|  |  |  |  |
| --- | --- | --- | --- |
| **武汉大学国家网络安全学院教学实验报告** | | | |
| 课程名称 | 操作系统及安全设计 | 实验日期 | 2019/10/18 |
| 实验名称 | 认识保护模式（一） | 实验序号 | 二 |
| 1. 实验目的及实验内容   （本次实验所涉及并要求掌握的知识；实验内容；必要的原理分析） | | | |
| **实验目的：**  理解x86架构下的段式内存管理  掌握实模式和保护模式下段式寻址的组织方式、  关键数据结构、代码组织方式  掌握实模式与保护模式的切换  掌握特权级的概念，以及不同特权之间的转移  **实验内容：**  1. 认真阅读章节资料，掌握什么是保护模式，弄清关键数据结构：  GDT、descriptor、selector、GDTR， 及其之间关系，阅读  pm.inc文件中数据结构以及含义，写出对宏Descriptor的分析  2. 调试代码，/a/ 掌握从实模式到保护模式的基本方法，画出代码  流程图，如果代码/a/中，第71行有dword前缀和没有前缀，编  译出来的代码有区别么，为什么，请调试截图。  3. 调试代码，/b/，掌握GDT的构造与切换，从保护模式切换回实  模式方法  4. 调试代码，/c/，掌握LDT切换  5. 调试代码，/d/掌握一致代码段、非一致代码段、数据段的权限  访问规则，掌握CPL、DPL、RPL之间关系，以及段间切换的基  本方法  6. 调试代码，/e/掌握利用调用门进行特权级变换的转移  **代码对应iso中chapter3**  **实验解决问题与课后动手改：**  1. GDT、Descriptor、Selector、GDTR结构，及其含义是什么？他  们的关联关系如何？pm.inc所定义的宏怎么使用？  2. 从实模式到保护模式，关键步骤有哪些？为什么要关中断？为  什么要打开A20地址线？从保护模式切换回实模式，又需要哪些  步骤？  3. 解释不同权限代码的切换原理，call, jmp，retf使用场景如何，  能够互换吗？  4. 课后动手改：  1. 自定义添加1个GDT代码段、1个LDT代码段，GDT段内要对一个内  存数据结构写入一段字符串，然后LDT段内代码段功能为读取并打  印该GDT的内容；  2. 自定义2个GDT代码段A、B，分属于不同特权级，功能自定义，要  求实现A-->B的跳转，以及B-->A的跳转。 | | | |
| 1. 实验环境及实验步骤   （本次实验所使用的器件、仪器设备等的情况；具体的实验步骤） | | | |
| **实验环境：**  VMwareWorkstationPro 15.5.0  Ubuntu 12.04.5 desktop i386 32位  bochs 2.6.9  **关键技术：**   1. bochs使用 2. 实模式，保护模式及其关键数据结构GDT,LDT，Descriptor、Selector等 3. 特权级变换   **实验步骤:**  **1.认真阅读章节资料，掌握什么是保护模式，弄清关键数据结构：**  **GDT、descriptor、selector、GDTR， 及其之间关系，阅读**  **pm.inc文件中数据结构以及含义，写出对宏Descriptor的分析**  GDT即为Global Descriptor Table（全局描述符表）又叫段描述符表,为保护模式下的一个数据结构。其中包含多个descriptor，定义了段的起始地址，界限属性等。  descriptor为段描述符，包含段基址，段界限，段属性。其结构如图    Selector为选择子，有其数据结构。在pmtest1.asm程序中，其作用就是偏移，对应描述符相对于GDT基址的偏移。    GDTR为GDT寄存器。结构与GDTPTR类似，6字节，前两字节GDT界限，后4字节GDT基地址。    四者关系：  GDT中包含多个descriptor，descriptor包含段的信息，包含段基址，界限属性等。多个selector包含对应descriptor相对于GDT的偏移，于是selector发挥了类似 指向descriptor的作用。而GDTR中包含了GDT基地址与界限。四者综合就可以获得某个descriptor的地址。而保护模式下寻址就先靠GDTR找到GDT，然后根据descriptor找到对应段的地址，然后再加上段内偏移offset，就得到某个线性地址。  如图所示    对宏Descriptor分析：  结构如图：    共8字节。从低地址开始前两字节为段界限1，然后三个字节为段基址1，然后两个字节byte5,byte6包含段属性以及段界限2，最后一字节为段基址2.由于历史原因，段界限和段基址都分开存放。程序中descriptor由pm.inc中的宏descriptor生成。  代码：  %macro Descriptor 3 ;macro定义宏。 3表示有三个参数  dw %2 & 0FFFFh ; 段界限1  dw %1 & 0FFFFh ; 段基址1  db (%1 >> 16) & 0FFh ; 段基址2  dw ((%2 >> 8) & 0F00h) | (%3 & 0F0FFh) ; 属性1 + 段界限2 + 属性2  db (%1 >> 24) & 0FFh ; 段基址3%endmacro ; 共 8 字节  macro代表宏开始。宏名Descriptor，3代表有三个参数。  参数1-3分别为段基址，界限，属性。  比如LABEL\_DESC\_VIDEO: Descriptor 0B8000h, 0ffffh, DA\_DRW ; 显存首地址  利用宏Descriptor定义了基址为0B8000H的段LABEL\_DESC\_VIDEO.  0B8000H为显存首地址。利用该段在屏幕中显示数据。  之后第一行dw 为两字节。 %2 & 0FFFFh， 相当于取段界限的低位，写入这两字节。  然后dw，dd去段基址1,2，构成三字节段基址，相当于上面结构图的段基址1.  然后dw两字节构成段属性，段界限2.  然后dw两字节构成段基址3.  其中段基址为该段起始地址，界限为长度。  **2. 