# 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 | 2020.10.22 | 实验项目 | 第5天：结构体、文字显示与GDT/IDT初始化 | | |

## 一、实验主要内容

**1、显示字符、字符串、鼠标**

**内容：**

1、通过像素点绘制出字符的形状，实验中采用的是每个字符都在一个8\*16的像素矩阵中来表示，每一个点对应一个二进制值，1表示该点着色、0表示该点无色；这样，一个字符就要16-bytes来表示，想要在画面上显示该字符，需要循环遍历这些这个8\*16像素矩阵中的每一个像素点，然后在相应的坐标上“打点”；

2、为了更方便的显示其他字符，实验沿用OSASK的字体，通过将hankaku.txt的字体信息编译后链接到bootpack.obj，使得c程序可以通过extern符号来使用外部字体；

3、打印字字符串的函数便是利用之前的putfont8()，循环遍历字符串，一个一个打印即可；

4、绘制鼠标，定义鼠标为一个16\*16的图形，所以需要16\*16大小的矩阵来存储像素点信息，鼠标图像中有三种颜色：鼠标本色、边界色、背景色；首先在初始化鼠标函数中准备好mouse[]颜色数组，之后再通过绘图函数绘制鼠标；

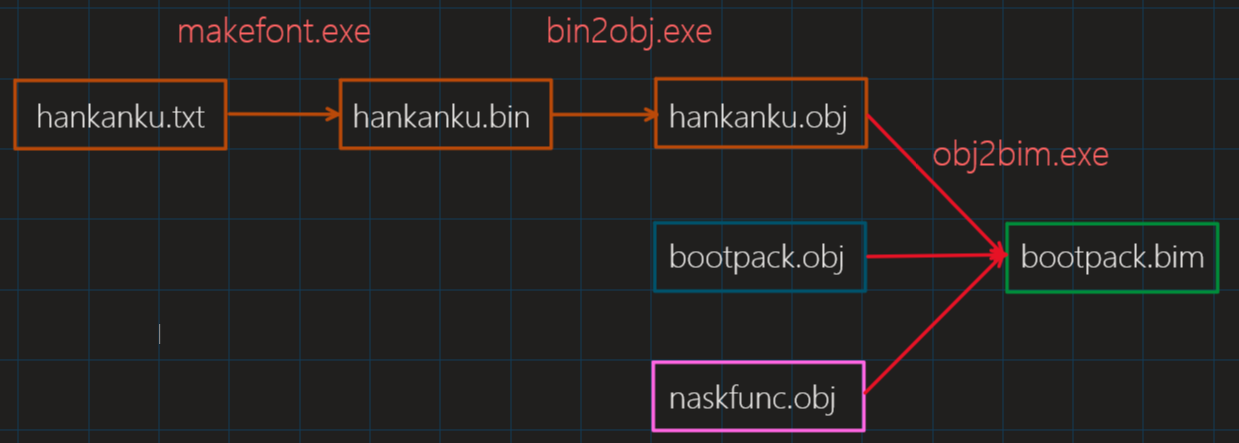
**重点总结：**

1、每一个字符在这种模式下需要16-bytes的存储空间；

2、绘制的过程实际上是对矩阵内每一个点的遍历，通过VRAM基址加上偏移值定位该点；

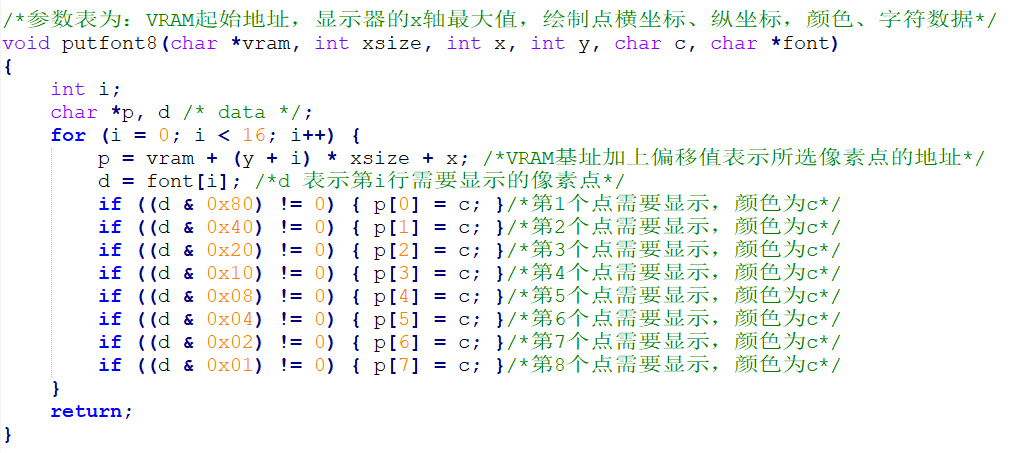
3、函数putfont8()中通过&运算来判断每一行的8个像素点中哪些被选中，注意不需要else分支，每一个点都有机会被选中；

