# 操作系统实验日志

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 学号 | 201808010718 | 姓名 | 肖鹏 | 专业年级班级 | 智能1802 |
| 实验日期 | 2020.11.17 | 实验项目 | 第8天：鼠标控制与32位模式切换 | | |

## 一、实验主要内容

### 1、完成鼠标移动

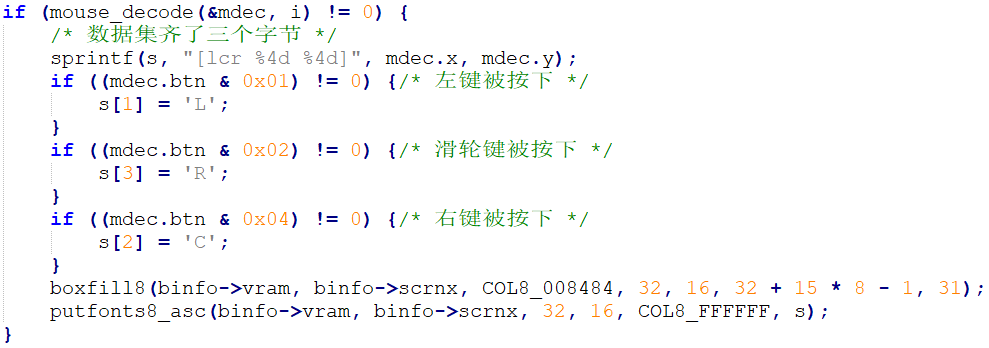
#### 内容：

a）当鼠标准备好之后，以3-bytes数组为一组，每当收集好一组数据之后，便把数据

打印出来；这里设置了一个变量mouse\_phase，用来记录当前处理数据的进度：等待数据、处理第1字节 、处理第2字节、处理第3字节；

b）把位于harimain函数中对键盘、鼠标数据处理的部分整理到专门的函数中，这样使 主函数内比较简洁，提高代码可读性；

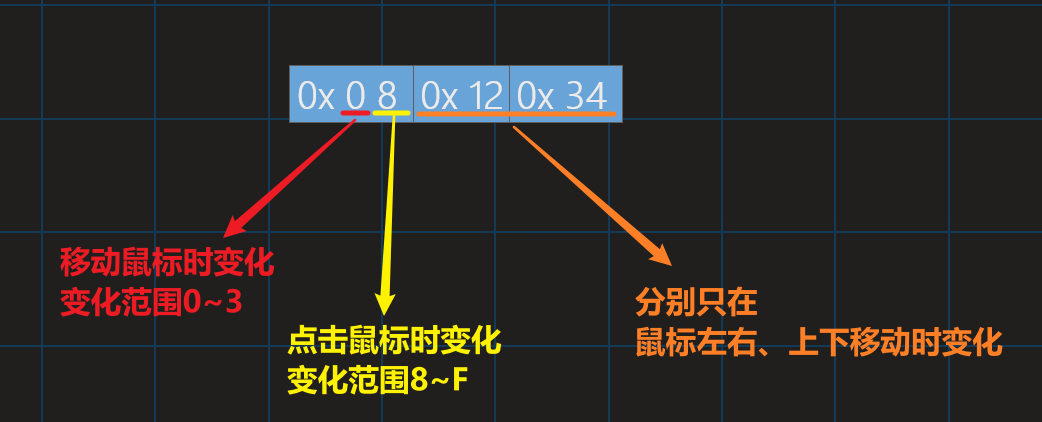
c）进一步解析3-bytes的数据；对第一字节的数据进行合法性检测，如果对鼠标移动 的变动的范围不是在规定的0~3内、对鼠标点击的变动范围不是在规定的8~F内， 那么这一个字节的值便会被舍去；这是为了防止让这些不该产生的数据造成内部 更大的问题；同时，根据所提取到的第一字节中的按键信息，把按下的按键对应的 字母显示成大写，未按下的字母呈小写；



d）让鼠标跟着移动；上一步已经把数据划分成三部分，我们只需要利用和鼠标移动有 关的x、y；通过变量记录鼠标的位置mx、my，每次接收到数据之后更新mx、my， 然后绘制出新位置的鼠标，即可实现“鼠标移动”；

#### 重点总结：

a）鼠标3-bytes数据的基本意义：



b）使用结构体MOUSE\_DEC，把一组从鼠标接收到的信息和数据处理进度放在一起， 方便处理；其中，归纳后的函数mouse\_decode只有在完整接收到3-bytes的数据 之后才会返回1，表示接收数据工作已结束，可以进行数据的集中处理；其他情况 ——处理进度未完成或者是其他异常情况导致的错误；

