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x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，多任务处理的雏形。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**

我们已经知道，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，当然，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3-21所示。

这样，执行ret之后，就可以转移到低特权级代码中了。我们还是在前文所写的程序pmtest4.asm基础上做一些修改，参考chapter3/e/pmtest5.asm。我们至少要添加一个ring3的代码段和一个ring3的堆栈段。首先添加一个代码段：

; CodeRing3

[SECTION .ring3]

ALIGN 32

[BITS 32]

LABEL\_CODE\_RING3:

mov ax, SelectorVideo

mov gs, ax ; 视频段选择子(目的)

mov edi, (80 \* 14 + 0) \* 2 ; 屏幕第 14 行, 第 0 列。

mov ah, 0Ch ; 0000: 黑底 1100: 红字

mov al, '3'

mov [gs:edi], ax

jmp $

SegCodeRing3Len equ $ - LABEL\_CODE\_RING3

; END of [SECTION .ring3]

这个代码非常简单，仍然跟[SECTION .la]和[SECTION .sdest]的内容差不多，同样是打印一个字符。只是需要注意，由于这段代码运行在ring3，而在其中由于需要写显存而访问到了VIDEO段，为了不会产生错误，我们需要把VIDEO段的DPL修改为3:

LABEL\_DESC\_VIDEO: Descriptor 0B8000h, 0ffffh, DA\_DRW + DA\_DPL3 ; 显存首地址

可以看到，在现实完数字3之后，执行了一句jmp $，从而程序不再继续执行。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。

然后添加新段对应的描述符：

LABEL\_DESC\_CODE\_RING3: Descriptor 0, SegCodeRing3Len - 1, DA\_C + DA\_32 + DA\_DPL3 ; 非一致代码段, 32

对应的选择子：

SelectorCodeRing3 equ LABEL\_DESC\_CODE\_RING3 - LABEL\_GDT + SA\_RPL3

我们用SA\_RPL3将RPL也设成了3。

初始化描述符的代码与初始化其他描述符的代码类似，在此略去。

然后添加一个堆栈段：

; 堆栈段ring3

[SECTION .s3]

ALIGN 32

[BITS 32]

LABEL\_STACK3:

times 512 db 0

TopOfStack3 equ $ - LABEL\_STACK3 - 1

; END of [SECTION .s3]

它的描述符是这样的：

LABEL\_DESC\_STACK3: Descriptor 0, TopOfStack3, DA\_DRWA + DA\_32 + DA\_DPL3

选择子是这样的：

SelectorStack3 equ LABEL\_DESC\_STACK3 - LABEL\_GDT + SA\_RPL3

至此，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：

.2: ; 显示完毕

call DispReturn

push SelectorStack3

push TopOfStack3

push SelectorCodeRing3

push 0

retf ; Ring0 -> Ring3，历史性转移！将打印数字 '3'。

此段代码放在显示完字符串“In Protect Mode now.”后立即执行。

编译，运行后，我看将会看到红色的3，这表明我们由ring0到ring3的历史性转移成功完成！这是我们第一次进入不同的特权级别！

**6.通过调用门进行有特权级变换的转移——实践篇**

既然已经位于ring3中了，就让我们试验一下调用门的使用。将[SECTION.ring3]的代码稍作修改：

LABEL\_CODE\_RING3:

mov ax, SelectorVideo

mov gs, ax ; 视频段选择子(目的)

mov edi, (80 \* 14 + 0) \* 2 ; 屏幕第 14 行, 第 0 列。

mov ah, 0Ch ; 0000: 黑底 1100: 红字

mov al, '3'

mov [gs:edi], ax

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

jmp $

SegCodeRing3Len equ $ - LABEL\_CODE\_RING3

; END of [SECTION .ring3]

在进入死循环之前，我们增加了使用调用门的指令，这个调用门是我们之前定义的，可是，为了满足CPL和RPL都小于等于调用门DPL的条件，我们必须同时修改调用门：

LABEL\_CALL\_GATE\_TEST: Gate SelectorCodeDest, 0, 0, DA\_386CGate + DA\_DPL3

编译，运行。什么？出现错误？你可能想起来了，从低特权级到高特权级转移的时候，需要用到TSS，我们就来准备一个TSS：

[SECTION .tss]

ALIGN 32

[BITS 32]

LABEL\_TSS:

DD 0 ; Back

DD TopOfStack ; 0 级堆栈

DD SelectorStack ;

DD 0 ; 1 级堆栈

DD 0 ;

DD 0 ; 2 级堆栈

DD 0 ;

DD 0 ; CR3

DD 0 ; EIP

DD 0 ; EFLAGS

DD 0 ; EAX

DD 0 ; ECX

DD 0 ; EDX

DD 0 ; EBX

DD 0 ; ESP

DD 0 ; EBP

DD 0 ; ESI

DD 0 ; EDI

DD 0 ; ES

DD 0 ; CS

DD 0 ; SS

DD 0 ; DS

DD 0 ; FS

DD 0 ; GS

DD 0 ; LDT

DW 0 ; 调试陷阱标志

DW $ - LABEL\_TSS + 2 ; I/O位图基址

DB 0ffh ; I/O位图结束标志

TSSLen equ $ - LABEL\_TSS

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。

对应TSS的描述符如下：

LABEL\_DESC\_TSS: Descriptor 0, TSSLen - 1, DA\_386TSS ; TSS

选择子：

SelectorTSS equ LABEL\_DESC\_TSS - LABEL\_GDT

另外，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：

.2: ; 显示完毕

call DispReturn

; Load TSS

mov ax, SelectorTSS

ltr ax ; 在任务内发生特权级变换时要切换堆栈，而内层堆栈的指针存放在当前任务的TSS中，所以要设置任务状态段寄存器 TR。

push SelectorStack3

push TopOfStack3

push SelectorCodeRing3

push 0

retf ; Ring0 -> Ring3，历史性转移！将打印数字 '3'。

在运行，好，成功了！运行结果如图3-23所示(这里不截图了)，我们不但看到了数字3，而且看到了字母C，这表明我们在ring3下对调用门的使用也是成功的！

好了，为了让我们的程序能够顺利地返回DOS，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DOS：

LABEL\_SEG\_CODE\_DEST:

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

; Load LDT

mov ax, SelectorLDT

lldt ax

jmp SelectorLDTCodeA:0 ; 跳入局部任务，将打印字母 'L'。

;retf

SegCodeDestLen equ $ - LABEL\_SEG\_CODE\_DEST

; END of [SECTION .sdest]

编译，运行，结果如图3-24所示。

图3-24 pmtest5.com的执行结果(3)

屏幕输出同时出现C、L和3,这是程序各个部分的输出，正是我们所期望的结果。

**3.2.3.4 关于“保护”二字的一点思考**

仅仅针对前面介绍的内容，我们了解到保护模式只是有：IA32的段式存储机制，特权级之间的变换。在涉及到特权级的每一步中，处理器都会对CPL、RPL、DPL等内容进行比较，这种比较无疑是动态的，实在运行过程中进行的，是发生在多个因素之间的行为。相对而言，段描述符中的界限、属性等内容则是静态，是对某一项内容的界定和约束。

那么，我们可以得出一个概括：保护模式其实是通过这样动静相宜的方式去见证“保护”二字的含义。

**3.3页式存储 P102**

**1.什么叫做“页”**

所谓“页”，就是一块内存，在80386中，页的大小是固定的4096字节(4KB)。在Pentium中，页的大小还可以是2MB或者4MB，并且可以访问到多于4GB的内存，在此我们不予讨论。下文中，我们只讨论页大小为4KB的情况。

**2.逻辑地址、线性地址、物理地址**

在未打开分页机制时，线性地址等同于物理地址，于是可以认为，逻辑地址通过分段机制直接转换成物理地址。但当分页开启时，情况发生变化，分段机制将逻辑地址转换成线性地址，线性地址再通过分页机制转换成物理地址转换成物理地址。

**3.为什么分页**

我们看到，分段管理机制已经提供了很好的保护机制，那为什么还要加上分页管理机制呢？其实它的主要目的在于实现虚拟存储器。稍后你可以看到，线性地址中任意一个页都能映射到物理地址中的任何一个页，这无疑使得内存管理变得相当灵活。

**3.3.1分页机制概述 P103**

从图3-25中我们知道，分页机制就像一个函数：

物理地址 = F(线性地址)

我们通过图3-26来看一下这个F是怎样的。

如图3-26所示，转换使用两级页表，第一级叫做页目录，大小为4KB，存储在一个物理页中，每个表项4字节长，共有1024个表项。每个表项对应第二级的一个页表，第二级的每一个页表也有1024个表项，每一个表项对应一个物理页。页目录表的表项简称PDE(Page Directory Entry)，页表的表项简称PTE(Page Table Entry)。

进行转换时，先是由寄存器cr3指定的页目录中根据线性地址的高10位得到页表地址，然后在页表中根据线性地址的第12~21位得到物理页首地址，将这个首地址加上线性地址低12位便得到了物理地址。

分页机制是否生效的开关位于cr0的最高位PG位(参见图3-6)。如果PG=1.则分页机制生效。所以，当我们准备好了页目录和页表，并将cr3指向页目录表之后，只需要置PG位，分页机制就开始工作了。下面我们就来写一段代码检验一下。

**3.3.2编写代码启动分页机制**

为简单起见，我们在pmtest2.asm的基础进行修改，将实验内存写入和读取的描述符、代码以及数据统统去掉，并添加这样一个函数(详见chapter3/f/pmtest6.asm):

; 启动分页机制 --------------------------------------------------------------

SetupPaging:

; 为简化处理, 所有线性地址对应相等的物理地址.

; 首先初始化页目录

mov ax, SelectorPageDir ; 此段首地址为 PageDirBase

mov es, ax

mov ecx, 1024 ; 共 1K 个表项

xor edi, edi

xor eax, eax

mov eax, PageTblBase | PG\_P | PG\_USU | PG\_RWW

.1:

stosd

add eax, 4096 ; 为了简化, 所有页表在内存中是连续的.

loop .1

; 再初始化所有页表 (1K 个, 4M 内存空间)

mov ax, SelectorPageTbl ; 此段首地址为 PageTblBase

mov es, ax

mov ecx, 1024 \* 1024 ; 共 1M 个页表项, 也即有 1M 个页

xor edi, edi

xor eax, eax

mov eax, PG\_P | PG\_USU | PG\_RWW

.2:

stosd

add eax, 4096 ; 每一页指向 4K 的空间

loop .2

mov eax, PageDirBase

mov cr3, eax

mov eax, cr0

or eax, 80000000h

mov cr0, eax

jmp short .3

.3:

nop

ret

; 分页机制启动完毕 ----------------------------------------------------------

在这段代码中，相关符号的定义如下：

PageDirBase equ 200000h ; 页目录开始地址: 2M

PageTblBase equ 201000h ; 页表开始地址: 2M + 4K

…...

LABEL\_DESC\_PAGE\_DIR: Descriptor PageDirBase, 4095, DA\_DRW

LABEL\_DESC\_PAGE\_TBL: Descriptor PageTblBase, 1023, DA\_DRW | DA\_LIMIT\_4K

…...  
SelectorPageDir equ LABEL\_DESC\_PAGE\_DIR - LABEL\_GDT

SelectorPageTbl equ LABEL\_DESC\_PAGE\_TBL - LABEL\_GDT

可以看到，PageDirBase和PageTblBase是两个宏，指定了页目录表和页表在内存中的位置。

页目录表位于地址2MB处，有1024个表项，占用1024个表项，占用4KB空间，紧接着页目录表便是页表，位于2MB+4KB处。在这里，我们假定最大的可能，共有1024个页表。由于每个页表占用4096字节，所以这些页表共占用4MB空间。也就是说，本程序所需要的内存至少大于6MB。

为了逻辑清晰和代码编写简便，我们分别定义两个段，用来存放页目录表和页表，大小分别是4KB和4MB。

为了简单起见，我们的程序将所有的线性地址映射到相同的物理地址，于是线性地址和物理地址的关系符合下面的公式：

物理地址 = F(线性地址) = 线性地址

所以程序的大部分代码都是写入页目录表和页表，以便让上面的公式成立。为了完全理解写入的内容，我们需要先来看一下PDE和PTE的结构。

**3.3.3 PDE和PTE P105**

图3-27和图3-28是PDE和PTE的结构和各位详细解释，具体可参见原文。

处理器会将最近经常用到的页目录和页表项保存在TLB(Translation Lookaside Buffer)中。只有在TLB中找不到被请求页的转换信息时，才会到内存中去寻找。这样就大大加快了访问页目录和页表的时间。

当页目录或页表项被更改时，操作系统应该马上使TLB中对应的条目无效，以便下次用到此条目时让它获得更新。

当cr3被加载时，所有的TLB都会自动无效，除非页或页表条目的G位被设置。

**3.3.4 cr3**

说起cr3，我们虽然提到它指向页目录表，但并未谈起过它的结构，cr3的结构如图3-29所示。

cr3又叫做PDBR(Page-Directory Base Register)。它的高20位将是页目录表首地址的高20位。页目录表首地址的低12位会是零，也就是说，页目录表会是4KB对齐的。类似地，PDE中的页表基址(Page-Table Base Address)以及PTE中的页基址(Page Base Address)也是用高20位来表示4KB对齐的页表和页。

至于第3位和第4位的两个标志，我们暂时可以忽略它们。

**3.3.5回头看代码 P108**



先前那边代码实现后的功能如图3-30所示，完成了页目录表和所有的页表的初始化，启动分页机制。运行后，表面上没有什么不同，不过有两个问题，你可能发现了：一是页表浪费得太多了，我们可能根本没有那么大的内存；二是我们除了“实现了”分页，并没有“得益于”分页，也就是说，我们还没有体会到分页的妙处。下面就继续修改我们的程序。

**3.3.6克勤克俭用内存 P110**

在前面的程序中，我们用了4MB的空间来存放页表，并用它映射了4GB的内存空间，这显然很浪费。因为如果内存总数是16MB的话，只是页表就占用了25%的内存空间，实际上，如果仅仅是对等映射的话，16MB的内存只要4个页表就够了，所以，我们有必要知道内存有多大，然后根据内存大小确定多少页表是够用的。而且，OS也必须知道内存的容量，以便进行内存管理。

那么程序如何知道机器有多少内存呢？实际上方法不止一个，在此我们仅介绍一种通用性比较强的方法，那就是利用中断15h，具体见原文中的各种表格。

由上面的说明我们看出，eax=0000E820h的int 15h得到的不仅仅是内存的大小，还包括对不同内存段的一些描述。而且，这些描述都被保存在一个缓冲区中。所以，在我们调用int 15h之前，必须先有一块缓冲区。详见chapter3/g/pmtest7.asm

\_MemChkBuf: times 256 db 0

我们可以在每次得到一次内存描述时都使用同一个缓冲区，然后对缓冲区里的数据进行处理，也可以将每次得到的数据放进不同的位置，比如一块连续的内存，然后在想要处理它们时再读取。后一种方式可能更方便一些，所以在这里定义了一块256B的缓冲区，它最多可以存放12个20B大小的结构体。我们现在还不知道它到底够不够用，这个大小仅仅是凭猜测设定。我们将把每次得到的内存信息写入这块缓冲区，形成一个结构体数组。然后在保护模式下把它们读出来，显示在屏幕上，并且凭借它们得到内存的容量。  
 得到内存信息的代码如下所示：

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

mov [\_wSPValueInRealMode], sp

; 得到内存数

mov ebx, 0

mov di, \_MemChkBuf

.loop:

mov eax, 0E820h

mov ecx, 20

mov edx, 0534D4150h

int 15h

jc LABEL\_MEM\_CHK\_FAIL

add di, 20

inc dword [\_dwMCRNumber]

cmp ebx, 0

jne .loop

jmp LABEL\_MEM\_CHK\_OK

LABEL\_MEM\_CHK\_FAIL:

mov dword [\_dwMCRNumber], 0

LABEL\_MEM\_CHK\_OK:

可以看到，代码使用了一个循环，一旦cf被置位或者ebx为零，循环将结束。在第一次循环开始之前，eax为0000E820h，ebx为0，ecx为20，edx为0534D4150h，es:di指向\_MemChkBuf的开始处。在每一次循环进行时，寄存器di的值将会递增，每次的增量为20字节。另外eax、ecx和edx的值都不会变，ebx的值我们置之不理。同时，每次循环我们让变量\_dwMCRNumber的值加1，这样到循环结束时它的值会是循环的次数，同时也是地址范围描述符结构的个数。

好了，下面我们来到保护模式下的32位代码，添加这样一个过程：

DispMemSize:

push esi

push edi

push ecx

mov esi, MemChkBuf

mov ecx, [dwMCRNumber] ;for(int i=0;i<[MCRNumber];i++) // 每次得到一个ARDS(Address Range Descriptor Structure)结构

.loop: ;{

mov edx, 5 ; for(int j=0;j<5;j++) // 每次得到一个ARDS中的成员，共5个成员

mov edi, ARDStruct ; { // 依次显示：BaseAddrLow，BaseAddrHigh，LengthLow，LengthHigh，Type

.1: ;

push dword [esi] ;

call DispInt ; DispInt(MemChkBuf[j\*4]); // 显示一个成员

pop eax ;

stosd ; ARDStruct[j\*4] = MemChkBuf[j\*4];

add esi, 4 ;

dec edx ;

cmp edx, 0 ;

jnz .1 ; }

call DispReturn ; printf("\n");

cmp dword [dwType], 1 ; if(Type == AddressRangeMemory) // AddressRangeMemory : 1, AddressRangeReserved : 2

jne .2 ; {

mov eax, [dwBaseAddrLow] ;

add eax, [dwLengthLow] ;

cmp eax, [dwMemSize] ; if(BaseAddrLow + LengthLow > MemSize)

jb .2 ;

mov [dwMemSize], eax ; MemSize = BaseAddrLow + LengthLow;

.2: ; }

loop .loop ;}

;

call DispReturn ;printf("\n");

push szRAMSize ;

call DispStr ;printf("RAM size:");

add esp, 4 ;

;

push dword [dwMemSize] ;

call DispInt ;DispInt(MemSize);

add esp, 4 ;

pop ecx

pop edi

pop esi

ret

这段代码的主体框架被注释写成了C代码，用来帮助我们理解这部分逻辑。

程序的主体是一个循环，循环的次数为地址范围描述符结构(下文用ADRStruct代替)的个数，每次循环都会读取一个ADRStruct。首先打印其中每一个成员的各项，然后根据当前的类型，得到可以被操作系统使用的内存上限。结果会被存放在变量dwMemSize中，并在此模块的最后打印到屏幕。

为了读起来方便，DispInt、DispStr连同DispAL、DispReturn都放在了lib.inc中，通过如下语句包含：

%include "lib.inc" ; 库函数

至此，我们新增的内容已经准备得差不多了，另外还需要提到的一点是，在数据段中，几乎每个变量都有类似的两个符号，比如：

\_dwMemSize: dd 0

和

dwMemSize equ \_dwMemSize - $$

在实模式下应该使用\_dwMemSize，而在保护模式下应该使用dwMemSize。因为程序实在实模式下编译的，地址只适用于实模式，在保护模式下，数据的地址应该是其相对于段基址的偏移。

我们添加了函数DispMemSize，调用它的代码如下：

push szMemChkTitle

call DispStr

add esp, 4

call DispMemSize ; 显示内存信息

在调用它之前，我们还显示了一个字符串作为将要打印的内存信息的表格头。我们运行后，会得到如下统计结果：

|  |  |  |
| --- | --- | --- |
| 内存段 | 属性 | 是否可被OS使用 |
| 00000000h~0009FBFFh | AddressRangeMemory | 可 |
| 0009FC00h~0009FFFFh | AddressRangeReserved | 不可 |
| 000E0000h~000FFFFFh | AddressRangeReserved | 不可 |
| 00100000h~01FFFFFFh | AddressRangeMemory | 可 |
| FFFC0000h~FFFFFFFFh | AddressRangeReserved | 不可 |

从这里我们可以直观地看到，操作系统所能使用的最大内存地址为01FFFFFFh，所以此机器拥有32MB的内存。而且，幸运地，我们指定的256B的内存MemChkBuf是够用的。

你可能没有想到，得到内存容量还要这么多代码，不过，实际上我们除了得到了内存的大小，还得到了可用内存的分布信息。由于历史原因，系统可用内存分布地并不连续，所以在使用的时候，我们要根据得到的信息小心行事。

内存容量得到了，你是否还记得我们为什么要得到内存？我们是为了节约使用，不再初始化所有PDE和所有页表，现在，我们已经可以根据内存大小计算应初始化多少PDE以及多少页表，下面来修改一下函数SetupPaging：

; 启动分页机制 --------------------------------------------------------------

SetupPaging:

; 根据内存大小计算应初始化多少PDE以及多少页表

xor edx, edx

mov eax, [dwMemSize]

mov ebx, 400000h ; 400000h = 4M = 4096 \* 1024, 一个页表对应的内存大小

div ebx

mov ecx, eax ; 此时 ecx 为页表的个数，也即 PDE 应该的个数

test edx, edx

jz .no\_remainder

inc ecx ; 如果余数不为 0 就需增加一个页表

.no\_remainder:

push ecx ; 暂存页表个数

; 为简化处理, 所有线性地址对应相等的物理地址. 并且不考虑内存空洞.

