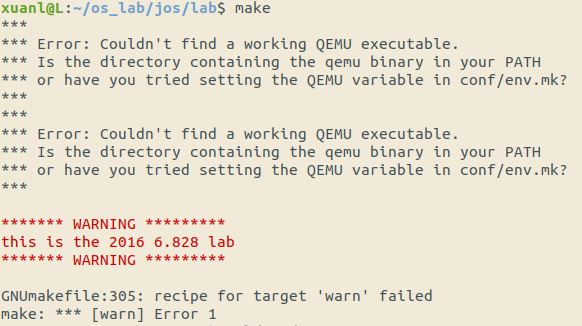
## Lab1

### Part1

注意.gitignore中忽略了/obj

Make失败，需要安装QEMU（是指Quick Emulator吗？）

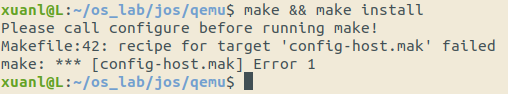


git clone http://web.mit.edu/ccutler/www/qemu.git -b 6.828-2.3.0

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

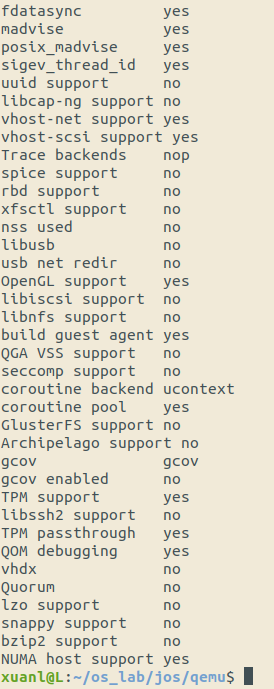
（注意：即使不配置安装路径，也要运行./configure）



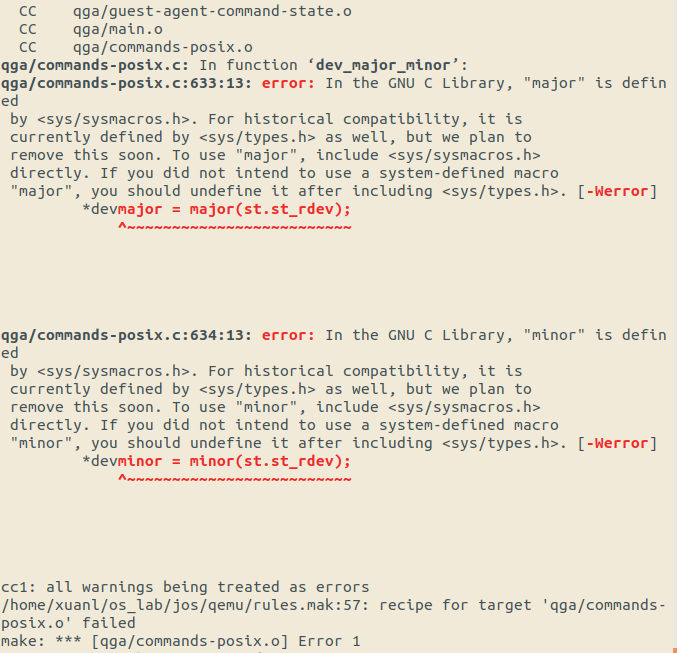
报错：

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。

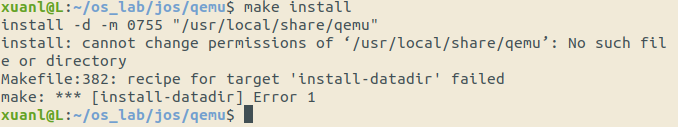