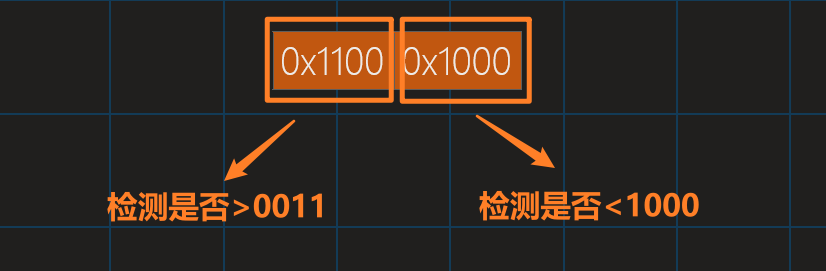
c）对字节处理之前，先检测第一字节数据的合法性；之后对于第二、三字节的处理， 需要将其转化为我们可读的描述鼠标移动方向的数据，即在x轴、y轴上移动的数 据；其余步骤：

①首先取出第一字节的低3-bits，描述的是鼠标键的状态；

②先直接把第二、第三字节的数据赋给鼠标的x、y成员变量（即水平、竖 直方向的移动量）；

③使用第一字节中与鼠标移动有关的部分数据调整x、y；

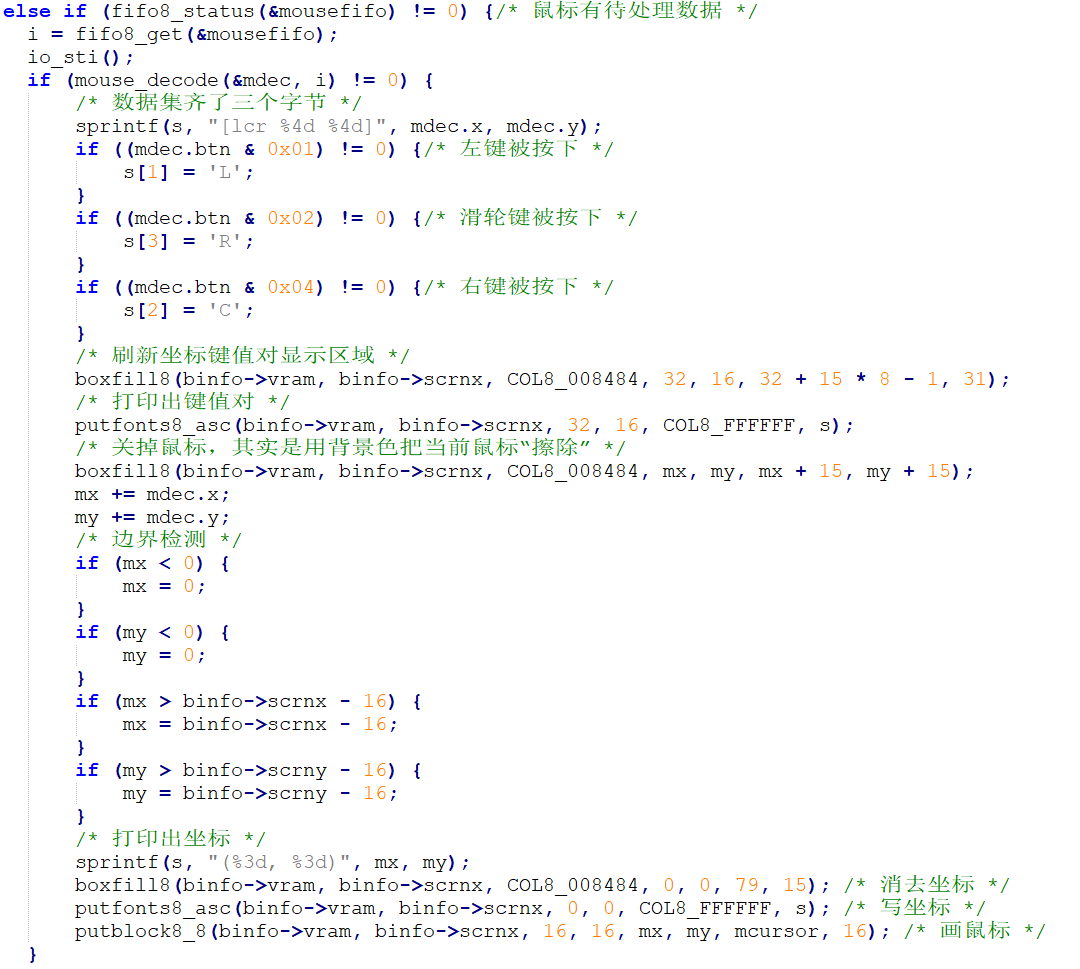
④由于鼠标移动时参考的y轴方向与屏幕中坐标y轴方向相反，故对y取 相反值；



d）在鼠标移动的实现中，需要进行边界检测，以免鼠标移动到了显示区以外的位置；

#### 关键代码及注释：



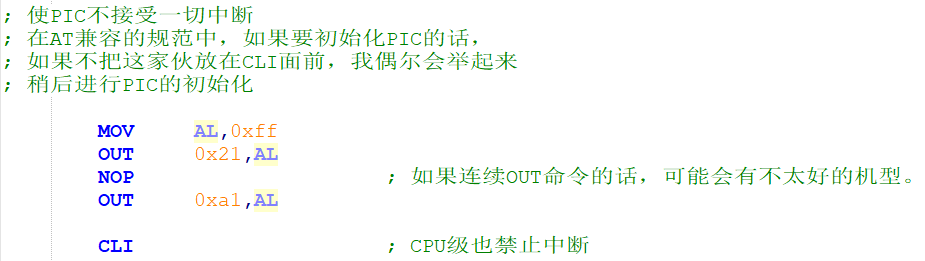
*（harib05a中Harimain中对鼠标字节的处理）*

*（harib05d中Harimain中对鼠标字节的处理）*

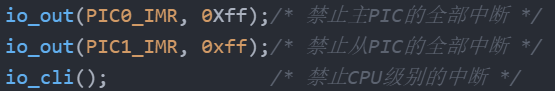
### 2、通往32位的路

#### 重点内容及代码注释：

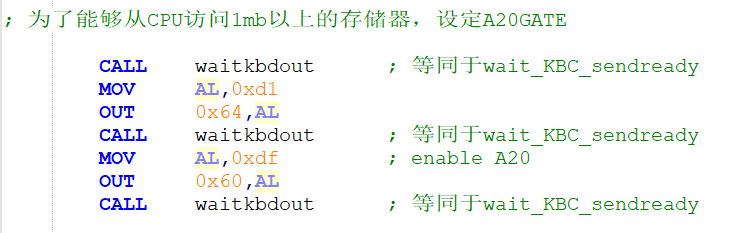
##### a）section#1：屏蔽中断



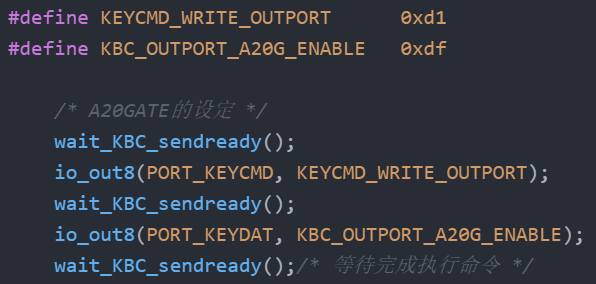
在进行CPU模式转换的时候，不允许被终端打扰，需要禁止所有中断；上述代码 相当于以下C程序：



##### b）section#2：获取1M以上内存



上述汇编代码相当于下面C代码：



有关waitkbdout的汇编实现如下：