; 首先初始化页目录

mov ax, SelectorPageDir ; 此段首地址为 PageDirBase

mov es, ax

xor edi, edi

xor eax, eax

mov eax, PageTblBase | PG\_P | PG\_USU | PG\_RWW

.1:

stosd

add eax, 4096 ; 为了简化, 所有页表在内存中是连续的.

loop .1

; 再初始化所有页表

mov ax, SelectorPageTbl ; 此段首地址为 PageTblBase

mov es, ax

pop eax ; 页表个数

mov ebx, 1024 ; 每个页表 1024 个 PTE

mul ebx

mov ecx, eax ; PTE个数 = 页表个数 \* 1024

xor edi, edi

xor eax, eax

mov eax, PG\_P | PG\_USU | PG\_RWW

.2:

stosd

add eax, 4096 ; 每一页指向 4K 的空间

loop .2

在函数的开头，我们就用内存大小除以4MB来得到应初始化的PDE个数(同时也是页表的个数)。在初始化页表的时候，通过刚刚计算出的页表个数乘以1024(每个页表包含1024个PTE)得到需要填充的PTE个数，然后通过循环完成对它的初始化。

个人说明一下这个计算原因，一个page中有1024个PDE(4Byte)，一个PDE对应1024个PTE(4Bytes)，一个PTE对应一个page，那么一个PDE就对应1024\*page=1024\*4k=4MB。

这样一来，页表所占的空间就小得多，在本里中，32MB的内存实际上只要32KB的页表就够了，所以在GDT中，这样初始化页表段：

LABEL\_DESC\_PAGE\_TBL: Descriptor PageTblBase, 4096 \* 8 - 1, DA\_DRW ; Page Tables

这样，程序所需的内存空间就小了许多。

**3.3.7进一步体会分页机制 P120**

上文中我们提到，我们还没有得益于分页。分页的益处其实体现在多个方面，这里，我们先举一个例子，以便先有初步的认识。

在此之前不知道你有没有注意过一个细节，在Windows下做开发时，如果写两个一样的程序员，然后同时打开两个VC，同时进行调试，你会发现，从变量地址到寄存器的值，几乎全部都是一样的！而这些“一样的”地址之间完全不会混淆起来，而是各自完成着自己的职责。这就是分页机制的功劳，下面我们就来模拟一下这个效果。

先执行某个线性地址处的模块，然后通过改变cr3来转换地址映射关系，再执行同一个线性地址处的模块，由于映射地址已经改变，所以两次得到的应该是不同的输出。

映射关系转换前的情形如图3-33所示。开始，我们让过程ProcPagingDemo中的代码实现向线性地址LinearAddrDemo处的转移，而LinearAddrDemo映射到物理地址空间中的ProcFoo处。我们让ProcFoo打印出红色的字符串Foo，所以执行时我们应该可以看到红色的Foo。随后我们改变地址关系，变化成图3-34所示的情形。

页目录表和页表的切换让LinearAddrDemo映射到物理地址空间的ProcBar处，所以当我们再一次调用过程ProcPagingDemo时，程序将转移到ProcBar处执行，我们将看到红色字符串Bar。

下面我们就来看一下实现这些需要对pmtest7.asm做哪些修改。

首先，我们用到了另外一套页目录和页表，所以原先的页目录段和页表段已经不再够用了。事实上，前面的程序中我们用两个段分别存放页目录表和页表，是为了让读者阅读时更加直观和形象。在pmtest8.asm中，我们把它们放到同一个段中，同时把增加的一套页目录和页表也放到这个段中。

为了操作方便，我们新增加一个段，其线性地址空间为0~4GB。由于分页机制启动之前线性地址等同于物理地址，所以通过这个段可以方便地存取特定的物理地址。此段的定义如下：

LABEL\_DESC\_FLAT\_C: Descriptor 0, 0fffffh, DA\_CR | DA\_32 | DA\_LIMIT\_4K

LABEL\_DESC\_FLAT\_RW: Descriptor 0, 0fffffh, DA\_DRW | DA\_LIMIT\_4K

SelectorFlatC equ LABEL\_DESC\_FLAT\_C - LABEL\_GDT

SelectorFlatRW equ LABEL\_DESC\_FLAT\_RW - LABEL\_GDT

我们之所以用了两个描述符来描述这个段，是因为我们不仅仅要读写这段内存，而且要执行其中的代码，而这对描述符的属性要求是不一样的。这两个段的段基址都是0，长度都是4GB。

下面我们就将启动分页的代码做相应的修改：

SetupPaging:

; 根据内存大小计算应初始化多少PDE以及多少页表

xor edx, edx

mov eax, [dwMemSize]

mov ebx, 400000h ; 400000h = 4M = 4096 \* 1024, 一个页表对应的内存大小

div ebx

mov ecx, eax ; 此时 ecx 为页表的个数，也即 PDE 应该的个数

test edx, edx

jz .no\_remainder

inc ecx ; 如果余数不为 0 就需增加一个页表

.no\_remainder:

mov [PageTableNumber], ecx ; 暂存页表个数

; 为简化处理, 所有线性地址对应相等的物理地址. 并且不考虑内存空洞.

; 首先初始化页目录

mov ax, SelectorFlatRW

mov es, ax

mov edi, PageDirBase0 ; 此段首地址为 PageDirBase0

xor eax, eax

mov eax, PageTblBase0 | PG\_P | PG\_USU | PG\_RWW

.1:

stosd

add eax, 4096 ; 为了简化, 所有页表在内存中是连续的.

loop .1

; 再初始化所有页表

mov eax, [PageTableNumber] ; 页表个数

mov ebx, 1024 ; 每个页表 1024 个 PTE

mul ebx

mov ecx, eax ; PTE个数 = 页表个数 \* 1024

mov edi, PageTblBase0 ; 此段首地址为 PageTblBase0

xor eax, eax

mov eax, PG\_P | PG\_USU | PG\_RWW

.2:

stosd

add eax, 4096 ; 每一页指向 4K 的空间

loop .2

mov eax, PageDirBase0

mov cr3, eax

我们原来并没有把页表个数保存起来，而现在，我们不只有页目录和页表，为了初始化另外的页表时方便起见，在这里增加了一个变量PageTableNumber，页表的个数就在里面。

在整个初始化页目录和页表的过程中，es始终为SelectorFlatRW。这样，想存取物理地址的时候，只需将地址赋值给edi，那么es:edi指向的就是相应物理地址。

比如页目录在物理地址PageDirBase0处，初始化PDE时将edi赋值为PageDirBase0，es:edi于是指向地址PageDirBase0处，赋值通过指令stosd实现，初始化页表也是同样的道理。

这样，页目录和页表的准备工作就完成了。不过我们不再在原来的位置调用它，而是新建一个函数PagingDemo，把所有与分页有关的内容全都放进里面，这样，程序看起来结构清晰一些。

根据图3-33和图3-34，我们可以认为在这个程序中的实现中有4个要关注的要素，分别是ProcPagingDemo、LinearAddrDemo、ProcFoo和ProcBar。因为程序开始时， LinearAddrDemo是指向ProcFoo的，而开始时线性地址和物理地址是对等的，所以LinearAddrDemo应该等于ProcFoo。而ProcFoo和ProcBar应该是指定的物理地址，所以 LinearAddrDemo也应该是指定的物理地址。也正因为如此，我们使用它们时应该确保使用的是FLAT段，即选择子应该是LABEL\_DESC\_FLAT\_C或者LABEL\_DESC\_FLAT\_RW。

为了将我们的代码放置在ProcFoo和ProcBar这两处地方，我们先写两个函数，在程序运行时将这两个函数的执行码复制过去就可以了。

ProcPagingDemo中是要调用 LinearAddrDemo的，而上面我们提到， LinearAddrDemo是在FLAT段中的，所以，如果不想使用段间转移的话，我们就需要把ProcPagingDemo也放进FLAT段中。我们需要写一个函数，然后把代码复制到ProcPagingDemo处。

这样看来，ProcPagingDemo、LinearAddrDemo、ProcFoo和ProcBar虽然都是当做函数来使用，但实际上却都是内存中指定的地址。我们把他们定义为常量：

LinearAddrDemo equ 00401000h

ProcFoo equ 00401000h

ProcBar equ 00501000h

ProcPagingDemo equ 00301000h

将代码填充进这些内存地址的代码就在上文我们提到的PaginDemo中：

PagingDemo:

mov ax, cs

mov ds, ax

mov ax, SelectorFlatRW

mov es, ax

push LenFoo

push OffsetFoo

push ProcFoo

call MemCpy

add esp, 12

push LenBar

push OffsetBar

push ProcBar

call MemCpy

add esp, 12

push LenPagingDemoAll

push OffsetPagingDemoProc

push ProcPagingDemo

call MemCpy

add esp, 12

mov ax, SelectorData

mov ds, ax ; 数据段选择子

mov es, ax

call SetupPaging ; 启动分页

call SelectorFlatC:ProcPagingDemo

call PSwitch ; 切换页目录，改变地址映射关系

call SelectorFlatC:ProcPagingDemo

ret

被复制的三个过程的代码如下：

PagingDemoProc:

OffsetPagingDemoProc equ PagingDemoProc - $$

mov eax, LinearAddrDemo

call eax

retf

; ---------------------------------------------------------------------------

LenPagingDemoAll equ $ - PagingDemoProc

foo:

OffsetFoo equ foo - $$

mov ah, 0Ch ; 0000: 黑底 1100: 红字

mov al, 'F'

mov [gs:((80 \* 17 + 0) \* 2)], ax ; 屏幕第 17 行, 第 0 列。

mov al, 'o'

mov [gs:((80 \* 17 + 1) \* 2)], ax ; 屏幕第 17 行, 第 1 列。

mov [gs:((80 \* 17 + 2) \* 2)], ax ; 屏幕第 17 行, 第 2 列。

ret

LenFoo equ $ - foo

bar:

OffsetBar equ bar - $$

mov ah, 0Ch ; 0000: 黑底 1100: 红字

mov al, 'B'

mov [gs:((80 \* 18 + 0) \* 2)], ax ; 屏幕第 18 行, 第 0 列。

mov al, 'a'

mov [gs:((80 \* 18 + 1) \* 2)], ax ; 屏幕第 18 行, 第 1 列。

mov al, 'r'

mov [gs:((80 \* 18 + 2) \* 2)], ax ; 屏幕第 18 行, 第 2 列。

ret

LenBar equ $ - bar

我们回过头看PaginDemo:代码，其中的大部分语句是内存复制工作，但实际上真正激动人心的语句却是代码最后的4个call指令。它们首先启动分页机制，然后调用ProcPagingDemo，再切换页目录，最后又调用一边ProcPagingDemo。

现在ProcPagingDemo、ProcFoo以及ProcBar的内容我们已经知道了，由于LinearAddrDemo和ProcFoo相等，并且函数SetupPaging建立起来的是对等的映射关系，所以第一次对ProcPagingDemo的调用反应的就是图3-33的情况。

接下来调用的是PSwitch，我们来看一下这个切换页目录的函数是怎样的。

PSwitch:

; 初始化页目录

mov ax, SelectorFlatRW

mov es, ax

mov edi, PageDirBase1 ; 此段首地址为 PageDirBase1

xor eax, eax

mov eax, PageTblBase1 | PG\_P | PG\_USU | PG\_RWW

mov ecx, [PageTableNumber]

.1:

stosd

add eax, 4096 ; 为了简化, 所有页表在内存中是连续的.

loop .1

; 再初始化所有页表

mov eax, [PageTableNumber] ; 页表个数

mov ebx, 1024 ; 每个页表 1024 个 PTE

mul ebx

mov ecx, eax ; PTE个数 = 页表个数 \* 1024

mov edi, PageTblBase1 ; 此段首地址为 PageTblBase1

xor eax, eax

mov eax, PG\_P | PG\_USU | PG\_RWW

.2:

stosd

add eax, 4096 ; 每一页指向 4K 的空间

loop .2

; 在此假设内存是大于 8M 的

mov eax, LinearAddrDemo

shr eax, 22

mov ebx, 4096

mul ebx

mov ecx, eax

mov eax, LinearAddrDemo

shr eax, 12

and eax, 03FFh ; 1111111111b (10 bits)

mov ebx, 4

mul ebx

add eax, ecx

add eax, PageTblBase1

mov dword [es:eax], ProcBar | PG\_P | PG\_USU | PG\_RWW

mov eax, PageDirBase1

mov cr3, eax

jmp short .3

.3:

nop

ret

这个函数前面初始化页目录表和页表的过程与SetupPaging是差不多的，只是紧接着程序增加了改变线性地址LinearAddrDemo对应的物理地址的语句。改变后，LinearAddrDemo将不再对应ProcFoo，而是对应ProcBar。

所以，此函数调用完成之后，对ProcPagingDemo的调用就变成了图3-34所示的情况。最后cr3被赋值PageDirBase1，这个切换过程宣告完成。运行一下，会发现两次调用打印Foo和Bar。

这个跟我们先前提到的Windows下做开发时看到的不同进程有相同的地址，原理跟本例是类似的，也是在任务切换时通过改变cr3的值来切换页目录，从而改变地址映射关系。

这样，操作系统全权负责了这其中的转换工作，而应用程序不必操作物理地址。

**3.4中断和异常**

说起中断，好像我们一直在用。最近的一次是我们通过int 15h得到了计算机内存信息。但是不知道你发现没有，我们所有中断的操作都是在实模式下进行的。我们在实模式下用int 15h得到内存信息，然后在保护模式下把它们显示出来。

并不是我们故意把问题搞复杂，而是在保护模式下，中断机制发生了很大变化，原来的中断向量表已经被IDT所代替，实模式下能用的BIOS中断在保护模式下已经不能用了。你可能没有听说过IDT，但看名字可以猜到，它跟GDT、LDT应该有相似的地方。没错，其实它也是描述表，叫做中断描述符表(Interrupt Descriptor Table)。IDT中的描述符可以是下面三种之一：

● 中断门描述符

● 陷阱门描述符

● 任务门描述符

IDT的作用是将每一个中断向量和一个描述符对应起来。从这个意义上说，IDT也是一种向量表，虽然它形式上跟实模式下的向量表非常不同。而我们在“调用门初体验”中也曾经提到，中断门和陷阱门是特殊的调用门，所以，虽然本节中我们接触的是新的概念，其核心却只是在原有内容的基础上的一点改变。

让我们看看中断向量到中断处理程序的对应过程(如图3-36所示)

联系调用门我们知道，其实中断门和陷阱门的作用机理几乎是一样的，只不过使用调用门时使用call指令，而这里我们使用int指令。

刚才我们提到，IDT中可以有中断门、陷阱门、或者任务门。但任务门在有些操作系统中根本就没有用到(比如Linux)，这里，我们也不做太多关注。中断门和陷阱门的结构如图3-37所示。

对比调用门的结构我们知道，在中断门和陷阱门中的BYTE4的低5位变成了保留位，而不再是Param Count。而且，表示TYPE的4位也将变为0xE(中断门)或0xF(陷阱门)。当然，S位仍将是0。

下面我们对中断进行一下全面的了解，再开展coding。

**3.4.1中断和异常机制 P129**

不管中断还是异常，通俗来讲，都是软件或者硬件发生了某种情形而通知处理器的行为。于是，由此引出两个问题：一是处理器可以对何种类型的通知做出反应；二是当接到某种通知时做出何种处理。

每一种中断(异常)都会对应一个中断向量号，而这个向量号通过IDT就与相应的中断处理程序对应起来。

那么，处理器到底能处理哪些中断或异常呢？可以查阅原文的表3-11。

看到“助记符”这一栏你可能想起来了，前文中我们对于#GP、#TS等异常已经有所提及。而“类型”一栏可能让你有些迷惑，这里由于怕引起歧义，所以没有进行翻译。实际上，Fault、Trap和Abort是异常的三种类型，他们的具体解释如下：

● Faults是一种可被更正的异常，而且一旦被更正，程序可以不失连续性地继续执行。当一个fault发生时，处理器会把产生fault的指令之前的状态保存起来。异常处理程序的返回地址将会是产生fault的指令，而不是其后的那条指令。

● Traps是一种在发生trap的指令之后立即被报告的异常，它允许程序或任务不失连续性地继续执行。异常处理程序的返回地址将会是产生trap的指令之后的那条指令。

● Aborts是一种不总是报告精确异常发生位置的异常，它不允许程序或任务继续执行，而是用来报告严重错误的。

**3.4.2外部中断 P131**

中断产生的原因有两种，一种是外部中断，也就是由硬件产生的中断，另一种是由指令int n产生的中断。

指令int n，n为向量号，它类似于调用门的使用。

外部中断的情况则复杂一些，因为需要建立硬件中断与向量号之间的对应关系。外部中断分为不可屏蔽中断(NMI)和可屏蔽中断两种，分别由CPU两根引脚NMI和INTR来接收，图3-38。



NMI不可屏蔽，因为它与IF是否被设置无关。NMI中断对应的中断向量号为2，这在表3-11中已经有所说明。

可屏蔽中断与CPU的关系是通过对可编程中断控制器8259A建立起来的。如果你是第一次听说8259A，那么你可以认为它是中断机制中所有外围设备的一个代理，这个代理不但可以根据优先级在同时发生中断的设备中选择应该处理的请求，而且可以通过对其寄存器的设置来屏蔽或打开相应的中断。

由图3-38我们知道，与CPU相连的不是一片，而是两片级联的8259A，每个8259A有8根中断信号线，于是两片级联总共可以挂在15个不同的外部设备。那么，这些设备发出的中断请求如何与中断向量对应起来呢？就是通过对8259A的设置完成的。在BIOS初始化的时候，IRQ0~IRQ7被设置为对应向量号08h~0Fh，而通过表3-11我们知道，在保护模式下向量号08h~0Fh已经被占用了，所以我们不得不重新设置主从8259A。

还好，8259A是可编程中断控制器，对它的设置并不复杂，是通过向相应的端口写入特定的ICW(Initialization Command Word)来实现的。主8259A对应的端口地址是20A和21A，从8259A对应的端口地址是A0h和A1h。ICW共有共有4个，每一个都是具有特定格式的字节。为了先对初始化8259A对应的过程有一个概括的了解，我们过一会再来关注每一个ICW的格式，现在，先来看一下初始化过程：

(1)往端口20h(主片)或A0h(从片)写入ICW1。

(2)往端口21h(主片)或A1h(从片)写入ICW2。

(1)往端口21h(主片)或A1h(从片)写入ICW3。

(1)往端口21h(主片)或A1h(从片)写入ICW4。

这5步的顺序是不能颠倒的，我们现在来看一下4个如图3-39所示的ICW的格式。

我们看到，在写入ICW2时涉及到了与中断向量号的对应，这便是窍门所在了。





这里加一句自己查阅的，21H/A1H是中断屏蔽寄存器端口地址，20H/A0H是接受中断结束命令的寄存器端口地址。

其实，需要做的工作不外乎设置8259A和建立IDT两大部分，我们以pmtest8.asm为基础对代码进行修改，形成pmtest9.asm。

**3.4.3编程操作8259A P133**

把设置8259A代码写进一个函数：

Init8259A:

mov al, 011h

out 020h, al ; 主8259, ICW1.

call io\_delay

out 0A0h, al ; 从8259, ICW1.

call io\_delay

mov al, 020h ; IRQ0 对应中断向量 0x20

out 021h, al ; 主8259, ICW2.

call io\_delay

mov al, 028h ; IRQ8 对应中断向量 0x28

out 0A1h, al ; 从8259, ICW2.

call io\_delay

mov al, 004h ; IR2 对应从8259

out 021h, al ; 主8259, ICW3.

call io\_delay

mov al, 002h ; 对应主8259的 IR2

out 0A1h, al ; 从8259, ICW3.

call io\_delay

mov al, 001h

out 021h, al ; 主8259, ICW4.

call io\_delay

out 0A1h, al ; 从8259, ICW4.

call io\_delay

mov al, 11111110b ; 仅仅开启定时器中断

;mov al, 11111111b ; 屏蔽主8259所有中断

out 021h, al ; 主8259, OCW1.

call io\_delay

mov al, 11111111b ; 屏蔽从8259所有中断

out 0A1h, al ; 从8259, OCW1.

call io\_delay

ret

这段代码分别往主、从两个8259A各写入了4个ICW。在往主8259A写入ICW2时，我们看到IRQ0对应了中断向量号20h，于是，IRQ0~IRQ7就对应中断向量20h~27h；类似地，IRQ8~

IRQ15对应中断向量28h~2Fh。对照表3-11我们知道，20h~2Fh处于用户定义中断的范围内。

在这段代码的后半部分，我们通过对端口21h和A1h的操作屏蔽了所有的外部中断，这一次写入的不再是ICW了，而是OCW(Operation Control Word)。OCW共有3个，OCW1、OCW2和OCW3。由于我们只在两种情况下用到它，因此并不需要了解所有的内容。这两种情况是：

● 屏蔽或打开外部中断。

● 发送EOI给8269A以通知它中断处理结束。

若想屏蔽或打开外部中断，只需要往8259A写入OCW1就可以了，OCW1的格式如下图3-40所示。

可见，若想屏蔽某一个中断，将对应那一位设成1就可以了。实际上，OCW1是被写入了中断屏蔽寄存器(IMR，全称Interrupt Mask Register)中，当一个中断到达，IMR会判断此中断是否应被丢弃。

说起EOI，如果你有过在实模式下的汇编经验，那么对它应该不会陌生。当每一次中断处理结束，需要发送一个EOI给8259A，以便继续接收中断。而发送EOI是通过往端口20h或A0h写OCW2来实现的。OCW2的格式如图3-41所示。