4、hankaku.txt如何被使用？

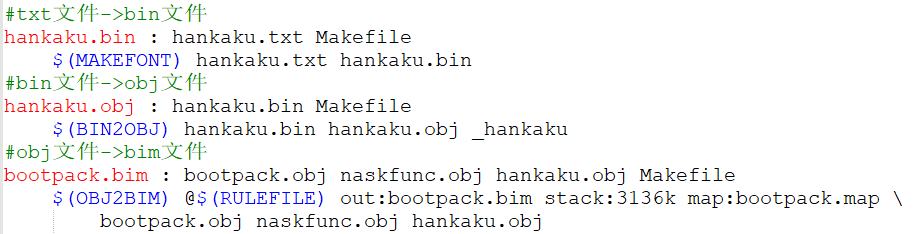


4、sprintf和putfonts8\_asc的组合使用显示变量信息，sprintf是C自带的函数，作用是将按一定格式输出的信息转化为字符串写在内存中，然后通过putfonts8\_asc打印sprintf生成的字符串，二者需要一起使用，sprintf不具有打印能力；

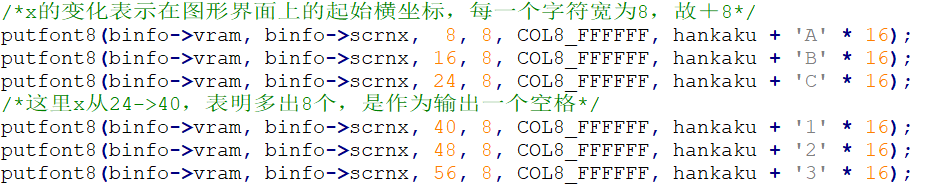
**关键代码及注释：**



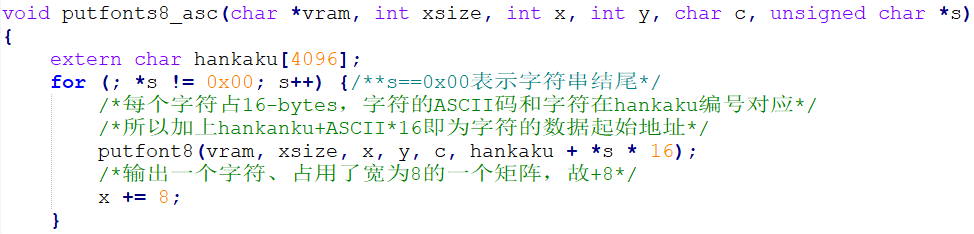
*（打印字符函数）*



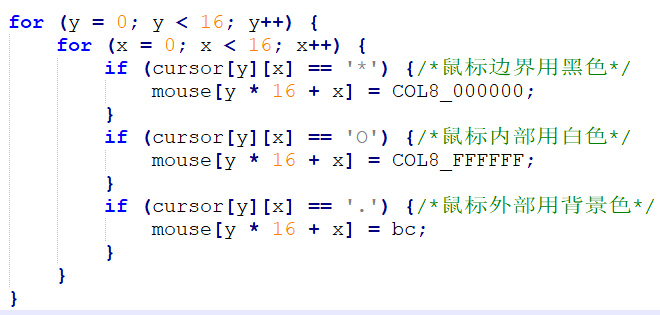
*（Makefile文件，对hankanku.txt的使用）*



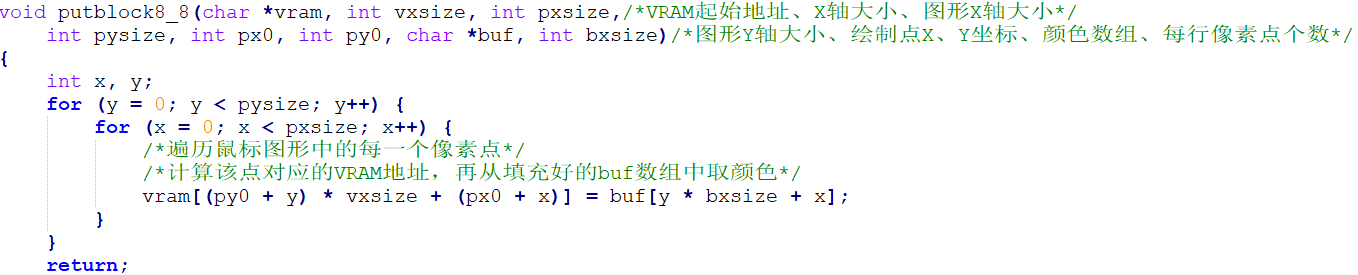
*（打印字符串“ABC 123”）*



*（打印指定字符串）*



*（设定鼠标的颜色数组）*



*（根据给定颜色数组，绘制指定大小图形）*

**2、GDT 初期**

**内容：**

1、GDT：全局段号记录表；这是用来管理内存分段的表，每一个GDT条目对应一个段，以结构体的形式，包含了该段的起始地址、大小、属性；在32-bits系统中，地址计算时是直接加上段寄存器DS的值作为基址，然后再加EBX；