*（waitkbdout汇编实现）*

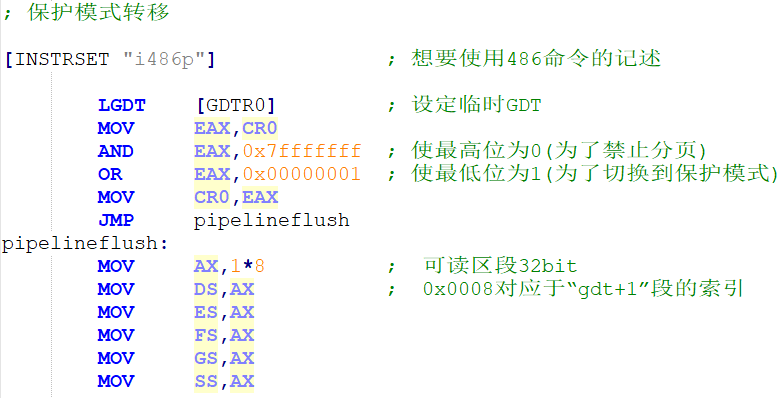
其中，添加了部分处理，就是从0x60进行读取数据，也就是说，如果控制器里有键 盘或鼠标的积累数据，就顺便读出来；

上面这一节代码首先等待键盘控制电路准备好接受指令，之后发送要模式修改指令， 模式设定成KEYCMD\_WRITE\_OUTPORT模式；之后等待控制电路准备，把具体的指令 （0xdf）发送到一个附属端口（0x60）；这个附属端口发出这条指令的功能是，让 A20GATE信号线变成ON的状态，从而可以使用1M以上的内存；

以我们要转化到的32位为例，如果A20GATE未开启，则32位地址其地址表示为 0xFFEFFFFF。这种表示的意思是如果A20Gate被禁止，则其第21条总线在CPU做地 址访问的时候是无效的，永远只能被作为0；如果A20 Gate被打开：则其第 21条总线是有效的，其值既可以是0，又可以是1。

也就是说，由于使用32位地址线，如果A20恒等于0，那么系统只能访问奇数兆的 内存，即只能访问0--1M、2-3M、4-5M......，这显然是不行的。很显然，在实模式下 要访问高端内存区，这个开关必须打开，所以在保护模式下，这个开关也必须打开。

##### c）section#3：切换到保护模式



保护模式就是刚刚提到的，在保护模式下才能够完全使用到内存，而在实模式下做不 到；之前16位模式下应该是处于实模式，所以在打开了A20GATE后还需要进行模式 转换才能够真正的利用到1M以上的空间；模式转换后，段寄存器的解释应该是段 号，每个段在GDT中注册之后有一个处于段表的序号，段寄存器中储存的不是地址 而是段位于段表中的索引，通过段表条目获得段的基址；