发送EOI给8259A可以由如下的代码完成：

mov al, 20h

out 20h或A0h, al

而对于OCW2其他各位的作用，我们完全可以暂时不予理会。

另外一点是，每一次I/O操作之后都调用了一个延迟函数io\_delay以等待操作完成。函数io\_delay很简单，调用了4个nop指令：

在相应的位置添加调用Init8259A的指令之后，对8259A的操作就结束了，我们下面就来建立一个IDT。

**3.4.1建立IDT**

为了操作方便，我们把IDT放进一个单独的段中，与原文不一样，因为这是最终版：

[SECTION .idt]

ALIGN 32

[BITS 32]

LABEL\_IDT:

; 门 目标选择子, 偏移, DCount, 属性

%rep 32

Gate SelectorCode32, SpuriousHandler, 0, DA\_386IGate

%endrep

.020h: Gate SelectorCode32, ClockHandler, 0, DA\_386IGate

%rep 95

Gate SelectorCode32, SpuriousHandler, 0, DA\_386IGate

%endrep

.080h: Gate SelectorCode32, UserIntHandler, 0, DA\_386IGate

IdtLen equ $ - LABEL\_IDT

IdtPtr dw IdtLen - 1 ; 段界限

dd 0 ; 基地址

看得出，这个IDT真的是不能再简单了，全部的255个描述符完全相同。这里利用了NASM的%rep预处理指令，将每一描述符都设置为指向Selector Code32:SpuriousHandler的中断门。SpuriousHandler也很简单，在屏幕的右上角打印红色的字符“！”，然后进入死循环：

\_SpuriousHandler:

SpuriousHandler equ \_SpuriousHandler - $$

mov ah, 0Ch ; 0000: 黑底 1100: 红字

mov al, '!'

mov [gs:((80 \* 0 + 75) \* 2)], ax ; 屏幕第 0 行, 第 75 列。

jmp $

iretd

加载IDT的代码与对GDT的处理非常类似：

; 为加载 IDTR 作准备

xor eax, eax

mov ax, ds

shl eax, 4

add eax, LABEL\_IDT ; eax <- idt 基地址

mov dword [IdtPtr + 2], eax ; [IdtPtr + 2] <- idt 基地址

; 保存 IDTR

sidt [\_SavedIDTR]

; 保存中断屏蔽寄存器(IMREG)值

in al, 21h

mov [\_SavedIMREG], al

; 加载 GDTR

lgdt [GdtPtr]

; 关中断

;cli

; 加载 IDTR

lidt [IdtPtr]

在执行lidt之前，用cli指令请IF位，暂时不响应可屏蔽中断。

回到DOS代码这里不降解了，参考源代码。下面要利用中断做些事情。

**3.4.5实现一个中断**

我们把080h号中断单独拿出来，调用一下：

call Init8259A

int 080h

因为IDT中的所有描述符都初始化成同一个样子，都指向SelectorCode32:SpuriousHandler处，所以我们把程序改进下，单独为第80h号中断准备处理函数，先修改IDT：

LABEL\_IDT:

; 门 目标选择子, 偏移, DCount, 属性

%rep 32

Gate SelectorCode32, SpuriousHandler, 0, DA\_386IGate

%endrep

.020h: Gate SelectorCode32, ClockHandler, 0, DA\_386IGate

%rep 95

Gate SelectorCode32, SpuriousHandler, 0, DA\_386IGate

%endrep

.080h: Gate SelectorCode32, UserIntHandler, 0, DA\_386IGate

新增加一个函数 UserIntHandler来处理它，它与 SpuriousHandler类似，只是在函数末尾通过iretd指令返回，而不是进入死循环：

\_UserIntHandler:

UserIntHandler equ \_UserIntHandler - $$

mov ah, 0Ch ; 0000: 黑底 1100: 红字

mov al, 'I'

mov [gs:((80 \* 0 + 70) \* 2)], ax ; 屏幕第 0 行, 第 70 列。

iretd

**3.4.6时钟中断实验**

我们需要将前面设置的8259A派上用场，我们即将打开时钟中断(IRQ0)。

我们提到过，可屏蔽中断于NMI的区别在于是否受到IF位的影响，而8259A的中断屏蔽寄存器(IMR)也影响着中断是否会被响应。所以，外部可屏蔽中断的发生就受到两个因素的影响，只有当IF位为1，并且IMR相应位为0时，才会发生。

那么，如果我们想打开时钟中断的话，一方面不仅要设计一个中断处理程序，另一方面还要设置IMR，并且设置IF位。设置IMR可以通过写OCW2来完成，而设置IF可以通过指令sti来完成。

我们先来写一个时钟中断处理程序，chapter3\i\pmtest9.asm

\_ClockHandler:

ClockHandler equ \_ClockHandler - $$

inc byte [gs:((80 \* 0 + 70) \* 2)] ; 屏幕第 0 行, 第 70 列。

mov al, 20h

out 20h, al ; 发送 EOI

iretd

看得出，这个中断处理程序当真是不能再简单了，除了发送EOI的两行语句以及iretd，只有一条指令，就是把屏幕第0行、第70列的字符曾一，变成ASCII码表中位于它后面的字符。如果我们在调用80h号中断之后打开中断的话，由于第0行、第70列处已被写入字符I，所以第一次中断发生时那里会变成字符J，再一次中断则变成K，以后每发生一次时钟中断，字符就会变化一次，就会看到不断变化中的字符。

修改初始化8259A的代码，时钟中断不再屏蔽：

mov al, 11111110b ; 仅仅开启定时器中断

;mov al, 11111111b ; 屏蔽主8259所有中断

out 021h, al ; 主8259, OCW1.

call io\_delay

mov al, 11111111b ; 屏蔽从8259所有中断

out 0A1h, al ; 从8259, OCW1.

call io\_delay

ret

IDT:

LABEL\_IDT:

; 门 目标选择子, 偏移, DCount, 属性

%rep 32

Gate SelectorCode32, SpuriousHandler, 0, DA\_386IGate

%endrep

.020h: Gate SelectorCode32, ClockHandler, 0, DA\_386IGate

%rep 95

Gate SelectorCode32, SpuriousHandler, 0, DA\_386IGate

%endrep

.080h: Gate SelectorCode32, UserIntHandler, 0, DA\_386IGate

按理说，现在在调用80h号中断之后执行sti来打开中断，效果就应该可以看到了。可是有一个问题：程序马上会继续执行，可能没等第一个中断发生，程序已经执行完并退出了。所以，我们需要让程序停留在某个地方，干脆让它死循环吧，这样虽然不雅，却简单易行：

int 080h

sti

jmp $

DOSBox运行后，你会发现，字符一直在变化跳动。

**3.4.7几点额外说明**

**3.4.7.1特权级变换**

为简单起见，我们上面的代码始终运行在ring0，但在实际应用中，中断的产生大多是带有特权级变换的。实际上，通过中断门和陷阱门的中断就相当于用call指令调用一个调用门，涉及到的特权级变换的规则是完全一样的。读者可以参考3.2.3.3中调用们的相关内容。

**3.4.7.2中断或异常发生时的堆栈变化**



如图3-44，如果中断或异常发生时没有特权级变换，那么eflags、cs、eip将依次被压入堆栈，如果有出错码的话，出错码将在最后被压栈。有特权级变换的情况下同样会发生堆栈切换，此时，ss和esp将被压入内层堆栈，然后是eflags、cs、eip、出错码(如果有的话)。

不总是会有出错码，具体情况参考表3-11。

从中断或异常返回时必须使用指令iretd，它与ret很相似，只是它同时会改变eflags的值。需要注意的是，只有当CPL为0时，eflags中的IOPL域才会改变，而且只有当CPL≤IOPL时，IF才会被改变。关于IOPL的更多细节请参考3.5节。

另外，iretd执行时Error Code不会被自动从堆栈中弹出，所以，执行它之前要先将它从栈中清除掉。

**3.4.7.3中断门和陷阱门的区别**

上文中，我们总是把中断门和陷阱门放在一起介绍。实际上，它们之间存在一个微小的差别，就是对中断允许标志IF的影响。由中断门向量引起的中断会复位IF，因为可以避免其他中断干扰当前中断的处理。随后的iret的指令会从堆栈上恢复IF的原值；而通过陷阱门产生的中断不会改变IF。

**3.5保护模式下的I/O**

毫无疑问，对I/O的控制权限是很重要的一项内容，保护模式对此也做了限制。用户进程如果不被许可是无法进行I/O操作的。这种限制通过两个方面来实现，它们就是IOPL和I/O许可位图。

**3.5.1 IOPL**

上文中我们曾提到IOPL，它是I/O保护机制的关键之一，位于寄存器eflags的第12、13位，如图3-45所示。



指令in、ins、out、outs、cli、sti只有在CPL≤IOPL时才能执行。

这些指令被成为I/O敏感指令(I/O Sensitive Instructions)。如果低特权级的指令试图访问这些I/O敏感指令将会导致常规保护错误(#GP)。

可以改变IOPL的指令只有popf和iretd，但只有运行在ring0的程序才能将其改变。运行在低特权级下的程序无法改变IOPL，不过，如果试图那样做的话并不会产生任何异常，只是IOPL不会改变，仍然保持原样。

指令popf同样可以用来改变IF(就好像执行了cli和sti)。然而，在这种情况下，popf也变成了I/O敏感指令。只有CPL≤IOPL时，popf才可以成功将IF改变，否则IF将维持原值，不会产生任何异常。

**3.5.2 I/O许可位图(I/O Permission Bitmap)**

如果你再回头看图3-19的话，会发现在TSS偏移102字节处有一个被称作“I/O位图基址”的东西，它是一个以TSS的地址为基址的偏移，指向的便是I/O许可位图。之所以叫位图，是因为它的每一位表示一个字节的端口地址是否可用。如果某一位为0，则表示此位对应的端口号可用，为1则不可用。由于每一个任务都可以有单独的TSS，所以每一个任务可以有它单独的I/O许可位图。比如：

由于I/O许可位图开始有12字节内容为0FFh，即有12\*8=96位被置为1，所以从端口00h到5fh共96个端口地址对此任务不可用。同理，接下来的1字节只有第1位(从0开始数)是0，表示这一位对应的端口(61h)可用。

I/O许可位图必须以0FFh结尾。

如果I/O位图基址大于或等于TSS段界限，就表示没有I/O许可位图，如果CPL≥IOPL，则所有I/O指令都会引起异常。

I/O许可位图的使用使得即便在同一特权级下不同的任务也可以有不同的I/O访问权限。

**3.6保护模式小结**

在GDT、LDT以及IDT中，每一个描述符都有自己的界限和属性等内容，是对描述符所描述对象的一种限定和保护。

分页机制中的PDE和PTE都含有R/W以及U/S位，提供页级保护。

页式存储的使用使应用程序使用的是线性地址空间而不是物理地址，于是物理内存就被保护起来。

中断不再像实模式下一样使用，也提供特权检验等内容。

I/O指令不再随便使用，于是端口被保护起来。

在程序运行过程中，如果遇到不同特权级间的访问等情况，会对CPL、RPL、DPL、IOPL等内容进行非常严格的检验，同时可能伴随堆栈的切换，这都对不同层级的程序进行了保护。

以上提到的内容并不能全部概括“保护”二字的含义，但至少可以让我们部分地看到保护模式的真谛所在。

**第四章 让操作系统走进保护模式**

先进入保护模式再说吧。

**4.1突破512字节的限制**

一个操作系统从开机到开始运行，大致经历“引导→加载内核入内存→跳入保护模式→开始执行内核”这样一个过程。也就是说，在内核开始执行之前不但要加载内核，而且还有准备保护模式等一系列工作，如果全部交给引导扇区来做，512字节很可能是不够用的，所以，不妨把这个过程交给另外的模块来完成，我们把这个模块叫做Loader。引导扇区负责把Loader加载入内存并且把控制权交给它，其他工作放心地交给Loader来做，因为它没有512字节的限制，将会灵活得多。

在这里，为了操作方便，不妨把软盘做成FAT12格式，这样对Loader以及今后的Kernel的操作将会非常简单易行。

**4.1.1 FAT12 P146**

FAT12是DOS时代就开始使用的文件系统，几乎所有的文件系统都会把磁盘划分为若干层次以方便组织和管理，这些层次包括：

● 扇区(Sector)：磁盘上的最小数据单元

● 簇(Cluster)：一个或多个扇区

● 分区(Parition)：通常指整个文件系统

我们已经接触过引导扇区，就让我们从这里开始。引导扇区是整个软盘的第0个扇区，在这个扇区中有一个很重要的数据结构叫做BPB(BIOS Parameter Block)，引导扇区的格式如表4-1所示(见原文)，其中名称以BPB\_开头的域属于BPB，以BS\_开头的域不属于BPB，只是引导扇区(Boot Sector)的一部分。

紧接着引导扇区的是两个完全相同的FAT表，每个FAT占用9个扇区。第二个FAT之后是根目录区的第一个扇区。根目录区的后面是数据区，如图4-1。

本来应该讲到FAT表了，但单纯数据结构的讲解太不活泼了，我们不妨想一想要用这张软盘做什么。我们是要把Loader复制到软盘上并让引导扇区找到并加载它，那么就来看一下引导扇区通过怎样的步骤才能找到文件，以及如何能够把文件内容全部读出来并放进内存里。

为简单起见，我们Loader只能放在根目录中，而根目录信息存放在FAT2后面的根目录区中。那么，先来看一下根目录区。

根目录区位于第二个FAT表之后，开始的扇区号为19，它由若干个目录条目(Directory Entry)组成，条目最多有BPB\_RootEntCnt个。由于根目录区的大小是依赖于BPB\_RootEntCnt的，所以长度不固定。

根目录区中的每一个条目占用32字节，它的格式如表4-2所示。

看来结构并不复杂，主要定义了文件的名称、属性、大小、日期以及在磁盘中的位置。你可能在想，要是能直观地看到一个真实的目录条目就好了。这并不难做到，我们先来创建一个虚拟软盘，假设是FOPPLY.IMG，然后把它作为Virtual PC的A盘，格式化后就可以方便地往其中添加文件和目录了。这样，当我们想查看它的格式时，只需用二进制查看器打开FOPPLY.IMG就可以了，跟实际的软盘看起来一样的(有兴趣的读者可以试着读取一张真实的软盘来对比)。

dd if=/dev/zero of=floppy.img bs=512 count=2880

mkdosfs -F 12 floppy.img

好了，通过Virtual PC在这张虚拟软盘中添加如下几个文本文件：

● RIVER.TXT，内容为riverriverriver。

● FLOWER.TXT，内容为30个单词flower，用来测试文件跨越扇区的情况。(可以先建一个小文件，最后再把它改长，这样可以让它对应的簇不连续，便于观察和理解。)

● TREE.TXT，内容为treetreetree。

再添加一个HOUSE目录，然后在目录\HOUSE下添加两个文本文件：

● CAT.TXT，内容为catcatcat。

● DOG.TXT，内容为dogdogdog。

由于根目录区从第19扇区开始，每个扇区512字节，所以其第一个字节位于偏移19\*512=9728=0x2600处。好的，就让我们用二进制查看器打开flooppy.img，定位到偏移

00002600 46 4c 4f 57 45 52 20 20 54 58 54 20 00 00 2c 90 |FLOWER TXT ..,.|

00002610 55 49 55 49 00 00 2c 90 55 49 6e 00 b5 00 00 00 |UIUI..,.UIn.....|

00002620 52 49 56 45 52 20 20 20 54 58 54 20 00 00 73 8f |RIVER TXT ..s.|

00002630 55 49 55 49 00 00 73 8f 55 49 23 00 10 00 00 00 |UIUI..s.UI#.....|

00002640 48 4f 55 53 45 20 20 20 20 20 20 10 00 64 6f 90 |HOUSE ..do.|

00002650 55 49 55 49 00 00 6f 90 55 49 9a 00 00 00 00 00 |UIUI..o.UI......|

00002660 e5 4f 47 54 58 54 7e 31 53 57 50 20 00 64 62 90 |.OGTXT~1SWP .db.|

00002670 55 49 55 49 00 00 62 90 55 49 00 00 00 00 00 00 |UIUI..b.UI......|

00002680 54 52 45 45 20 20 20 20 54 58 54 20 00 64 54 90 |TREE TXT .dT.|

00002690 55 49 55 49 00 00 54 90 55 49 87 00 0d 00 00 00 |UIUI..T.UI......|

看到这个画面，你可能一下就注意到了RIVER、FLOWER、TREE等单词，是的，它们就在这里。我们以RIVER.TXT为例，它的各项值如表4-3所示。

|  |  |
| --- | --- |
| 名称 | 值 |
| DIR\_Name | RIVER TXT |
| DIR\_Attr | 0x20 |
| DIR\_WrtTime | 0x8f73 |
| DIR\_WrtDate | 0x4955 |
| DIR\_FstClus | 0x0023 |
| DIR\_FileSize | 0x00000010 |

00000000 eb 3c 90 6d 6b 66 73 2e 66 61 74 00 02 01 01 00 |.<.mkfs.fat.....|

00000010 02 e0 00 40 0b f0 09 00 12 00 02 00 00 00 00 00 |...@............|

00000020 00 00 00 00 00 00 29 4a 80 e8 6a 4e 4f 20 4e 41 |......)J..jNO NA|

00000030 4d 45 20 20 20 20 46 41 54 31 32 20 20 20 0e 1f |ME FAT12 ..|

00000040 be 5b 7c ac 22 c0 74 0b 56 b4 0e bb 07 00 cd 10 |.[|.".t.V.......|

00000050 5e eb f0 32 e4 cd 16 cd 19 eb fe 54 68 69 73 20 |^..2.......This |

00000060 69 73 20 6e 6f 74 20 61 20 62 6f 6f 74 61 62 6c |is not a bootabl|

00000070 65 20 64 69 73 6b 2e 20 20 50 6c 65 61 73 65 20 |e disk. Please |

00000080 69 6e 73 65 72 74 20 61 20 62 6f 6f 74 61 62 6c |insert a bootabl|

00000090 65 20 66 6c 6f 70 70 79 20 61 6e 64 0d 0a 70 72 |e floppy and..pr|

000000a0 65 73 73 20 61 6e 79 20 6b 65 79 20 74 6f 20 74 |ess any key to t|

000000b0 72 79 20 61 67 61 69 6e 20 2e 2e 2e 20 0d 0a 00 |ry again ... ...|

000000c0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|

由于本文制作出来的FAT12镜像不符合，这里面有两篇文章可供参考

<http://blog.csdn.net/pacosonswjtu/article/details/48391467>

<http://www.tuicool.com/articles/rAJ7zmV>

通过一个文件所描述的DIR\_FstClus，去FAT1表中查到对应的簇号，去数据区域查找。

**4.1.2 DOS可识别的引导盘 P151**

既然引导扇区需要有BPB等头信息才能被微软识别，我们就先加上它，让程序一开头变成下面的形式。

;%define \_BOOT\_DEBUG\_ ; 做 Boot Sector 时一定将此行注释掉!将此行打开后用 nasm Boot.asm -o Boot.com 做成一个.COM文>件易于调试

%ifdef \_BOOT\_DEBUG\_

org 0100h ; 调试状态, 做成 .COM 文件, 可调试

%else

org 07c00h ; Boot 状态, Bios 将把 Boot Sector 加载到 0:7C00 处并开始执行

%endif

jmp short LABEL\_START ; Start to boot.

nop ; 这个 nop 不可少

; 下面是 FAT12 磁盘的头

BS\_OEMName DB 'ForrestY' ; OEM String, 必须 8 个字节

BPB\_BytsPerSec DW 512 ; 每扇区字节数

BPB\_SecPerClus DB 1 ; 每簇多少扇区

BPB\_RsvdSecCnt DW 1 ; Boot 记录占用多少扇区

BPB\_NumFATs DB 2 ; 共有多少 FAT 表

BPB\_RootEntCnt DW 224 ; 根目录文件数最大值

BPB\_TotSec16 DW 2880 ; 逻辑扇区总数

BPB\_Media DB 0xF0 ; 媒体描述符

BPB\_FATSz16 DW 9 ; 每FAT扇区数

BPB\_SecPerTrk DW 18 ; 每磁道扇区数

BPB\_NumHeads DW 2 ; 磁头数(面数)

BPB\_HiddSec DD 0 ; 隐藏扇区数

BPB\_TotSec32 DD 0 ; 如果 wTotalSectorCount 是 0 由这个值记录扇区数

BS\_DrvNum DB 0 ; 中断 13 的驱动器号

BS\_Reserved1 DB 0 ; 未使用

BS\_BootSig DB 29h ; 扩展引导标记 (29h)

BS\_VolID DD 0 ; 卷序列号

BS\_VolLab DB 'Tinix0.01 '; 卷标, 必须 11 个字节

BS\_FileSysType DB 'FAT12 ' ; 文件系统类型, 必须 8个字节

LABEL\_START:

把生成的Boot.bin，写入磁盘引导扇区，运行的效果没有变，仍然会是图1-1的样子。但是，现在的软盘已经能够被DOS以及Windows识别了，我们已经可以方便地往上添加或删除文件了。

**4.1.3一个最简单的Loader**

要写代码加载Loader入内存了，可是我们还没有Loader，怎么办？不要紧，我们先写一个最小的，让它显示一个字符，然后进入死循环，这样，如果加载成功并成功交出了控制权的话，应该可以看到这个字符。