去看了18年的，发现configure的时候加上了--disable-werror，

确实忽略了警告，不会fail了

Make就整了好久

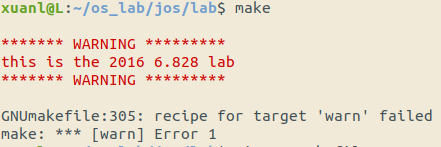
Make file又出问题



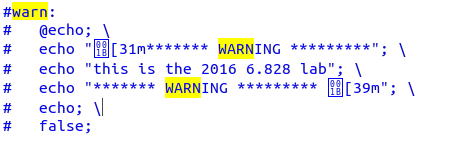
这是权限问题么？

切成root，就可以了。（所以上图中就是安装路径，顺便记下来了）

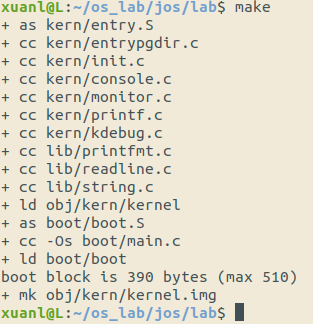
Lab1的make还是有问题，居然因为时2016年的就不让用了



手动修改makefile文件，把所有warn的部分注释掉

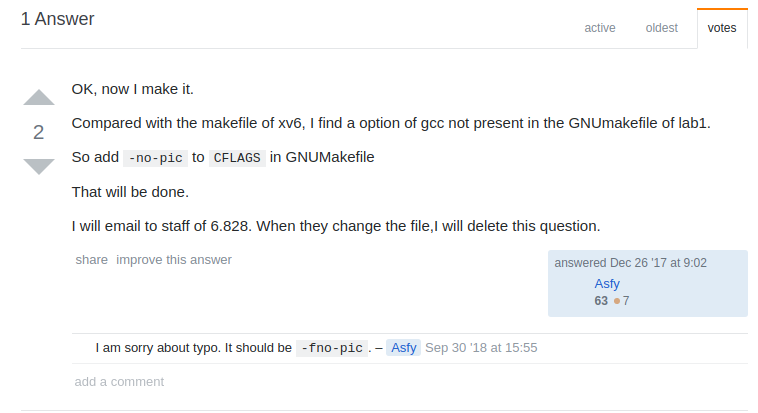
、

然后make，终于成功了！



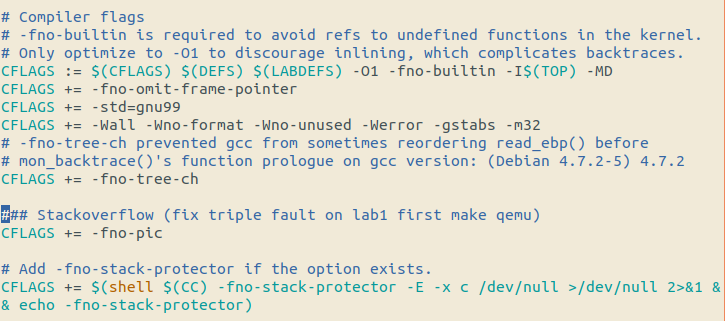
但是，继续make qemu遇到了大问题，出现了Triple fault，然后一直卡死了。