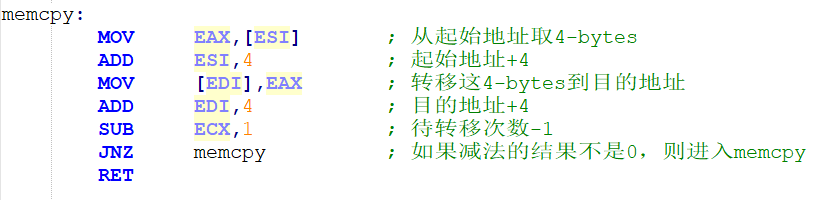
所以在切换到32位保护模式后，需要把段寄存器的内容修改成对应的意思，这里代 码做的是把几个段寄存器的内容改成gdt+1段对应的索引；

##### d）section#4：转移bootpack



进入32位保护模式的工作基本上完成，剩下的就是把我们软盘中的用户代码转移到 相应的地址；这里的汇编代码只是在调用memcpy程序，分两部分（bootpack和软盘 全部的1440KB）转移到内存，bootpack转移到0x280000起始处、软盘数据位于 0x100000~0x267fff保存；

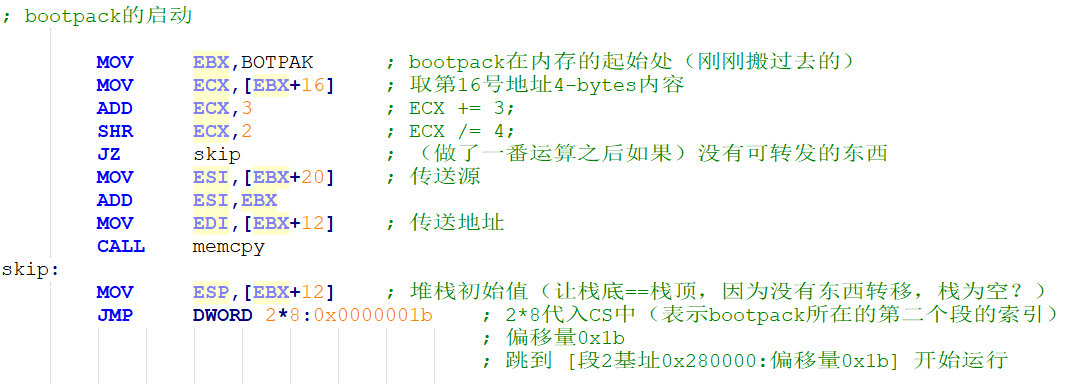
这里需要注意的就是：①memcpy函数每次以4-bytes为转移单位，即每次复制4- bytes的内容到目的地址，所以先把待转移的数据字节大小/4，得到一个需要转移的 次数n，然后在memcpy函数中，每转移一次，n--，直到==0；（在代码中，n保存于 寄存器ECX）；



*（memcpy汇编实现）*

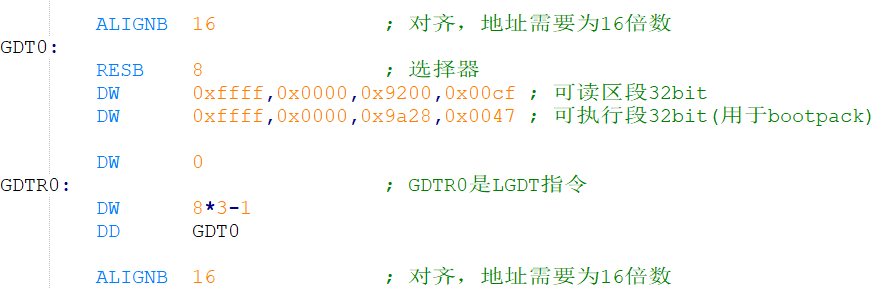
②转移除启动扇区外的扇区，要先把扇区数转化成字节数，软盘一共有18扇区、每 个扇区2面、一面512-bytes；最后需要减去启动区的转移次数；

##### e）section#5：为bootpack运行准备栈空间



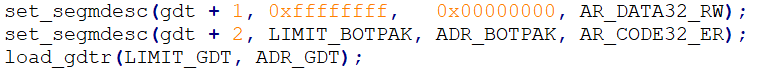
上一步只是把bootpack整个以程序的形式搬到了内存里，但真正要运行起来，还需 要在内存中为其分配栈空间，并且把将用到的数据等内容转移过去；从这里可以看到 在32位保护模式下对段寄存器的解释，通过段寄存器保存的段索引和偏移量拼接形 成地址；

##### f）section#6：初始GDT



首先第一条指令“ALIGNB 16”，其实只是一条简单的对齐指令，对齐的要求是起始地 址都需要是16的倍数；用在这里即GDT0的首地址必须是16的倍数。为什么呢？因 为如果起始地址不是8的倍数的话，向段寄存器复制的MOV指令就会慢一些，所以 其实用“ALIGNB 8”也可以；