新建一个文件loader.asm，短短几行指令如代码4-2所示。

org 0100h

mov ax, 0B800h

mov gs, ax

mov ah, 0Fh ; 0000: 黑底 1111: 白字

mov al, 'L'

mov [gs:((80 \* 0 + 39) \* 2)], ax ; 屏幕第 0 行, 第 39 列。

jmp $ ; Start

这段代码被编译成.COM文件，可以直接在DOS下执行，效果是在屏幕第一行中央出现字符L，然后进入死循环。在这里，我们用命令行nasm loader.asm -o loader.bin来编译它。需要注意的是，虽然编译出的二进制代码加载到内存的任意位置都可以正确执行，但我们要扩展它，为了将来的执行不会出现问题，要保证把它放入某个段内偏移0x100位置。

**4.1.4加载Loader入内存**

要加载一个文件入内存的话，免不了要读软盘，这时候就用到BIOS中断int 13h。它的用法如表4-4所示。



我们看到，中断需要的参数不是原来提到的从第0扇区开始的扇区号，而是柱面号、磁头号以及在当前柱面上的扇区号3个分量，所以需要我们自己来转换一下。对于1.44MB的软盘来讲，总共有两面(磁头号0和1)，每面80个磁道(磁道号0~79)，每个磁道有18个扇区(扇区号1~18)。下面的公式就是软盘容量的由来：

2\*80\*18\*512 = 1.44MB

于是，磁头号、柱面(磁道)号和起始扇区号可以用图4-6所示的方法来计算。

我们就先写一个读软盘扇区的函数吧：

ReadSector: ;从第 ax 个 Sector 开始, 将 cl 个 Sector 读入 es:bx 中

; -----------------------------------------------------------------------

; 怎样由扇区号求扇区在磁盘中的位置 (扇区号 -> 柱面号, 起始扇区, 磁头号)

; -----------------------------------------------------------------------

; 设扇区号为 x

; ┌ 柱面号 = y >> 1

; x ┌ 商 y ┤

; -------------- => ┤ └ 磁头号 = y & 1

; 每磁道扇区数 │

; └ 余 z => 起始扇区号 = z + 1

push bp

mov bp, sp

sub esp, 2 ; 辟出两个字节的堆栈区域保存要读的扇区数: byte [bp-2]

mov byte [bp-2], cl

push bx ; 保存 bx

mov bl, [BPB\_SecPerTrk] ; bl: 除数

div bl ; y 在 al 中, z 在 ah 中

inc ah ; z ++

mov cl, ah ; cl <- 起始扇区号

mov dh, al ; dh <- y

shr al, 1 ; y >> 1 (其实是 y/BPB\_NumHeads, 这里BPB\_NumHeads=2)

mov ch, al ; ch <- 柱面号

and dh, 1 ; dh & 1 = 磁头号

pop bx ; 恢复 bx

; 至此, "柱面号, 起始扇区, 磁头号" 全部得到 ^^^^^^^^^^^^^^^^^^^^^^^^

mov dl, [BS\_DrvNum] ; 驱动器号 (0 表示 A 盘)

.GoOnReading:

mov ah, 2 ; 读

mov al, byte [bp-2] ; 读 al 个扇区

int 13h

jc .GoOnReading ; 如果读取错误 CF 会被置为 1, 这时就不停地读, 直到正确为止

add esp, 2

pop bp

ret

说到这里，我想提一句，我们写了这个函数，当然不是一蹴而就的，中间很可能会有错误，最好能够同时调试。所以，在开发过程中，我们就把“%define \_BOOT\_DEBUG\_”这个定义打开，编译成.COM文件，这样可以随时在DOS下进行调试。

还要注意的是，由于这段代码中用到了堆栈，要在程序开头初始化ss和esp：

%ifdef \_BOOT\_DEBUG\_

BaseOfStack equ 0100h ; 调试状态下堆栈基地址(栈底, 从这个位置向低地址生长)

%else

BaseOfStack equ 07c00h ; Boot状态下堆栈基地址(栈底, 从这个位置向低地址生长)

%endif

mov ax, cs

mov ds, ax

mov es, ax

mov ss, ax

mov sp, BaseOfStack

好了，读扇区的函数写好了，下面我们就开始编写在软盘中寻找Loader.bin的代码。

代码4-5

mov sp, BaseOfStack

xor ah, ah ; ┓

xor dl, dl ; ┣ 软驱复位

int 13h ; ┛

; 下面在 A 盘的根目录寻找 LOADER.BIN

mov word [wSectorNo], SectorNoOfRootDirectory

LABEL\_SEARCH\_IN\_ROOT\_DIR\_BEGIN:

cmp word [wRootDirSizeForLoop], 0 ; ┓

jz LABEL\_NO\_LOADERBIN ; ┣ 判断根目录区是不是已经读完

dec word [wRootDirSizeForLoop] ; ┛ 如果读完表示没有找到 LOADER.BIN

mov ax, BaseOfLoader

mov es, ax ; es <- BaseOfLoader

mov bx, OffsetOfLoader ; bx <- OffsetOfLoader 于是, es:bx = BaseOfLoader:OffsetOfLoader

mov ax, [wSectorNo] ; ax <- Root Directory 中的某 Sector 号

mov cl, 1

call ReadSector

mov si, LoaderFileName ; ds:si -> "LOADER BIN"

mov di, OffsetOfLoader ; es:di -> BaseOfLoader:0100 = BaseOfLoader\*10h+100

cld

mov dx, 10h

LABEL\_SEARCH\_FOR\_LOADERBIN:

cmp dx, 0 ; ┓循环次数控制,

jz LABEL\_GOTO\_NEXT\_SECTOR\_IN\_ROOT\_DIR ; ┣如果已经读完了一个 Sector,

dec dx ; ┛就跳到下一个 Sector

mov cx, 11

LABEL\_CMP\_FILENAME:

cmp cx, 0

jz LABEL\_FILENAME\_FOUND ; 如果比较了 11 个字符都相等, 表示找到

dec cx

lodsb ; ds:si -> al

cmp al, byte [es:di]

jz LABEL\_GO\_ON

jmp LABEL\_DIFFERENT ; 只要发现不一样的字符就表明本 DirectoryEntry 不是

; 我们要找的 LOADER.BIN

LABEL\_GO\_ON:

inc di

jmp LABEL\_CMP\_FILENAME ; 继续循环

LABEL\_DIFFERENT:

and di, 0FFE0h ; else ┓ di &= E0 为了让它指向本条目开头

add di, 20h ; ┃

mov si, LoaderFileName ; ┣ di += 20h 下一个目录条目

jmp LABEL\_SEARCH\_FOR\_LOADERBIN; ┛

LABEL\_GOTO\_NEXT\_SECTOR\_IN\_ROOT\_DIR:

add word [wSectorNo], 1

jmp LABEL\_SEARCH\_IN\_ROOT\_DIR\_BEGIN

LABEL\_NO\_LOADERBIN:

mov dh, 2 ; "No LOADER."

call DispStr ; 显示字符串

%ifdef \_BOOT\_DEBUG\_

mov ax, 4c00h ; ┓

int 21h ; ┛没有找到 LOADER.BIN, 回到 DOS

%else

jmp $ ; 没有找到 LOADER.BIN, 死循环在这里

%endif

LABEL\_FILENAME\_FOUND: ; 找到 LOADER.BIN 后便来到这里继续

jmp $ ; 代码暂时停在这里

这段代码看上去稍微有一点复杂，但逻辑是很清晰的，就是遍历根目录区所有的扇区，将每一个扇区加载入内存，然后从中寻找文件名为Loader.bin的条目，直到找到为止。找到的那一刻，es:di是指向条目中字母N后面的那个字符。代码中注释比较多，仔细看还是不困难的，在这里就不做过多解释了。其中用到的变量和字符串的定义请参见代码4-6。里面还包含其他一点变量和字符串的值，在以后的代码中会用到。

BaseOfLoader equ 09000h ; LOADER.BIN 被加载到的位置 ---- 段地址

OffsetOfLoader equ 0100h ; LOADER.BIN 被加载到的位置 ---- 偏移地址

RootDirSectors equ 14 ; 根目录占用空间

SectorNoOfRootDirectory equ 19 ; Root Directory 的第一个扇区号

;变量

;----------------------------------------------------------------------------

wRootDirSizeForLoop dw RootDirSectors ; Root Directory 占用的扇区数, 在循环中会递减至零.

wSectorNo dw 0 ; 要读取的扇区号

bOdd db 0 ; 奇数还是偶数

;============================================================================

;字符串

;----------------------------------------------------------------------------

LoaderFileName db "LOADER BIN", 0 ; LOADER.BIN 之文件名

; 为简化代码, 下面每个字符串的长度均为 MessageLength

MessageLength equ 9

BootMessage: db "Booting "; 9字节, 不够则用空格补齐. 序号 0

Message1 db "Ready. "; 9字节, 不够则用空格补齐. 序号 1

Message2 db "No LOADER"; 9字节, 不够则用空格补齐. 序号 2

需要注意的一点是，由于在读取过程中打印一些字符串，我们需要一个函数来做这项工作。为了节省代码长度，字符串的长度都设为9字节，不够则用空格补齐，这样就相当于一个二维数组，定位的时候通过数字就可以了，非常方便。显示字符串的函数名叫做DispStr，调用它的时候只要保证寄存器dh的值是字符串的序号就可以了。

DispStr:

mov ax, MessageLength

mul dh

add ax, BootMessage

mov bp, ax ; ┓

mov ax, ds ; ┣ ES:BP = 串地址

mov es, ax ; ┛

mov cx, MessageLength ; CX = 串长度

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

mov bx, 0007h ; 页号为0(BH = 0) 黑底白字(BL = 07h)

mov dl, 0

int 10h ; int 10h

ret

同时我们看到，在找到Loader.bin之后，程序死循环在那里，我们暂时先这样做，等到调试通过再继续进行，下面我们先编译一下生成一个Boot.com。

Loader.bin复制到虚拟软盘(\*.IMG)上，具体调试过程见原文。

现在我们已经有了Loader.bin的起始扇区号，我们需要用这个扇区号来做两件事：一件是把起始扇区装入内存，另一件则是通过它找到FAT中的项，从而找到Loader占用的其余所有扇区。

在这里，我们把Loader装入内存的BaseOfLoader:OffsetOfLoader处，你可能发现了，我们在代码4-5中根目录区也是装到这个位置，这算是资源的回收利用吧。因为我们装入根目录区仅仅是为了寻找相应的目录条目而已，找到之后，根目录区对我们就没有用了。所以，现在我们尽管用Loader把它覆盖掉，没有关系。

装入一下扇区对我们来说已经是轻松的事了，可从FAT中找到一个项还多少有些麻烦。而且，如果Loader占用多个扇区的话，我们可能需要重复从FAT中找相应的项，所以，还是写一个函数来做这件事。函数的输入就是扇区号，输出则是其对应的FAT项的值。

代码4-8，chapter4/c/boot.asm

GetFATEntry:

push es

push bx

push ax

mov ax, BaseOfLoader ; ┓

sub ax, 0100h ; ┣ 在 BaseOfLoader 后面留出 4K 空间用于存放 FAT

mov es, ax ; ┛

pop ax

mov byte [bOdd], 0

mov bx, 3

mul bx ; dx:ax = ax \* 3

mov bx, 2

div bx ; dx:ax / 2 ==> ax <- 商, dx <- 余数

cmp dx, 0

jz LABEL\_EVEN

mov byte [bOdd], 1

LABEL\_EVEN:;偶数

xor dx, dx ; 现在 ax 中是 FATEntry 在 FAT 中的偏移量. 下面来计算 FATEntry 在哪个扇区中(FAT占用不止一个扇区)

mov bx, [BPB\_BytsPerSec]

div bx ; dx:ax / BPB\_BytsPerSec ==> ax <- 商 (FATEntry 所在的扇区相对于 FAT 来说的扇区号)

; dx <- 余数 (FATEntry 在扇区内的偏移)。

push dx

mov bx, 0 ; bx <- 0 于是, es:bx = (BaseOfLoader - 100):00 = (BaseOfLoader - 100) \* 10h

add ax, SectorNoOfFAT1 ; 此句执行之后的 ax 就是 FATEntry 所在的扇区号

mov cl, 2

call ReadSector ; 读取 FATEntry 所在的扇区, 一次读两个, 避免在边界发生错误, 因为一个 FATEntry 可能跨越两个扇

区

pop dx

add bx, dx

mov ax, [es:bx]

cmp byte [bOdd], 1

jnz LABEL\_EVEN\_2

shr ax, 4

LABEL\_EVEN\_2:

and ax, 0FFFh

LABEL\_GET\_FAT\_ENRY\_OK:

pop bx

pop es

ret

其中，SectorNoOfFAT1的定义如下：

SectorNoOfFAT1 equ 1 ; FAT1 的第一个扇区号 = BPB\_RsvdSecCnt

连同前面的RootDirSectors、SectorNoOfRootDirectory等宏一起，与FAT12有关的几个数字我们都定义成了宏，而不是在程序中进行计算。

。。。

好了，现在一切俱备了，让我们开始加载Loader吧，代码4-9。

LABEL\_FILENAME\_FOUND: ; 找到 LOADER.BIN 后便来到这里继续

mov ax, RootDirSectors

and di, 0FFE0h ; di -> 当前条目的开始

add di, 01Ah ; di -> 首 Sector

mov cx, word [es:di]

push cx ; 保存此 Sector 在 FAT 中的序号

add cx, ax

add cx, DeltaSectorNo ; 这句完成时 cl 里面变成 LOADER.BIN 的起始扇区号 (从 0 开始数的序号)

mov ax, BaseOfLoader

mov es, ax ; es <- BaseOfLoader

mov bx, OffsetOfLoader ; bx <- OffsetOfLoader 于是, es:bx = BaseOfLoader:OffsetOfLoader = BaseOfLoader \* 10h + OffsetOfLoader

mov ax, cx ; ax <- Sector 号

LABEL\_GOON\_LOADING\_FILE:

push ax ; ┓

push bx ; ┃

mov ah, 0Eh ; ┃ 每读一个扇区就在 "Booting " 后面打一个点, 形成这样的效果:

mov al, '.' ; ┃

mov bl, 0Fh ; ┃ Booting ......

int 10h ; ┃

pop bx ; ┃

pop ax ; ┛

mov cl, 1

call ReadSector

pop ax ; 取出此 Sector 在 FAT 中的序号

call GetFATEntry

cmp ax, 0FFFh

jz LABEL\_FILE\_LOADED

push ax ; 保存 Sector 在 FAT 中的序号

mov dx, RootDirSectors

add ax, dx

add ax, DeltaSectorNo

add bx, [BPB\_BytsPerSec]

jmp LABEL\_GOON\_LOADING\_FILE

LABEL\_FILE\_LOADED:

在代码4-9中我们又看到一个新的宏 DeltaSectorNo，那么它是做什么的呢？我们还是拿4.1.1节中的例子来看，文件RIVER.TXT对应的目录条目中的开始族号是2，但我们显然不能根据这个数字2来读取扇区。实际上，开始族号是2对应的是数据区的第一个扇区。所以，我们需要有一个方法来计算簇号为X代表从引导扇区开始算起是第几个扇区。

我们已经知道，根目录区占用的空间是RootDirSectors，即14个扇区，虽然根目录区的开始扇区号是19，但我们却要用“X+RootDirSectors+19-2”算出来的扇区号才是“33”这个正确的值。所以，我们又定义了一个宏 DeltaSectorNo为17(即19-2)来帮助计算正确的扇区号：

DeltaSectorNo equ 17 ; DeltaSectorNo = BPB\_RsvdSecCnt + (BPB\_NumFATs \* FATSz) - 2

**4.1.5向Loader交出控制权**

上面的代码调试通过后，我们就已经成功地将Loader加载如内存，下面让我们来一个跳转，开始执行Loader。

jmp BaseOfLoader:OffsetOfLoader ; 这一句正式跳转到已加载到内存中的 LOADER.BIN 的开始处

; 开始执行 LOADER.BIN 的代码

; Boot Sector 的使命到此结束

**4.1.6整理boot.asm**

最后，我们来对boot.asm进行整理，以便让执行的效果更好一点。而且，把它写进引导扇区(别忘了编译前注释掉\_BOOT\_DEBUG\_宏)。运行结果如图4-9所示。

图4-9 Loader开始执行

代码4-11首先清屏，然后显示字符串Booting。这样，加载Loader时打印的圆点也会出现在这个字符串的后面。

mov ss, ax

mov sp, BaseOfStack

; 清屏

mov ax, 0600h ; AH = 6, AL = 0h

mov bx, 0700h ; 黑底白字(BL = 07h)

mov cx, 0 ; 左上角: (0, 0)

mov dx, 0184fh ; 右下角: (80, 50)

int 10h ; int 10h

mov dh, 0 ; "Booting "

call DispStr ; 显示字符串

LABEL\_FILE\_LOADED:

mov dh, 1 ; "Ready."

call DispStr ; 显示字符串

;

jmp BaseOfLoader:OffsetOfLoader

代码4-12实现的是在全部加载完毕准备跳入Loader之前打印字符串“Ready.”。

Loader.bin本质上是.COM文件，最大也不可能超过64KB。但是，我们已经成功突破512字节限制，这个进步无疑是巨大的。

回想一下，我们做了如此多的工作，但代码长度仍然保存在512字节之内，是不是很精妙/很有趣呢？

4.2保护模式下的“操作系统”

Linux的引导扇区代码boot.s比我们的代码简单，它直接把内核移动到目标内存。我们的代码之所以负责一些，是因为我们想和MSDOS的磁盘格式兼容，以便调试的时候容易一些。虽然开始时我们多做了工作，但是从长远来看，我认为是值得的。

Loader毕竟不是OS，所以Loader要做的事情至少有两件：

● 加载内核入内存

● 跳入保护模式

不过，内核是在Linux下编译链接出的ELF格式文件，直接放进内存肯定是不行的，没关系，下一章开始就会研究ELF的格式。

**第五章 内核雏形**

研究ELF需要一个样本，我们恰好可以通过写一个小的内核来做样本。现在就让我们来到Linux，写一个尽可能小但足以做加载试验的“内核”。

**5.1用NASM在Linux下写Hello World P166**

我们提到，完成从实模式到保护模式跳转这一任务的应该是Loader，那么Loader应该走多远呢？只完成跳转，还是应该把GDT、IDT、8259A等内容准备完备？实际上，从逻辑上讲，Loader不是操作系统的一部分，所以不应该越俎代庖。而且，你一定也希望早早结束Loader的工作进入正题。所以，我们还是要让Loader尽量简单，其余的工作留给内核来做。

之所以提到这个问题，是因为我们需要知道内核应该由哪些编程语言来完成。我们当然希望尽早摆脱汇编进入C的世界。但事实上却的确还有一些功能需要继续使用汇编来完成其中的部分代码，比如时钟中断处理、异常处理等。你将会发现，甚至于进程调度的一部分代码也是用汇编来完成的。

所以，在Linux下我们仍然离不开汇编。还好，我们选择了NASM这一在各种平台都能使用的工具。下面就来体验一下用NASM在Linux下编程的感觉。

代码5-1 chapter5/a/hello.asm

[section .data] ; 数据在此

strHello db "Hello, world!", 0Ah

STRLEN equ $ - strHello

[section .text] ; 代码在此

global \_start ; 我们必须导出 \_start 这个入口，以便让链接器识别。

\_start:

mov edx, STRLEN

mov ecx, strHello

mov ebx, 1

mov eax, 4 ; sys\_write

int 0x80 ; 系统调用

mov ebx, 0

mov eax, 1 ; sys\_exit

int 0x80 ; 系统调用

编译、链接和运行，由于本人的机器是54位，所以参数是elf64，i386的参数是elf

nasm -f elf64 hello.asm -o hello.o

ld -s hello.o -o hello

wujian at wujian in ~/MyOS/bochs-work/chapter5/a (master●●)

$ ./hello

Hello, world!

