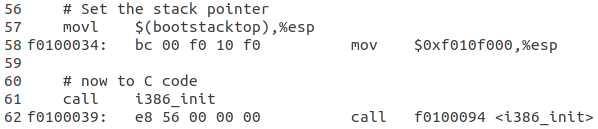
这里77行修改了%esp的值，建立了栈。最底下可以看到，这个栈就是data段，在memlayout.h中可以看到内核栈大小为8页（8\*4K）。这里用了.space 预留了栈的空间，空间大小为KSTKSIZE。



根据之前的objdump查看到的目标文件的信息，.data段的虚拟地址为0xf0107000。所以，内核栈开始的虚拟地址也就是这里。栈大小8\*4K=0x8000，加上去后得到栈顶为0xf010f000。

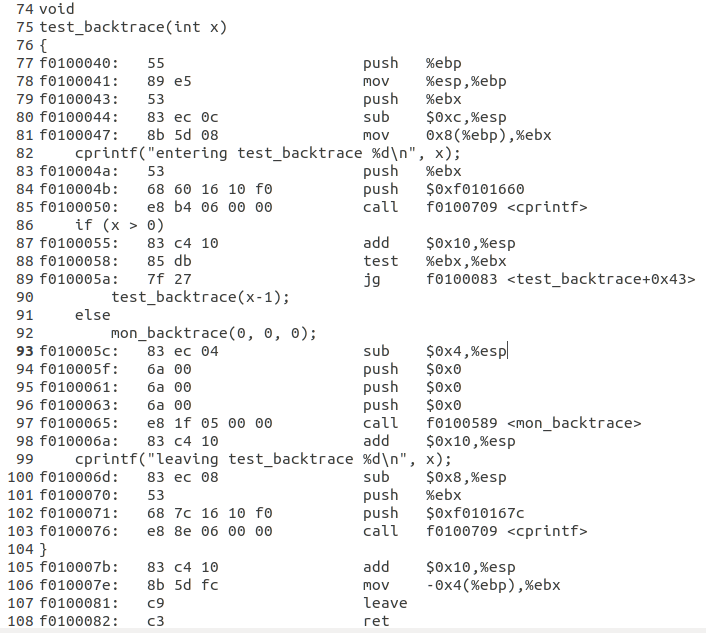


下面检查上述推理是否正确，查看objdump得到的反汇编文件，发现是0xf010f000。



#### Exercise 10

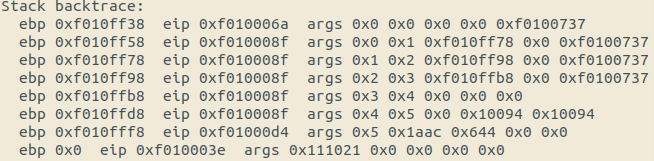
查看反汇编文件，根据%esp的变化可以发现，每个test\_traceback函数占用的栈空间为32B（8个双字，分别用于保存返回地址，保存栈帧%ebp，保存%ebx以存放变量x，保存函数调用所需的参数，这里因为调用了cprintf，需要两个参数，而mon\_backtrace需要3个参数，因此最多总共要6个双字，为了对齐就占用了8个双字）



x, %eip, %ebp, %ebx, (), (), 0, 0, 0, %eip, ...

