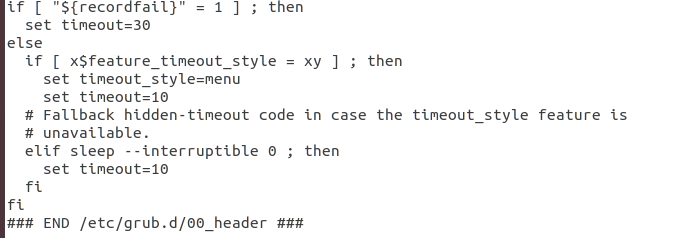
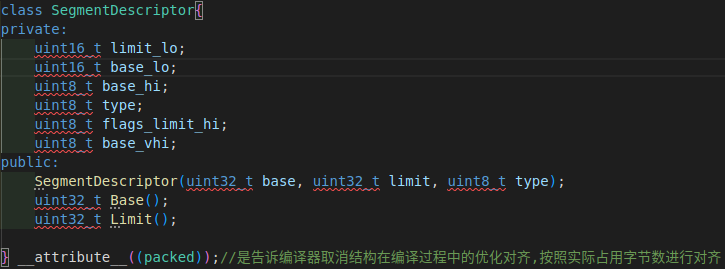
操作系统编写问题总集：

1. **把所写的.bin文件加入启动菜单的时候会出现无法进入系统选择界面**

在grub.cfg 中需要设置**timeout\_style**为menu，并按自己需求设置timeout（超时后自动选择第一个启动项）

1. **在CentOS中，将manuentry 写入grub.cfg后，reboot，启动自己的系统后会出现文件无法找到**

暂时还没找到解决问题，在centos中只有grub2文件，将系统改为使用ubuntu后则可以正常启动

1. **设置全局描述符条目的每一位**



0x9A表示可执行，一致性代码段

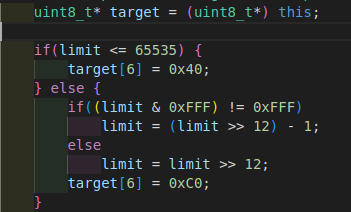
0x92示只读，向下扩展的数据段

**P:**高32位中的第15位，用来标识段是否存在内存中，1表示存在。如果为0，CPU将抛出异常，跳转到我们编写的异常处理程序，在处理完成以后，需要将该字段置为1

**DPL：**高32位中的13~14位（共2位），表示描述符特权级，特权级的数字越小，特权级越大。CPU进入保护模式（操作系统的代码）默认为0，用户应用程序通常处于3特权级，权限最小

**S：**高32位中12位，S用来表明是系统端（S为0）还是数据段（S为1），在CPU的世界里，系统段主要是指各种称为门的结构，如调用门，任务门；数据段主要是指操作系统及应用程序的代码、数据以及栈。

**TYPE：**高32位中的8~11位（共4位），TYPE用来表示段描述符的类型，在不同S标志位下，TYPE的含义不同

target[6]的高8位是flags



**G：**高32位中23位，段界限只是单位量，G用来标识单位量的大小，G为0，单位量为字节，G为1，单位量为4K

**D/B：**高32位中的第22位，用来指示有效地址（段内偏移地址）及操作数的大小。如果段是代码段，0表示指令中的有效地址地址和操作数是16位，指令地址使用IP寄存器，1代表指令中的有效地址和操作数是32位，指令有效地址用EIP寄存器；对于栈段来说，0使用的是SP寄存器，1使用的是ESP寄存器

**L：**高32位中的第21位，用来设置是否是64位代码段，L为1表示64位代码段，否则表示32位代码段

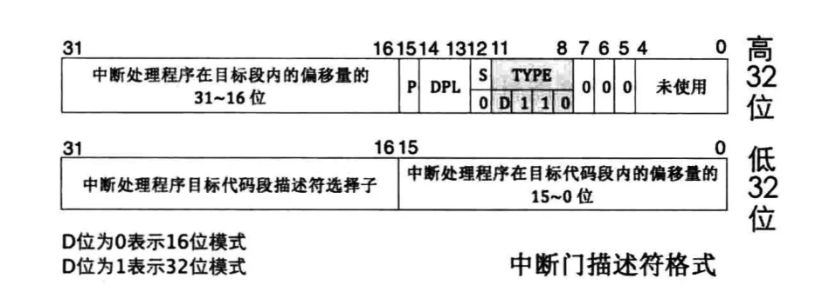
**AVL：**高32位中的第20位，对于操作系统来说，此位没有效果

0x40: D/B = 1

0xC0: G = 1; D/B = 1

1. **屏幕的大小是80 \* 25的，内存中的初始位置是0xb8000，每一个字符为16bit高8bit为颜色信息，低8bit是内容**
2. **如何从.cpp文件的函数跳转到.s文件中**