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。

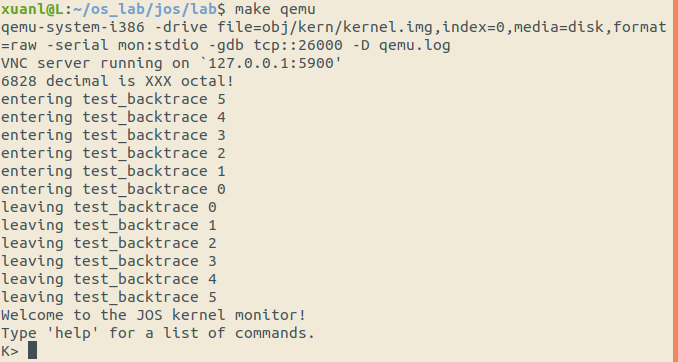


在makefile中的CFLAGS加上一行，CLAFGS += -fno-pic

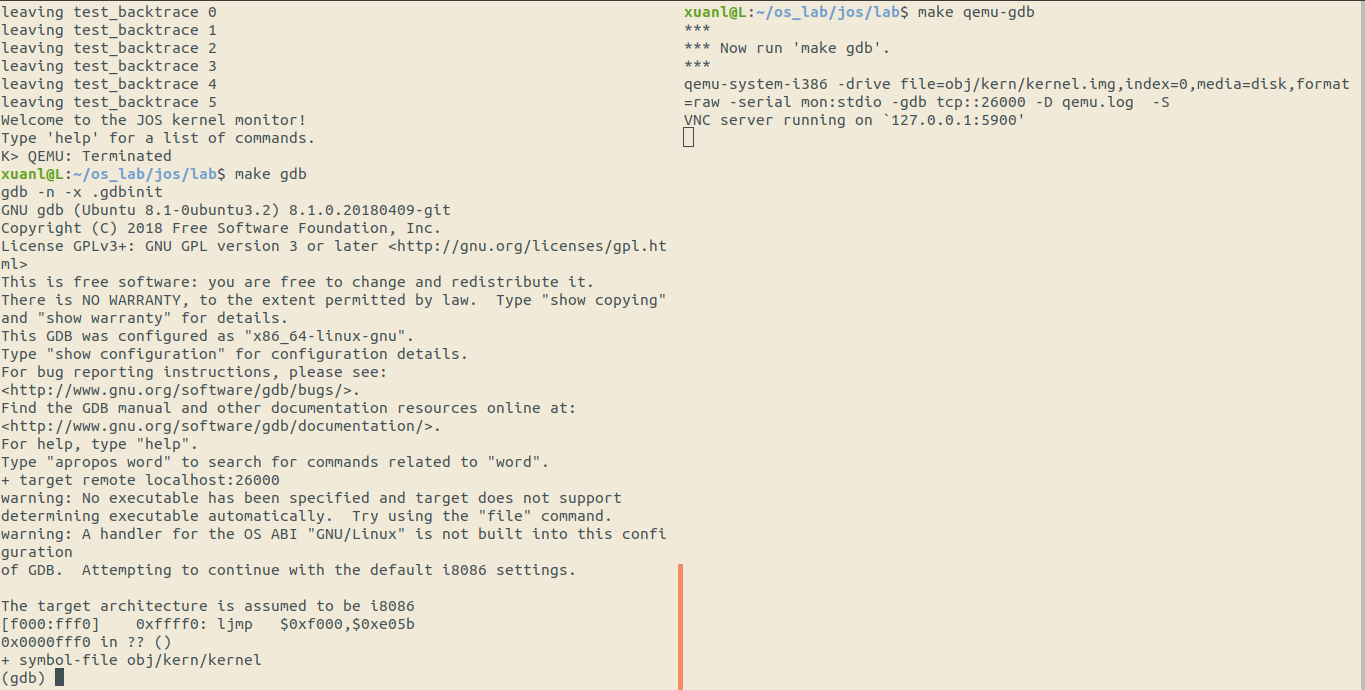
这个地方的pic表示position-independent code位置无关代码，用于生成动态链接的代码，那这里用-fno-pic，应该就是生成的代码里使用绝对地址而不是相对地址。但为什么必须要这样呢？



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）

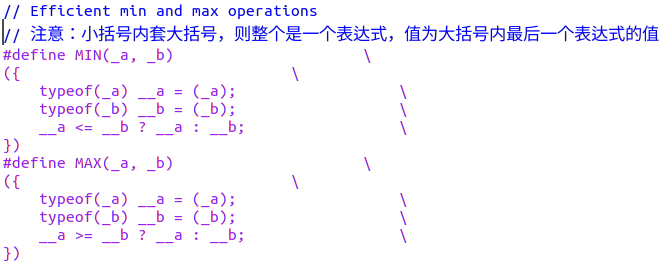
### Part2

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++;});