2、对GDT的初始化和设置；相关函数在本实验中出现，函数说明见详细代码注释；

3、IDT：终端记录表，存放中断号和对应的中断处理函数（本次实验仅简单提及，相关实现须在GDT之后）；

**重点总结：**

1、每个段由64-bits描述：基址（32-bits分三段short int、char、char，为了与80286兼容）、段的大小（24位分为short int、char）、段属性（8-bits）；

2、由于段寄存器DS第三位不能使用，所以最多可定义8192个段，内存中需要64k-bytes来存放GDT；

3、段属性：

高四位：GD00

G是G bit，1表示段上限是以页面为单位,0以字节；

D是指段的模式,1是指32位模式,0是指16位模式；

这里出现的16位模式主要只用于运行80286的程序；

低四位：

0x00 : 未使用的记录表(descriptor table)0x92 (1001 0010): 系统专用，可读写的段、不可执行；0x9a (1001 1010): 系统专用，可执行的段、可读不可写；0xf2 (1111 0010): 应用程序用，可读写的段、不可执行；0xfa (1111 1010): 应用程序用，可执行的段、可读不可写；

4、GDTR——48位寄存器

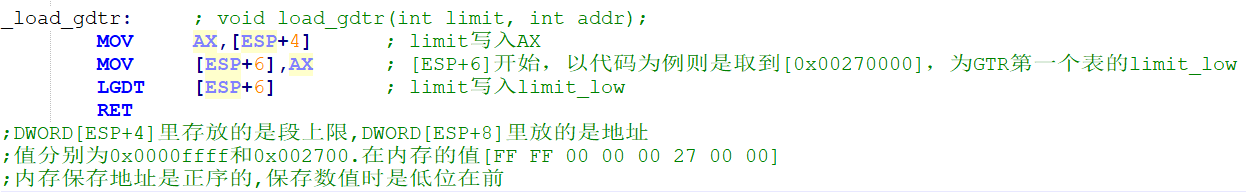
前16位：段上限，它等于"GDT的有效字节数-1"，偶尔会用到上限这个词，意思都是表示量的大小，一般为"字节数-1"；