然后看下面的指令，首先是“RESB 8”，空了8-bytes的0，表示这里不能定义任何 段；之后的两条语句表示定义的两个段：



下面的GDTR0，书上说的是“GDTR0是LGDT指令”，一看GDTR0里面刚好6-bytes 的内容，分别表示段上限、段起始地址；和之前使用LGDT指令的时候的描述是一致 的；

Question： 至于为什么asmhead.nas里面明明设置了两个段，在bootpack.c里为什么 还要再初始化？

按上面的“GDTR0是LGDT指令”理解，当在bootpack.c初始化时第一次调用 load\_gdtr之前，GDT还不在内存中（GDT表是通过int\_gdtidt时创建到内存的），此 时GDT还在asmhead里面，IDT压根都不存在；所以在执行bootpack时，需要尽快 重建GDT和IDT；所以asmhead里面的GDT0是给内存中GDT还没有创建的时候用 的，这就是为什么初始化时又需要根据数据所在内存地址为参数再创建一个GDT；

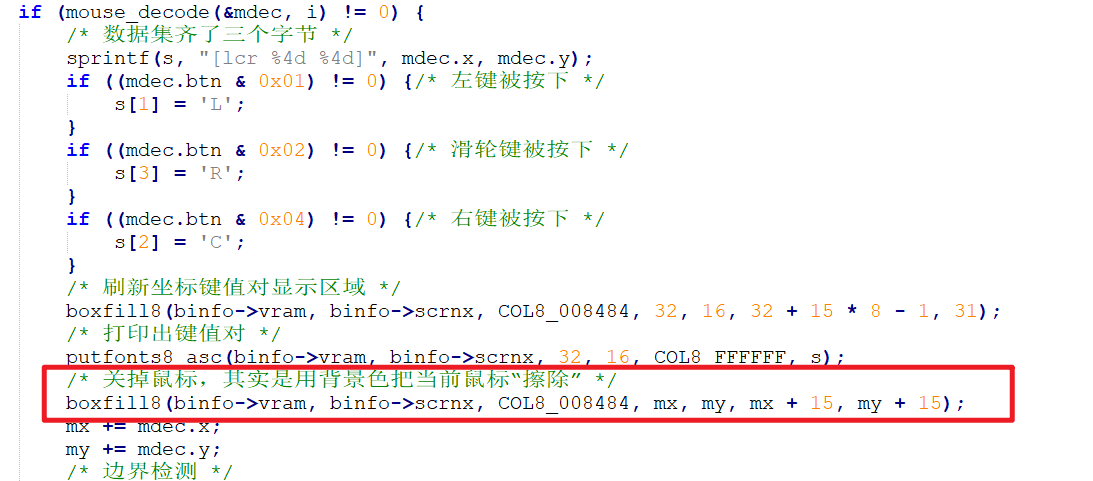
## 二、遇到的问题及解决方法

### 1、为什么鼠标会破坏桌面？

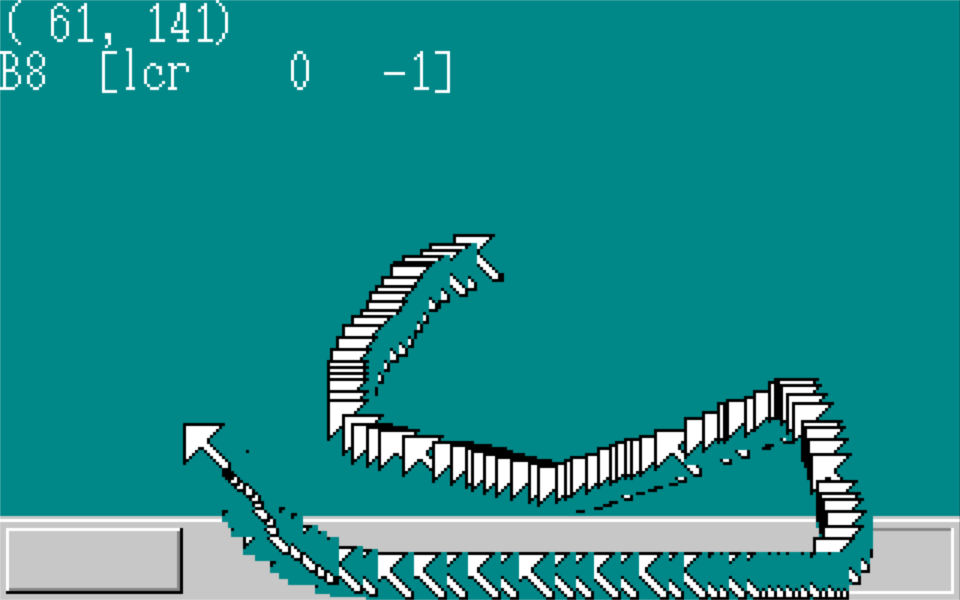
**问题描述**：如图，当鼠标滑动到任务栏区域时，会出现以下“橡皮擦”效果，仿佛把桌面 擦掉了；