选项“-s”中字母s是strip的简写，可以去掉符号表等内容，可起到对生成的可执行代码减肥之功效。

我们回头看看代码5-1，程序中定义了两个节(Section)，一个放数据，一个放代码。在代码中值得注意的一点是，入口默认的是“\_start”，我们不但要定义它，而且要通过global这个关键字将它导出，这样链接程序才能找到它。至于代码本身，你需要知道它们是两个系统调用，用来显示字符串并退出就够了。至于为什么这么做倒不用研究，因为在我们自己的OS中根本用不到Linux的系统调用。

**5.2再进一步，汇编和C同步使用 P168**

本次的例子中，汇编和C会进行同步调用

之所以这样安排，是因为它包含了汇编代码和C代码之间相互的调用，今后会用到。

代码5-2，chapter5/b/foo.asm

extern choose ; int choose(int a, int b);

[section .data] ; 数据在此

num1st dd 3

num2nd dd 4

[section .text] ; 代码在此

global \_start ; 我们必须导出 \_start 这个入口，以便让链接器识别。

global myprint ; 导出这个函数为了让 bar.c 使用

\_start:

push num2nd ; ┓

push num1st ; ┃

call choose ; ┣ choose(num1st, num2nd);

add esp, 4 ; ┛

mov ebx, 0

mov eax, 1 ; sys\_exit

int 0x80 ; 系统调用

; void myprint(char\* msg, int len)

myprint:

mov edx, [esp + 8] ; len

mov ecx, [esp + 4] ; msg

mov ebx, 1

mov eax, 4 ; sys\_write

int 0x80 ; 系统调用

ret

(1)函数myprint由关键字global导出，给bar.c用

(2)由于用到C函数choose，所以要用extern声明

(3)无论是myprint还是choose，遵循的都是C的调用约定，后面的参数先入栈，并由调用者清理堆栈

代码5-3，bar.c文件

void myprint(char\* msg, int len);

int choose(int a, int b)

{

if(a >= b){

myprint("the 1st one\n", 13);

}

else{

myprint("the 2nd one\n", 13);

}

return 0;

}

**5.3 ELF(Executable and Linkable Format) P170**

本章节是对ELF的非常入门级别的介绍，旨在了解programmer header所描述的code segment和data segment，根据这些信息，可以知道foobar加载进内存之后的信息。

下一步就要扩充Loader了。

**5.4从Loader到内核**

研究过ELF，回忆一下Loader需要做的工作：加载内核到内存，跳入保护模式，我们先来做第一项，把内核加载到内存。

**5.4.1用Loader加载ELF**

加载内核到内存这一步和引导扇区的工作非常相似，只是处理内核时我们需要根据Program header table中的值把内核中相应的段放到正确的位置。我们可以这样来做，首先像引导扇区处理Loader那样把内核放入内存，只要内核进入了内存，如何处理它便是一件容易的事情了，我们可以在保护模式下挪动它的位置。

依旧是寻找文件、定位文件以及读入内存，实际上，单就把内核读入内存这一部分，除了文件名和读入的内存地址变了，其余其实都一样。之所以没有把它写成一个函数分别在boot.asm和loader.asm，是因为函数在调用时堆栈操作会占用更多的空间，在引导扇区中，每一个字节都是珍贵的。

不过，一些常量的定义却可以在boot.asm和loader.asm之间共享。我们不妨把与FAT12文件有关的内容写进一个单独的文件(文件名为fat12hdr.inc，见代码5-4，详细在chapter5\c\)，在两个文件的开头相应的位置分别包含进去。

然后就是把内放进内存，loader.asm，这里就不贴出来代码了。

最后在准备一个kernel，kernel.asm，没有做任何工作，因为本章节只是为了把kernel.bin找到并读入内存，运行一下。

**5.4.2跳入保护模式 P181**

现在，内核已经被我们加载进内存了，该是跳入保护模式的时候了。

首先是GDT以及对应的选择子，我们只定义三个描述符，分别是一个0~4GB的可执行段、一个0~4GB的可读写段和一个指向显存开始地址的段(见代码5-9)，chapter5/d/loader.asm

jmp LABEL\_START ; Start

; 下面是 FAT12 磁盘的头, 之所以包含它是因为下面用到了磁盘的一些信息

%include "fat12hdr.inc"

%include "load.inc"

%include "pm.inc"

; GDT ------------------------------------------------------------------------------------------------------------------------------------------------------------

; 段基址 段界限 , 属性

LABEL\_GDT: Descriptor 0, 0, 0 ; 空描述符

LABEL\_DESC\_FLAT\_C: Descriptor 0, 0fffffh, DA\_CR | DA\_32 | DA\_LIMIT\_4K ; 0 ~ 4G

LABEL\_DESC\_FLAT\_RW: Descriptor 0, 0fffffh, DA\_DRW | DA\_32 | DA\_LIMIT\_4K ; 0 ~ 4G

LABEL\_DESC\_VIDEO: Descriptor 0B8000h, 0ffffh, DA\_DRW | DA\_DPL3 ; >显存首地址

; GDT ------------------------------------------------------------------------------------------------------------------------------------------------------------

GdtLen equ $ - LABEL\_GDT

GdtPtr dw GdtLen - 1 ; 段界限

dd BaseOfLoaderPhyAddr + LABEL\_GDT ; 基地址

; GDT 选择子 ----------------------------------------------------------------------------------

SelectorFlatC equ LABEL\_DESC\_FLAT\_C - LABEL\_GDT

SelectorFlatRW equ LABEL\_DESC\_FLAT\_RW - LABEL\_GDT

SelectorVideo equ LABEL\_DESC\_VIDEO - LABEL\_GDT + SA\_RPL3

在第3章我们学习保护模式时，大部分描述符的段基址都是运行时计算后填入相应位置的，因为那时我们的程序是由DOS加载的，我们不知道段地址，于是也就不知道程序运行时在内存中的位置。如今，Loader是由我们自己加载的，段地址已经被确定为BaseOfLoader，所以在Loader中出现的标号(变量)的物理地址可以用下面的公式来表示：

标号(变量)的物理地址 = BaseOfLoader \* 10h + 标号(变量)的偏移

不过，这样一来， BaseOfLoader就同时在boot.asm和loader.asm两个文件中使用，我们也把它以及相应的声明放在同一文件load.inc中，见代码5-10

BaseOfLoader equ 09000h ; LOADER.BIN 被加载到的位置 ---- 段地址

OffsetOfLoader equ 0100h ; LOADER.BIN 被加载到的位置 ---- 偏移地址

BaseOfLoaderPhyAddr equ BaseOfLoader \* 10h ; LOADER.BIN 被加载到的位置 ---- 物理地址 (= BaseOfLoader \* 10h)

BaseOfKernelFile equ 08000h ; KERNEL.BIN 被加载到的位置 ---- 段地址

OffsetOfKernelFile equ 0h ; KERNEL.BIN 被加载到的位置 ---- 偏移地址

我们即将进入保护模式，仍然像过去一样，进入之后只是打印一个字符，并不做太多工作。

初始化各个寄存器的值，分配一个够用的堆栈1KB，等到我们进入内核时，可以重新设置堆栈。

下面，我们打开分页机制，打开之前还是应该先知道可使用内存的情况。因此，我们在Loader的开头再增加一些代码：

; 得到内存数

mov ebx, 0 ; ebx = 后续值, 开始时需为 0

mov di, \_MemChkBuf ; es:di 指向一个地址范围描述符结构（Address Range Descriptor Structure）

.MemChkLoop:

mov eax, 0E820h ; eax = 0000E820h

mov ecx, 20 ; ecx = 地址范围描述符结构的大小

mov edx, 0534D4150h ; edx = 'SMAP'

int 15h ; int 15h

jc .MemChkFail

add di, 20

inc dword [\_dwMCRNumber] ; dwMCRNumber = ARDS 的个数

cmp ebx, 0

jne .MemChkLoop

jmp .MemChkOK

.MemChkFail:

mov dword [\_dwMCRNumber], 0

.MemChkOK:

得到内存信息之后，就可以添加启动分页的代码了。

运行，结果如图5-13所示。

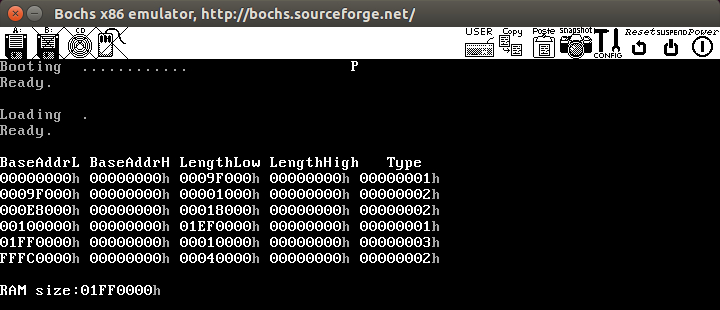


图5-13 Loader列出内存情况

**5.4.3重新放置内核**

我们的loader.asm正在飞速膨胀，不要紧，马上就要冲刺了，终点就在前方，因为我们马上就要整理内存中的内核并把控制权交给它了。

实际上，我们要做的工作是根据内核的Program header table的信息进行类似下面这个C语言的内存复制：

memcpy(p\_vaddr, BaseOfLoaderPhyAddr + p\_offset, p\_filesz);

复制可能不止一次，如果Program header有n个，复制就进行n次。

我们说过，每一个Program header都描述一个段。语句中的p\_offset为段在文件中的偏移，p\_files为段在文件中的长度，p\_vaddr为段在内存中的虚拟地址。

说到这里，你可能想起一件事情，就是由ld生成的可执行文件中p\_vaddr的值总是一个类似于0x8048XXX的值，至少我们的例子中是一个这样的值。可是我们启动分页机制时地址都是对等映射的，内存地址0x8048XXX已经处在128MB内存以外(128MB的十六进制表示是0x8000000)，如果计算机的内存小于128MB的话，这个地址显然已经超出了内存大小。

即便计算机有足够大的内存，显然，我们也不能让编译器来决定内核加载到什么地方。我们得让它受控制，解决它有两个办法，一是通过修改页表让0x8048XXX映射到较低的地址，另一种方法就是通过修改ld的选项让它生成的可执行代码中p\_vaddr的值变小。

显然，第二种方法更加简单易行，下面我们就把编译链接时的命令行改为：

[root@XXX XXX]# nasm -f elf kernel.asm -o kernel.o

[root@XXX XXX]# ld -s -Ttext 0x30400 -o kernel.bin kernel.o

程序的入口地址就变成0x30400了，ELF header等信息会位于0x30400之前。此时的ELF header和Program header table的情况如表5-3和表5-4所示。

我们应该这样放置内核：

memcpy(30000h, 90000h + 0, 40Dh);

也就是说，我们应该把文件从开头开始共40Dh字节的内容放到内存30000h处。由于程序的入口在30400h处，所以从这里可以看出，实际上代码只有0Dh+1个字节。我们来看Kernel.bin的内容。

00000060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|

\*

00000400 b4 0f b0 4b 65 66 a3 ee 00 00 00 eb fe 00 54 68 |...Kef........Th|

00000410 65 20 4e 65 74 77 69 64 65 20 41 73 73 65 6d 62 |e Netwide Assemb|

从中可以看出，从400h到40Dh是仅有的代码，0xebfe正是代码最后“jmp $”。

代码5-22实现了将Kernel.bin根据ELF文件信息转移到正确的位置。它很简单，找出每个Program header，根据其信息进行内存复制。

代码5-22 InitKernel，chapter5/e/loader.asm

; InitKernel ---------------------------------------------------------------------------------

; 将 KERNEL.BIN 的内容经过整理对齐后放到新的位置

; --------------------------------------------------------------------------------------------

InitKernel: ; 遍历每一个 Program Header，根据 Program Header 中的信息来确定把什么放进内存，放到什么位置，以及放多少。

xor esi, esi

mov cx, word [BaseOfKernelFilePhyAddr + 2Ch]; ┓ ecx <- pELFHdr->e\_phnum

movzx ecx, cx ; ┛

mov esi, [BaseOfKernelFilePhyAddr + 1Ch] ; esi <- pELFHdr->e\_phoff

add esi, BaseOfKernelFilePhyAddr ; esi <- OffsetOfKernel + pELFHdr->e\_phoff

.Begin:

mov eax, [esi + 0]

cmp eax, 0 ; PT\_NULL

jz .NoAction

push dword [esi + 010h] ; size ┓

mov eax, [esi + 04h] ; ┃

add eax, BaseOfKernelFilePhyAddr ; ┣ ::memcpy( (void\*)(pPHdr->p\_vaddr),

push eax ; src ┃ uchCode + pPHdr->p\_offset,

push dword [esi + 08h] ; dst ┃ pPHdr->p\_filesz;

call MemCpy ; ┃

add esp, 12 ; ┛

.NoAction:

add esi, 020h ; esi += pELFHdr->e\_phentsize

dec ecx

jnz .Begin

ret

; InitKernel ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

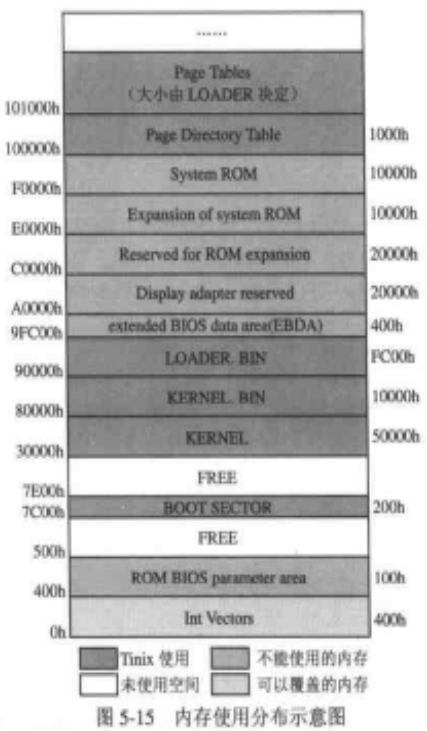


图5-15是一个内存使用分布图示。看第一眼的时候你可能有些惊讶，我们不是才往里面放了两个文件吗，怎么这么复杂？是的，虽然我们往里存放的内容不多，但它并不单纯。比如我们一直以来用做显示的以0xB8000为开始的内存，显然就不能被OS用在常规用途；再比如0x400~0x4FF这段内存，里面存放了许多参数，为了保证在用得着它们的时候它们还在，我们还是暂时保留不覆盖它为妙。

当你看到9FC00h这个数字的时候，不知道你是不是感到面熟，回头看看图3-32和表3-10就明白了，通过中断15h得到的内存信息已经明确地告诉哦我们，09FC00h~09FFFFh这段内存不能被用做常规使用。即便0h~09FBFFh可以被使用，仍然应该把BIOS参数区保护起来以备后用，所以，我们真正可以使用的内存是0500h~09FBFFh这一段。那么，为什么指定的入口地址0x30400离0x500还那么远呢？其实，之所以这么做是为了调试方便。因为大多数的DOS都不占用0x30000以上的内存地址，把内核加载到这里，即便在DOS下调试也不会覆盖掉ODS内存。之前说过，通过对代码中宏定义的一点修改，我们可以让引导程序运行在DOS下，如果有错误可以很方便地进行调试。如果你亲自实践的话，这种调试一定是少不了的。(Loader调试详见5.4.5节)

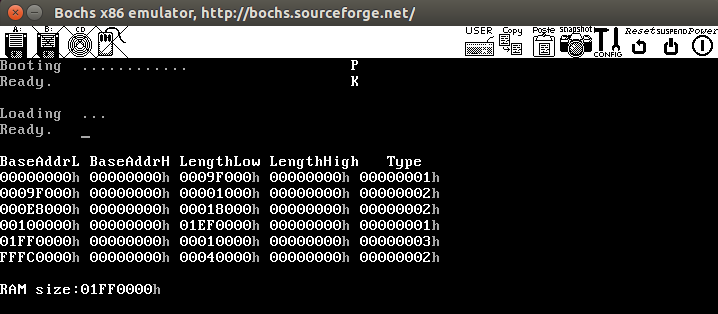
那么，现在情况很清楚了，0x90000开始的63KB留给了Loader.bin，0x80000开始的64KB留给了Kernel.bin，0x30000开始的320KB留给整理后的内核，而页目录和页表放置在了1MB以上的空间。

**5.4.4向内核交出控制权**

该是我们进行试验的时候了，下面我们就试着向内核跳转：

jmp SelectorFlatC:KernelEntryPointPhyAddr ; 正式进入内核 \*

其中， KernelEntryPointPhyAddr定义在load.inc中，值为0x30400。当然，它必须跟我们的ld的参数-Ttext指定的值是一致的。其实，将来如果我们想将内核放在另外的位置(比如1MB以上的内存)，只需改动这两个地方就可以了。

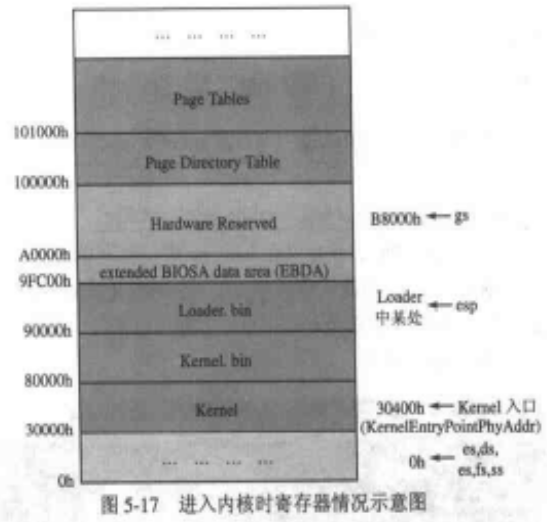
图5-16 向内核跳转成功

第二行中央出现字符K，这表明我们的内核在执行了。在下面的章节中，可以使用C语言进行扩充。

不过，在继续之前，让我们先来回顾一下在内核获得控制权之时各个寄存器的情况。

如图5-17所示，cs、ds、es、fs、ss表示的段统统指向内存地址0h，gs表示的段则指向显存，这是我们在进入保护模式之后设置的。

同时，esp、GDT等内容也在Loader中，下面对内核进行扩充时，我们会将它们都挪到内核中，以便于控制。



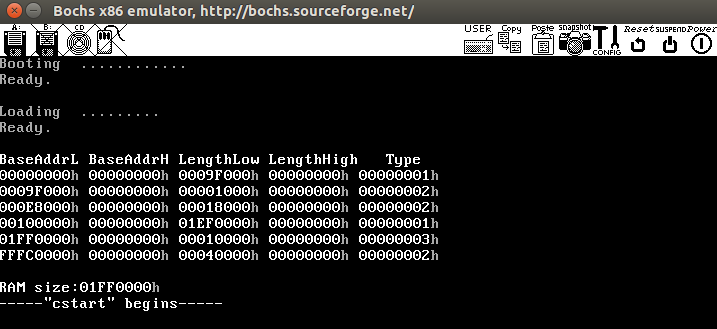
**5.4.5 操作系统的调试方法**

假如loader出现了问题，该如何解决，bochsdbg，跟gdb功能很类似。

**5.5扩充内核**

**5.5.1切换堆栈和GDT P204**

把esp和GDT等内容从loader中切换到内核中，可以用C语言来实现，编译运行下。



**5.5.2整理我们的文件夹**

**5.5.3 Makefile**

**5.5.4添加中断处理**

设置8259A和建立IDT，我们先来写一个函数设置8259A。

代码5-40，chapter5/i/kernel/i8259.c

#include "type.h"

#include "const.h"

#include "protect.h"

#include "proto.h"

/\*======================================================================\*

init\_8259A

\*======================================================================\*/

PUBLIC void init\_8259A()

{

out\_byte(INT\_M\_CTL, 0x11); // Master 8259, ICW1.

out\_byte(INT\_S\_CTL, 0x11); // Slave 8259, ICW1.

out\_byte(INT\_M\_CTLMASK, INT\_VECTOR\_IRQ0); // Master 8259, ICW2. 设置 '主8259' 的中断入口地址为 0x20.

out\_byte(INT\_S\_CTLMASK, INT\_VECTOR\_IRQ8); // Slave 8259, ICW2. 设置 '从8259' 的中断入口地址为 0x28

out\_byte(INT\_M\_CTLMASK, 0x4); // Master 8259, ICW3. IR2 对应 '从8259'.

out\_byte(INT\_S\_CTLMASK, 0x2); // Slave 8259, ICW3. 对应 '主8259' 的 IR2.

out\_byte(INT\_M\_CTLMASK, 0x1); // Master 8259, ICW4.

out\_byte(INT\_S\_CTLMASK, 0x1); // Slave 8259, ICW4.

out\_byte(INT\_M\_CTLMASK, 0xFD); // Master 8259, OCW1.

out\_byte(INT\_S\_CTLMASK, 0xFF); // Slave 8259, OCW1.

}

我们把初始化8259A的函数命名为init\_8259A，它本质上跟第三章中的代码3-38是一样的，只是由汇编代码转换成C代码，而且对应的端口被定义成宏。宏定义见代码5-41和5-42。

代码5-41，chapter5/i/include/const.h

/\* 8259A interrupt controller ports. \*/

#define INT\_M\_CTL 0x20 /\* I/O port for interrupt controller <Master> \*/

#define INT\_M\_CTLMASK 0x21 /\* setting bits in this port disables ints <Master> \*/

#define INT\_S\_CTL 0xA0 /\* I/O port for second interrupt controller <Slave> \*/

#define INT\_S\_CTLMASK 0xA1 /\* setting bits in this port disables ints <Slave> \*/

代码5-42，include/protect.h

/\* 中断向量 \*/

#define INT\_VECTOR\_IRQ0 0x20

#define INT\_VECTOR\_IRQ8 0x28

函数init\_8259A中只用到一个函数，就是用来写端口的**out\_byte**，它的函数体位于klib.asm中。其中，不但有out\_byte，用来对端口进行写操作，还有**in\_byte**，用来对端口进行读操作。由于端口操作可能需要时间，所以两个函数中都加了点空操作以便有微小的延迟。

out\_byte和in\_byte，还有start.c中函数disp\_str的声明也被挪到了里面。

PUBLIC void out\_byte(t\_port port, t\_8 value);

PUBLIC t\_8 in\_byte(t\_port port);

PUBLIC void disp\_str(char \* info);

其中又使用了一个新的类型t\_port，它的定义在chapter5/i/include/type.h中

代码5-45

typedef unsigned int t\_port;

说起IDT让我们不能不想起GDT，当初初始化它所用的方法，我们同样可以拿过来用。首先修改start.c。

代码5-47，chapter5/i/kernel/start.c

…...

#include "global.h"

…...

// idt\_ptr[6] 共 6 个字节：0~15:Limit 16~47:Base。用作 sidt 以及 lidt 的参数。

t\_16\* p\_idt\_limit = (t\_16\*)(&idt\_ptr[0]);

t\_32\* p\_idt\_base = (t\_32\*)(&idt\_ptr[2]);

\*p\_idt\_limit = IDT\_SIZE \* sizeof(GATE) - 1;

\*p\_idt\_base = (t\_32)&idt;

…...

