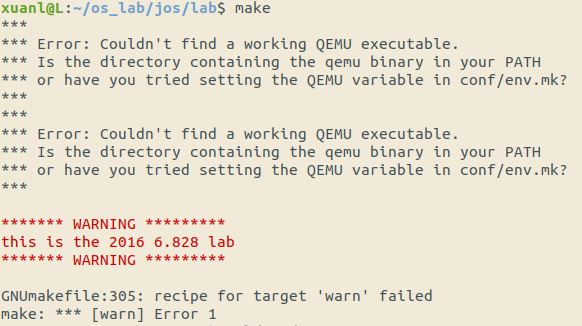
## Lab1

### Part1

Make失败，需要安装QEMU（是指Quick Emulator吗？）

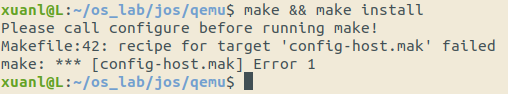


git clone http://web.mit.edu/ccutler/www/qemu.git -b 6.828-2.3.0

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

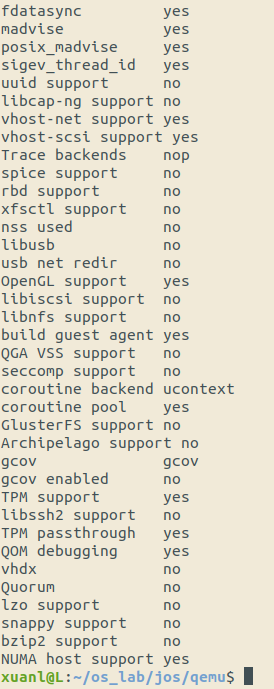
（注意：即使不配置安装路径，也要运行./configure）



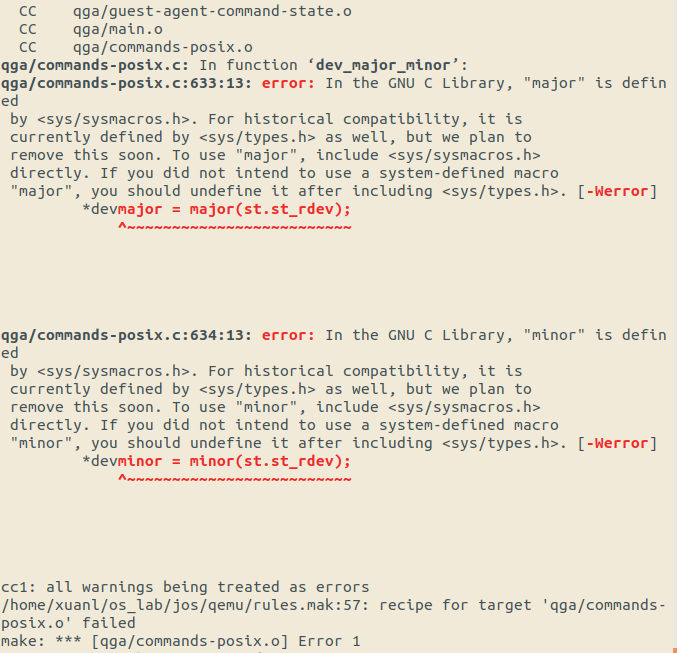
报错：

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。

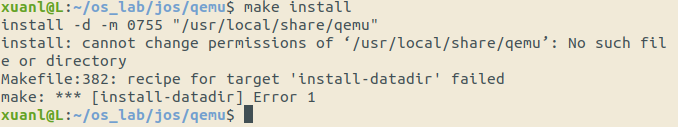


去看了18年的，发现configure的时候加上了--disable-werror，

确实忽略了警告，不会fail了

Make就整了好久

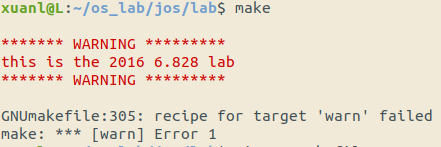
Make file又出问题