**出现原因：**分析鼠标中断时使用的函数代码，发现有一个这样的语句，应该是导致这个问 题的原因；



可以看到，每次鼠标送来一组数据时，都会先把鼠标“关掉”，为什么要关掉呢，可以尝 试一下不关会有什么影响：



可以看到，鼠标移动路径上出现了无数个重影；所以为了避免重影出现，在把鼠标移动 到目标坐标之前先把鼠标变成“背景色”来实现把鼠标“关掉”的效果；但也正是这个 操作，导致了鼠标在背景色与COL8\_008484不一样时，就会有“橡皮擦”的效果；

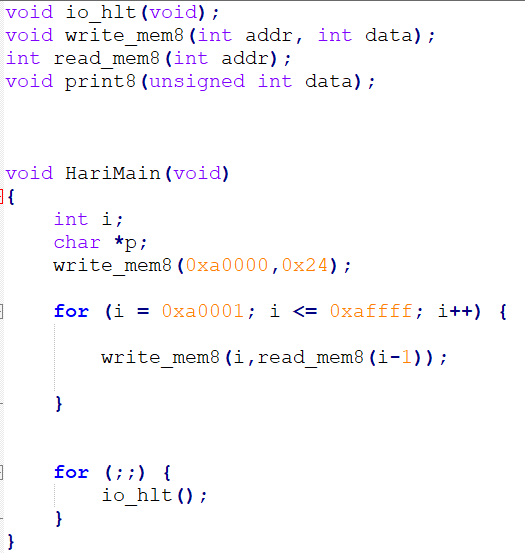
**解决方法：见创新点1；**

### 2、asmhead.nas中的GDT0与内存中的GDT

**问题描述：**在asmhead.nas中明明已经设置了一个段表GDT0，为什么在bootpack.c里

还要创建GDT？

**出现原因及分析理解：**没有联系之前的实验；在之前的实验中，我们在还没有涉及GDT、 IDT的时候，就已经可以运行bootpack.c中的C代码了：



*（harib01a中的bootpack.c）*

因为这时其实在asmhead.nas中已经有一个段表GDT0的存在，在GDT0中已经为 bootpack.hrb分配好了段（段号2）、运行时的栈空间、以及有相应的指令将其从磁盘中 搬运到内存对应的地址，所以函数已经能照常运行；那为什么又要重新创建一个GDT 呢？

个人理解：asmhead.nas中的GDT0在编译生成映像文件之后，其实是固化到了软盘 中 的，也就是这个GDT0是处于软盘固定位置的，不能后续拓展段表；段表需要在后续的 实现中，根据程序文件的大小、权限实时的创建独立的段表条目，然后动态地注册到段 表中，而这些只有当段表位于内存中才能实现，如果在asmhead.nas中创建，那么只能够 在生成映像文件之前就要确定好要用到的所有程序、并为他们分配好段号；这样看起来 应该是不现实的，首先对于一些动态执行的程序，无法预测，用C编程时也无法直接对 汇编代码进行修改；其次由于段数目有限制，也行一次性全把段分配完了，有的程序还 没有得到内存空间；所以，在内存中重建一个GDT的好处是：①GDT的地址容易获得， 因为这是我们自己指定的，这样全局都能简单地访问到段表；②可以在C程序中动态的 分配段索引；