调试代码，/a/ 掌握从实模式到保护模式的基本方法，画出代码**  **流程图，如果代码/a/中，第71行有dword前缀和没有前缀，编**  **译出来的代码有区别么，为什么，请调试截图。**  流程图：pmtest1.asm 用文字描述如下  （1）定义GDT [SECTION .gdt]  其中定义了一个空descriptor，一个32位代码段，一个显存descriptor  其中32位代码段只初始化了段界限，段属性  （2）进入[SECTION .s16] 16位代码段（实模式）  修改GDT值：修改32位段描述符值  将LABEL\_SEG\_CODE32的物理地址（即 [SECTION .s32]这个段的物理地址）赋给eax，然后把它分成三部分赋给描述符DESC\_CODE32中的相应位置。由于DESC\_CODE32的段 界限和属性已经指定，所以至此，DESC\_CODE32的初始化全部完成。  **（将段寄存器段界限段属性由符合实模式要求到符合保护模式要求）**  之后赋值gdtr寄存器：  把GDT的物理地址填充到了GdtPtr这个6字节的数据结构中。  **lgdt** [GdtPtr] 将GdtPtr指示的6字节加载到寄存器gdtr  之后关中断。  之后打开A20地址线。  修改cr0寄存器：PE位置1。  此时cs的值仍然是实模式下的值，把代码段的选择子装入cs：  **jmp** dword SelectorCode32:0 ，进入32位代码段[SECTION .s32]  （3）进入32位代码段[SECTION .s32]  进行屏幕显示操作。  **调试代码a：**  将程序编译为.com文件，使用dos运行。（因为引导扇区只有512字节，程序高于512字节就不方便了）  代码a有dword前缀调试：   1. 准备freedocs.img 2. bximage生成pm.img 3. 修改bochs     重点是  floppya: 1\_44=freedos.img, status=inserted  floppyb: 1\_44=pm.img, status=inserted  boot: a   1. 用bochs格式化B盘   Sudo bochs  在dos format b:     1. 修改pmtest1，org改为0100h，并编译为pmtest1.com        1. 将pmtest1.com复制到pm.img   sudo mount -o loop pm.img /mnt/floppy  会出现了错误  mount point /mnt/floppy does not exist  先创建文件夹IMG_256  然后 sudo losetup /dev/loop0 pm.img  创建loop设备，然后操作loop设备，就是对pm.img数据的操作了  IMG_257  sudo mount /dev/loop0/ /mnt/floppy loop设备挂载到/mnt/floppy上  IMG_258  然后   sudo cp pmtest1.com /mnt/floppy/ 赋值  然后卸载  sudo umount /mnt/floppy/  IMG_259  之后再做一次遇到问题  IMG_260   解决，卸载  IMG_261   另外发现了 sudo cp pmtest2.com /mnt/floppy/ 赋值并不是覆盖。也就是说cp了先cp了pmtest1.com，然后不格式化（format b:），直接cp  pmtest2.com，那么两个程序都可以运行。   1. 在dos下运行pmtest1.com   Sudo bochs  B:\pmtest1.com 运行  可见右侧出现一个红色的P    代码a无dword前缀调试：   1. 修改pmtest1.asm，删掉第71行的dword，存为pmtestd.asm，并编译为pmtestd.com      1. 在dos运行   陷入循环并且无红色的P在屏幕右侧    失败原因：  jmp dword SelectorCode32:0 ; 执行这一句会把 SelectorCode32 装入 cs,  //selector16位，dword两字节，高位selector，低位偏移0.(因为声明了这段是16位代码，所以一个字两字节) ; 并跳转到 Code32Selector:0 处  删除dword 后只有16位。cs寄存器没有正确设置，没有跳转到32位代码段，故显示失败  **3. 调试代码，/b/，掌握GDT的构造与切换，从保护模式切换回实**  **模式方法**  **分析：pmtest2.asm**  在前面程序的基础上，新建一个段，这个段以5MB为基址，远远超出实模式下1MB的界限。我们  先读出开始处8字节的内容，然后写入一个字符串，再从中读出8字节。如果读写成功的话，两次读出的内容应该是不同的，而且第  二次读出的内容应该是我们写进的字符串。字符串是保存在数据段中的，也是新增加的。  （1）LABEL\_DESC\_STACK: Descriptor 为全局堆栈段[SECTION .gs]的descriptor，初始化在[SECTION .gs]和[SECTION.16]完成。Descriptor属性为DA\_DRWA+DA\_32，DA\_32表明是32位堆栈段。  （2）LABEL\_DESC\_DATA:Descriptor 为[SECTION .data1] ; 数据段的descriptor，初始化在[SECTION .data1] 完成，其中包含了要写入的字符串  （3）LABEL\_DESC\_CODE32: Descriptor 为32位代码段（保护模式）[SECTION .s32]. 由实模式跳入.  在[SECTION .s32]中我们改变了ss和esp（代码3.5第174行到177行），这样，在32位代码段中所有的堆栈操作将会在新增的 堆栈段中进行。  这个段的开头初始化了ds、es和gs，让ds指向新增的数据段，es指向新增的5MB内存 的段，gs指向显存（第167行到第172行）。接着显示一行字符串，之后就开始读写大地址内存了（第198行到第200行）。由于要读 两次相同的内存，我们把读的过程写进一个函数TestRead，写内存的内容也写进函数TestWrite，这两个函数的入口分别在第206行 和第222行。