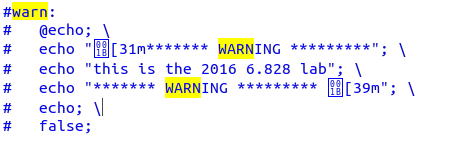
这是权限问题么？

切成root，就可以了。（所以上图中就是安装路径，顺便记下来了）

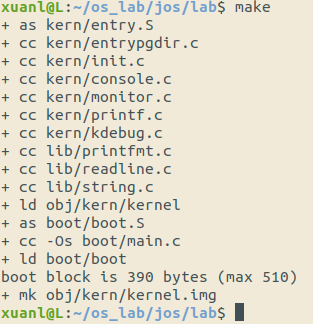
Lab1的make还是有问题，居然因为时2016年的就不让用了



手动修改makefile文件，把所有warn的部分注释掉

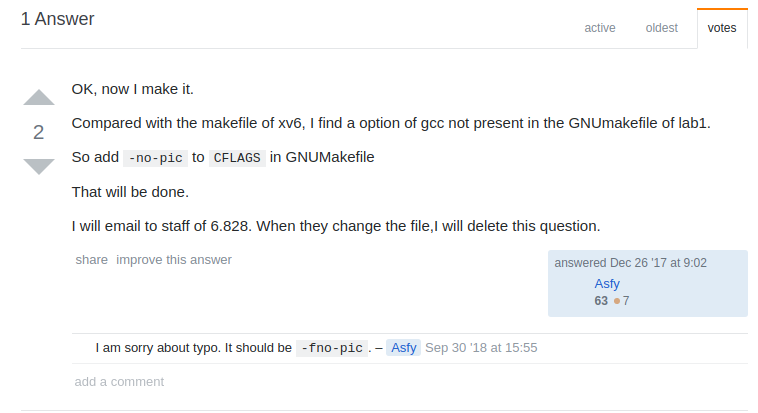
、

然后make，终于成功了！



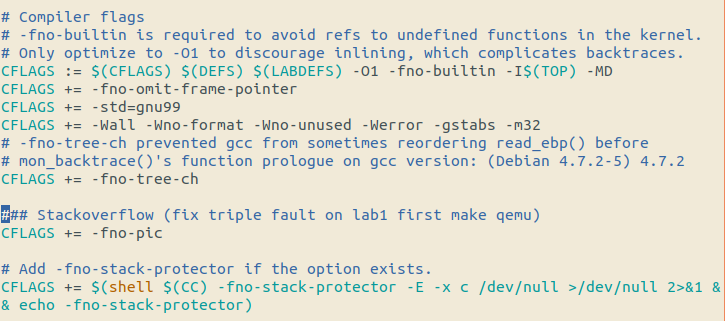
但是，继续make qemu遇到了大问题，出现了Triple fault，然后一直卡死了。

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。

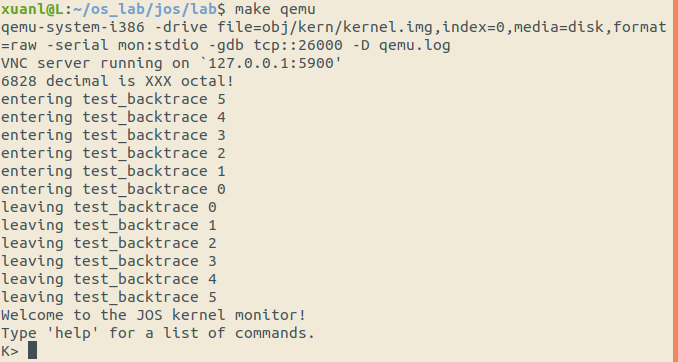


在makefile中的CFLAGS加上一行，CLAFGS += -fno-pic

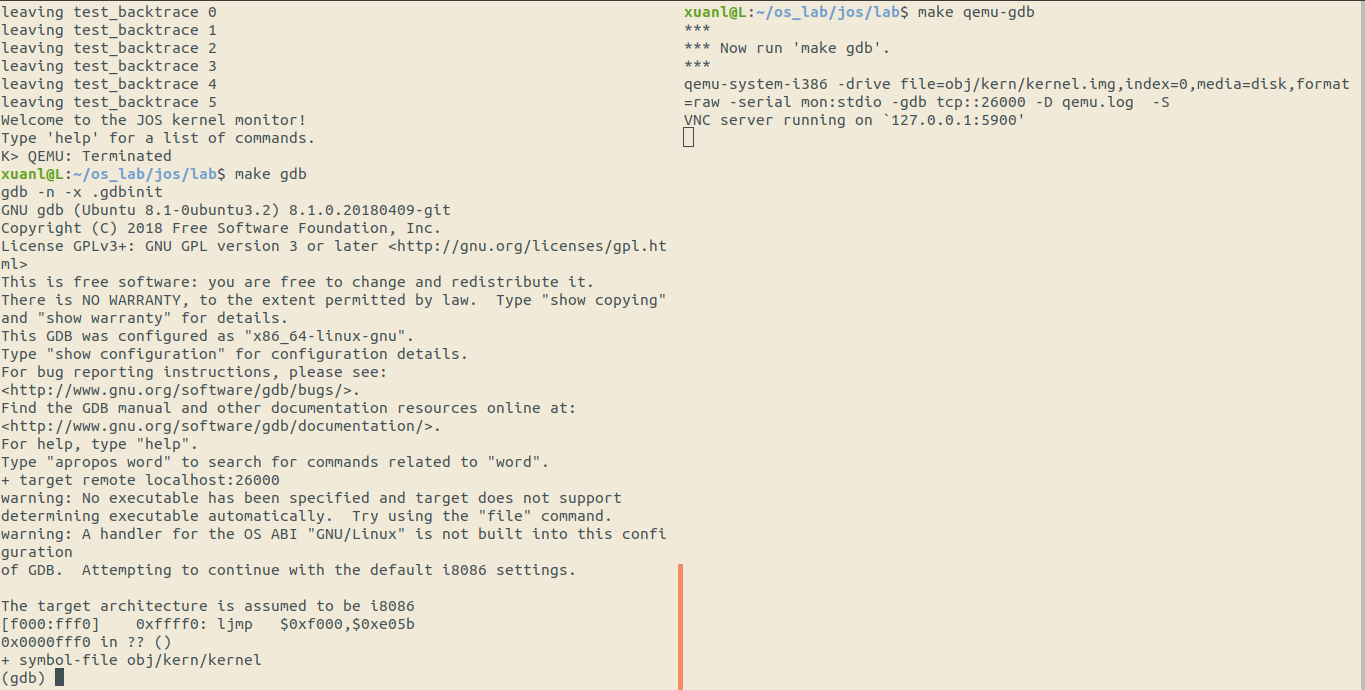
这个地方的pic表示position-independent code位置无关代码，用于生成动态链接的代码，那这里用-fno-pic，应该就是生成的代码里使用绝对地址而不是相对地址。但为什么必须要这样呢？