代码跟先前初始化GDT的部分基本上是一样的，只是所有的GDT字眼变成了IDT。不过你会发现，原来位于start.c开头的gdt[]和gdt\_ptr[]等的声明不在了，取而代之的是对头文件global.h的包含。

start.c修改完之后，我们在kernel.asm中添加如下两句：

代码5-52，加载IDT的语句，chapter5/i/kernel/kernel.asm

…...

extern idt\_ptr

…...

lidt [idt\_ptr]

…...

回顾图3-44，从中可以看到，中断或异常发生时eflags、cs、eip已经被压栈，如果有错误码的话，错误码被压栈。所以我们对异常处理的总体思想是，如果有错误码，则直接把向量号压栈，然后执行一个函数exception\_handler；如果没有错误码，则先在栈中压入一个0xFFFFFFFF，再把向量号压栈并随后执行exception\_handler。

PUBLIC void exception\_handler(int vec\_no, int err\_code, int eip, int cs, int eflags)

由于C调用约定是调用者恢复堆栈，所以不用担心 exception\_handler会破坏堆栈中的eip、cs以及eflags。

代码5-53，chapter5/i/kernel/kernel.asm

…...

global divide\_error

global single\_step\_exception

global nmi

global breakpoint\_exception

global overflow

global bounds\_check

global inval\_opcode

global copr\_not\_available

global double\_fault

global copr\_seg\_overrun

global inval\_tss

global segment\_not\_present

global stack\_exception

global general\_protection

global page\_fault

global copr\_error

…...

; 中断和异常 -- 异常

divide\_error:

push 0xFFFFFFFF ; no err code

push 0 ; vector\_no = 0

jmp exception

single\_step\_exception:

push 0xFFFFFFFF ; no err code

push 1 ; vector\_no = 1

jmp exception

nmi:

push 0xFFFFFFFF ; no err code

push 2 ; vector\_no = 2

jmp exception

breakpoint\_exception:

push 0xFFFFFFFF ; no err code

push 3 ; vector\_no = 3

jmp exception

overflow:

push 0xFFFFFFFF ; no err code

push 4 ; vector\_no = 4

jmp exception

bounds\_check:

push 0xFFFFFFFF ; no err code

push 5 ; vector\_no = 5

jmp exception

…...

我们看到，在这段代码的最后，栈顶被调整为指向eip，堆栈中从顶向下依次是：eip、cs、

eflags。

下面我们就来看一下函数exception\_handler，它的实现实际上也很简单，首先把屏幕的前5行通过打印空格的方式清空，然后把堆栈中的参数打印出来，在protect.c中。

现在我们已经有了异常处理函数，该是设置IDT的时候了。我们把设置IDT的代码放进函数init\_prot()中，它也位于protect.c中。

protect.c通篇几乎只调用一个函数，就是init\_idt\_desc()，它用来初始化一个门描述符。

/\*======================================================================\*

init\_idt\_desc

\*----------------------------------------------------------------------\*

初始化 386 中断门

\*======================================================================\*/

PUBLIC void init\_idt\_desc(unsigned char vector, t\_8 desc\_type, t\_pf\_int\_handler handler, unsigned char privilege)

{

GATE \* p\_gate = &idt[vector];

t\_32 base = (t\_32)handler;

p\_gate->offset\_low = base & 0xFFFF;

p\_gate->selector = SELECTOR\_KERNEL\_CS;

p\_gate->dcount = 0;

p\_gate->attr = desc\_type | (privilege << 5);

p\_gate->offset\_high = (base >> 16) & 0xFFFF;

}

在init\_port()中，所有描述符都被初始化成中断门。

我们现在制造一个异常，Intel为我们准备了一个指令叫做ud2，能够产生一个#UD异常，我们就在kernel.asm中添加一条ud2指令。

代码5-62，chapter5/i/kernel/kernel.asm

csinit: ; “这个跳转指令强制使用刚刚初始化的结构”——<<OS:D&I 2nd>> P90.

;jmp 0x40:0

;ud2

sti

hlt

由于本文是64环境，所以编译出问题，就不进行截图演示了。

8259A设置虽然完成了，但是我们要真正使用它。复习一下图3-38，我们知道，两片级联的8259A可以挂接15个不同的外部设备，我们也理应有15个中断处理程序。

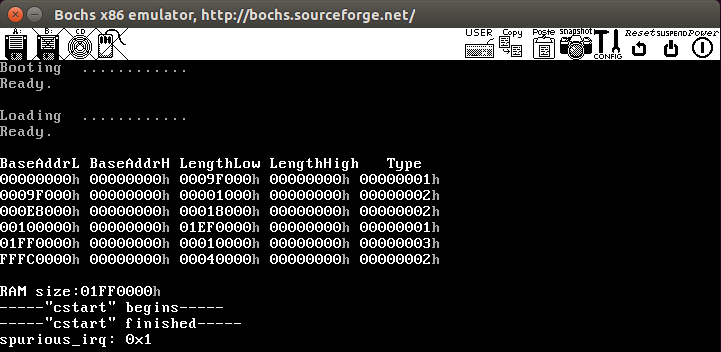
打开键盘中断，

out\_byte(INT\_M\_CTLMASK, 0xFD); // Master 8259, OCW1.

out\_byte(INT\_S\_CTLMASK, 0xFF); // Slave 8259, OCW1.

最后，在kernel.asm中添加sti指令设置IF位。

运行后，敲击任意按键，

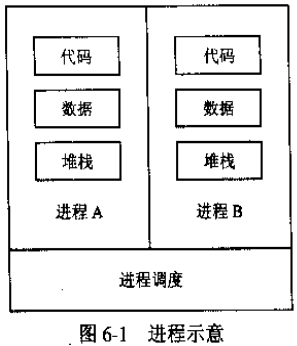
图5-35 键盘中断发生

**5.6小结**

略

**第六章 进程**

**6.2概述**

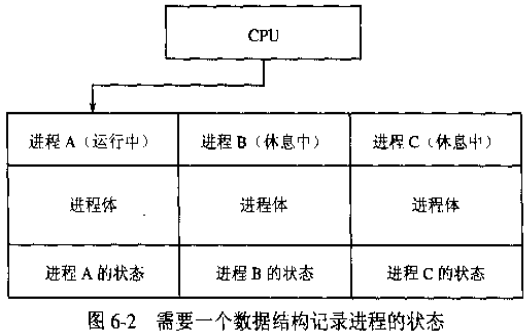
**6.2.1进程介绍**

我们先形成一个最简陋的进程，然后模仿它再写一个，变成两个。我们试着让它们同时运行，并让我们的系统对着它们进行调度，当然，使用的是最简单的调度算法。最后，再试着扩展进程的功能。

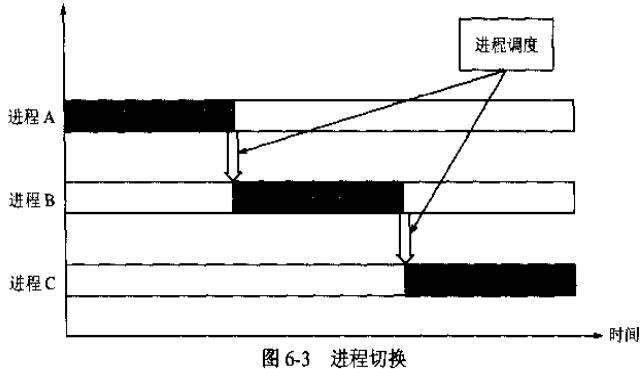
**6.2.2未雨绸缪——形成进程的必要考虑**

CPU大部分情况下只有一个，哪怕我们可以有多个CPU，我们也不能每增加一个进程增加一个CPU。这就决定了，在同一时刻，总是有“正在运行的”和“正在休息的”进程。所以，对于“正在休息的”进程，我们需要让它在重新醒来时记住自己挂起之前的状态，以便让原来的任务继续执行下去。

所以，我们需要一个数据结构记录一个进程的状态，在进程要被挂起的时候，进程信息就被写入这个数据结构，等到进程重新启动的时候，这个信息重新被读出来，见图6-2。

事情其实还要更加复杂些，因为在很多情况下，进程和进程调度是运行在不同的层级上。这里，本着简单的原则，我们让所有任务运行在ring1，而让进程切换运行在ring0。

不过，进程自己是不知道什么时候被挂起，什么时候又被启动的，诱发进程切换的原因不只一种，比较典型的情况是发生了时钟中断。当时钟中断发生时，中断处理程序会将控制权交给进程调度模块。这时，如果系统认为应该进行进程切换，进程切换就发生了，当前进程的状态会被保存起来，队列中的下一个进程将被恢复执行。图6-3表示了单CPU系统中进程切换的情况，黑色条表示进程处在运行态，白色表示进程处在休息态。在同一时刻，只能有一个进程处在运行态。进程切换的操作者是操作系统的进程调度模块。

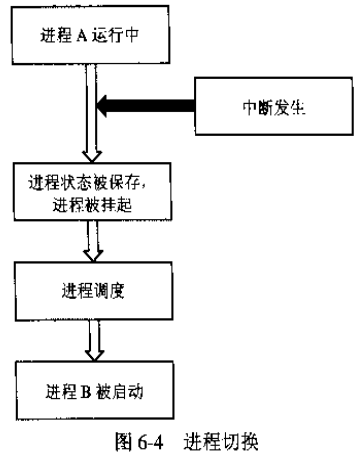
这里要说明的一点是，并非在每一次时钟中断时都一定会发生进程切换，不过为了容易理解和实现。

**6.2.3参考的代码**

我们正在写的操作系统的名字叫做Tinix，是Try+Minix的组合。

**6.3最简单的进程**

一个进程正在运行着，这时候时钟中断发生了，特权级从ring1跳到ring0，开始执行时钟中断处理程序，中断处理程序这时调用进程调度模块，指定下一个应该运行的进程，当中断处理程序结束时，下一个进程准备就绪并开始运行，特权级又从ring0跳回ring1，如图6-4所示。我们把这个过程按照时间顺序整理如下：

****

(1)进程A运行中。

(2)时钟中断发生，ring1->ring0，时钟中断处理程序启动。

(3)进程调度，下一个应运行的进程(假设为进程B)被指定。

(4)进程B被恢复，ring0->ring1。

(5)进程B运行中。

要想实现这些功能，我们必须完成的应该有以下几项：

● 时钟中断处理程序

● 进程调度模块

● 两个进程

**6.3.1简单进程的关键技术预测**

**6.3.1.1进程的哪些状态需要被保存**

只有可能被改变的才有保存的必要。我们的进程要运行，不外乎CPU和内存在相互协作，而不同进程的内存互不干涉(我们考虑最简单的情况，假设内存足够大)，但是我们提到过，CPU只有一个，不同进程共用一个CPU的一套寄存器。所以，我们要把寄存器的值统统保存起来，准备进程被恢复执行时使用。

**6.3.1.2进程的状态需要何时以及怎样被保存**

为了保证进程状态完整，不被破坏，我们当然希望在进程刚刚被挂起时保存所有寄存器的值。你一定在想，保存寄存器我已经很拿手，push就可以了，没错，push就够了。不过，Intel想得更周到，不但有push，更有pushad，一条指令可以保存许多寄存器值。而这些代码，我们应该把它写在时钟例程的最顶端，以便中断发生时马上被执行。

**6.3.1.3如何恢复进程B的状态**

不用说，你一定早就想到了，保存是为了恢复，既然保存用的是push，恢复一定用pop了。等所有寄存器的值都已经被恢复，执行指令iretd，就回到了进程B。

**6.3.1.4 进程表的引入**

对于“进程状态”，有个专业的称呼，那就是“进程表”。我们通过进程表对进程进行管理。

从代码编写的这个角度看，除中断处理的部分内容我们不得不使用汇编之外，我们还是要用C来编写大部分进程管理的内容。

我们会建立一个进程表数组，进程表是用来描述进程的，所以它必须独立于进程之外。所以，当我们把寄存器值压到进程表内的时候，已经处在进程管理模块中了。

**6.3.1.5进程栈和内核栈**

当寄存器的值已经被保存到进程表内，进程调度模块就开始执行了。但这时有一个很重要的问题容易被忽视，就是esp现在指向何处。

毫无疑问，我们在进程调度模块中会用到堆栈，而寄存器被压到进程表之后，esp是指向进程表某个位置的。这就有了问题，如果接下来进程任何的堆栈操作，都会破坏掉进程表的值，从而在下一次进程恢复时产生严重的错误。

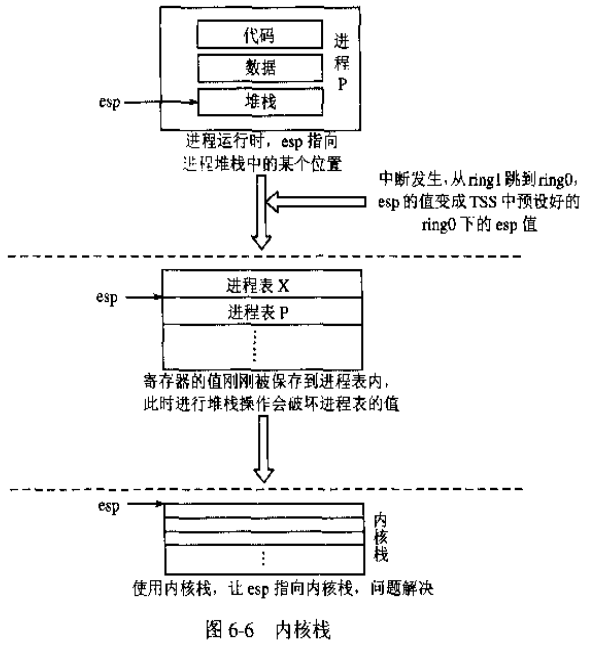
为解决这个问题，避免错误的出现，一定要记得将esp指向专门的内核栈区域。这样，在短短的进程切换过程中，esp的位置出现在3个不同的区域，如图6-6

● 进程栈 —— 进程运行时自身的堆栈

● 进程表 —— 存储进程状态信息的数据结构

● 内核栈 —— 进程调度模块运行时使用的堆栈

在具体编写代码的过程中，一定要清楚当前使用的是哪个堆栈，以免破坏掉不应破坏的数据。

**6.3.1.6特权级变换: ring1->ring0**

为每个进程准备好LDT。

**6.3.1.7特权级变换: ring0->ring1**

第一个进程，我们面临一个从ring0到ring1的转移，并启动进程A。这跟我们从进程B恢复的情况很相似，所以我们完全可以在准备就绪之后跳转到中断处理程序的后半部分，“假装”发生了一次时钟中断来启动进程A，利用iretd来实现ring0到ring1的转移。

**6.3.2第一步——ring0->ring1 P247**

我们已经看到，即便是想象中最简单的进程，仍然需要不少的关键技术。我们注意到，在开始第一个进程时，我们打算使用iretd来实现由ring0->ring1的转移，一旦转移成功，便可以认为已经在一个进程中运行了。下面就开始这一部分。

为了对这一部分的实现有一个感性认识，我们先来看一下第6章最终实现的代码(\chapter6\i)

中kernel.asm的一小部分：

代码6-1

restart:

mov esp, [p\_proc\_ready]

lldt [esp + P\_LDT\_SEL]

lea eax, [esp + P\_STACKTOP]

mov dword [tss + TSS3\_S\_SP0], eax

restart\_reenter:

dec dword [k\_reenter]

pop gs

pop fs

pop es

pop ds

popad

add esp, 4

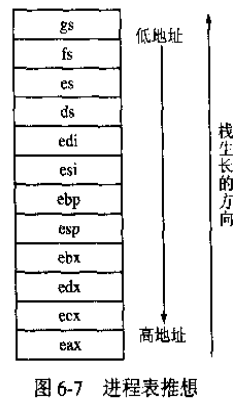
iretd

为了容易理解，先来看一看本章所附的代码的部分内容。因为进程毕竟是个新鲜事物，它涉及若干方面，如果一开始就下手行动，很可能会无所适从。  
 在文件夹\kernel中，你会发现多了一个main.c，里面有一个函数tinix\_main()，从中可以找到这样一行：

restart();

它调用的便是代码6-1这一段，它是进程调度的一部分，同时也是我们的操作系统启动第一个进程时的入口。

第二行mov esp, [p\_proc\_ready]设置了esp的值，而在下方不远处就是若干个pop以及一个popad指令。结合过去的分析我们不难推断，p\_proc\_ready应该是一个指向进程表的指针，存放的便是下一个要启动进程的进程表的地址。而且。其中的内容必然是以图6-7所示的顺序进行存放。

这样，才会使pop和popad指令执行后各寄存器的内容更新一遍。

打开代码，我们发现p\_proc\_ready结构体(struct s\_proc\*)的第一个成员的数据结构如下

typedef struct s\_stackframe { /\* proc\_ptr points here ↑ Low \*/

t\_32 gs; /\* ┓ │ \*/

t\_32 fs; /\* ┃ │ \*/

t\_32 es; /\* ┃ │ \*/

t\_32 ds; /\* ┃ │ \*/

t\_32 edi; /\* ┃ │ \*/

t\_32 esi; /\* ┣ pushed by save() │ \*/

t\_32 ebp; /\* ┃ │ \*/

t\_32 kernel\_esp; /\* <- 'popad' will ignore it │ \*/

t\_32 ebx; /\* ┃ ↑栈从高地址往低地址增长\*/

t\_32 edx; /\* ┃ │ \*/

t\_32 ecx; /\* ┃ │ \*/

t\_32 eax; /\* ┛ │ \*/

t\_32 retaddr; /\* return address for assembly code save() │ \*/

t\_32 eip; /\* ┓ │ \*/

t\_32 cs; /\* ┃ │ \*/

t\_32 eflags; /\* ┣ these are pushed by CPU during interrupt │ \*/

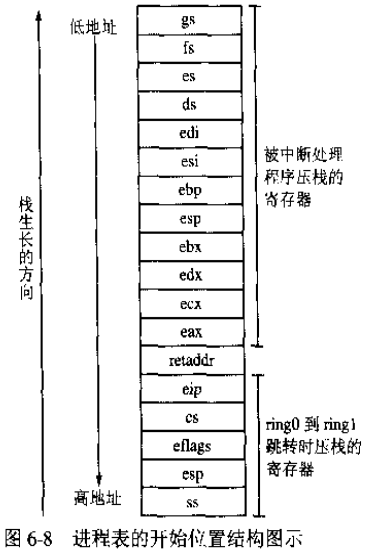
t\_32 esp; /\* ┃ │ \*/

t\_32 ss; /\* ┛ ┷High \*/

}STACK\_FRAME;

看来，它的内容安排与我们的推断完全一致。

现在我们知道了，原来进程的状态统统放在了s\_proc这个结构体中，而且位于前部的是所有相关寄存器的值，s\_proc这个结构就应该是我们提到过的“进程表”。当要恢复一个进程时，便将esp指向这个结构体的开始处，然后运行一系列的pop命令将寄存器至弹出。进程表的开始位置如图6-8所示。

第三行很明显是设置ldtr的，esp+P\_LDT\_SET恰好就是s\_proc中的成员ldt\_sel。同时可以猜测，在执行restart()之前，在某个地方一定是做了ldt\_sel的初始化工作，以便lldt可以正确执行。对于这一点，我们留到下文中进行验证。

第4、5行的作用是将s\_proc的第一个结构体成员将的末地址赋给TSS中的ring0堆栈指针域(esp)。我们可以想象，在下一次中断发生时，esp将变成regs的末地址，然后进程ss和esp两个寄存器值，以及eflags，还有cs、eip这几个寄存器值将依次被压栈，即放到s\_stackframe这个结构体最后的五个成员中。

至此，我们只剩下两行代码没有分析，一行是将k\_reenter的值减1，而另一行则是将esp加4。esp加4恰好跳过了retaddr这个成员，以便执行iretd这个指令，这两行后面会分析。

接下来，我们在有一定程度感性认识上，做以下工作：时钟中断处理程序、进程调度模块和进程体。

**6.3.2.1时钟中断处理程序**

先来做最简单的。完善的时钟中断处理未必会很简单，但前面说过，我们打算且只打算实现由ring0到ring1的转移，做到这一点用一个iretd指令就够了。此时并不需要关于进程调度的任何内容，所以时钟中断处理程序在这一步并不重要，我们完全可以做得最简单。

目前中断例程中什么也不做，直接返回，必要的时候会丰富它。

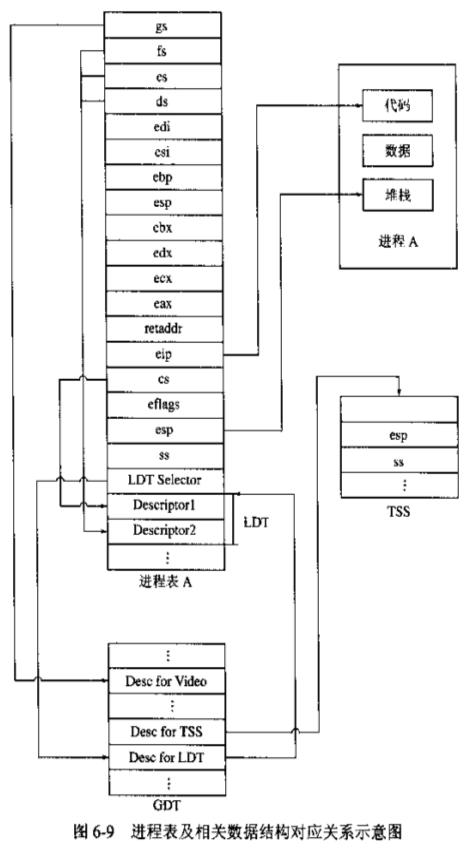
**6.3.2.2化整为零：进程表、进程体、GDT、TSS**

将寄存器初始化，cs、ds、es、fs、gs、ss、esp、eip、eflags。

注意，这里的cs、ds等段寄存器对应的将是LDT，而不再是GDT的描述符。所以我们的另一个任务是初始化局部描述符表。可以把它放置在进程表中，从逻辑上看，由于LDT是进程的一部分，所以这样安排也是合理的。同时，我们还必须在GDT中增加相应的描述符，并在合适的时间将相应的选择子加载给ldtr。

初始化任务状态段TSS，然后初始化里面的ring0的ss和esp。

在第一个进程正式开始之前，我们的准备工作已经做得差不多了，其核心内容便是一个进程表以及与之相关的TSS等内容。他们之间的对应关系如图6-9所示。



图中四个部分，进程表、进程体、GDT和TSS。它们之间的关系大致为：

● 进程表和GDT。进程表内的LDT Selector对应GDT中的一个描述符，而这个描述符所指向的内存空间就存在于进程表内。

● 进程表和进程。进程表是进程的描述，进程运行过程中如果被中断，各个寄存器的值都会被保存进进程表中。但是，在我们的第一个进程开始之前，并不需要初始化太多内容，只需要知道进程的入口地址就足够了。另外，由于程序免不了用到堆栈，而堆栈是不受程序本身控制的，所以还需要先指定esp。

● GDT和TSS。GDT中需要有一个描述符来对应TSS，需要事先初始化这个描述符。

**1.准备一个小的进程体**

虽然你身上穿着经过无数道工序做出的华丽衣装，但是若想遮蔽风寒的话，一张兽皮其实已经足够了。同样，我们知道以后会有无所不能的进程，但此刻，并不需要，我们只需要一个极小的进程执行体，它只有不到10行：

void TestA()

{

int i = 0;

while(1){

disp\_str("A");

disp\_int(i++);

disp\_str(".");

delay(1);

}

}

看到这个“进程”，你会说，这是个函数。是的，不但是个函数，而且是个及其简单的函数，但已经可以满足它作为一个进程执行体的功能。在它执行时会不停地循环，每循环一次就打印一个字符和一个数字，并且稍停片刻。

我们希望进程开始运行时能看到屏幕上打印出源源不断的A——那无疑将是激动人心的时刻。

注意，这段代码被放置在mian.c这个文件中。我们上文提到过，这是个新建立的文件，在第5章的最后，我们调用指令sti打开中断之后就用hlt指令让程序停止以等待中断的发生。显然，在本章中，我们将最终让进程运行起来，而不能仅仅停在那里，所以程序需要继续进行下去。我们将hlt注释掉，并让程序跳转到tinix\_main()这个函数中，见代码6-5，这个函数放在main.c中，目前除了显示一行字符之外并不完成其他工作。不过，由于在完成进程的编写之前，要让程序停在这里，所以我们用一个死循环作为它的结束。

代码6-4

PUBLIC int tinix\_main()

{

disp\_str("-----\"tinix\_main\" begins-----\n");

while(1){}

}

代码6-5

csinit: ; “这个跳转指令强制使用刚刚初始化的结构”——<<OS:D&I 2nd>> P90.