将所需要调用的函数设置为静态函数，编译后获得的目标文件中会有他作为编译后的函数名，用于.s文件中

1. **中断描述符**

中断类型：

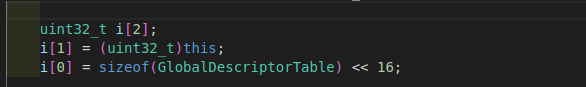
任务门描述符:Type = 0101

中断门描述符:Type = D110

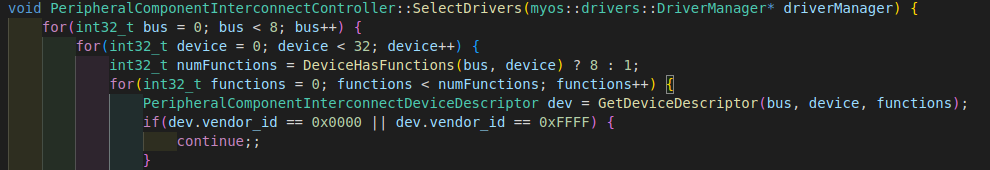
陷阱门描述符:Type = D111

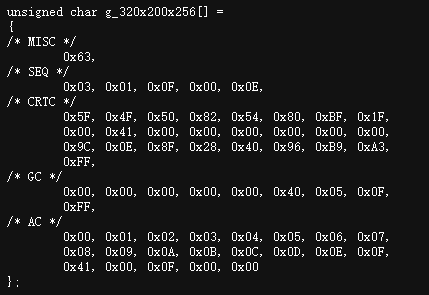
调用门描述符:Type = D100

DPL为DescriptorPrivilegeLevel

1. **在教程中，出现开启中断后系统崩溃可能是GDT中指针指向错误的位置让CPU崩溃，需要修改以下位置的顺序，****this 放在i[1]。**
2. **鼠标驱动中，如果鼠标出现问题可以尝试修改3bit数组的初始offset**

Buff[1] = x,buff[2]=y方向相反

1. **PCI信息**
2. **在****PCI中，设备功能编号并不是连续的**
3. **在Memery Mapping中，16位地址原信息是地址掩码，表示地址映射的范围，当使用完成address后要还原为原来的掩码**
4. **查询不同模式需要传输给显卡的数据**

//https://files.osdev.org/mirrors/geezer/osd/graphics/modes.c

320 x 200 x 8bit;

1. **VGA端口含义 （https://wiki.osdev.org/VGA\_Hardware#Port\_0x3C8）**

**Port 0x3C0**

This port is a confusing one: you write both the index and data bytes to the same port. The VGA keeps track of whether the next write is supposed to be the index or the data byte. However, the initial state is unknown. By reading from port 0x3DA it'll go to the index state. To read the contents, feed the index into port 0x3C0, then read the value from 0x3C1 (then read 0x3DA as it is not defined whether the VGA expects a data byte or index byte next).

**Port 0x3C2**

This is the miscellaneous output register. It uses port 0x3C2 for writing, and 0x3CC for reading. Bit 0 of this register controls the location of several other registers: if cleared, port 0x3D4 is mapped to 0x3B4, and port 0x3DA is mapped to 0x3BA. For readability, only the first port is listed and bit 0 is assumed to be set.

**Port 0x3C4, 0x3CE, 0x3D4**

These are the most used indexed registers. The index byte is written to the port given, then the data byte can be read/written from port+1. Some programs use a single 16-bit access instead of two byte accesses for writing, which does effectively the same. (take care of byte ordering when doing so)

Port 0x3D4 has some extra requirements - it requires bit 0 of the **Miscellaneous Output Register** to be set before it responds to this address (if cleared, these ports appears at 0x3B4). Also, registers 0-7 of 0x3D4 are write protected by the protect bit (bit 7 of index 0x11)

**Port 0x3C6**

Port 0x3C6 only contains the DAC Mask Register, which can easily be accessed by a simple read/write operation on this port. Under normal conditions it should contain 0xff.