## 三、程序设计创新点

### 1、尝试修复鼠标“擦除桌面”现象

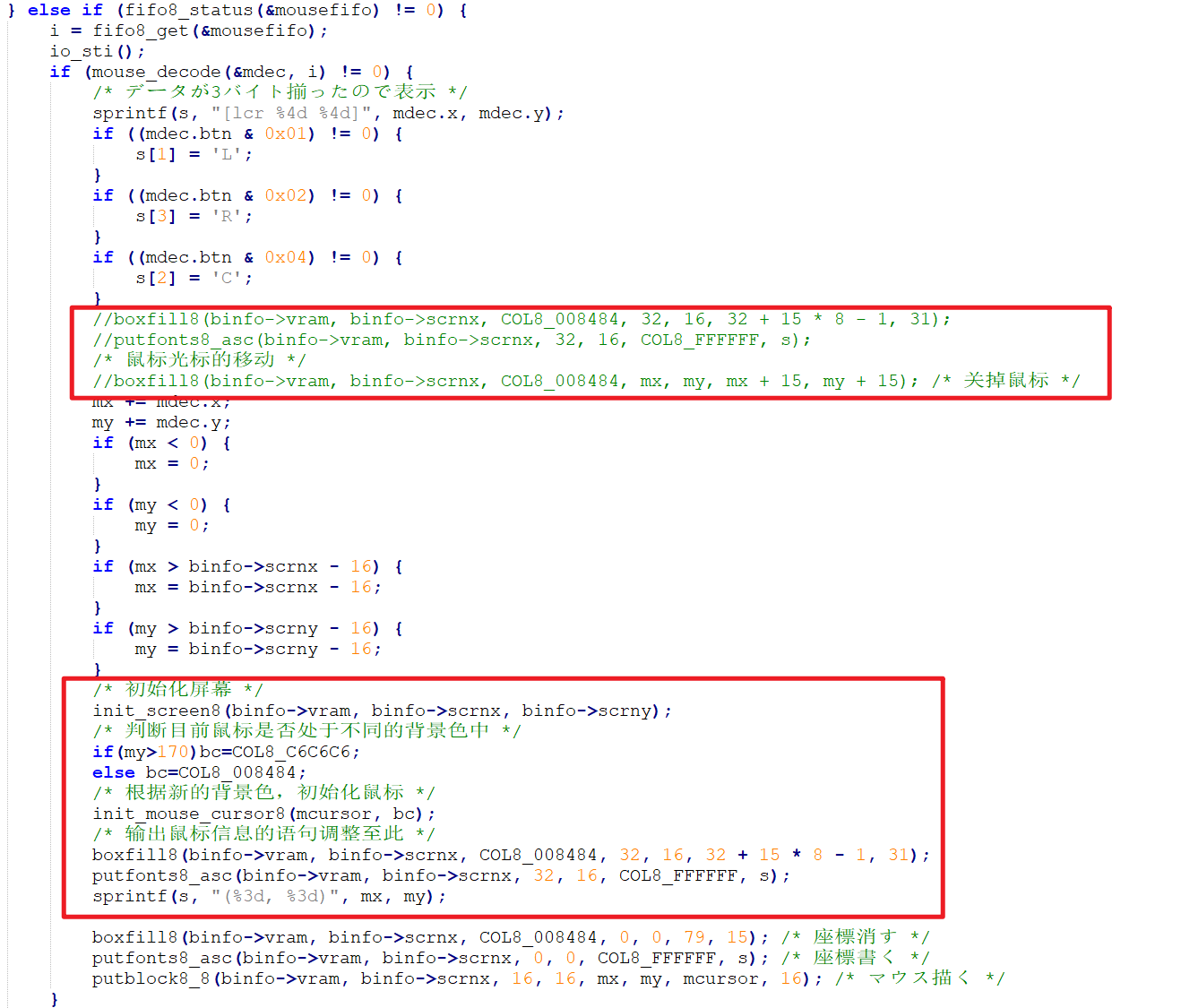
**创新点：**在问题1中可以看到鼠标的效果不是很好，我们尝试看看能不能用一些简单的方 法解决；首先在上面提到的那一句“关闭鼠标”的代码先把他注释掉，那这样我们的鼠标 就会出现重影现象，怎么解决这些重影呢，可以简单粗暴的在绘制出最终位置的鼠标之前 加上一句屏幕的初始化，这样即使在中间过程中会出现重影，但是在通过初始化屏幕，我 们能够保证最终的鼠标一定是绘制在一张“干净”的桌面上的，见效果图（1）；当然这个 简单的办法也有显而易见的bug，就是鼠标始终有一个固定的背景色，就算到了任务栏， 背景色依然是蓝色的；不仅如此，就连显示鼠标信息的键值对都消失了——噢！初始化背 景后被覆盖了，调整一下代码位置可以解决！

那么对于背景色，我们根据鼠标所在的不同区域，来调整背景色可以吗？

尝试一下，当鼠标my>170也就是到了任务栏部分时，把背景色bc改成COL8\_C6C6C6；最 终的效果见效果图（2）；

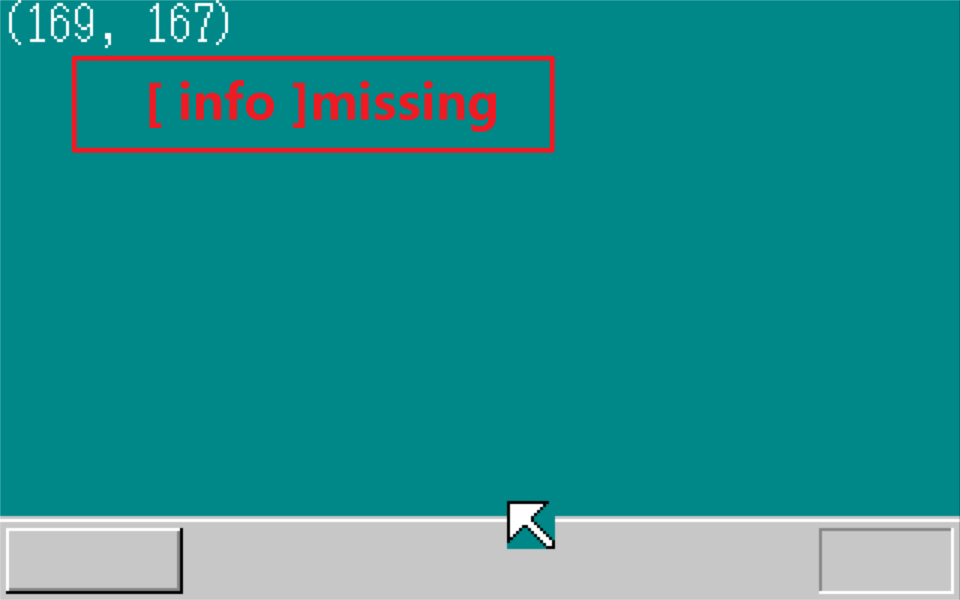
**存在的问题：**实现的时候便发现，这样通过鼠标的位置来判断背景色，然后初始化鼠标、 整个背景，这样的作法肯定不是最终实现的；首先这样的方式开销比较大，重新初始化需 要遍历整个屏幕像素点，而且就开销不说，效果也不尽如人意，比如在不同背景交接的地 方，还是会出现颜色“断层的现象”；百思不得其解后，看了看后面的实验，发现对于这个 问题，是通过不同“图层”的方式实现的，所以，笔者这里还是too young too simple……， 但作为一次尝试，便记录在创新设计点中。