#### Exercise 11

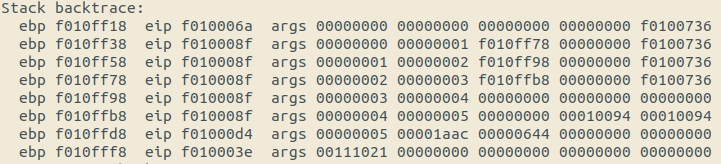


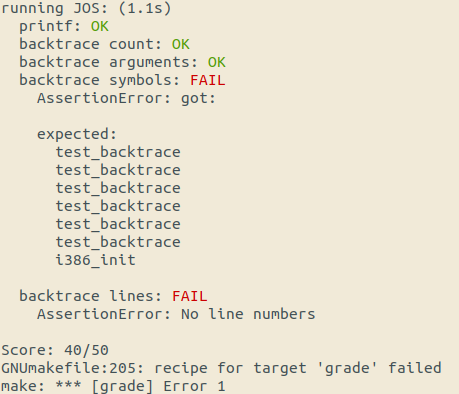


最后多的这一样不知道应不应该打印？以及感觉不能用%p，而应该用%8x。修改一下。此外，这里第一个打印的也不是当前的ebp，而是\*ebp，也不对。



修改后结果应该对了



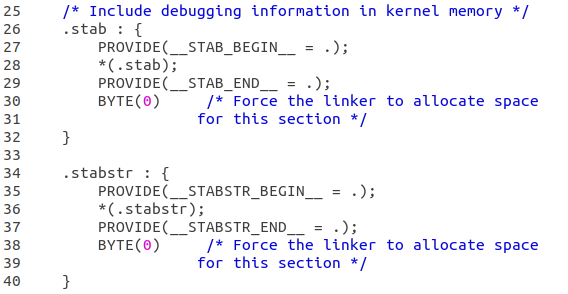


这里没有打印函数名，所以接下来的Exercise 12内完成这个内容。

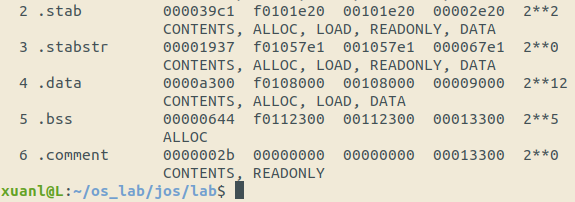
#### Exercise 12

最后这个练习，首先按照要求要完善debuginfo\_eip的功能。

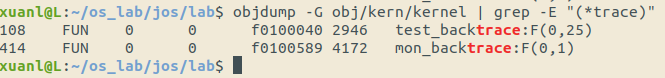
先再回去看kernel.ld文件，之前不明所以的PROVIDE现在就明白了，其实就是给kdebug提供\_\_STAB\_\*这一系列全局变量，方便获取符号表中的数据。



使用objdump -h 可以看到，.stab符号表段和.stabstr段都会被装载到内存中。



接下来使用objdump -G 查看符号表的内容。这里内容很多，可以用grep搜索一下test\_traceback和mon\_backtrace，发现可以找到这两个符号。这里-E表示扩展正则表达式。



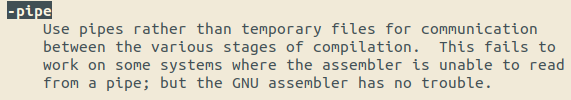
此外，这里要注意，不能只看函数。整体来看，先出现的是源文件的文件名符号（类型S0），然后紧随其后的是这个源文件里出现的符号，而下一个源文件中的符号必然在那个S0类型的符号之后！只有明白了这一点，才能看懂后面的kdebug.c中的查询函数。

最后在终端中输入gcc -pipe -nostdinc -O2 -fno-builtin -I. -MD -Wall -Wno-format -DJOS\_KERNEL -gstabs -c -S kern/init.c