终于完成了PART1！快落泪了！



退出的话，似乎Ctrl+C不行，得Ctrl+A，X



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）

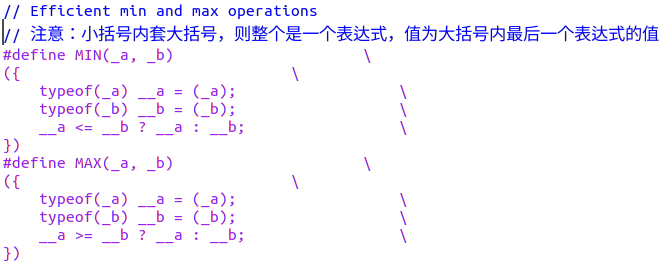
### Part2

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++;});

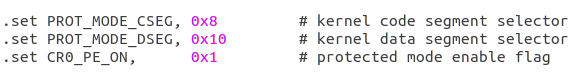


Elf.h里面主要是ELF头部数据结构，以及程序头表，和节头表，这三个数据结构。

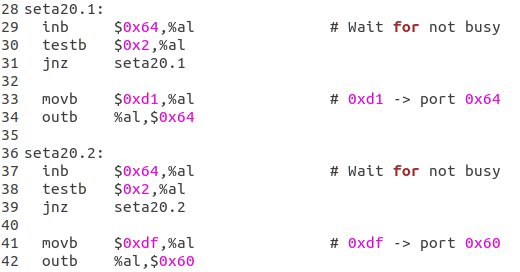
最后是一些常量宏定义

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

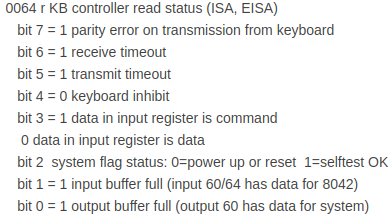


0x64号端口是什么呢？是键盘的状态寄存器。



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

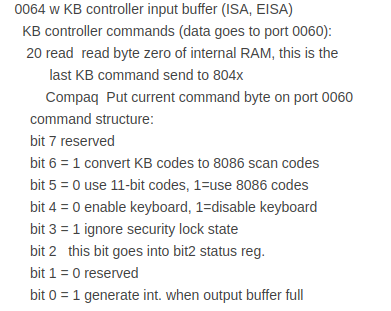
其实可以看bochs的资料，也可以（<http://bochs.sourceforge.net/techspec/PORTS.LST>）



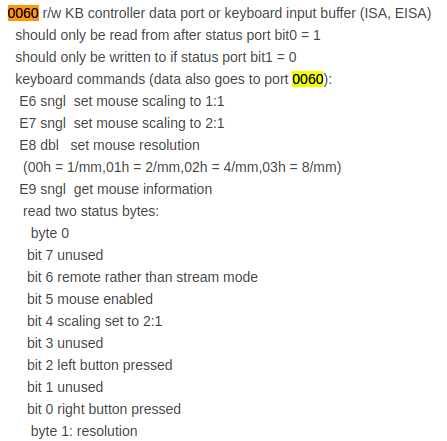
按照这个意思，如果testb $0x2, %al 之后标志位为0，说明input buffer为空，否则为满。那这一段代码就是等input buffer为空的时候继续。

