动手写操作系统笔记

本文的编译的codes在IA32平台上

ubuntu14.04安装环境bochs，按照这个文章可以成功安装bochs，记得安装nasm

http://blog.csdn.net/time4/article/details/25079417

**第一章 软盘模拟启动P21**

当PC上电后，如果是从软盘启动，计算机就会检查软盘的0面0磁道1扇区，sector[510]=0x55，且sector[511]=0xAA结束，则BIOS认为它是一个引导扇区，也就是我们说的Boot Sector。当然一个正确的Boot Sector除了以0xAA55结束之外，还应该包含一段少于512B的执行码。

一旦BIOS发现了Boot Sector，就会将这512B的内容装在到内存的0000:7c00处(所以后面的代码开始会定义在0x7c00位置)，然后跳转到0000:7c00处将控制权彻底交给这段引导代码。到此为止，计算机不再由BIOS中固有的程序来控制，而变成操作系统的一部分来控制。

**代码**

chapter1/a/boot.asm

org 07c00h ; 告诉编译器程序加载到7c00处

mov ax, cs ; ds es两个段寄存器指向与cs相同的段，以便在以后进行数据操作的时候能定位到正确的位置

mov ds, ax

mov es, ax

call DispStr ; 调用显示字符串例程

jmp $ ; 无限循环

DispStr:

mov ax, BootMessage

mov bp, ax ; ES:BP = 串地址

mov cx, 16 ; CX = 串长度

mov ax, 01301h ; AH = 13, AL = 01h

mov bx, 000ch ; 页号为0(BH = 0) 黑底红字(BL = 0Ch,高亮)

mov dl, 0

int 10h ; 10h 号中断

ret

BootMessage: db "Hello, OS world!"

times 510-($-$$) db 0 ; 填充剩下的空间，使生成的二进制代码恰好为512字节

dw 0xaa55 ; 结束标志

(1)在NASM中，任何不被方括号[]括起来的标签或变量名都被认为是地址

foo dw 1

mov ax, foo 把foo的地址传给ax

mov bx, [foo] 把bx的值赋为1

(2)关于$和$$，$表示当前行被汇编的地址，$$一个节section的汇编后的起始地址，由于本程序只有一个section，那么实际上它表示程序被编译后的开始地址

ndisasm -o 0x7c00 boot.bin >> disboot.asm

00007C09 EBFE jmp short 0x7c09

$ = 0x7c09

$$ = 0x7c00

times 510-($-$$) db 0 ; 重复510-($-$$)遍，填充0x00,也就是说程序到这里有510B大小，加上结束标志正好512B

**配置文件**

~/MyOS/bochs-work/chapter1/a/bochsrc

**命令**

bximage，选fd，然后一路回车

nasm boot.asm -o boot.bin

dd if=boot.bin of=a.img bs=512 count=1 conv=notrunc

bochs，即可出现如下截图



**第三章 保护模式(Protect Mode) P57**

**3.1认识保护模式P57**

开头给了一段程序，bochs-work/chapter3/a/pmtest1.asm，这段代码主要分三大块儿：

1. [SECTION .gdt]定义一些宏

2. [SECTION .s16]一块16位代码段，处在实模式中，然后跳转到32位代码段

3. [SECTION .s32]一块32位代码段，由实模式跳入，最后打印给红色的字母P

编译一下

nasm pmtest1.asm -o pmtest1.com

本文是在WIN7下安装的DOSBox环境中运行，如下图所示



图3-1 pmtest1.com执行结果

通过上面的介绍，我们了解到的内容如下：

● 程序定义了一个叫做GDT的数据结构

● 16位代码段进行了一些与GDT有关的操作

● 程序最后跳到32位代码中做了一点操作显存的工作

那么问题来了

● GDT是什么？它是干什么用的？

● 程序对GDT做了什么？

● jmp dword SelectorCode32:0 跟我们以前用过的jmp有什么不同？

**3.1.1 GDT(Global Descriptor Table) P62**

在IA32下，CPU有两种工作模式：实模式和保护模式。当我们打开自己的PC，开始时CPU是工作在实模式下的，经过某种机制之后，才进入保护模式。在保护模式下，CPU有着巨大的寻址能力，并为强大的32位操作系统提供了更好的硬件保障。类比与新旧政策的更替。

旧政策，Intel 8086是16位的CPU，16位的Register，16位的Data Bus，20位的Address Bus和1MB的寻址能力。

物理地址(Physical Address) = 段值(Segment)\*16 + 偏移(Offset)