可以看到，在TestRead中还调用了DispAL和DispReturn这两个函数（第253行和第286行），DispAL将al中的字节用十 六进制数形式显示出来，字的前景色仍然是红色；DispReturn模拟一个回车的显示，实际上是让下一个字符显示在下一行的开头 处。要注意的一个细节是，在程序的整个执行过程中，edi始终指向要显示的下一个字符的位置。所以，如果程序中除显示字符外 还用到edi，需要事先保存它的值，以免在显示时产生混乱。  （4）保护模式中字符串寻址：在TestWrite中用到一个常量OffsetStrTest，它的定义在代码3.4第47行。注意，我们用到这个字符串的时候并没有用直接标 号StrTest，而是又定义了一个符号OffsetStrTest，它等于StrTest-$$。$$的含义代表当前 节（section）开始处的地址。所以StrTest-$$表示字符串StrTest相对于本节的开始处（即LABEL\_DATA处）的偏移。容易发现数据段的基址便是LABEL\_DATA的物理地址。于是OffsetStrTest既是字符串相对LABEL\_DATA的偏移，也是其在数据段中的偏移。我们在保护模式下需要用到的正是这个偏移，而不再是实模式下的地址。前文中提到过的section的一点妙用指 的便是这里的$$，它不是没有替代品，而是这样做思路会比较清晰。OffsetPMMessage的情形与此类似。  **（6）返回实模式**  **概述：**  **先回忆开中断：**加载寄存器，之后关中断。之后打开A20地址线。修改cr0寄存器：PE位置1。此时cs的值仍然是实模式下的值，把代码段的选择子装入cs（修改段界限，段属性。）  **关中断差不多就是完成上述的逆向操作：**  加载一个合适的描述符选择子到有关段寄存器，以使对应段描述符高速缓冲寄存器中含有合适的段界限和属性，重新设置各个段寄存器的值，比如cr0PE位置0.恢复sp（堆栈指针寄存器）的值，修改段界限，段属性，然后关闭A20，打开中断，重新回到原来的样子。  **（将段寄存器段界限段属性由符合保护模式要求到符合实模式要求）**  为了能从保护模式恢复实模式的寄存器，需要先保存到系统自己的堆栈段。在[SECTION.16]中完成。  mov sp, 0100h  ...  然后32位代码段的操作在自定义的堆栈段[SECTION .STACK]完成。二者互不干扰，方便了恢复。  **详述：**  从实模式进入保护模式时直接用一个跳转就可以了，但是返回的时候却稍稍复杂一些。因为在准备结束保护模式回到实模 式之前，需要加载一个合适的描述符选择子到有关段寄存器，以使对应段描述符高速缓冲寄存器中含有合适的段界限和属性。而 且，我们**不能从32位代码段返回实模式，只能从16位代码段中返回**。**这是因为无法实现从32位代码段返回时cs高速缓冲寄存器中的 属性符合实模式的要求（实模式不能改变段属性）。**  所以，在这里，我们新增一个Normal描述符（代码3.4第15行）。在返回实模式之前把对应选择子SelectorNormal加载到ds、 es和ss，就是上面所说的这个原因。  LABEL\_DESC\_NORMAL: Descriptor对应选择子SelectorNormal。对应段 [SECTION .s16code]，16 位代码段. 由 32 位代码段跳入, 跳出后到实模式。  这个段是由[SECTION .s32]中的jmp SelectorCode16:0跳进来的。开头的语句把 SelectorNormal赋给ds、es、fs、gs和ss，完成我们刚刚提到的使命。然后就清cr0的PE位，接下来的跳转看上去好像不太对，因 为段地址是0。其实这里只是暂时这样写罢了，在程序的一开始处可以看到代码3.8中的这几句。  67 **mov** ax, cs  ...  73 **mov** [LABEL\_GO\_BACK\_TO\_REAL+3], ax  mov [LABEL\_GO\_BACK\_TO\_REAL+3], ax的作用就是为回到实模式的这个跳转指令指定正确的段地址，这条指令的机器码如图3.9 所示。    图3.9告诉我们，LABEL\_GO\_BACK\_TO\_REAL+3恰好就是Segment的地址，而第73行执行之前ax的值已经是实模式下的cs（我们记 做cs\_real\_mode）了，所以它将把cs保存到Segment的位置，等到jmp指令执行时，它已经不再是：  **jmp** 0:LABEL\_REAL\_ENTRY  而变成了：  **jmp** cs\_real\_mode:LABEL\_REAL\_ENTRY  它将跳转到标号LABEL\_REAL\_ENTRY处。  在跳回实模式之后，程序重新设置各个段寄存器的值，恢复sp的值，然后关闭A20，打开中断，重新回到原来的样子  144 LABEL\_REAL\_ENTRY: ; 从保护模式跳回到实模式就到了这里  ...  159 **int** 21h ; / 回到 *DOS*  **调试：**  编译pmtest2.asm为pmtest2.com  在bochs dos 下运行    结果如图    第一行为开始内存5MB处全是零。然后写入了41,42，...48，也就是16进制的A,B,C,D...H，在代码pmtest2.asm中DATA段的写入的str。  同时看到，程序执行结束后不再像上一个程序那样进入死循环，而是重新出现了DOS提示符。这说明我们重新回到了实模式下  的DOS。   1. **调试代码，/c/，掌握LDT切换**   **分析：**  LDT与GDT都是描述符table，L代表Local，局部。简单来说，LDT是一种描述符表，与GDT差不多，只不过它的选择子的TI位必 须置为1。在运用它时，需要先用lldt指令加载ldtr，lldt的操作数selector是GDT中用来描述LDT的描述符。（也就是说LDT相当于GDT中描述的一个段，对应有特殊的寄存器ldtr，而该段中又有一些描述符描述一些LDT段，只属于这个LDT。）  pmtest3.asm中增加了两个节[SCTION .ldt][SECTION .la]。（原来有omtest2.asm中的各个段）。其中[SCTION .ldt]在GDT中有对应的descriptor和selector LABEL\_DESC\_LDT: 。