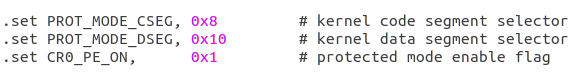


Elf.h里面主要是ELF头部数据结构，以及程序头表，和节头表，这三个数据结构。

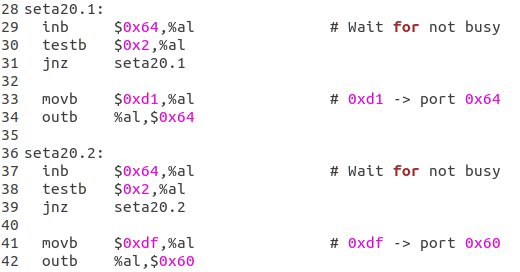
最后是一些常量宏定义

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

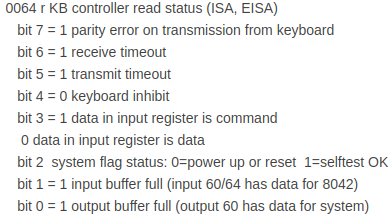


0x64号端口是什么呢？是键盘的状态寄存器。



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

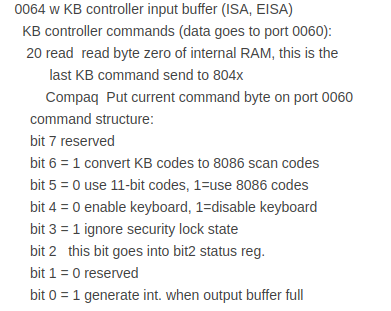
其实可以看bochs的资料，也可以（<http://bochs.sourceforge.net/techspec/PORTS.LST>）



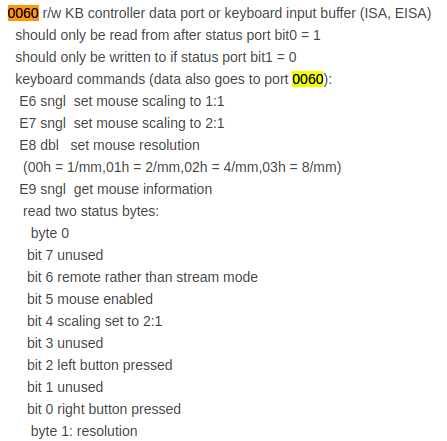
按照这个意思，如果testb $0x2, %al 之后标志位为0，说明input buffer为空，否则为满。那这一段代码就是等input buffer为空的时候继续。

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。

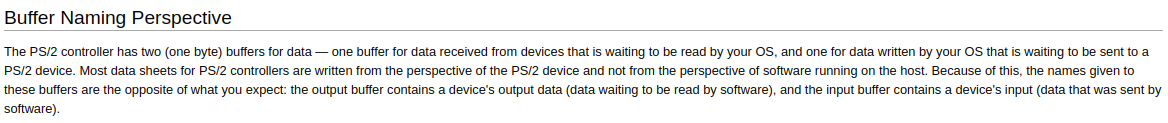


虽然还是不太明白，这个地方实在不知道要看什么了。

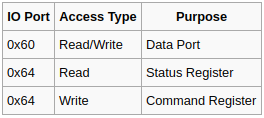
找到个这个，专门讲键盘鼠标部分的，不知道能不能看懂。（[https://wiki.osdev.org/%228042%22\_PS/2\_Controller](https://wiki.osdev.org/"8042"_PS/2_Controller)）

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

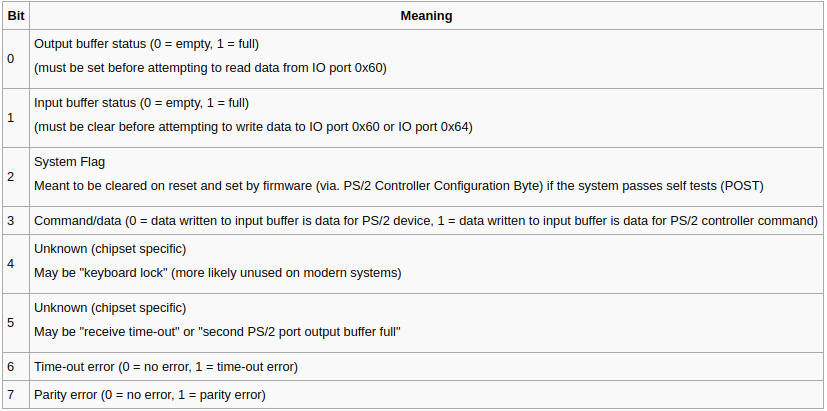


上面这一段提到了命名的问题。端口的命名是站在硬件的视角来的，因此output buffer其实是键盘输出的数据，对于软件来说需要用in指令来读取。反过来，input buffer则是软件输出，对于硬件是输入。