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。

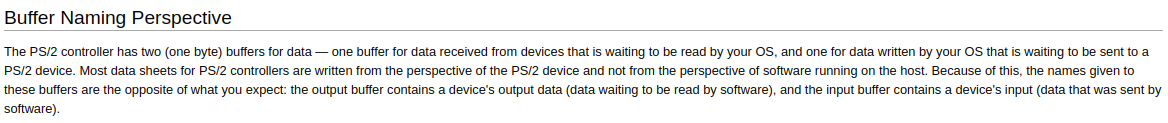


虽然还是不太明白，这个地方实在不知道要看什么了。

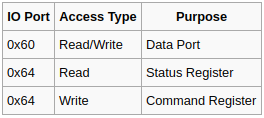
找到个这个，专门讲键盘鼠标部分的，不知道能不能看懂。（[https://wiki.osdev.org/%228042%22\_PS/2\_Controller](https://wiki.osdev.org/"8042"_PS/2_Controller)）

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

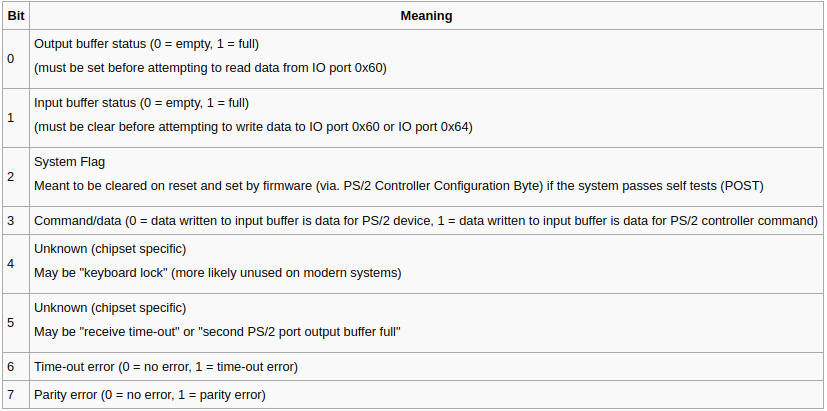


上面这一段提到了命名的问题。端口的命名是站在硬件的视角来的，因此output buffer其实是键盘输出的数据，对于软件来说需要用in指令来读取。反过来，input buffer则是软件输出，对于硬件是输入。



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。

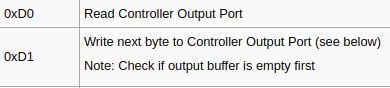
同一个端口地址的读取和写入，可能在实际的物理硬件内部对应的是不同的硬件寄存器。



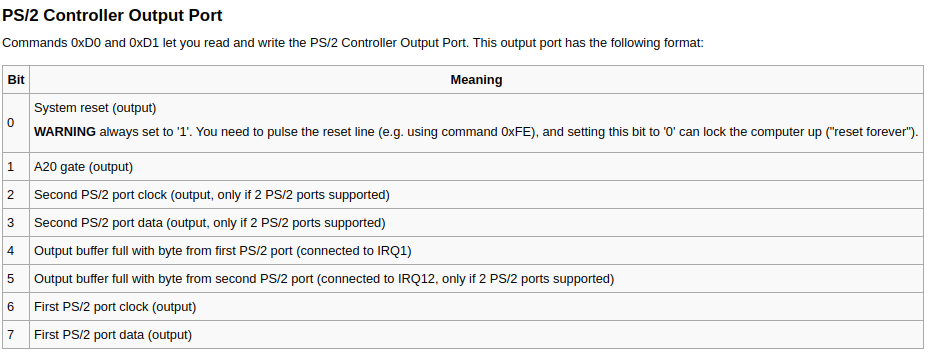
从这个表可以看出，首先，任何需要写数据的操作（写入0x60/0x64），都需要先等待0x64的1号比特位为0时才能写入（可见这一位为1，表示当前设备正在被其他什么东西写入，即写繁忙状态）。但之后写入0xd1到0x64寄存器，这一步感觉还是不能看这个表。继续往下看看。

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。



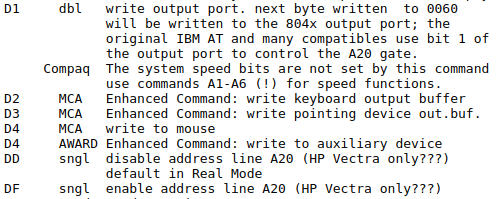
可见，0xd1送到0x64之后，外设此时会等待next byte送入0x60。



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。