而[SECTION .la]是LDT描述的段，在GDT无定义。  [SCTION .ldt]是增加的LDT,其中有一个descriptor，对应[SECTION .la]。  [SECTION .la]中包含显示的字符L，在屏幕显示。实现时调用了GDT中 的SelectorVideo。  **转换到LDT的过程：**先由实模式跳转到GDT中的32位代码段[SECTION .s32](保护模式)，然后在[SECTION .s32]中  mov ax, SelectorLDT  lldt ax  加载ldtr（成为当前LDTR），  然后。jmp SelectorLDTCodeA。因为SelectorLDTCodeA的TI位为1，所以系统从当前LDT寻找相应描述符。跳转到LDT中descriptor描述的段[SECTION .la]显示L后，然后jmp SelectorCode16:0，跳回GDT中描述的16位代码段，然后返回实模式。其中SelectorLDT在GDT中定义，指向LDT地址。  [SECTION .s32]第217行到第220行，指令lldt，功能和lgdt也差不多， 负责加载ldtr，它的操作数是一个选择子，这个选择子对应的就是用来描述LDT的那个描述符（标号LABEL\_DESC\_LDT）。  本例用到的LDT中只有一个描述符（标号LABEL\_LDT\_DESC\_CODEA处），这个描述符跟GDT中的描述符没什么分别。选择子却不一样，多出了一个属性SA\_TIL。可以在pm.inc中找到它的定义：  **SA\_TIL EQU 4**  由图3.5可知，SA\_TIL将选择子SelectorLDTCodeA的TI位置为1。实际上，这一位便是区别GDT的选择子和LDT的选择子的关键所在。如果TI被置位，那么系统将从当前LDT中寻找相应描 述符。也就是说，当代码3.10中用到SelectorLDTCodeA时，系统会从LDT中找到LABEL\_LDT\_DESC\_CODEA描述符，并跳转到相应的段中。  这个LDT很简单，只有一个代码段。我们还可以在其中增加更多的段，比如数据段、堆栈段等，这样一来，我们可以把一个单独的任务所用到的所有东西封装在一个LDT中。  通过几个简单的例子，我们对IA32的分段机制大致已经有所了解了。“保护模式”中“保护”二字到底是什么含义？ 在描述符中段基址和段界限定义了一个段的范围，对超越段界限之外的地址的访问是被禁止的，这无疑是对段的一种保护。另外，有点复杂的段属性作为对一个段各个方面的定义规定和限制了段的行为和性质，从功能上来讲，这仍然是一种保护。  **调试：**  编译pmtest3.asm为pmtest3.com，在dos运行    **5. 调试代码，/d/掌握一致代码段、非一致代码段、数据段的权限**  **访问规则，掌握CPL、DPL、RPL之间关系，以及段间切换的基**  **本方法**  **分析：**   1. 特权级   在IA32的分段机制中，特权级总共有4个特权级别，从高到低分别是0、1、2、3。数字越小表示的特权级越大，较为核心的代码和数据，将被放在特权级较高的层级中。处理器将用这样的机制来避免低特权级的任务在不被 允许的情况下访问位于高特权级的段。如果处理器检测到一个访问请求是不合法的，将会产生常规保护错误（#GP）。     1. CPL,DPL,RPL   CPL是存寄存器如CS中,  RPL是代码中根据不同段跳转而确定,以动态刷新CS里的CPL.  DPL是在GDT/LDT描述符表中,静态的。 **一致代码段:**  　简单理解，就是操作系统拿出来被共享的代码段,可以被低特权级的用户直接调用访问的代码。通常这些共享代码，是"不访问"受保护的资源和某些类型异常处理。比如一些数学计算函数库,为纯粹的数学运算计算，被作为一致代码段。 **一致代码段的限制作用：** 特权级高的程序不允许访问特权级低的数据:核心态不允许调用用户态的数据.  特权级低的程序可以访问到特权级高的数据.但是特权级不会改变:用户态还是用户态. **非一致代码段:** 为了避免低特权级的访问而被操作系统保护起来的系统代码. **非一致代码段的限制作用** 只允许同级间访问.  绝对禁止不同级访问:核心态不用用户态.用户态也不使用核心态.   通常低特权代码必须通过"门"来实现对高特权代码的访问和调用。不同级别代码段之间转移规则,是通过CPL/RPL/DPL来校验。先来理解这几个概念。  **CPL（Current PrivilegeLevel）**  CPL是当前执行的程序或任务的特权级。它被存储在cs和ss的第0位和第1位上。在通常情况下，CPL等于代码所在的段的 特权级。当程序转移到不同特权级的代码段时，处理器将改变CPL。  在遇到一致代码段时，情况稍稍有点特殊，一致代码段可以被相同或者更低特权级的代码访问。当处理器访问一个与 CPL特权级不同的一致代码段时，CPL不会被改变。  **DPL（Descriptor Privilege Level）**  DPL表示段或者门的特权级。它被存储在段描述符或者门描述符的DPL字段中，正如我们先前所看到的那样。当当前代码段试图访问一个段或者门时，DPL将会和CPL以及段或门选择子的RPL相比较，根据段或者门类型的不同，DPL将会被区别 对待，下面介绍一下各种类型的段或者门的情况。  **数据段**：DPL规定了可以访问此段的最低特权级。比如，一个数据段的DPL是1，那么只有运行在CPL为0或者 1的程序才有权访问它。  **非一致代码段**（不使用调用门的情况下）：DPL规定访问此段的特权级。比如，一个非一致代码段的特 权级为0，那么只有CPL为0的程序才可以访问它。  调用门：DPL规定了当前执行的程序或任务可以访问此调用门的最低特权级（这与数据段的规则是一致的）。  **一致代码段和通过调用门访问的非一致代码段**：DPL规定了访问此段的最高特权级。比如，一个一致代 码段的DPL是2，那么CPL为0和1的程序将无法访问此段。  TSS：DPL规定了可以访问此TSS的最低特权级（这与数据段的规则是一致的）。（TSS 全称task state segment，是在操作系统进程管理的过程中，任务（进程）切换时的任务现场信息。）    **RPL（Requested PrivilegeLevel）**  RPL是通过段选择子的第0位和第1位表现出来的。处理器通过检查RPL和CPL来确认一个访问请求是否合法。即便提出访问请求的段有足够的特权级，如果RPL不够也是不行的。也就是说，如果RPL的数字比CPL大（数字越大特权级越低）， 那么RPL将会起决定性作用，反之亦然。  操作系统过程往往用RPL来避免低特权级应用程序访问高特权级段内的数据。当操作系统过程（被调用过程）从一个应用程序（调用过程）接收到一个选择子时，将会把选择子的RPL设成调用者的特权级。于是，当操作系统用这个选择子 去访问相应的段时，处理器将会用调用过程的特权级（已经被存到RPL中），而不是更高的操作系统过程的特权级（CPL）进行特权检验。这样，RPL就保证了操作系统不会越俎代庖地代表一个程序去访问一个段，除非这个程序本身是有权限的。  **例子：**  的数据段的选择子的RPL改为3：  SelectorData **equ** LABEL\_DESC\_DATA-LABEL\_GDT+SA\_RPL3  再运行一下，发生了什么？  Bochs重启了，系统崩溃了，在控制台你能看到这样的字样：  load\_seg\_reg(DS): RPL & CPL must be <= DPL  容易理解，崩溃的原因在于我们违反了特权级的规则，用RPL=3的选择子去访问DPL=1的段，于是引起异常。而我们又没有相应 的异常处理模块，于是最为严重的情况就发生了。   1. 不同特权级代码段间转移   程序从一个代码段转移到另一个代码段之前，目标代码段的选择子会被加载到cs中。作为加载过程的一部分，处理器将会检查描述符的界限、类型、特权级等内容。如果检验成功，cs将被加载，程序控制将转移到新的代码段中，从eip指示的位置开始执 行。  程序控制转移的发生，可以是由指令jmp、call、ret、sysenter、sysexit、int n 或iret引起的，也可以由中断和异常机制 引起。  使用jmp或call指令可以实现下列4种转移：  1. 目标操作数包含目标代码段的段选择子。  2. 目标操作数指向一个包含目标代码段选择子的调用门描述符。  3. 目标操作数指向一个包含目标代码段选择子的TSS。  4. 目标操作数指向一个任务门，这个任务门指向一个包含目标代码段选择子的TSS。  这4 种方式可以看做是两大类，一类是通过jmp和call的直接转移（上述第1种），另一类是通过某个描述符的间接转移（上述 第2、3、4种）。下面就来分别看一下。   1. 通过jmp或call直接转移   如果目标是非一致代码段，要求CPL必须等于目标段的  DPL，同时要求RPL小于等于DPL；如果目标是一致代码段，则要求CPL大于或者等于目标段的DPL，RPL此时不做检查。当转移到一致  代码段中后，CPL会被延续下来，而不会变成目标代码段的DPL。也就是说，通过jmp和call所能进行的代码段间转移是非常有限  的，对于非一致代码段，只能在相同特权级代码段之间转移。遇到一致代码段也最多能从低到高，而且CPL不会改变。如果想自由  地进行不同特权级之间的转移，显然需要其他几种方式，即运用门描述符或者TSS。   1. 基本的调用门进行段转移（先不涉及特权级转换，用门特权级转换见6./e/）   门：门也是一种描述符，门描述符的结构如图3.13    可以看到，门描述符和我们前面提到的描述符有很大不同，它主要是定义了目标代码对应段的选择子、入口地址的偏移和一些 属性等。可是，虽然这样的结构跟代码段以及数据段描述符大不相同，我们仍然看到，第5个字节（BYTE5）却是完全一致的，都表 示属性。在这个字节内，各项内容的含义与前面提到的描述符也别无二致，这显然是必要的，以便识别描述符的类型。在这里，S 位将是0  直观来看，一个门描述了由一个选择子和一个偏移所指定的线性地址，程序正是通过这个地址进 行转移的。门描述符分为4种：  调用门（Call gates）  中断门（Interrupt gates）  陷阱门（Trap gates）  任务门（Task gates）  其中，中断门和陷阱门是特殊的调用门，将会在后面提到，我们先来介绍调用门。在这个例子中，我们用到调用门。为简单起见，先不涉及任何特权级变换，而是先来关注它的工作方法。  在pmtest3.asm的基础上修改为pmtest4.asm  增加一个代码段作为通过调用门转移的目标段  添加[SECTION .sdset]：调用selectvideo在屏幕上显示C。因为打算用call指令调用将要建立的调用门，所以，在这段代码的结尾处调用了一个retf指令。  然后加入该段的descriptor以及selector,并初始化    然后添加调用门的descriptor以及selector  使用宏GATE（在pm.inc定义）初始化门的descriptor  SelectorCodeDest就是这个调用门要调用的段的selector，也就是我们刚刚在上面定义的段的selector        然后就准备好了要被调用的段以及调用门  下面进行调用  Call 测试调用门后retf，相当于继续运行，从235行开始继续。  调用门准备就绪，它指向的位置是SelectorCodeDest:0，即标号LABEL\_SEG\_CODE\_DEST处的代码  用一个call指令来使用这个调用门是个好主意 ：  233 ; 测试调用门（无特权级变换），将打印字母*'C'*  ⇒ 234 **call** SelectorCallGateTest:0  ...  241 **jmp** SelectorLDTCodeA:0 ; 跳入局部任务，将打印字母*'L'*。  这个call指令被放在进入局部任务之前，由于我们新加的代码以指令retf结尾，所以最终代码将会跳回 到call指令的下面继续执行。所以，我们最终看到的结果应该是在pmtest3.exe执行结果的基础上多出一个红色的字母C。  其实调用门本质上只不过是个入口地址，只是增加了若干的属性而已。在我们的例子中所用到的调用门完全等同于一个地址，我们甚至可以把使用调用门进行跳转的指令修改为跳转到调用门内指定的地址的指令：  **call** SelectorCodeDest:0  运行一下，效果是完全相同的。（下面是更复杂的情况）  （6）使用调用门进行转移时特权级检验的规则。  