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。

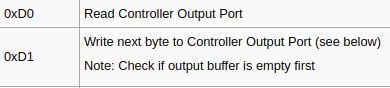
同一个端口地址的读取和写入，可能在实际的物理硬件内部对应的是不同的硬件寄存器。



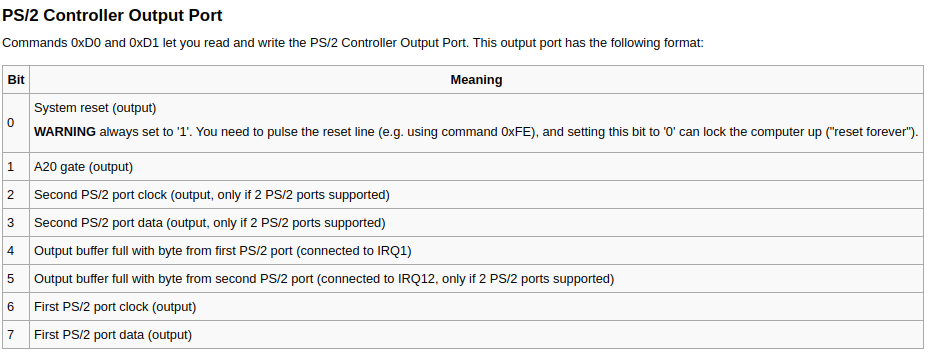
从这个表可以看出，首先，任何需要写数据的操作（写入0x60/0x64），都需要先等待0x64的1号比特位为0时才能写入（可见这一位为1，表示当前设备正在被其他什么东西写入，即写繁忙状态）。但之后写入0xd1到0x64寄存器，这一步感觉还是不能看这个表。继续往下看看。

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。



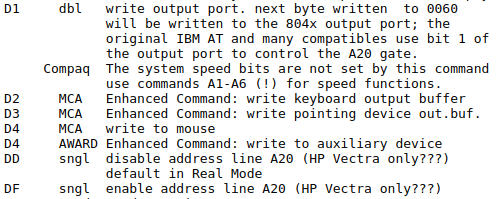
可见，0xd1送到0x64之后，外设此时会等待next byte送入0x60。



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

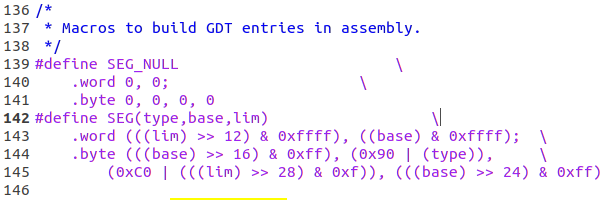
lgdt    gdtdesc

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。

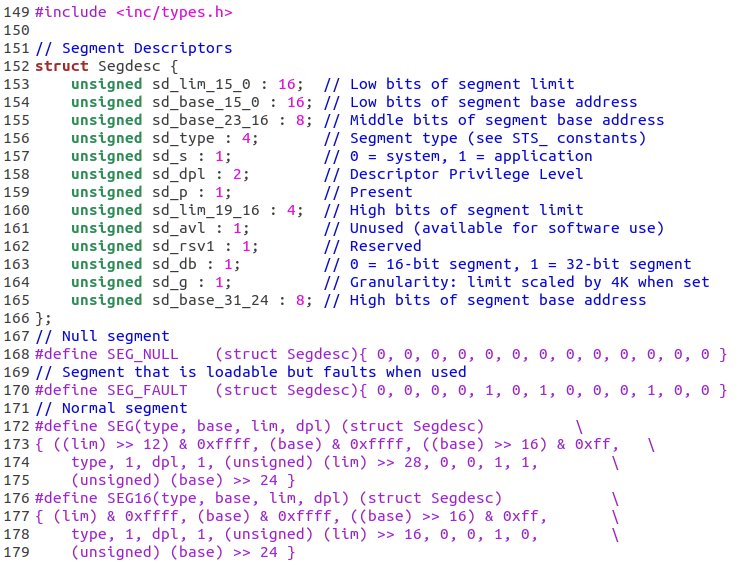
具体这个指令什么意思，可以参考（<https://www.felixcloutier.com/x86/index.html>）这个网站上的，上面各种x86指令都挺全的，而且比较新（19年5月更新）