**Port 0x3C8**

Port 0x3C8, 0x3C9 and 0x3C7 control the DAC. Each register in the DAC consists of 18 bits, 6 bits for each color component. To write a color, write the color index to port 0x3C8, then write 3 bytes to 0x3C9 in the order red, green, blue. If you want to write multiple consecutive DAC entries, you only need to write the first entry's index to 0x3C8 then write all values to 0x3C9 in the order red, green, blue, red, green, blue, and so on. The accessed DAC entry will automatically increment after every three bytes written. To read the DAC entries, write the index to be read to 0x3C7, then read the bytes from port 0x3C9 in a similar fashion (as with writing, the index will increment after every three bytes read)

**（14）硬盘控制器主要端口寄存器**

**data 寄存器** 16位

在读硬盘时，硬盘准备好数据后，硬盘控制器将其放在内部缓冲区中，不断读此寄存器便是读出缓冲区中的数据。

在写硬盘时，我们要把数据源源不断地输送到此端口，数据便被存入到缓冲区里，硬盘控制器发现这个缓冲区中有数据了，便将此处的数据写入相应的扇区中。

**Error** 8位

使用场景：读

只在读取硬盘失败时有用，里面记录失败的信息，尚未读取的扇区保存在 Sector count 寄存器中。

**Features** 8位

使用场景：写

写硬盘时，有些命令需要指定额外参数，这些参数就写在 Fea ture 寄存器中。

**Sector count**8位

使用场景：读写

用来指定待读取或写入的扇区数。硬盘每完成一个扇区，就会将此寄存器的值减 1，所以如果中间失败了，此寄存器中的值便是尚未完成的扇区。

8位寄存器，最大值位 255，若指定为 0，则表示操作 256 个扇区。（因为：1 0000 0000 舍去高位 1 从而保证 8 位，结果为 0）

**LBA low | mid | high**8位

使用场景：读写

LBA 有两种，一种 LBA28，另一种 LBA48，均用来描述一个扇区的地址。

LBA low：存储 28 位地址的第 0~7 位

LBA mid：存储第 8~15 位

LBA high：存储第 16~23 位

device 寄存器的低 4 位：存储第 24~27 位

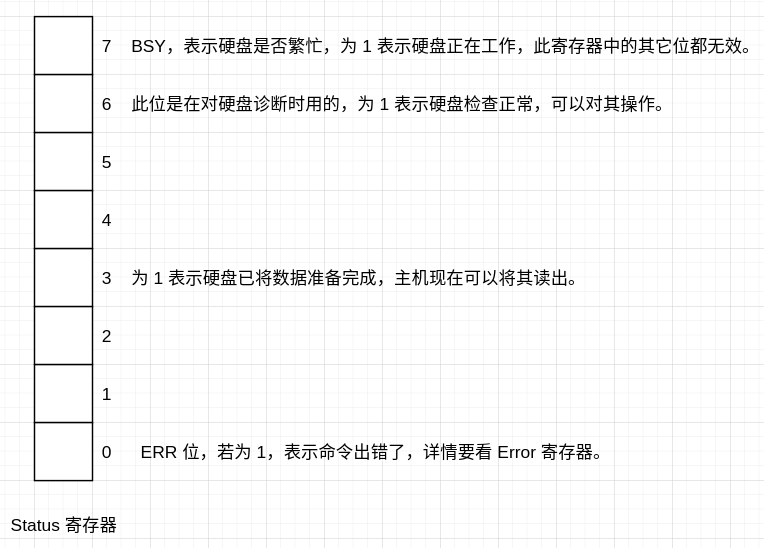
**device 寄存器**8位

使用场景：读写

**Status**8位

使用场景：读

作用：保存硬盘的状态信息。

图示：

**Command**8位

使用场景：写

作用：存储让硬盘执行的命令，将命令写入该寄存器，硬盘便开始工作了。

主要命令：

1 identify：0xEC 硬盘识别

2 read sector：0x20 读扇区

3 write sector：0x30 写扇区