假设我们想由代码A转移到代码B，运用一个调用门G，即调用门G中的目标选择子指向代码B的段。实际上，我们涉及了这么几个要素：CPL、RPL、代码B的DPL（记做DPL\_B）、调用门G的DPL（记做DPL\_G）。根据3.2.3.1中提到的，A访问G这个调用门时，规则相当于访问一个数据段，要求CPL和RPL都小于或者等于DPL\_G。换句话说，CPL和RPL需在更高的特权级上。  除了这一步要符合要求之外，系统还将比较CPL和DPL\_B。如果是一致代码段的话，要求DPL\_B≤CPL；如果是非一致代码段的话，call指令和jmp指令又有所不同。在用call指令时，要求DPL\_B≤CPL；在用jmp指令时，只能是DPL\_B=CPL。  综上所述，调用门使用时特权检验的规则如表所示。    也就是说，通过调用门和call指令，可以实现从低特权级到高特权级的转移，无论目标代码段是一致的还是非一致的。  **调试：**  编译pmtest4.asm为pmtest4.com，在dos运行  在pmtest3.asm的基础上又多显示了C。是调用门调用的段的输出    **6.调试代码，/e/掌握利用调用门进行特权级变换的转移**  分析：   1. 跳转与堆栈   通过调用门和call指令，可以实现从低特权级到高特权级的转移，无论目标代码段是一致的还是非一致的。 那么如何进行高特权级向低特权级转换？  有特权级变换的转移的复杂之处，不但在于严格的特权级检验，还在于特权级变化的时候，堆栈也要发生变化。处理器的这种 机制避免了高特权级的过程由于栈空间不足而崩溃。而且，如果不同特权级共享同一个堆栈的话，高特权级的程序可能因此受到有意或无意的干扰。  在我们的程序中，指令call DispReturn和call SelectorCodeDest:0显然不同。与在实模式下类似，如果一个调用或跳转指 令是在段间而不是段内进行的，那么我们称之为“长”的（Far jmp/call），反之，如果在段内则是“短”的（Near jmp/call）。 （与windows不同）  那么长的和短的jmp或call有什么分别呢？对于jmp而言，仅仅是结果不同罢了，短跳转对应段内，而长跳转对应段间；而call 则稍微复杂一些，因为call指令是会影响堆栈的，长调用和短调用对堆栈的影响是不同的。我们下面的讨论只考虑32位的情况.  对于短调用来说，call指令执行时下一条指令的eip压栈，到ret指令执行时，这个eip会被从堆栈中弹出，如图所示。  先从右向左压栈参数，然后压栈下一条指令eip，（从高地址到低地址压栈）eip寄存器存储着我们cpu要读取指令的地址**每次cpu执行都要先读取eip寄存器的值，然后定位eip指向的内存地址。Esp是当前堆栈的指针寄存器，指向当前堆栈的底部位置。**    可以看出，调用者的eip被压栈，而在此之前参数已经入栈。图中的“调用者  eip”对应nop指令地址。而在函数foo调用最后一条指令ret（带有参数）返回之前和之后，堆栈的变化如图所示。可见esp指向的内存中，存放着call后下一条指令的地址（nop）    长调用的情况与此类似，容易想到，返回的时候跟调用的时候一样也是“长”转移，所以返回的时候也需  要调用者的cs，于是call指令执行时被压栈的就不仅有eip，还应该有cs，如图所示。    带参数的ret指令执行前后的情形如图所示。     1. 通过调用门进行特权级转换   call一个调用门也是长调用，情况跟上面 所说的长调用差不多。可是由于一些原因堆栈发生了切换，也就是说，call指令执行前后的堆栈已经 不再是同一个。我们在堆栈A中压入参数和返回时地址，等到需要使用它们的时候堆栈已经变成B了。Intel提供了这样一种机制，将堆栈A的诸多内容复制到堆栈B中，如图所示。    事实上，由于每一个任务最多都可能在4个特权级间转移，所以，每个任务实际上需要4个堆栈。可 是，我们只有一个ss和一个esp，那么当发生堆栈切换，我们该从哪里获得其余堆栈的ss和esp呢？这里涉及一样TSS（Task-State Stack），它是一个数据结构，里面包含多个字段，32位TSS如图所示。    可以看出，TSS包含很多个字段，但是在这里，我们只关注偏移4到偏移27的3个ss和3个esp。当发生堆栈切换时，内层的ss和 esp就是从这里取得的。  比如，我们当前所在的是ring3，当转移至ring1时，堆栈将被自动切换到由ss1和esp1指定的位置。由于只是在由外层到内层 （低特权级到高特权级）切换时新堆栈才会从TSS中取得，所以TSS中没有位于最外层的ring3的堆栈信息。  新堆栈的问题已经解决，下面就是CPU在整个过程中所做的工作：  1. 根据目标代码段的DPL（新的CPL）从TSS中选择应该切换至哪个ss和esp。  2. 从TSS中读取新的ss和esp。在这过程中如果发现ss、esp或者TSS界限错误都会导致无效TSS异常（#TS）。  3. 对ss描述符进行检验，如果发生错误，同样产生#TS 异常。  4. 暂时性地保存当前ss和esp的值。  5. 加载新的ss和esp。  6. 将刚刚保存起来的ss和esp的值压入新栈。  7. 从调用者堆栈中将参数复制到被调用者堆栈（新堆栈）中，复制参数的数目由调用门中Param Count一项来决定。如果 Param Count是零的话，将不会复制参数。  8. 将当前的cs和eip压栈。  9. 加载调用门中指定的新的cs和eip，开始执行被调用者过程。  在第7步中，解释了调用门中Param Count的作用，Param Count只有5位，也就是说，最多只能复制31个参数。如果参数多于31个该怎么办呢？这时可以让其中的某个参数变成指向一 个数据结构的指针，或者通过保存在新堆栈里的ss和esp来访问旧堆栈中的参数。  此刻结合TSS结构和上述步骤，可以理解通过调用门进行由外层到内层调用的全过程。那么，正如call指令对 应ret，调用门也面临返回的问题。通过长短call和ret的堆栈变化这两组对比，我们发现，ret基本上是call的反过程，只  是带参数的ret指令会同时释放事先被压栈的参数。  实际上，ret这个指令不仅可以实现短返回和长返回，而且可以实现带有特权级变换的长返回。由被调用者到调用者的返回过 程中，处理器的工作包含以下步骤：  1. 