后32位：代表GDT的开始地址；

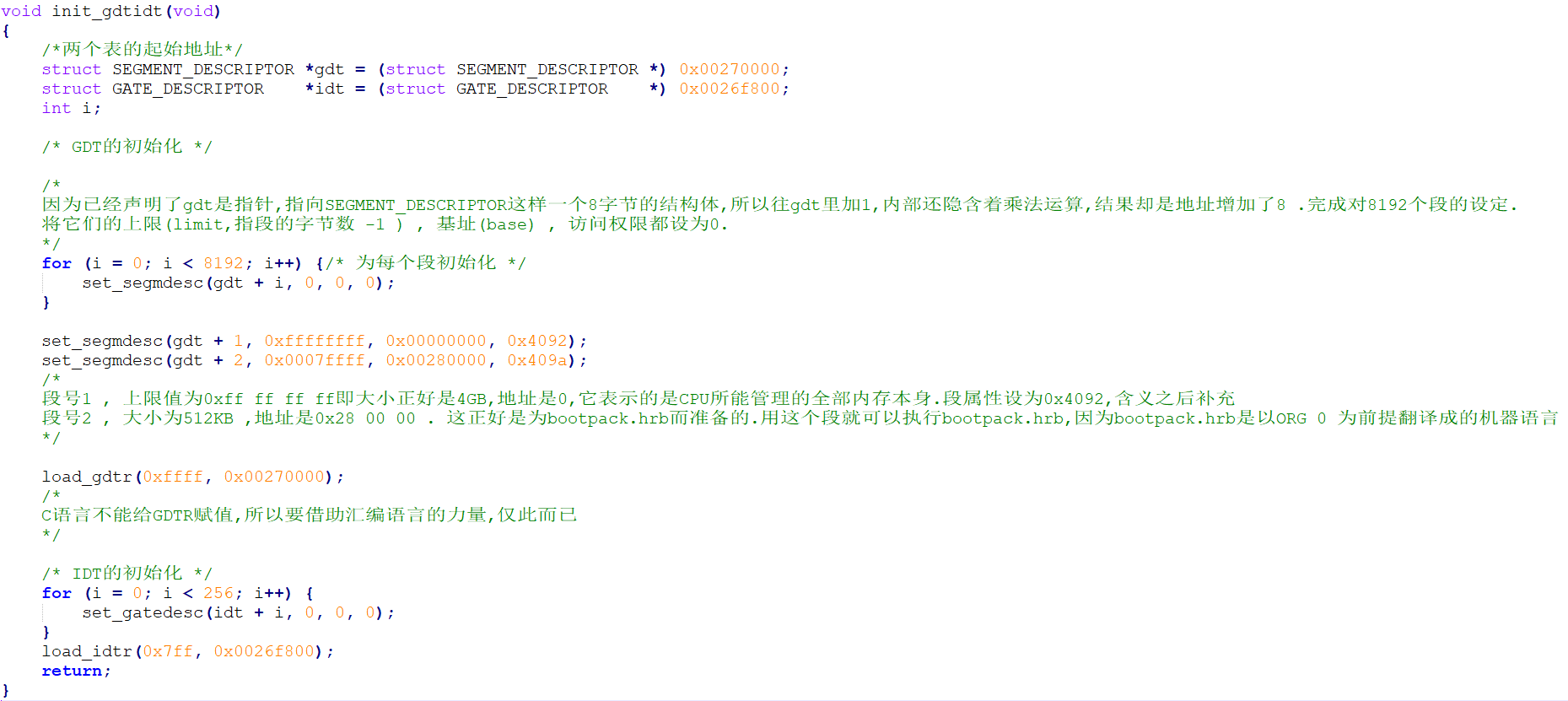
5、LGDT指令

GDTR的值不能用MOV指令来赋值。给它赋值的时候，唯一的办法就是制定一个内存的地址，从制定的内存地址读取6个字节，然后赋值给GDTR寄存器。完成这一任务的指令，就是LGDT。