**实现代码：**

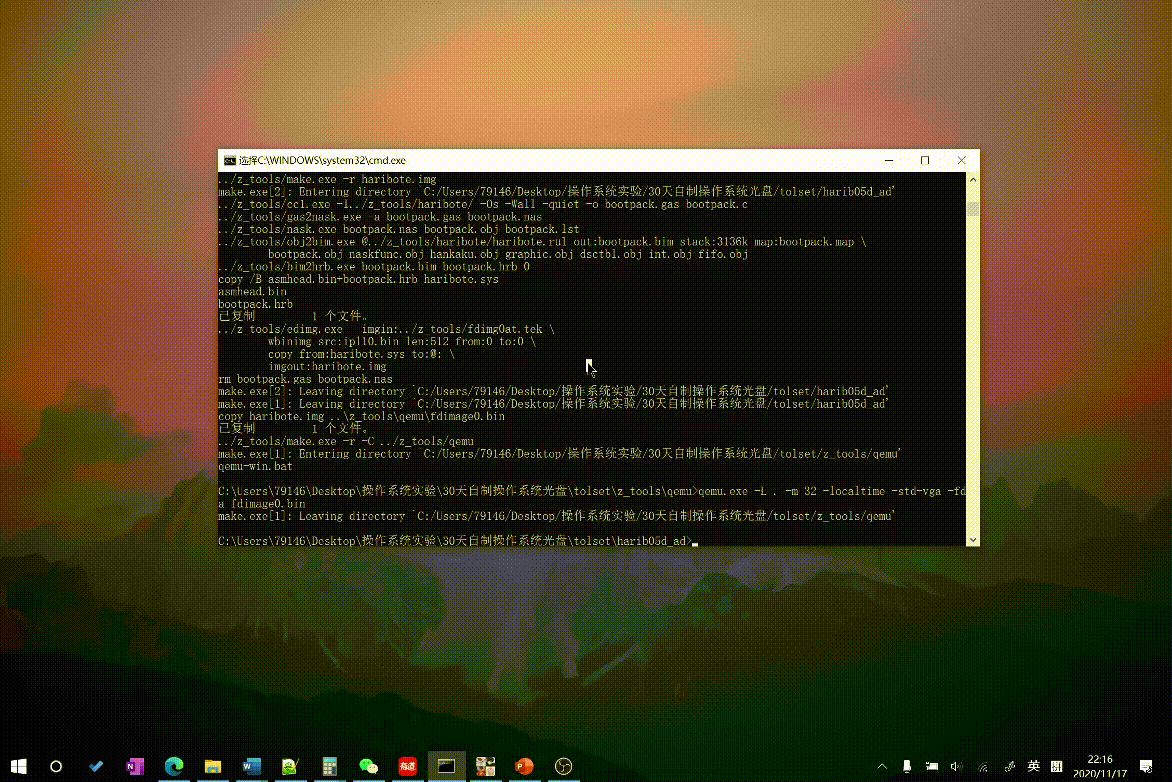


*（harimain中代码，其中红框表示变动的部分）*

**效果截图：**



*（效果图1）*



*（效果图2）*

## 四、实验心得体会

这次实验由于要做预习报告，所以对于涉及概念的部分弄了比较久……好吧，其实这一次的实验的概念部分本身久很多！这次主要内容就是①鼠标移动②通往32位之路；感觉内容难度1：5……有点像Day 3的实验；主要就是花时间理解asmhead.nas里面的代码，其中遇到的难的一点就是对GDT0、GDTR0这最后一段代码的理解，其实就是两个阶段的段表，在初期还没有设计GDT时，分段是通过GDT0实现的，我们初期的实验也都是在GDT0为bootpack.c分好的段（和栈）里面运行，直到调用了GDTR0（也就是LGDT）创建了GDT，才开始使用内存中的GDT，这样便能更好地理解了。