检查保存的cs中的RPL以判断返回时是否要变换特权级。  2. 加载被调用者堆栈上的cs和eip（此时会进行代码段描述符和选择子类型和特权级检验）。  3. 如果ret指令含有参数，则增加esp的值以跳过参数，然后esp将指向被保存过的调用者ss和esp。注意，ret的参数必须 对应调用门中的Param Count 的值。  4. 加载ss和esp，切换到调用者堆栈，被调用者的ss和esp被丢弃。在这里将会进行ss描述符、esp以及ss段描述符的检验。  5. 如果ret指令含有参数，增加esp的值以跳过参数（此时已经在调用者堆栈中）。  6. 检查ds、es、fs、gs的值，如果其中哪一个寄存器指向的段的DPL小于CPL（此规则不适用于一致代码段），那么一个空描述符会被加载到该寄存器。  如图所示    综上所述，使用调用门的过程实际上分为两个部分，一部分是从低特权级到高特权级，通过调用门和call指令来实现；另一部  分则是从高特权级到低特权级，通过ret指令来实现。   1. 进入ring3   在ret指令执行前，堆栈中应该已经准备好了目标代码段的cs、eip，以及ss和esp，另外，还可能有参数。这些可以是处理器压入栈的，也可以由我们自己压栈。在我们的例子中，在ret前的堆栈如图3.22所示。    这样，执行ret之后就可以转移到低特权级代码中了。在（pmtest4.asm）基础上做一下修改（形成 pmtest5a.asm）。如上面的图3.22所示，我们至少要添加一个ring3的代码段和一个ring3的堆栈段。   1. pmtest5a.asm 由ring0到ring3转移   **首先，我们之前的代码都运行在ring0！**  添加一个ring3代码段[SECTION .ring3]，一个ring3堆栈段[SECTION .s3]  这个ring3代码段非常简单，跟[SECTION .la]和[SECTION .sdest]的内容差不多，同样是打印一个字符。  需要注意，由于这段代码运行在ring3，而在其中由于要写显存而访问到了VIDEO段，为了不会产生错误，我们把VIDEO段的DPL 修改为3。  25 LABEL\_DESC\_VIDEO: Descriptor 0B8000h, 0ffffh, DA\_DRW + DA\_DPL3  第392行让程序不再继续执行。392 **jmp** $  之所以这样做，是为了先验证一下由ring0到ring3的转移是否成功。如果屏幕上出 现红色的3，并且停住不动，不再返回DOS，则说明转移成功。  新段对应的描述符LABEL\_DESC\_CODE\_RING3的属性加上了DA\_DPL3，让它的DPL变成了3  相应选择子SelectorCodeRing3的SA\_RPL3将RPL也设成了3。  同时有堆栈段的descriptor LABEL\_DESC\_STACK3以及selector SelectorStack3，以及初始化，在此略去。  这样，代码段和堆栈段都已经准备好了。让我们将ss、esp、cs、eip依次压栈，并且执行retf指令。  266 **push** SelectorStack3  267 **push** TopOfStack3  107268 **push** SelectorCodeRing3  269 **push** 0  270 **retf**  此段代码放在显示完字符串“In Protect Mode now.”后立即执行。  编译，运行。  会看到了红色的3在“In Protect Mode now.”下方显示。在这表明我们由ring0到ring3的转移成功完成。   1. pmtest5b.asm 在ring3中使用调用门   修改pmtest4中提到的调用门的selectorSelectorCallGateTest以及descriptorLABEL\_CALL\_GATE\_TEST:的DPL，RPL  然后修改[SECTION .ring3]代码，在死循环前添加  **call** SelectorCallGateTest:0。  修改描述符和选择子是为了满足CPL和RPL 都小于等于调用门DPL的条件。  编译运行  出现错误。因为从低特权级到高特权级转移的时候，需要用到 TSS。   1. pmtest5c.asm 添加TSS,在ring3中使用调用门   因为从低特权级到高特权级转移的时候，需要用到 TSS，在pmtest5c.asm中准备一个TSS  TSS作为数据结构有其descriptor LABEL\_DESC\_TSS，selector SelectorTSS以及段[SECTION .TSS]。定义及初始化见代码  可以看出，除了0级堆栈之外，其他各个字段我们都没做任何初始化。因为在本例中，我们只用到这一部分。  添加初始化TSS描述符的代码之后，TSS就准备好了，我们需要在特权级变换之前加载它  311 **call** DispReturn  312  ⇒ 313 **mov** ax, SelectorTSS  ⇒ 314 **ltr** ax  315  316 **push** SelectorStack3  317 **push** TopOfStack3  318 **push** SelectorCodeRing3  319 **push** 0  320 **retf**  之后编译运行，成功。显示call调用门的C以及ring3段的3.   1. pmtest5.asm 返回实模式   到目前为止，我们已经成功实现了两次从高特权级到低特权级以及一次从低特权级到高特权级的转移（ring0-ring3-ring-0-ring3，ring0打印“In protect mode”，然后到ring3打印3，然后ring3callgate到ring0打印L，然后返回ring3），最终在低特权级的代码[SECTION .ring3] 中让程序停住。我们已经具备了在各种特权级下进行转移的能力，并且熟悉了调用门这种典型门描述符的用法。  为了让我们的程序能够顺利地返回实模式，我们将调用局部任务的代码加入到调用门的目标代码（[SECTION .sdest]）。最后，程序将由这里进入局部任务，然后经由原路返回实模式。