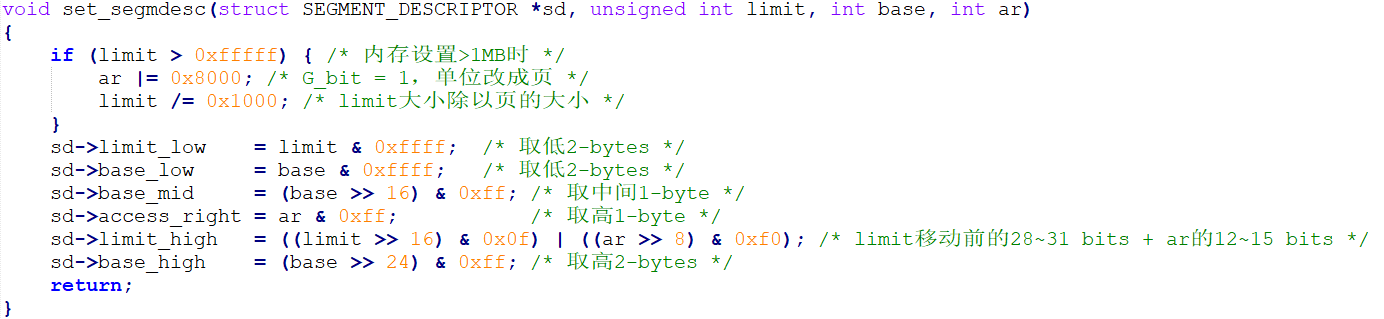
**关键代码及注释：**



*（load\_gdtr函数注释）*



*（GDT&IDT初始化函数）*

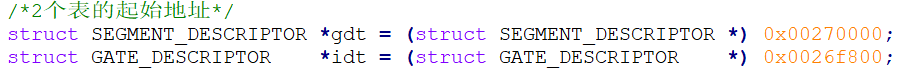


*(设置段表条目)*

## 二、遇到的问题及解决方法

1、内存地址以及装载时的相关指令问题

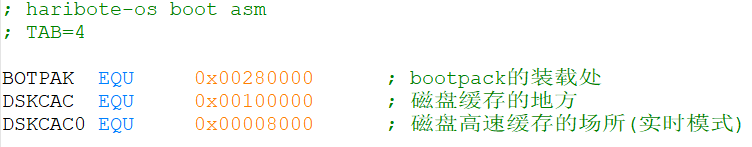
问题描述：相关内存装载地址不清楚



出现原因：

教材中说GDT、IDT的初始化是笔者随意的决定，地址值本身没有特殊的意义，只是在内存分布图中观察到这一段地址没有被占用；随后提到了0x280000~0x2fffff是属于bootpack.h的地方，对这个声明有点陌生，因为连续两天的实验都没有涉及到详细的地址问题了，按照提示打开了asmhead.nas；

理解： 在asmhead.nas文件开始可以看到，声明了几个宏定义：

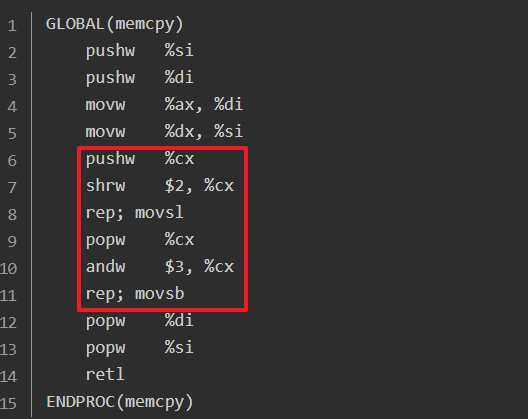


可以看到，直接将0x28000作为bootpack的预装载地址，具体的装载动作呢？



可以看到，传输指令先把bootpack的起始地址传给了源址寄存器%ESI、把要装载到的内存起始地址给了目的寄存器%EDI；之后向%ECX中写入了一个512\*1024/4的数据，现在来分析一下这个数据；

512\*1024/4，明显的bootpack作为启动区，只有一个扇区，所以为512KB，所以前半部分表示的是文件的字节总数，那/4是为什么？参考以下Linux中void \*memcpy(void \*dest, const void \*src, size\_t n)的函数实现：