;jmp 0x40:0

;ud2

;sti

xor eax, eax

mov ax, SELECTOR\_TSS

ltr ax

jmp tinix\_main

;hlt

**2.初始化进程表**

代码结构体的定义见6-8，结构体STACK\_FRAME的定义见6-7。

代码6-7

typedef struct s\_stackframe { /\* proc\_ptr points here ↑ Low \*/

t\_32 gs; /\* ┓ │ \*/

t\_32 fs; /\* ┃ │ \*/

t\_32 es; /\* ┃ │ \*/

t\_32 ds; /\* ┃ │ \*/

t\_32 edi; /\* ┃ │ \*/

t\_32 esi; /\* ┣ pushed by save() │ \*/

t\_32 ebp; /\* ┃ │ \*/

t\_32 kernel\_esp; /\* <- 'popad' will ignore it │ \*/

t\_32 ebx; /\* ┃ ↑栈从高地址往低地址增长\*/

t\_32 edx; /\* ┃ │ \*/

t\_32 ecx; /\* ┃ │ \*/

t\_32 eax; /\* ┛ │ \*/

t\_32 retaddr; /\* return address for assembly code save() │ \*/

t\_32 eip; /\* ┓ │ \*/

t\_32 cs; /\* ┃ │ \*/

t\_32 eflags; /\* ┣ these are pushed by CPU during interrupt │ \*/

t\_32 esp; /\* ┃ │ \*/

t\_32 ss; /\* ┛ ┷High \*/

}STACK\_FRAME;

代码6-8

typedef struct s\_proc {

STACK\_FRAME regs; /\* process' registers saved in stack frame \*/

t\_16 ldt\_sel; /\* selector in gdt giving ldt base and limit\*/

DESCRIPTOR ldts[LDT\_SIZE]; /\* local descriptors for code and data \*/

/\* 2 is LDT\_SIZE - avoid include protect.h \*/

t\_32 pid; /\* process id passed in from MM \*/

char p\_name[16]; /\* name of the process \*/

}PROCESS;

现在，结构体的定义有了，我们在global.c中声明一个进程表：

PUBLIC PROCESS proc\_table[NR\_TASKS];

其中，NR\_TASKS定义了最大允许进程，我们把它设为1。虽然目前只试验一个进程的运行，但为了以后的扩展，我们还是声明了一个数组而不是单独一个变量，当NR\_TASK为1的时候数组和变量是一样的。

代码6-9，初始化进程表

PUBLIC int tinix\_main()

{

disp\_str("-----\"tinix\_main\" begins-----\n");

PROCESS\* p\_proc = proc\_table;

p\_proc->ldt\_sel = SELECTOR\_LDT\_FIRST;

memcpy(&p\_proc->ldts[0], &gdt[SELECTOR\_KERNEL\_CS >> 3], sizeof(DESCRIPTOR));

p\_proc->ldts[0].attr1 = DA\_C | PRIVILEGE\_TASK << 5; // change the DPL

memcpy(&p\_proc->ldts[1], &gdt[SELECTOR\_KERNEL\_DS >> 3], sizeof(DESCRIPTOR));

p\_proc->ldts[1].attr1 = DA\_DRW | PRIVILEGE\_TASK << 5; // change the DPL

p\_proc->regs.cs = ((8 \* 0) & SA\_RPL\_MASK & SA\_TI\_MASK) | SA\_TIL | RPL\_TASK;

p\_proc->regs.ds = ((8 \* 1) & SA\_RPL\_MASK & SA\_TI\_MASK) | SA\_TIL | RPL\_TASK;

p\_proc->regs.es = ((8 \* 1) & SA\_RPL\_MASK & SA\_TI\_MASK) | SA\_TIL | RPL\_TASK;

p\_proc->regs.fs = ((8 \* 1) & SA\_RPL\_MASK & SA\_TI\_MASK) | SA\_TIL | RPL\_TASK;

p\_proc->regs.ss = ((8 \* 1) & SA\_RPL\_MASK & SA\_TI\_MASK) | SA\_TIL | RPL\_TASK;

p\_proc->regs.gs = (SELECTOR\_KERNEL\_GS & SA\_RPL\_MASK) | RPL\_TASK;

p\_proc->regs.eip = (t\_32)TestA;

p\_proc->regs.esp = (t\_32) task\_stack + STACK\_SIZE\_TOTAL;

p\_proc->regs.eflags = 0x1202; // IF=1, IOPL=1, bit 2 is always 1.

p\_proc\_ready = proc\_table;

restart();

while(1){}

}

进程表初始化的主要有3个部分：寄存器、LDT Selector和LDT。

LDT Selector被赋值为 SELECTOR\_LDT\_FIRST，这个宏定义在代码6-10中。LDT里面共有两个描述符，为简化起见，分别被初始化成内核代码段和内核数据段，只是改变了一下DPL以让其运行在低的特权级下。

cs指向LDT中的第一个描述符，ds、es、fs、ss都设为指向LDT中，ds、es、fs、ss都设为指向LDT中的第二个描述符，gs仍然指向显存，只是其RPL发生改变。

接下来，eip指向TestA，这表明进程将从TestA的入口地址开始运行。另外，esp指向了单独的栈，栈的大小为STACK\_SIZE\_TOTAL。

最后一行是设置eflags，结合图3-45，0x1202恰好设置了IF位并把IOPL设为1。这样，进程就可以使用I/O指令，并且中断会在iretd执行时被打开。

代码中用到的宏大部分定义在protect.h中，见代码6-10

/\* GDT \*/

/\* 描述符索引 \*/

#define INDEX\_DUMMY 0 // ┓

#define INDEX\_FLAT\_C 1 // ┣ LOADER 里面已经确定了的.

#define INDEX\_FLAT\_RW 2 // ┃

#define INDEX\_VIDEO 3 // ┛

#define INDEX\_TSS 4

#define INDEX\_LDT\_FIRST 5

/\* 选择子 \*/

#define SELECTOR\_DUMMY 0 // ┓

#define SELECTOR\_FLAT\_C 0x08 // ┣ LOADER 里面已经确定了的.

#define SELECTOR\_FLAT\_RW 0x10 // ┃

#define SELECTOR\_VIDEO (0x18+3) // ┛<-- RPL=3

#define SELECTOR\_TSS 0x20 // TSS. 从外层跳到内存时 SS 和 ESP 的值从里面获得.

#define SELECTOR\_LDT\_FIRST 0x28

#define SELECTOR\_KERNEL\_CS SELECTOR\_FLAT\_C

#define SELECTOR\_KERNEL\_DS SELECTOR\_FLAT\_RW

#define SELECTOR\_KERNEL\_GS SELECTOR\_VIDEO

/\* 每个任务有一个单独的 LDT, 每个 LDT 中的描述符个数: \*/

#define LDT\_SIZE 2

/\* 描述符类型值说明 \*/

#define DA\_32 0x4000 /\* 32 位段 \*/

#define DA\_LIMIT\_4K 0x8000 /\* 段界限粒度为 4K 字节 \*/

#define DA\_DPL0 0x00 /\* DPL = 0 \*/

#define DA\_DPL1 0x20 /\* DPL = 1 \*/

#define DA\_DPL2 0x40 /\* DPL = 2 \*/

#define DA\_DPL3 0x60 /\* DPL = 3 \*/

/\* 存储段描述符类型值说明 \*/

#define DA\_DR 0x90 /\* 存在的只读数据段类型值 \*/

#define DA\_DRW 0x92 /\* 存在的可读写数据段属性值 \*/

#define DA\_DRWA 0x93 /\* 存在的已访问可读写数据段类型值 \*/

#define DA\_C 0x98 /\* 存在的只执行代码段属性值 \*/

#define DA\_CR 0x9A /\* 存在的可执行可读代码段属性值 \*/

#define DA\_CCO 0x9C /\* 存在的只执行一致代码段属性值 \*/

#define DA\_CCOR 0x9E /\* 存在的可执行可读一致代码段属性值 \*/

/\* 系统段描述符类型值说明 \*/

#define DA\_LDT 0x82 /\* 局部描述符表段类型值 \*/

#define DA\_TaskGate 0x85 /\* 任务门类型值 \*/

#define DA\_386TSS 0x89 /\* 可用 386 任务状态段类型值 \*/

#define DA\_386CGate 0x8C /\* 386 调用门类型值 \*/

#define DA\_386IGate 0x8E /\* 386 中断门类型值 \*/

#define DA\_386TGate 0x8F /\* 386 陷阱门类型值 \*/

/\* 选择子类型值说明 \*/

/\* 其中, SA\_ : Selector Attribute \*/

#define SA\_RPL\_MASK 0xFFFC

#define SA\_RPL0 0

#define SA\_RPL1 1

#define SA\_RPL2 2

#define SA\_RPL3 3

#define SA\_TI\_MASK 0xFFFB

#define SA\_TIG 0

#define SA\_TIL 4

代码6-10不但定义了SELECTOR\_LDT\_FIRST，而且定义了SELECTOR\_TSS，因为从图6-9中可知，我们还需要一个用来使用TSS的描述符。

这里，一定要记得LDT跟GDT是联系在一起的，别忘了填充GDT中进程的LDT的描述符：

init\_descriptor函数。

**3. GDT和TSS**

现在，再看一下图6-9，会发现剩下的没有初始化的只有TSS(定义请见代码6-16)和它对应的描述符了。让我们来到init\_port()，填充TSS以及对应的描述符：

/\* 填充 GDT 中 TSS 这个描述符 \*/

memset(&tss, 0, sizeof(tss));

tss.ss0 = SELECTOR\_KERNEL\_DS;

init\_descriptor(&gdt[INDEX\_TSS],

vir2phys(seg2phys(SELECTOR\_KERNEL\_DS), &tss),

sizeof(tss) - 1,

DA\_386TSS);

代码6-15，加载tr

xor eax, eax

mov ax, SELECTOR\_TSS

ltr ax

jmp tinix\_main

代码6-16，结构体TSS

typedef struct s\_tss {

t\_32 backlink;

...

t\_16 iobase; /\* I/O位图基址大于或等于TSS段界限，就表示没有I/O许可位图 \*/

/\*t\_8 iomap[2];\*/

}TSS;

**6.3.2.3 iretd**

代码6-1，我们对restart进行了分析，先参考着做一个简化版本的restart。

如果，我们正在制造一把枪，那么现在大部分的工作已经完成，只剩下扳机了。是的，扳机。让我们回到tinix\_main()，添加以下代码：

代码6-19，扳机

p\_proc\_ready = proc\_table;

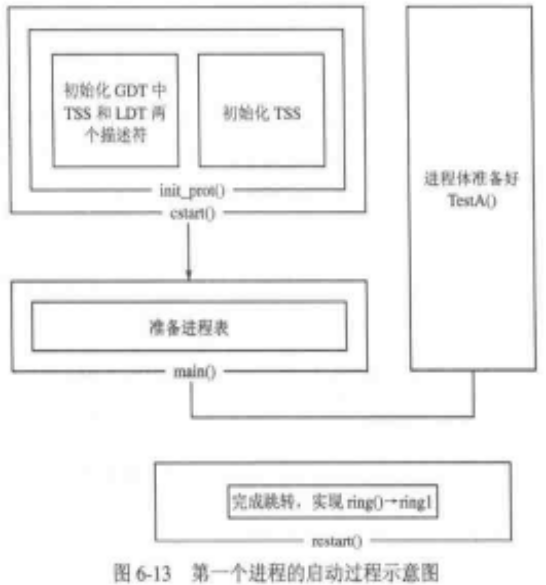
restart();

**6.3.2.4进程启动**

截图，略

**6.3.2.5第一个进程回顾**

在6.3.2中，进程表并没有担当起保存和恢复的作用，没有进程调度，中断也仅仅是开始的时候发生了一次。如图6-13所示，不过我们第一步完成了ring0到ring1的跳转，下面打开进程调度的大门。



**6.3.3 第二步——丰富中断处理程序 P263**

**6.3.3.1让时钟中断开始起作用**

打开时钟中断，添加改变屏幕第0行第0列的字符功能。

**6.3.3.2现场的保护与恢复**

代码6-23

ALIGN 16

hwint00: ; Interrupt routine for irq 0 (the clock).

sub esp, 4

pushad ; ┓

push ds ; ┃

push es ; ┣ 保存原寄存器值

push fs ; ┃

push gs ; ┛

mov dx, ss

mov ds, dx

mov es, dx

mov esp, StackTop ; 切到内核栈

inc byte [gs:0] ; 改变屏幕第 0 行, 第 0 列的字符

mov al, EOI ; ┓reenable master 8259

out INT\_M\_CTL, al ; ┛

push clock\_int\_msg

call disp\_str

add esp, 4

mov esp, [p\_proc\_ready] ; 离开内核栈;

lea eax, [esp + P\_STACKTOP]

mov dword [tss + TSS3\_S\_SP0], eax

pop gs ; ┓

pop fs ; ┃

pop es ; ┣ 恢复原寄存器值

pop ds ; ┃

popad ; ┛

add esp, 4

iretd

**6.3.3.3赋值tss.esp0**

ring1和ring0之间的切换，会有ss和esp的参与，在初始化时，对ring0的ss赋了值

tss.ss0 = SELECTOR\_KERNEL\_DS;

当下一次ring1->ring0时，在iretd返回之前要保证tss.esp0是正确的。也就是说在进程A将要运行时，我们要将tss.esp0设置正确。

**6.3.3.4内核栈**

当进行复杂的进程调度程序时，我们要将esp切换到内核栈

ALIGN 16

hwint00: ; Interrupt routine for irq 0 (the clock).

sub esp, 4

pushad ; ┓

push ds ; ┃

push es ; ┣ 保存原寄存器值

push fs ; ┃

push gs ; ┛

mov dx, ss

mov ds, dx

mov es, dx

mov esp, StackTop ; 切到内核栈

inc byte [gs:0] ; 改变屏幕第 0 行, 第 0 列的字符

mov al, EOI ; ┓reenable master 8259

out INT\_M\_CTL, al ; ┛

push clock\_int\_msg

call disp\_str

add esp, 4

mov esp, [p\_proc\_ready] ; 离开内核栈;

lea eax, [esp + P\_STACKTOP]

mov dword [tss + TSS3\_S\_SP0], eax

**6.3.3.5中断重入**

通过设置全局变量，处理中断嵌套问题。

**6.3.4进程体设计技巧 P274**

void TestA()

{

int i = 0;

while(1){

disp\_str("A");

disp\_int(i++);

disp\_str(".");

delay(1);

}

}

进程体这样设计，而不是简单的打印固定字符，为了确认进程确实成功恢复了。

**6.4多进程 P276**

到目前为止，我们的功能功能是，一个进程，然后被中断，再恢复，即两种状态，运行和睡眠。我们已经具备了多个进程的能力，只需要让其中一个进程处在运行态，其余进程处在睡眠态就可以了。

那么，现在我们就来试一试实现多个进程。

**6.4.1添加一个进程体**

代码6-32，进程B

void TestB()

{

int i = 0x1000;

while(1){

disp\_str("B");

disp\_int(i++);

disp\_str(".");

delay(1);

}

}

**6.4.2相关变量和宏**

增加B进程，需要将A进程的代码复制一份，但是这么做不妥，我们需要数据结构。

代码6-33，首先在proc.h中声明一个数组类型：

typedef struct s\_task {

t\_pf\_task initial\_eip;

int stacksize;

char name[32];

}TASK;

其中， t\_pf\_task是这样定义的，在type.h中：

typedef void (\*t\_pf\_task) ();

下面，我们在global.c中增加这样一个定义：

PUBLIC TASK task\_table[NR\_TASKS] = {{TestA, STACK\_SIZE\_TESTA, "TestA"},

{TestB, STACK\_SIZE\_TESTB, "TestB"},

NR\_TASKS改成2

#define NR\_TASKS 2

代码6-35，进程栈大小定义：

#define STACK\_SIZE\_TESTB 0x8000

#define STACK\_SIZE\_TOTAL (STACK\_SIZE\_TESTA + \

STACK\_SIZE\_TESTB

**6.4.3进程表初始化代码扩充**

TASK\* p\_task = task\_table;

PROCESS\* p\_proc = proc\_table;

char\* p\_task\_stack = task\_stack + STACK\_SIZE\_TOTAL;

t\_16 selector\_ldt = SELECTOR\_LDT\_FIRST;

int i;

for(i=0;i<NR\_TASKS;i++){

strcpy(p\_proc->p\_name, p\_task->name); // name of the process

p\_proc->pid = i; // pid

p\_proc->ldt\_sel = selector\_ldt;

memcpy(&p\_proc->ldts[0], &gdt[SELECTOR\_KERNEL\_CS >> 3], sizeof(DESCRIPTOR));

p\_proc->ldts[0].attr1 = DA\_C | PRIVILEGE\_TASK << 5; // change the DPL

memcpy(&p\_proc->ldts[1], &gdt[SELECTOR\_KERNEL\_DS >> 3], sizeof(DESCRIPTOR));

p\_proc->ldts[1].attr1 = DA\_DRW | PRIVILEGE\_TASK << 5; // change the DPL

p\_proc->regs.cs = ((8 \* 0) & SA\_RPL\_MASK & SA\_TI\_MASK) | SA\_TIL | RPL\_TASK;

p\_proc->regs.ds = ((8 \* 1) & SA\_RPL\_MASK & SA\_TI\_MASK) | SA\_TIL | RPL\_TASK;

p\_proc->regs.es = ((8 \* 1) & SA\_RPL\_MASK & SA\_TI\_MASK) | SA\_TIL | RPL\_TASK;

p\_proc->regs.fs = ((8 \* 1) & SA\_RPL\_MASK & SA\_TI\_MASK) | SA\_TIL | RPL\_TASK;

p\_proc->regs.ss = ((8 \* 1) & SA\_RPL\_MASK & SA\_TI\_MASK) | SA\_TIL | RPL\_TASK;

p\_proc->regs.gs = (SELECTOR\_KERNEL\_GS & SA\_RPL\_MASK) | RPL\_TASK;

p\_proc->regs.eip = (t\_32)p\_task->initial\_eip;

p\_proc->regs.esp = (t\_32)p\_task\_stack;

p\_proc->regs.eflags = 0x1202; // IF=1, IOPL=1, bit 2 is always 1.

p\_task\_stack -= p\_task->stacksize;

p\_proc++;

p\_task++;

selector\_ldt += 1 << 3;

}

k\_reenter = -1;