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



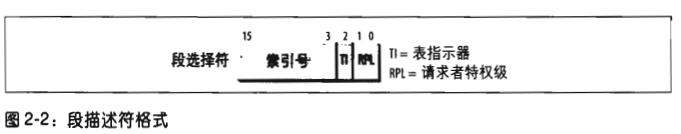
为了更好地理解汇编的写法，我觉得还是得参考下面的C语言的写法。注意这里用到了C语言位域的特性，即struct中成员名后面冒号加位数，可以节约空间。



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。

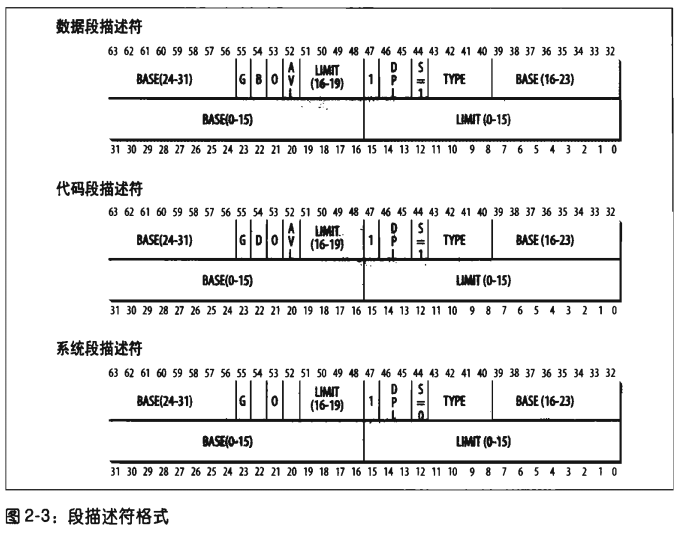


段选择符的高位是索引号，也就是“段描述符”在描述符表中的偏移，通过索引才能找到段描述符。

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。

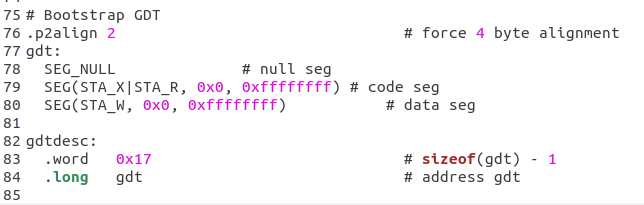
当sd\_g被置为1时，则limit的单位是页（即4K），所以此时段大小最大为4G。



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。

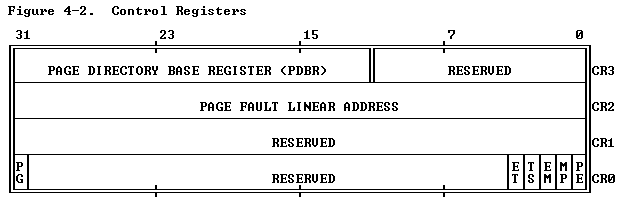
对于普通段，sd\_s=1,sd\_p=1,sd\_db=1,sd\_g=1，保留位都是0，其余需要根据传入的参数来设置。

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。

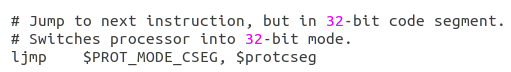


由上图可见，实际操作系统中的GDT里面，首先有一个空段（不清楚干啥的），然后是一个代码段，以及一个数据段，两个段的地址都是0，而且limit都是4G，除了保护作用外其实相当于没用到太多分段的功能，此时虚拟地址与物理地址实际上相等（见注释）。

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。



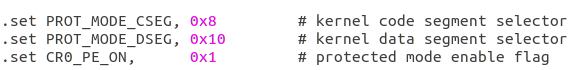
最低位的PE位表示Protection Enable，把这一位置1则将使CPU进入保护模式。



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。



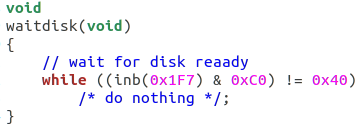
开头的这个常量设置现在也明白了，因为第一个段是SEG\_NULL，占8字节，所以代码段的段描述符在表中的偏移就是8了，然后代码段描述符为16，即0x10。