其中，段值和偏移都是16位的。

从80386开始，Intel家族的CPU进入32位时代。80386有32位地址线，所以寻址空间可以达到4GB。所以，单从寻址这方面来说，使用16位寄存器的方法已经不够用了。这时候，我们需要新的方法来提供更大的寻址能力，当然能猜到的是保护模式也不仅仅在这一方面。

在实模式下，16位寄存器需要用“段:偏移”这种方法才能达到1MB的寻址能力。新政策下，地址线仍然用“SEG:OFFSET”这样形式表示，只不过保护模式下“段”的概念发生了根本性的变化。实模式下，段值还是可以看做是地址的一部分的；而在保护模式下，虽然段值仍然由原来的16位的cs、ds等寄存器表示，但此时它仅仅变成了一个索引，这个索引指向一个数据结构的一个表项，表项中详细定义了段的起始地址、界限、属性等内容。这个数据结构就是GDT(实际上还可能是LDT，这个以后说明)。GDT中的表项也有一个专门的名字，叫做描述符(Descriptor)。

也就是说，GDT的作用是用来提供段式存储机制，这种机制是通过段寄存器和GDT中的描述符共同组成。

图3-2 代码段和数据段描述符

由图3-2可以看出来，段描述符定义一个段的基址和界限，除了BYTE5和BYTE6中的一堆属性看着有点复杂，先不去管它。

本例的GDT中共有3个描述符，为方便起见，分别成为DESC\_DUMMY、DESC\_CODE32和DESC\_VIDEO。其中DESC\_VIDEO = 0b8000h，指向的是显存。

在[SECTION.s32]中有两行代码

mov ax, SelectorVideo

mov gs, ax

其中 SelectorVideo的定义是

SelectorVideo equ LABEL\_DESC\_VIDEO - LABEL\_GDT

直观地看，它好像是DESC\_VIDEO这个描述符相对于GDT基址的偏移。实际上，它有一个专门的名称，叫做选择子(Selector)，它也不是一个偏移，而是稍稍复杂一些，它的结构如图3-3所示。



图3-3 选择子(Selector)的结构

不难理解，当TI和RPL都为零时，选择子就变成了对应，描述符相对于GDT基址的偏移。

在程序快结束的位置，你可以找到这一行：

mov [gs:edi], ax

看到这里，大家肯定已经明白了，gs的值是SelectorVideo，它对应DESC\_VIDEO，这条指令把ax的值写入显存中偏移位edi的位置。

我们可以看一下示意图，注意这只是个示意图，真实情况比这个复杂的多。

下图3-4可以看出，“段:偏移”形式的逻辑地址(Logical Address)经过段机制转化成“线性地址”(Linear Address)，而不是“物理地址”(Physical Address)，其中的原因以后会提到。在上面的程序中，线性地址就是物理地址。另外，包含描述符的，不仅可以是GDT，也可以是LDT。

这样，我们分析了整个程序两个部分，第一部分宏定义和第三部分[SECTION .s32]，剩下第二部分[SECTION .s16],它的任务一定就是从实模式向保护模式跳转了。下面我们就看一下实模式如何转换到保护模式的。

**3.1.2 实模式到保护模式，不一般的jmp P65**

对应这代码，先看一下初始化 32 位代码段描述符，先将LABEL\_SEG\_CODE32的物理地址赋给eax，然后把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。

接下来的动作把GDT的物理地址填充到了GdtPtr这个6字节的数据结构中，然后执行了指令

lgdt [GdtPtr]

这一句的作用是将GdtPtr指示的6字节加载到寄存器gdtr，gdtr的结构如图3-5所示



下面是关中断，之所以是关中断，是因为保护模式下中断处理的机制是不同的，不关掉中断将会出现错误。

再下面几句的作用是打开A20地址线，那么什么是A20呢？这又是一个历史问题。8086只有20位地址总线，只能寻址到1MB，那么如果试图访问超过1MB的地址时，系统会回卷回去(wrap-around)，重新从地址零开始寻址。可是，到了80286时，真的可以访问1MB以上的内存了，

那么就不应该回卷寻址，为了保证向上兼容，IBM相出一个办法，使用键盘控制器(8042芯片)来控制第20个(从零开始数)地址位，这就是A20地址线，如果不被打开，第20个地址位将会总是零。