由于堆栈是从高地址往低地址生长的，所以在给每一个进程分配地址空间的时候也是从高地址往低地址进行。

在这里，我们为每一个进程都在GDT中分配一个描述符用来对应进程的LDT。

此外，p\_proc->p\_name，p\_proc->pid是锦上添花的事情，没有什么实际作用。

**6.4.4 LDT**

代码6-37，为每一个LDT进行赋值，chapter6/d/kernel/protect.c

int i;

PROCESS\* p\_proc = proc\_table;

t\_16 selector\_ldt = INDEX\_LDT\_FIRST << 3;

for(i=0;i<NR\_TASKS;i++){

init\_descriptor(&gdt[selector\_ldt>>3],

vir2phys(seg2phys(SELECTOR\_KERNEL\_DS), proc\_table[i].ldts),

LDT\_SIZE \* sizeof(DESCRIPTOR) - 1,

DA\_LDT);

p\_proc++;

selector\_ldt += 1 << 3;

}

进程切换时，需要重新加载ldt

mov esp, [p\_proc\_ready] ; 离开内核栈;

lldt [esp + P\_LDT\_SEL]

lea eax, [esp + P\_STACKTOP]

mov dword [tss + TSS3\_S\_SP0], eax

**6.4.5修改中断处理程序 P281**

如果现在运行程序的话，也只是对进程A进行休眠和恢复。我们需要修改时钟中断处理程序，该程序位于kernel.asm中。

回忆一下，一个进程如何由“睡眠”状态变成“运行”状态？无非是将esp指向进程表项的开始处，然后在执行lldt之后经历一系列pop指令恢复各个寄存器的值。一切信息都包含在进程表中，所系，要想恢复不同的进程，只需要将esp指向不同的进程表就可以了。

在离开内核栈的时候，有一个语句是为esp赋值的：

mov esp, [p\_proc\_ready] ; 离开内核栈;

全局变量 p\_proc\_ready是指向进程表结构的指针，我们只需要在这一句执行之前把它赋予不同的值就可以了。

我们参照Minix创建一个clock.c

PUBLIC void clock\_handler(int irq)

{

disp\_str("#");

p\_proc\_ready++;

if (p\_proc\_ready >= proc\_table + NR\_TASKS) {

p\_proc\_ready = proc\_table;

}

}

在时钟中断例程中，调用一下clock\_handler，每次让p\_proc\_ready指向进程表中的下一个表项，如果切换前已经到达进程表结尾则回到第一个表项。

mov esp, StackTop ; 切到内核栈

sti

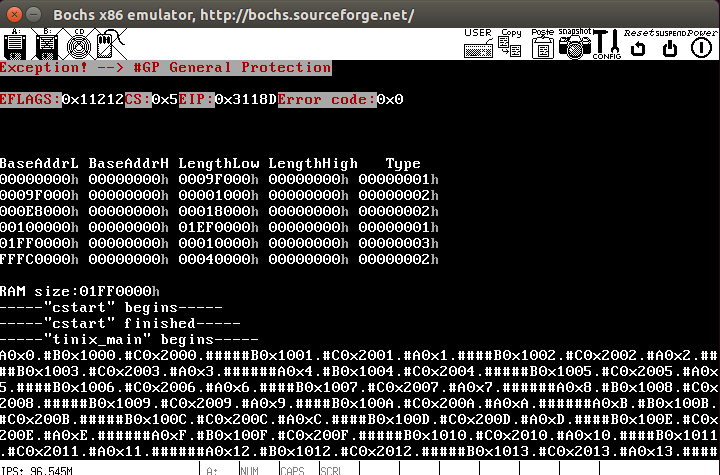
push 0

call clock\_handler

add esp, 4

cli

mov esp, [p\_proc\_ready] ; 离开内核栈;



**6.4.6 添加一个任务的步骤总结 P283**

添加一个进程体：

void TestC()

{

int i = 0x2000;

while(1){

disp\_str("C");

disp\_int(i++);

disp\_str(".");

delay(1);

}

}

在task\_table中添加一个进程:

PUBLIC TASK task\_table[NR\_TASKS] = {{TestA, STACK\_SIZE\_TESTA, "TestA"},

{TestB, STACK\_SIZE\_TESTB, "TestB"},

{TestC, STACK\_SIZE\_TESTC, "TestC"}};

然后在proc.h中修改NR\_TASKS的值：

#define NR\_TASKS 3

分配任务堆栈：

#define STACK\_SIZE\_TESTA 0x8000

#define STACK\_SIZE\_TESTB 0x8000

#define STACK\_SIZE\_TESTC 0x8000

#define STACK\_SIZE\_TOTAL (STACK\_SIZE\_TESTA + \

STACK\_SIZE\_TESTB + \

STACK\_SIZE\_TESTC)

在proto.h中添加函数声明：

void TestC();

make，运行，最后再总结一下

(1) 在task\_table中增加一项(global.c)。

(2) 让NR\_TASK加1(proc.h)。

(3) 定义任务堆栈(proc.h)。

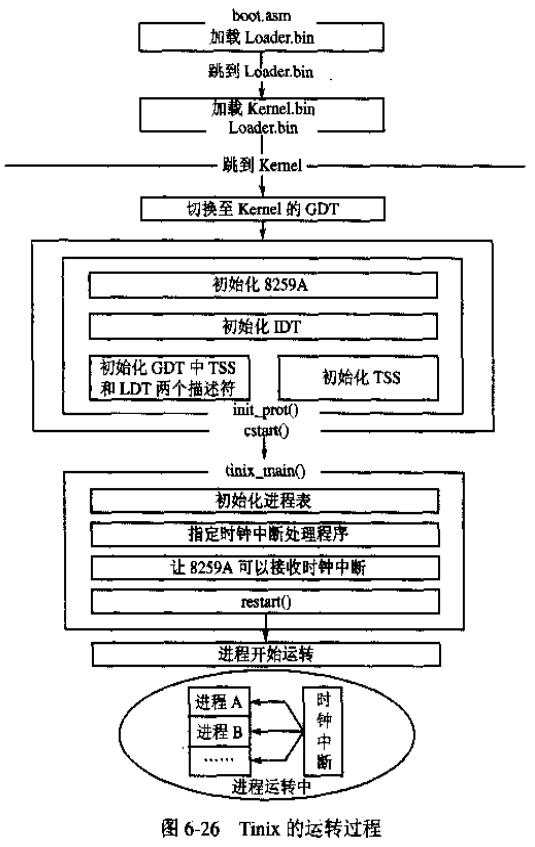
(4) 修改STACK\_SIZE\_TOTAL(proc.h)。

(5) 添加新任务执行体的函数声明(proto.h)。

**6.4.7号外：Minix的中断处理**

略

**6.4.8 代码回顾与整理**



**6.5系统调用 P300**

通过系统调用，让操作系统帮助应用程序完成一些事情，当然涉及到特权级变换。

**6.5.1实现一个简单的系统调用**

我们实现一个函数int get\_ticks()的函数作为例子来说明。我们用它来得到总共发生多少次时钟中断。

现在，假设进程P想得到当前的ticks，就问操作系统：“OS先生，请问当前ticks是多少？”然后OS先生低头在自己的记录本中查找了一下，然后告诉P：“当前的ticks是1234。”

根据这段形象的对话，读者肯定已经知道系统调用的过程是怎样的了。首先是“问”，就是告诉操作系统自己要什么；然后是操作系统“找”，即处理；最后是“回答”，也就是把结果返回给进程。

下面就按照这个顺序，实现我们的第一个系统调用：get\_ticks。

使用堆栈不是明智的，不如寄存器来的干脆利落。

代码6-62，chapter6/g/kernel/syscall.asm

\_NR\_get\_ticks equ 0 ; 要跟 global.c 中 sys\_call\_table 的定义相对应！

INT\_VECTOR\_SYS\_CALL equ 0x90

; 导出符号

global get\_ticks

bits 32

[section .text]

; ====================================================================================

; get\_ticks

; ====================================================================================

get\_ticks:

mov eax, \_NR\_get\_ticks

int INT\_VECTOR\_SYS\_CALL

ret

首先将eax的值赋为\_NR\_get\_ticks，这样，在中断处理程序中，OS先生看到eax的值是\_NR\_get\_ticks，就知道问题是“请问当前的ticks是多少”。

这里将系统调用对应的中断号设为0x90,它只要不和原来的中断号重复即可，Linux相应的中断号是0x80。

马上来定义INT\_VECTOR\_SYS\_CALL对应的中断门，在protect.c的init\_prot()中，紧跟初始化其他中断门的语句：

init\_idt\_desc(INT\_VECTOR\_SYS\_CALL, DA\_386IGate, sys\_call, PRIVILEGE\_USER);

INT\_VECTOR\_SYS\_CALL号中断对应的中断例程是sys\_call。我们先实现保存现场，模仿hwint\_master宏来做，但是需要用esi来保存，应为eax存放P先生OS先生的问题。

现在可以写sys\_call了，请看代码6-64，call [sys\_call\_table+eax\*4]，即调用sys\_call\_table[eax]

与irq\_table类似，sys\_call\_table是一个函数指针数组，每一个成员都指向一个函数，用以处理相应的系统调用。

代码6-64，chapter6/g/kernel/kernel.asm

sys\_call:

call save

sti

call [sys\_call\_table + eax \* 4]

mov [esi + EAXREG - P\_STACKBASE], eax ;返回值放在进程表中eax的位置

cli

ret

sys\_call\_table的定义在global.c中，实际上目前它只有一个成员：

PUBLIC t\_sys\_call sys\_call\_table[NR\_SYS\_CALL] = {sys\_get\_ticks};

chapter6/g/kernel/clock.c

PUBLIC void clock\_handler(int irq)

{

disp\_str("#");

ticks++;

if (k\_reenter != 0) {

disp\_str("!");

return;

}

p\_proc\_ready++;

if (p\_proc\_ready >= proc\_table + NR\_TASKS) {

p\_proc\_ready = proc\_table;

}

}

添加函数声明后，在TestA中可以调用了，get\_ticks()。

**6.5.2 get\_ticks的应用 P306**

改进get\_ticks，让其达到延时10ms功能。

**6.6进程调度**

**6.6.1避免对称——进程的节奏感**

我们可以通过修改3个进程各自delay的时间来达到“时间片”的效果，但是这不能说明进程就有了优先级，我们需要把这种ticks移植到进程调度的程序中，对进程表的成员加入一个变量，来表示他的运行周期。

代码6-75，为进程表添加新的成员ticks, priority

typedef struct s\_proc {

STACK\_FRAME regs; /\* process' registers saved in stack frame \*/

t\_16 ldt\_sel; /\* selector in gdt giving ldt base and limit\*/

DESCRIPTOR ldts[LDT\_SIZE]; /\* local descriptors for code and data \*/

/\* 2 is LDT\_SIZE - avoid include protect.h \*/

int ticks; /\* remained ticks \*/

int priority;

t\_32 pid; /\* process id passed in from MM \*/

char name[16]; /\* name of the process \*/

}PROCESS;

ticks是递减的，从某个初值到0，priority是记录初值的，当ticks=0，重新赋值。

代码6-76，ticks和proiority初始化，chapter6/i/kernel/main.c

proc\_table[0].ticks = proc\_table[0].priority = 15;

proc\_table[1].ticks = proc\_table[1].priority = 5;

proc\_table[2].ticks = proc\_table[2].priority = 3;

对于进程调度，每一个进程获得一个运行周期，ticks-1，当减到0时，此进程就不再获得执行的机会，直到所有进程的变量都减到0为止。我们独单写一个函数，叫做schedule，放在proc.c中，代码略。

然后始终中断调用schedule。

**6.6.2优先级调度总结**

略

**第七章 输入输出系统 P322**

我们刚刚实现了简单的进程，你现在可能很想把它做得更加完善，比如进一步改进调度算法、增加通信机制等。但是这些工作不但做起来没有尽头，而且有些也是难以实现的，因为进程必须与I/O、内存管理等其他模块一起工作。而且，简单的进程更有利于我们思考和控制。

那么现在，我们就来实现简单的I/O。

**7.1键盘**

到目前为止，Tinix一旦启动就不再受我们的控制了，只能静静地等待结果的出现。但正如眼下人们看新闻的方式正在从电视转向因特网一样，我们的操作系统显然也更需要交互。而交互的手段，首先当然是键盘。

**7.1.1从中断开始——键盘初体验**

新建一个文件keyboard.c，添加如下函数：

PUBLIC void keyboard\_handler(int irq)

{

disp\_str("\*");

}

结果是每按一次键，打印一个星号，有点像在输入密码。

为了不受其他进程输出的影响，我们把其他进程的输出都注释掉。

最后添加指定中断处理程序和打开键盘中断的代码：

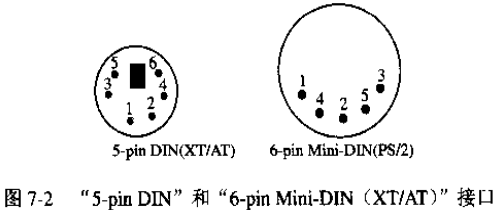
put\_irq\_handler(KEYBOARD\_IRQ, keyboard\_handler); /\* 设定键盘中断处理程序 \*/

enable\_irq(KEYBOARD\_IRQ); /\* 开键盘中断 \*/

实际上，出现一个星号之后键盘就不再响应了？这倒是比较奇怪的，看来事情比我们想象的要复杂一些。我们还是要从头说起。

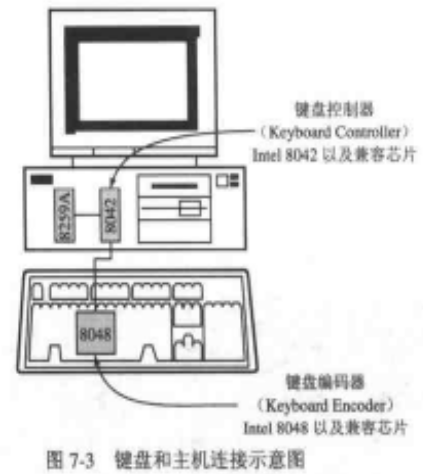
**7.1.2 AT、PS/2键盘**

图7-2中，左图是PS/2键盘，相对右图AT键盘做了一点点的扩展，在最初接触它们时，你可以认为他们是一样的。

****

**7.1.3键盘敲击的过程**

在键盘中存在一枚叫做键盘编码器(Keyboard Encoder)的芯片，用于传递数据给计算机，另外，在计算机主板上还有一个键盘控制器(Keyboard Controller)，用来接收和解码来自键盘的数据，并与8259A以及软件等进行通信(如图7-3所示)。

敲击键盘所产生的编码被称作扫描码(Scan Code)，它分为Make Code和Break Code两类。当一个键被按下或者保持住按下时，将会产生Make Code，当键弹起时，产生Break Code。除了Pause键之外，每一个按键都对应一个Make Code和一个Break Code。

当8048检测到一个键的动作后，会把相应的扫描码发送给8042，8042会把它转换成相应的Scan code set 1扫描码，并将其放置在输入缓冲区中，然后8042将不再接收，一直到缓冲区被晴空，8042才会收到更多的扫描码。

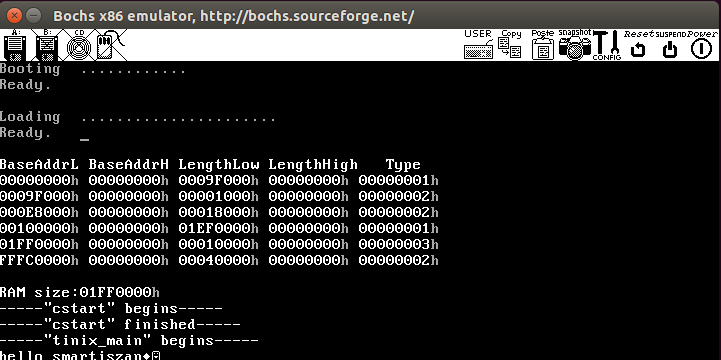
所以，我们只有把扫描码从缓冲区读出来后，8042才能继续响应新的按键。

我们在keyboard\_handler中添加in\_byte(KB\_DATA); 结果就会出现，按一下键，出现两个星号，没错，每次按键都产生一个Make Code和一个Bread Code。

进一步修改，加入disp\_int(scan\_code); 再次运行，按下a和9，出现4组代码：0x1E、0x9E、0xA、0x8A，对照表7-2发现，它们恰恰就是字符a和9的Make Code和Break Code。

**7.1.4解析扫描码**

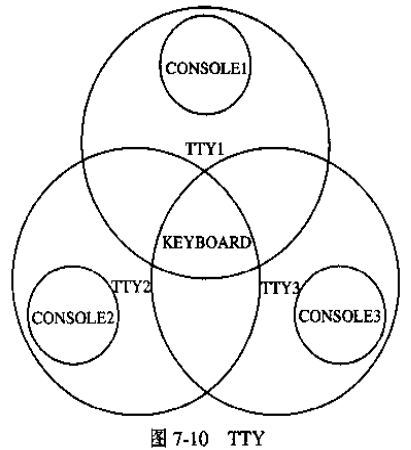
扫描码获得后，我们要对其进行解析，然后获得正确的按键字符。



**7.2显示器 P345**

**7.2.1初识TTY**

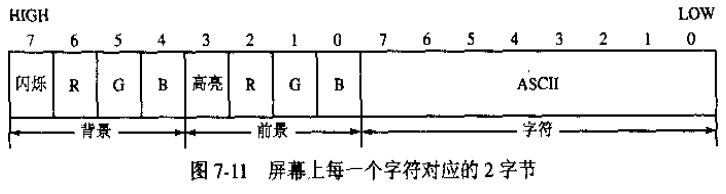
在Linux的某个终端，输入tty，会显示当前的终端号。对于不同的TTY，我们可以理解成图7-10的样子。

虽然不同的TTY对应的输入设备是同一个键盘，但输出却好比是在不同的显示器上，因为不同的TTY对应的屏幕画面可能是迥然不同的。实际上，我们当然是在使用同一个显示器，画面的不同只不过是因为显示了显存的不同位置罢了。

既然3个CONSOLE共用一块显存，就必须有一种方式，在切换CONSOLE的瞬间，让屏幕显示现存中某个位置的内容。

**7.2.2基本概念**

我们使用的是80x25文本模式显存范围为0xB8000~0xBFFFF，共计32KB。每2字节代表一个字符，其中低字节表示字符的ASCII码，高字节表示字符的属性。一个屏幕总共显示25行，每行80个字符。

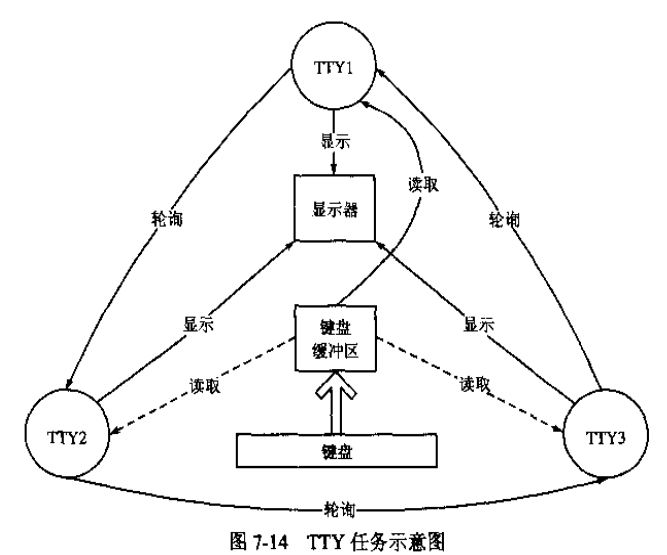
一个屏幕映射到显存中所占空间大小：80x25x2=4000字节

现存有32KB，如果我们有3个TTY，分别让他们使用10KB的空间还有剩余，而且在每一个TTY内，还可以实现简单的滚屏功能。

**7.2.3寄存器**

通过设置VGA寄存器，实现按下Shift+↑后，屏幕向上卷动15行。

**7.3 TTY任务**



在TTY任务中执行一个循环，这个循环将轮询每一个TTY，处理它的时间，包括从键盘缓冲区读取数据、显示字符等内容。

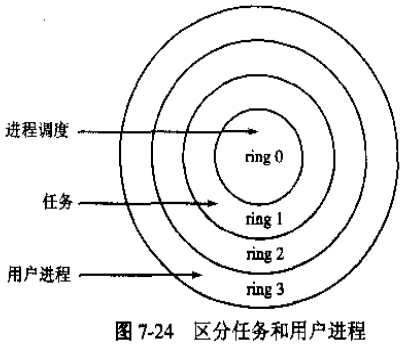
运行d目录的程序，shift+↑和shift+↓上下滚动屏幕，Alt+F3，切换到控制台2。

运行e目录的程序，Caps Lock和小键盘起作用了。

**7.4区分任务和用户进程**

现在，我们有了4个进程，分别是TTY、A、B、C。其中A、B和C是可有可无的，它其实不是操作系统的一部分，而更像用户在执行的程序。而TTY则不同，它肩负着重大的职责，没有它我们连键盘都无法使用。

所以，我们有必要把它们区分开来，分为两类。我们称TTY为“任务”，而称A、B、C为“用户进程”。程序在e目录



**7.5 printf**

当某个进程调用printf()时，操作系统必须知道往哪个控制台输出才行，所以，我们必须为每个进程指定一个与之相对应的TTY，这可以通过在进程表中增加一个成员来实现：

本例，A与第0个TTY绑定，B和C与第1个TTY绑定。