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？

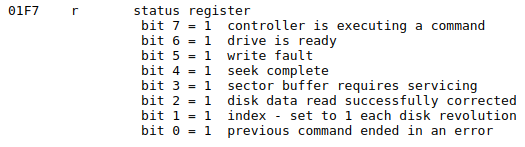
Main.c

这里面有一些声明在x86.h以及elf.h当中。

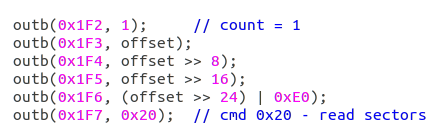


这里的inb(port)函数其实就是从0x1f7（port）端口读入数据作为返回值。之后和0xc0与。

查阅bochs手册之后，果然这个端口是硬盘的状态寄存器。



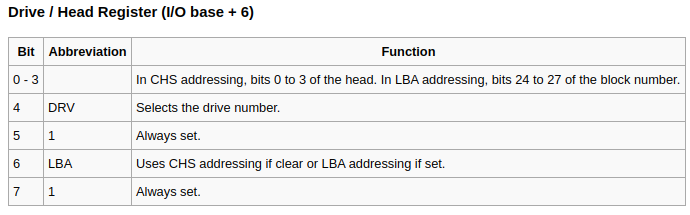
由上面可见，如果返回值和0xc0相与后等于0x40，说明两件事：其一是控制器不在执行指令（bit 7)，其二是设备准备就绪（bit 6)。



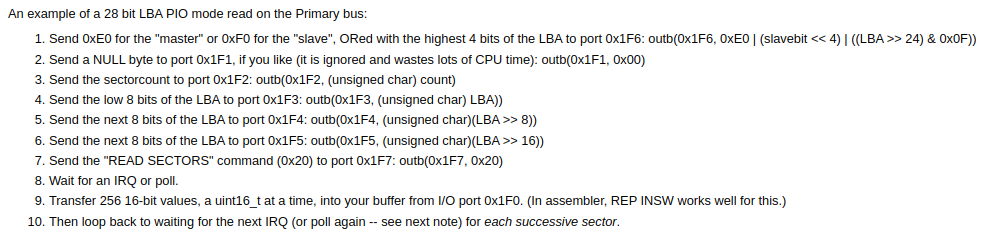
这个地方，0x1f2端口表示读取的扇区数量，0x1f3表示扇区号，0x1f4和0x1f5共同表示柱面号(cylinder low/high)，0x1f6端口有点没懂。查阅资料（[https://wiki.osdev.org/ATA\_PIO\_Mode#Writing\_28\_bit\_LBA](https://wiki.osdev.org/ATA_PIO_Mode" \l "Writing_28_bit_LBA)）可知，这里使用的是ATA硬盘规范。有且仅有一根线（一位）选择总线上的哪一个设备处于活动状态，也即总线上最多只有两个设备，被称为主设备（master）和从设备（slave）。

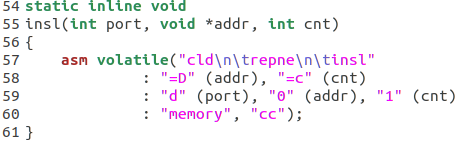
ATA规范有三种寻址模式，为28-bit LBA、48-bit LBA以及CHS。这里LBA即logical block address，而CHS则表示Cylinder-Head-Sector。

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。

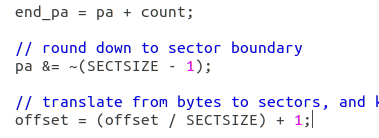
这一个函数，相当于读取port端口的内容，并存入addr地址中。

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。

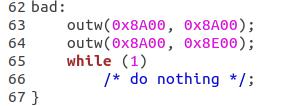


这里注意看注释，读的是第一页，而不是第一块，一页是4K，所以这里SECTSIZE\*8。

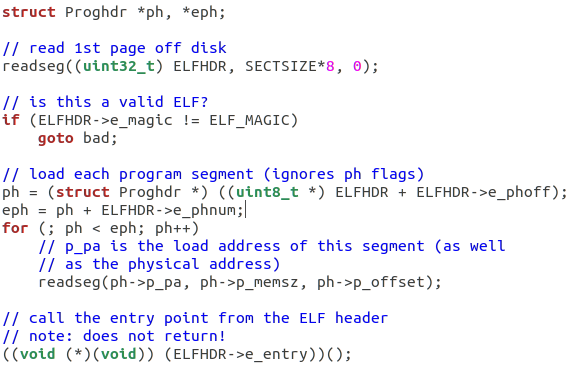
这里读进来的ELFHDR的类型为(struct Elf \*)，但是其实整个4K内存中只有开头部分是Elf首部，而不是说首部就占了4K。