（ring3打印3，调用门，调用门打印C，调用局部任务LDT打印L，然后在局部任务jmp SelectorCode16:0返回16位代码段，之后返回实模式）  346 [**SECTION** .sdest]; 调用门目标段  347 [**BITS** 32]  ...  ⇒ 359 **mov** ax, SelectorLDT  ⇒ 360 **lldt** ax  361  ⇒ 362 **jmp** SelectorLDTCodeA:0 ; 跳入局部任务，将打印字母*'L'*。  编译运行，结果应为显示in protect mode ，3，c，l，然后返回实模式可以继续运行  **调试：**  编译为.com文件运行  pmtest5a    pmtest5b    pmtest5c    pmtest5    **7.课后手动改：**  （1）自定义添加1个GDT代码段、1个LDT代码段，GDT段内要对一个内  存数据结构写入一段字符串，然后LDT段内代码段功能为读取并打印该GDT的内容；  参考pmtest3.com  修改[SECTION .data1]，修改字符串为StrTest: db "JUST MONIKA", 0  修改[SECTION .s32]; 32 位代码段. 由实模式跳入.  改为如下，相当于直接跳到LDT中的descriptor  .........................  [SECTION .s32]; 32 位代码段. 由实模式跳入.  [BITS 32]  LABEL\_SEG\_CODE32:  ; Load LDT  mov ax, SelectorLDT  lldt ax  jmp SelectorLDTCodeA:0 ; 跳入局部任务  SegCode32Len equ $ - LABEL\_SEG\_CODE32  ; END of [SECTION .s32]  ........................  修改LDT中的段; CodeA (LDT, 32 位代码段)[SECTION .la]  功能改为显示GDT中[SECTION .DATA]段的字符串StrTest  ........................  ; CodeA (LDT, 32 位代码段)  [SECTION .la]  ALIGN 32  [BITS 32]  LABEL\_CODE\_A:  mov ax, SelectorData  mov ds, ax ; 数据段选择子  mov ax, SelectorVideo  mov gs, ax ; 视频段选择子  mov ax, SelectorStack  mov ss, ax ; 堆栈段选择子  mov esp, TopOfStack  ; 下面显示一个字符串  mov ah, 0Ch ; 0000: 黑底 1100: 红字  xor esi, esi  xor edi, edi  mov esi, OffsetStrTest ; 源数据偏移  mov edi, (80 \* 10 + 0) \* 2 ; 目的数据偏移。屏幕第 10 行, 第 0 列。  cld  .1:  lodsb  test al, al  jz .2  mov [gs:edi], ax  add edi, 2  jmp .1  .2: ; 显示完毕  jmp SelectorCode16:0  CodeALen equ $ - LABEL\_CODE\_A  ; END of [SECTION .la]  .........................  然后编译运行。运行时顺序为实模式跳转保护模式[SECTION .s32]，然后[SECTION .s32]加载LDT的ldtr，然后跳转LDT的[SECTION .la]段，该段中先在屏幕显示[SECTION .DATA]段的字符串StrTest，然后跳回实模式  代码保存为pmtestmy.asm，编译为pmtestmy.com。  编译    运行如图，说明成功    （2）自定义2个GDT代码段A、B，分属于不同特权级，功能自定义，要  求实现A-->B的跳转，以及B-->A的跳转。  参考pmtest5,实现了ring0->ring3->ring0->ring3d的跳转 | | | |
| 1. 实验过程分析   （实验分工，详细记录实验过程中发生的故障和问题，进行故障分析，说明故障排除的过程及方法。根据具体实验，记录、整理相应的数据表格等） | | | |
| **遇到的问题及解决：**   1. 将asm文件编译为.com，copy到pm.img失败   sudo mount -o loop pm.img /mnt/floppy  会出现了错误  mount point /mnt/floppy does not exist  解决：  先创建文件夹  IMG_256  然后 sudo losetup /dev/loop0 pm.img  创建loop设备，然后操作loop设备，就是对pm.img数据的操作了  IMG_257  sudo mount /dev/loop0/ /mnt/floppy loop设备挂载到/mnt/floppy上  IMG_258  然后   sudo cp pmtest1.com /mnt/floppy/ 赋值  然后卸载  sudo umount /mnt/floppy/  IMG_259  之后再做一次遇到问题  IMG_260   解决，卸载  IMG_261   另外发现了 sudo cp pmtest2.com /mnt/floppy/ 赋值并不是覆盖。也就是说cp了先cp了pmtest1.com，然后不格式化（format b:），直接cp  pmtest2.com，那么两个程序都可以运行。  **实验结果记录：**  pmtest1.com 实模式到保护模式    pmtest2.com GDT的构造与切换，从保护模式切换回实  模式方法    pmtest3.asm LDT切换    pmtest4.asm 段间切换基本方法（call gate）    pmtest5a ring0到ring3转移    pmtest5b ring3调用gate（低级到高级），未添加TSS    pmtest5c ring3调用gate（低级到高级），添加TSS    pmtest5 返回实模式 | | | |