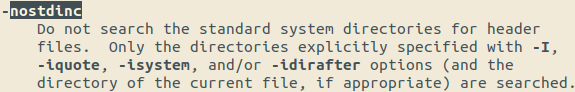
在init.s文件中找到相关的符号。

上面gcc的一些选项可以查看man

-pipe：使用管道而不是临时文件来传输汇编过程中的临时数据。

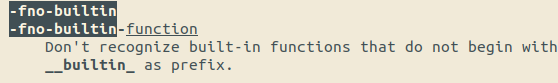


-nostdinc：应该是no standard include，不要搜索操作系统提供的C标准库头文件

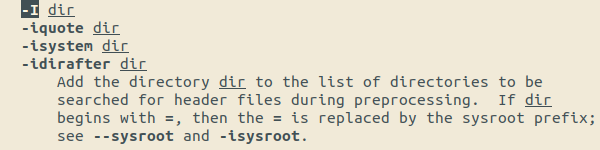


-O2：优化

-fno-builtin：不对一些builtin函数进行优化



-I：提供寻找头文件的目录（但上面的命令中似乎没提供？）

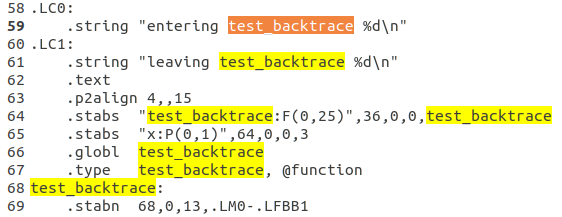


-MD：在man中未找到，但这个似乎挺关键的。这个参数用于不是输出预编译过程的结果，而是输出一个用于make的规则，该规则描述了这个源文件的依赖关系。预编译器输出的这个make规则包含名字与原文件相同的目标文件，冒号和所有include文件的名字。

-Wall：显示所有警告

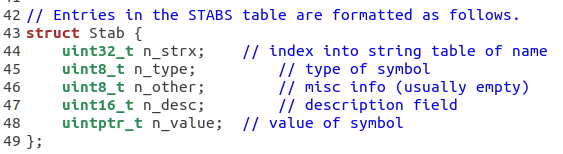
-Wno-format：未找到

-gstabs：首先-g表示增加调试信息，这里stab应该就是符号表。具体这个选项的含义man中也没有，还是得去官网看。注：表示不生成行号（desc项全0）



接下来就开始完成实验了。

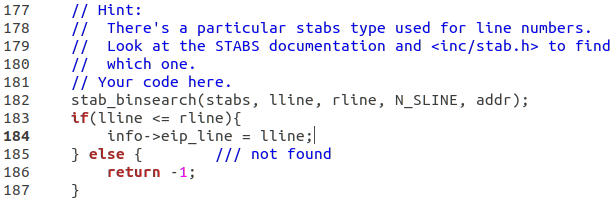
先看stab.h头文件，知道struct Stab的声明



然后找到line number对应的宏定义，注意应该是text segment的line number



这里要补充代码，选择N\_SLINE，然后addr其实应该是相对偏移（可以看-G生成的信息），但上面已经修改好了，所以这里直接传addr应该就可以了。

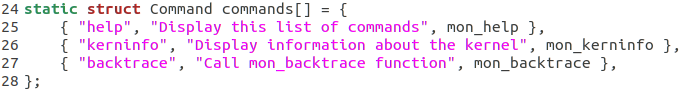


根据这里的代码可以知道，符号表中的PSYM类型的符号其实就是变量，因此可以使用这个信息来获取函数的参数个数！

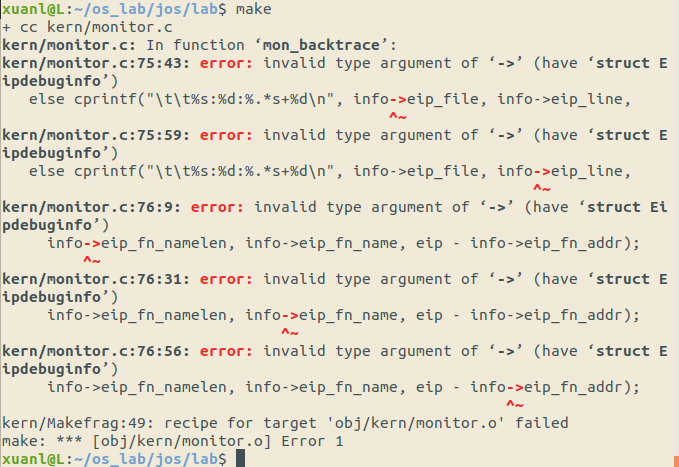
Monitor函数通过调用readline函数，来将命令读入到内核的buf中。然后，调用runcmd函数来解析buf中的字符串。