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>）



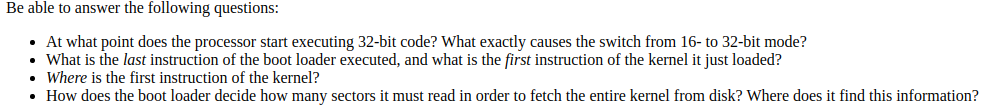
看起来，这是一个模拟器提供的端口，用于调试。首先发送的指令是0x8A00，表示激活调试设备。然后0x8E00应该是一个命令，但是在bochs的手册上没有查到这个命令，其他地方也实在查不到，这个可能是QEMU的命令？在bochs上有一个0x8AE0，效果是返回debuger的命令行，可能这两者效果类似？存疑。。。



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。

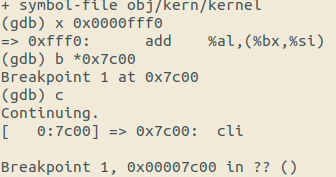
最后这个((void(\*)(void)) (ELFHDR->e\_entry))();是一个函数调用，感觉非常高级。这先是把这个地址的类型转成一个函数指针（参数为void，返回值void），然后用（）操作符作用于这个函数指针，则相当于call 了这个地址。



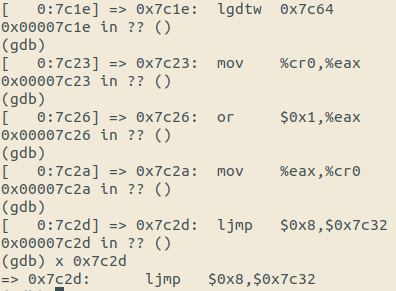
到此为止，代码分析得差不多了。可以回答一下实验的四个问题。

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



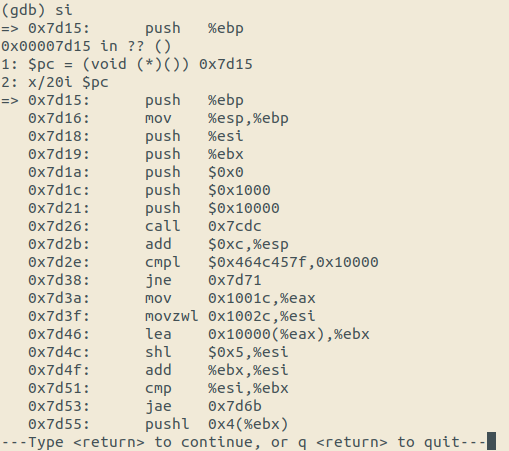
先用b \*0x7c00在boot程序的入口处设置断点，然后c继续执行到断点处。然后si单步执行机器指令。



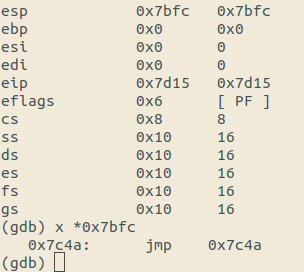
这里可以看到，对于lgdtw和ljmp指令都能够正常地反汇编，而使用ojbdump得到的反汇编文件中，这两条指令是错误的。一条指令被拆成了两条指令。



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**

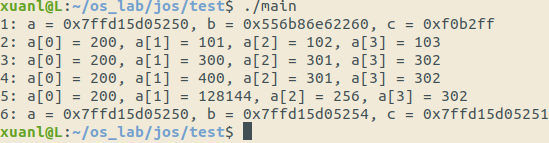
这里一开始的Excercise 4 要求阅读C语言教材的指针部分。本来感觉自己会了，但看这里说除非已经精通C语言，否则不要跳过这个阅读材料。Trust us; you don't want to find out what "the hard way" is.搞的挺吓人的，还是看一看吧。

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。

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？这里试试看改成0x7c01。