关注红框内的代码：首先获取%cx（放的是传入的字节数n），然后逻辑右移两位（n/4），之后每次通过movsl每次都拷贝4个字节到目的地址中，rep表示一直执行；看到这便明白了，汇编中的/4是预处理，毕竟我们使用的memcpy没有传参，参数需要预先设定好在寄存器中，到这里问题就算解决了，但有的人可能会问——万一文件大小不是4的倍数呢？看红框中剩余的代码，当从rep;movsl跳出来后，说明剩余字节数<4，把%cx取出来AND 0x11，就取得了%cx的最低两位，也就是剩余字节数，之后rep;movsb，一个一个字节取出剩余字节即可！

## 三、程序设计创新点

1、绘制一个新字符

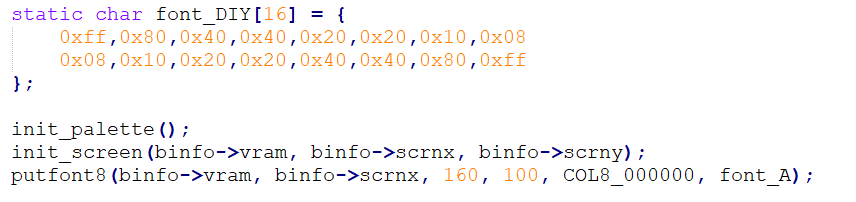
创新点：

根据教材中提到的方法，在8\*16的像素矩阵中，选取一部分点，使得能够绘制出Σ的符号；矩阵数据如下：

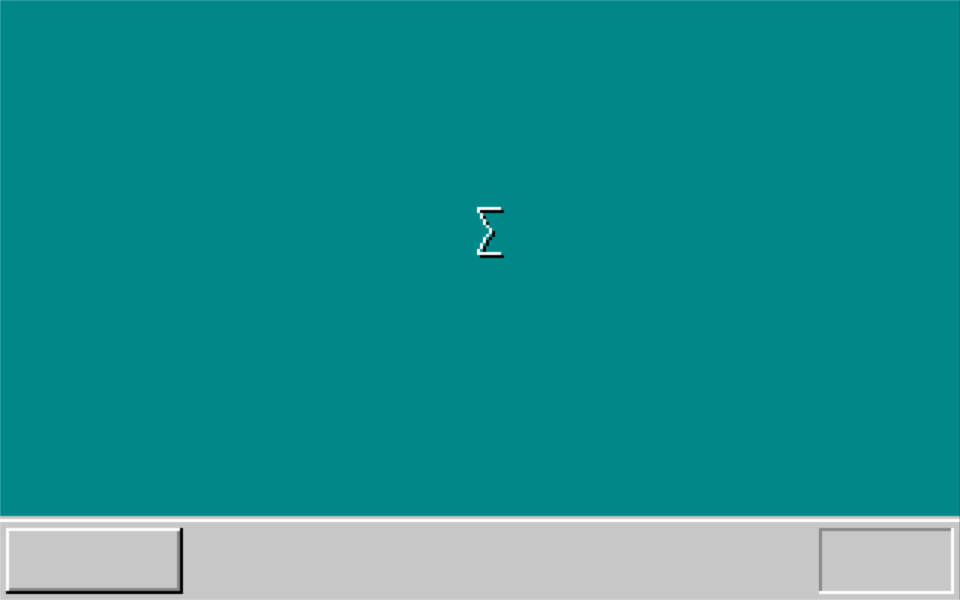
**0xff,0x80,0x40,0x40,0x20,0x20,0x10,0x08**

**0x08,0x10,0x20,0x20,0x40,0x40,0x80,0xff**

**关键代码：**



**结果截图：**



## 四、实验心得体会

这次的实验内容主要是对字符的显示，由于不能直接使用现成的打印函数，所以要自己从底层实现，这一过程让我对打印输出有了更清楚的理解；对于鼠标的实现，这次先只是绘制出了一个形状，并不能真正的移动；此外，引入了GDT和IDT，对二者的概念、底层数据结构做了初步介绍，涉及到了初始化和设置函数，这些内容在Day5中没有解释太多，详细的内容放在了Day6里面，确实想要一股脑的把详细的地址、属性这些东西弄清楚还是很难的，这次遗留了一些问题，发现在Day6里面有要求详细的理解，便把问题放一放了。