这里可以看到，其实argv字符串数组，指向的是内存中同一个字符串（即buf缓冲区）上的不同位置而已。在解析的过程中，把空白符都变成了’\0’。

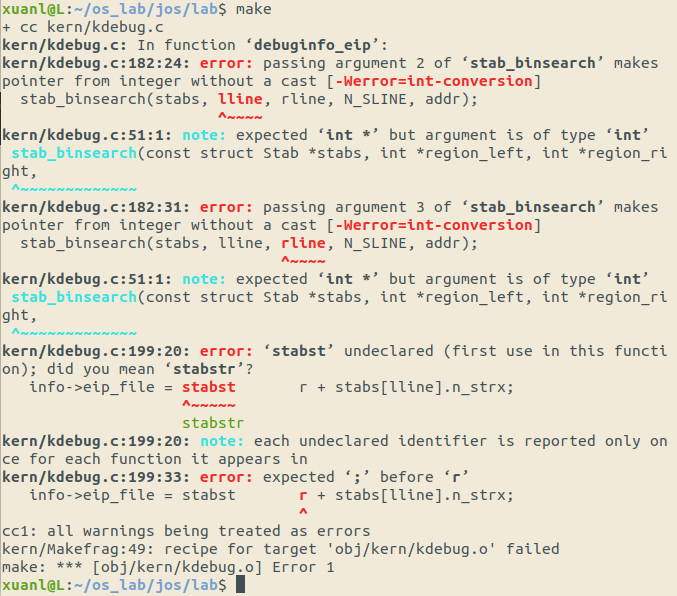
所有的命令都以结构体的方式存在一个数组commands中。这里要添加一个新的命令，也就是backtrace。



尝试编译



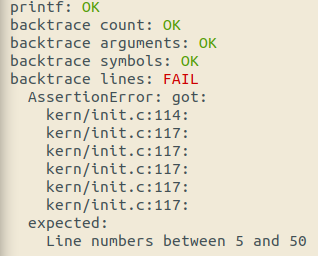
这个应该用.操作符，或者把info换成指针类型应该就行了，但后者要额外分配内存，比较麻烦。



修改好了，但还有一堆问题。这里忘记加取地址符号了。

这是最终版本





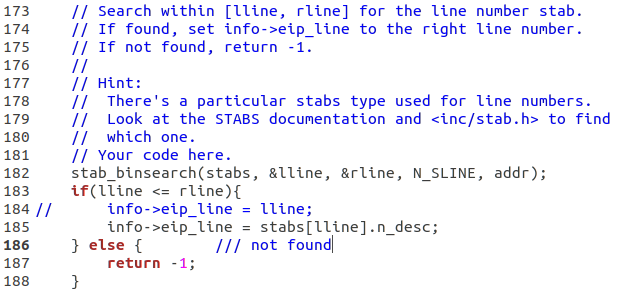
还有最后的问题，行号不对。这可能是调用二分查找的地方有问题吗？

是的，这里本来是直接Info->eip\_line = lline，但是这个lline根本不是在源文件里的行号，而是在符号文件中的索引号！

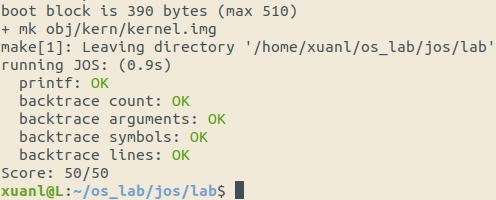
正确的写法应该是info->eip\_line = stabs[lline].n\_desc，这里的desc就是源文件语句行号！

此外，之前的gcc编译选项中，出现过-gstabs选项，其实这个就表示不生成行号！而-gstabs+

表示生成行号。（<https://blog.csdn.net/iteye_14514/article/details/81684249?fps=1&locationNum=2>）



最后使用make grade检查，全部通过！



到此为止，lab1终于全部完成了。