那么，原文中通过操作端口92h将A20地址线打开，注意方法并不是唯一的。

然后是把寄存器cr0的第0位置为1,这一位是决定实模式和保护模式的关键



如图3-6,cr0的第0位是PE位，此位为0时，CPU运行于实模式，为1时，CPU运行于保护模式。但是，此时cs的值仍然是实模式下的值，我们需要把代码段的选择子转入cs，所以，我们需要jmp指令：

jmp dword SelectorCode32:0

至此，新政策代替旧政策，我们进入保护模式。

不过，这个jmp比看起来还要复杂一点，因为它不得不放在16位的段中，目标地址却是32位的。从这一点看，它是混和16位和32位的代码。所以，这个jmp跟一般的jmp是很不相同的，直接这样写是不严谨的

jmp SelectorCode32:0；错误！

因为偏移地址应该是32位的，这样编译出来的只是16位的代码。假设目标地址的偏移不是0,而是一个32位的值，比如jmp SelectorCode32:0x12345678，则编译后偏移会被截断，只剩下0x5678。

所以，这个特殊的跳转需要特殊方法来处理。Linux采用DB指令，NASM则加一个dowrd。

至此，我们已经成功进入保护模式，下面总结一下进入保护模式的主要步骤：

(1)准备GDT

(2)用lgdt加载gdtr

(3)打开A20

(4)置cr0的PE位

(5)跳转，进入保护模式

**3.1.3 描述符属性 P67**

类似Linux的ELF描述，比如描述是否可执行。这里说一下一致代码段，他就是关于特权级别的限制，原文描述不一定完全正确，先列举一例吧，比如对于非一致代码段：

高特权级下不能调用低特权级的代码，为了避免低特权级代码获得高特权级从而能执行某些系统指令；需要使用调用门。

**3.2保护模式进阶 P70**

**3.2.1海阔凭鱼跃**

体验保护模式访问1MB以上内存的能力，关闭A20,并跳回实模式。有些地方需要注意：

代码目录bochs-work/chapter3/b/

[SECTION .data1] ; 数据段

ALIGN 32

[BITS 32]

LABEL\_DATA:

SPValueInRealMode dw 0

; 字符串

PMMessage: db "In Protect Mode now. ^-^", 0 ; 进入保护模式后显示此字符串

OffsetPMMessage equ PMMessage - $$

StrTest: db "ABCDEFGHIJKLMNOPQRSTUVWXYZ", 0

OffsetStrTest equ StrTest - $$

DataLen equ $ - LABEL\_DATA

; END of [SECTION .data1]

$$表示当前节(section)的开始处的地址。也就是说PMMessage-$$表示字符串PMMessage相对于本节的开始处(即LABEL\_DATA处)的偏移，再联系下我们定义的段基址

; 初始化数据段描述符

xor eax, eax

mov ax, ds

shl eax, 4

add eax, LABEL\_DATA

mov word [LABEL\_DESC\_DATA + 2], ax

shr eax, 16

mov byte [LABEL\_DESC\_DATA + 4], al

mov byte [LABEL\_DESC\_DATA + 7], ah

从这里我们看出，数据段的基址便是LABEL\_DATA的物理地址。于是OffsetPMMessage 既是字符串相对于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，而不在是实模式下的地址。

另外一个就是保护模式回到实模式比之前实模式跳到保护模式要复杂一些。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，程序中新增了一个Normal描述符

LABEL\_DESC\_NORMAL: Descriptor 0, 0ffffh, DA\_DRW ; Normal 描述符

在返回实模式之前把对应的选择子SelectorNormal加载到ds、es、和ss。

[SECTION .s16code]

ALIGN 32

[BITS 16]

LABEL\_SEG\_CODE16:

; 跳回实模式:

mov ax, SelectorNormal

mov ds, ax

mov es, ax

mov fs, ax

mov gs, ax

mov ss, ax

mov eax, cr0

and al, 11111110b

mov cr0, eax

LABEL\_GO\_BACK\_TO\_REAL:

jmp 0:LABEL\_REAL\_ENTRY ; 段地址会在程序开始处被设置成正确的值

Code16Len equ $ - LABEL\_SEG\_CODE16

这个段是通过[SECTION .s32]中的jmp SelectorCode16:0跳进来的，段界限和属性和处理好后，

然后就是请cr0的PE位，接下来跳转的段地址是0，我们先看下程序的开始处

[SECTION .s16]

[BITS 16]

LABEL\_BEGIN:

mov ax, cs

mov ds, ax

mov es, ax

mov ss, ax

mov sp, 0100h

mov [LABEL\_GO\_BACK\_TO\_REAL+3], ax

ax的作用就是为了回到实模式的这个跳转指令正确的段地址，这条指令的机器码如图3-7所示。



上图告诉我们，LABEL\_GO\_BACK\_TO\_REAL+3 恰好就是Segemtn的地址，而mov [LABEL\_GO\_BACK\_TO\_REAL+3], ax 这一语句ax值已经是实模式下的cs(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。

然后重新设置各个寄存器的值，恢复sp的值，然后关闭A20,打开中断，回到原来的样子。

程序运行结果如下，回车后，又返回了DOS



**3.2.2 LDT(Local Descriptor Table)**

程序中起了一个单独的任务在LDT中，LDT类似GDT，不同的是T1位置需要置1。在运用它时，需要先用lldt指令加载ldtr。步骤总结如下：

(1)增加一个32位的代码段，内容不妨尽量简单。

(2)增加一个段，内容是一个描述符表(LDT)，可以只有一个代码段描述符，也可以添加更多的描述符描述更多的段。注意，涉及的选择子的T1位是1。

(3)在GDT中增加一个描述符，用以描述这个LDT，同时要定义其描述符。

(4)增加新添的描述符的初始化代码，主要是针对段基址。

(5)用新加的LDT描述的局部任务准备完毕。

(6)使用前用lldt指令夹在ldtr，用jmp指令跳转等方式运行。

我们已经看到，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于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。下面就介绍涉及到保护的进阶知识，特权级。

**3.2.3特权级 P82**

DPL Descriptor Privilege Level，RPL Requested Privilege Level，前面所有的例子都是运行在最高特权级下，所以涉及到的DPL和RPL都是0(最高特权级)。

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。如果特权级低的任务在不被允许的情况下访问特权级高的段，将会产生常规保护错误(#GP)。



这里，由于数据越大表示的特权级越小，所以有时为避免混淆，也将高特权级称做内层，而把低特权级称做外层。

**3.2.3.1 CPL、DPL、RPL**

**1. CPL(Current Privilege Level)**

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。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，一直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。

**2. 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，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，根据段或者门类型的不同，DPL将会被区别对待，下面介绍一个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的程序才有权限访问它。

●**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，规则同数据段

●**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，规则同数据段

**3. RPL(Requested Privilege Level)**

RPL是通话段选择子的第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)，而不是更低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。

**3.2.3.2一个小试验**

复制文件pmtest3.asm → pmtest3.2.3.2.asm，修改LABEL\_DESC\_DATA的DPL属性修改为1

LABEL\_DESC\_DATA: Descriptor 0, DataLen - 1, DA\_DRW+DAPL1

继续修改，把对刚才修改过的数据段的选择子的RPL改为3:

SelectorData equ LABEL\_DESC\_DATA - LABEL\_GDT + SA\_RPL3

用DOSbox运行后，会卡住，DOSbox自己退出了，系统崩溃了。原因就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发生了。

**3.2.3.3 不同特权代码段之间的转移**

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。作为夹在过程的一部分，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、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：

● 目标操作数包含目标代码段的段选择子

● 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

● 目标操作数指向一个包含目标代码段选择子的TSS

● 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中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进行直接转移**

我们在3.1.3节中对通过jmp或call进行直接转移已经有过一些讨论，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；如果目标是一直代码段，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。

**2.调用门初体验**

门描述符的结构，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，将会在后面的章节中提到，我们先来介绍调用门。来看一个例子，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的基础上增加一个代码段作为通过调用门转移的目标段，见d/pmtest4.asm

[SECTION .sdest]; 调用门目标段

[BITS 32]

LABEL\_SEG\_CODE\_DEST:

;jmp $

mov ax, SelectorVideo

mov gs, ax ; 视频段选择子(目的)

mov edi, (80 \* 12 + 0) \* 2 ; 屏幕第 12 行, 第 0 列。

mov ah, 0Ch ; 0000: 黑底 1100: 红字

mov al, 'C'

mov [gs:edi], ax

retf

SegCodeDestLen equ $ - LABEL\_SEG\_CODE\_DEST

; END of [SECTION .sdest]

这个段的代码沿用我们以前的方法打印一个字符。我们打算用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。

现在来加入这个代码段的描述符：

LABEL\_DESC\_CODE\_DEST: Descriptor 0,SegCodeDestLen - 1, DA\_C + DA\_32

同时加入初始化这个描述符的代码：

xor eax, eax

mov ax, cs

shl eax, 4

add eax, LABEL\_SEG\_CODE\_DEST

mov word [LABEL\_DESC\_CODE\_DEST + 2], ax

shr eax, 16

mov byte [LABEL\_DESC\_CODE\_DEST + 4], al

mov byte [LABEL\_DESC\_CODE\_DEST + 7], ah

这段代码想必你已经非常熟悉了，我们每初始化一个描述符都会进行这项操作，以后再添加一个描述符时也是这样，到时为节省篇幅，类似代码将略过不提。

目标代码段的描述符如下：

SelectorCodeDest equ LABEL\_DESC\_CODE\_DEST - LABEL\_GDT

好了，现在添加调用门：

LABEL\_CALL\_GATE\_TEST: Gate SelectorCodeDest, 0, 0, DA\_386CGate + DA\_DPL0

这里，我们用了一个宏Gate来初始化这个门描述符，Gate的定义在pm.inc中可以找到：

%macro Gate 4

dw (%2 & 0FFFFh) ; 偏移 1 (2 字节)

dw %1 ; 选择子 (2 字节)

dw (%3 & 1Fh) | ((%4 << 8) & 0FF00h) ; 属性 (2 字节)

dw ((%2 >> 16) & 0FFFFh) ; 偏移 2 (2 字节)

%endmacro ; 共 8 字节

这个宏和Descriptor宏有点类似，也是将描述符的构成要素分别安置在相应的位置，使代码看起来非常清晰。

我们的门描述符的属性是DA\_386CGate，表明它是一个调用门。里面指定的选择子是SelectorCodeDest，表明目标代码段是刚刚添加的代码段。偏移地址是0,表示将跳转到目标代码段的开头处。另外，我们把DPL指定为0。

调用门对应的选择子的定义如下：

SelectorCallGateTest equ LABEL\_CALL\_GATE\_TEST - LABEL\_GDT

好了，现在我们的调用门准备就绪了，它指向的位置是SelectorCodeDest:0，即标号LABEL\_SEG\_CODE\_DEST处的代码。

我们刚刚说过，用一个call指令来使用这个调用门是个好主意：

call SelectorCallGateTest:0 ; 测试调用门（无特权级变换），将打印字母 'C'。

它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。



调用门这种听起来很可怕的东西本质上只不过是个入口地址，只是增加了若干的属性而已。其实，在我们的例子中所用到的调用门完全等同一个地址，我们甚至可以把使用调用门进行跳转的指令修改到调用门内指定的地址的指令：

call SelectorCodeDest:0

运行一下，效果完全相同的。

可是，调用门显然不是多此一举的东西，因为我们将要用它来实现不同特权级的代码之间的转移。下面我们就来介绍一下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。

除了这一步符合要求之外。系统还将比较CPL和DP\_B。如果一致代码段，需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，需DPL\_B≦CPL，在用jmp指令，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3-4所示。

表3-4 调用门特权级规则

|  |  |  |
| --- | --- | --- |
|  | call | jmp |
| 目标是一致代码段 | CPL≦DPL\_G, RPL≦DPL\_G, DPL\_B≦CPL | |
| 目标是非一致代码段 | CPL≦DPL\_G, RPL≦DPL\_G,  DPL\_B≦CPL | CPL≦DPL\_G, RPL≦DPL\_G  DPL\_B=CPL |

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。

在coding实现一个特权级变化之前，还需要学习的一件事情是堆栈，因为特权级发生变化的时候，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。

**3.回忆——关于堆栈**

在我们程序中，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)。

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3-14所示。



push param1

push param2

push param3

call foo

nop

…...

foo:

…...

ret 3

图3-14中的调用者eip对应这里的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3-15所示。

这是短调用的情况，长调用的情况也与此类似，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3-16所示。

相应地，带参数的ret指令执行前后的情形如图3-17所示。



**4.通过调用门进行有特权级变换的转移——理论篇**

上面是我们对前面内容的一点再讨论，联系当前通过调用门的转移，我们想到，call一个调用门也是长调用，情况应该跟上面所说的长调用差不多才对。可是，正如我们已经提到的，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。这样一来问题出现了，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，这该怎么办呢？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3-18所示。



这里，我们涉及到两个堆栈。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？实际上，这里涉及到一样新事物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3-19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。

好了，新堆栈的问题已经解决，就让我们看一下整个的转移过程是怎样的。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中选择应该切换至那个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(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，我们终于明白了调用门中Param Count的作用，至此，调用门中各个部分的作用不再留有疑问。要说明的是，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。

好了，此刻如果你结合图3-19和上述步骤，一定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。通过图3-14和图3-15、图3-16和图3-17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。

图3-20可以比较形象地表示出这个过程。

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。说到这里，我想你一定明白了，通过ret的指令可以实现由高特权级到低特权级的转移。

**5.进入ring3**