第 12 章　存储器的保护

处理器引入保护模式的目的是提供保护功能，其中很重要的一个方面就是存储器保护。存储器的保护功能可以禁止程序的非法内存访问，比如，向代码段写入数据、访问段界限之外的内存位置等。很多时候，这类问题都是由于编程疏漏引起的，属于有缺陷的软件，但也不排除软件的功能本身就是恶意的。不过，一旦能够及时发现和禁止这些非法操作，在程序失去控制之前引发异常中断，就可以提高软件的可靠性，降低整个计算机系统的安全风险。

凡事都有两面。利用存储器的保护功能，也可以实现一些有价值的功能，比如虚拟内存管理。当处理器访问一个实际上不存在的段时，会引发异常中断。操作系统可以利用这一点，通过接管异常处理过程，并用硬盘来进行段的换入和换出，从而实现在较小的内存空间运行尽可能大、尽可能多的程序。本章的学习目标是：

1. 通过实例来认识处理器是如何进行存储器的保护的。
2. 了解别名段的意义和作用。
3. 以一个字符串排序过程作为例子，演示保护模式下的内存数据访问，体验一下它们与在实模式下访问数据段有什么不同。同时，在这个过程中学习用汇编语言实现冒泡排序算法，以及一条新的 x86 处理器指令 xchg。

12.1　代码清单 12-1

本章有配套的汇编语言源程序，并围绕这些源程序进行讲解，请对照阅读。

本章代码清单：12-1（主引导扇区程序）

源程序文件：c12_mbr.asm

12.2　进入 32 位保护模式

12.2.1　话说 mov ds,ax 和 mov ds,eax

本章的代码和上一章有几分类似，但实质上有很大区别。

我们知道，段寄存器（选择器）的值只能用内存单元或者通用寄存器来传送，一般的指令格式为

```
mov sreg,r/m16
```

这里有一个常见的例子：

```
    mov ds,ax
```

在 16 位模式下，传送到 DS 中的值是逻辑段地址；在 32 位保护模式下，传送的是段描述符的选择子。无论传送的是什么，这都不重要，重要的是，在 16 位模式和 32 位模式下，一些老式的编译器会生成不同的机器代码。下面是一个例证：

```
    [bits 16]
    mov ds,ax          ;8E D8

    [bits 32]
    mov ds,ax          ;66 8E D8
```

由于在 16 位模式下，默认的操作数大小是字（2 字节），故生成 8E D8 也不难理解。在 32 位模式下，默认的操作数大小是双字（4 字节）。由于指令中的源操作数是 16 位的 AX，故编译后的机器码前面应当添加前缀 0x66 以反转默认的操作数大小，即 66 8E D8。

很遗憾，由于这一点点区别，有前缀的和没有前缀的相比，处理器在执行时会多花一个额外的时钟周期。问题在于，这样的指令用得很频繁，而且牵扯到内存段的访问，自然也很重要。因此，它们在 16 位模式和 32 位模式下的机器指令被设计为相同。即都是 8E D8，不需要指令前缀。

这可难倒了很多编译器，它们固执地认为，在 32 位模式下，源操作数是 16 位的寄存器 AX 时，应当添加指令前缀。好吧，为了照顾它们，很多程序员习惯使用这种看起来有点别扭的形式：

```
    mov ds,eax
```

你别说，还真有效，果然生成的是不加前缀的 8E D8。

说到这里，我觉得 NASM 编译器还是非常优秀的，起码它不会有这样的问题。因此，不管处理器模式如何变化，也不管指令形式如何变化，以下代码编译后的结果都一模一样：

```
    [bits 16]
    mov ds,ax          ;8E D8
    mov ds,eax         ;8E D8

    [bits 32]
    mov ds,ax          ;8E D8
    mov ds,eax         ;8E D8
```

和这个示例一样，其他从通用寄存器到段寄存器的传送也符合这样的编译规则。因此，代码清单 12-1 第 7、8 行，用于通过寄存器 EAX 来初始化栈段寄存器 SS。

12.2.2 创建 GDT 并安装段描述符

准备进入保护模式。

首先是创建 GDT，并安装刚进入保护模式时就要使用的描述符。第 12～15 行，首先计算 GDT 在实模式下的逻辑地址。在上一章里，GDT 的大小和线性基地址分别是用两个标号 gdt_size 和 gdt_base 声明和初始化的：

```
    gdt_size dw 0
    gdt_base dd 0x0000007e00
```

但是，如后面的第 107、108 行所示，现在已经改成

```
        pdgt dw 0
             dd 0x00007e00
```

另外一个区别是计算 GDT 逻辑地址的方法。在 32 位处理器上，即使是在实模式下，也可以使用 32 位寄存器。所以，第 12 行，直接将 GDT 的 32 位线性基地址传送到寄存器 EAX 中。

我们知道，32 位处理器可以执行以下除法操作：

```
        div r/m32
```

其中，64 位的被除数在 EDX:EAX 中，32 位被除数可以在 32 位通用寄存器中，也可以在 32 位内存单元中。因此，第 13～15 行，用 64 位的被除数 EDX:EAX 除以 32 位的除数 EBX。指令执行后，EAX 中的商是段地址，仅低 16 位有效；EDX 中的余数是段内偏移地址，仅低 16 位有效。

第 17、18 行，初始化段寄存器 DS，使其指向 GDT 所在的逻辑段。

第 21、22 行，安装空描述符。该描述符的槽位号是 0，处理器不允许访问这个描述符，任何时候，使用索引字段为 0 的选择子来访问该描述符，都会被处理器阻止，并引发异常中断。在现实中，一个忘了初始化的指针往往默认值就是 0，所以空描述符的用意就是阻止不安全的访问。很多人喜欢用这个槽位来记载一些私人信息，做一些特殊的用途，认为反正处理器也不用它。但是，这样做可能是不安全的，还没有证据表明 Intel 公司保证决不会使用这个槽位。

第 25、26 行，安装保护模式下的数据段描述符。参考前面的段描述符格式，可以看出，该段的线性基地址位于整个内存的最低端，为 0x00000000；属于 32 位的段，段界限是 0xFFFFF。但是要注意，段的粒度是以 4KB 为单位的。对于以 4KB（十进制数 4096 或者十六进制数 0x1000）为粒度的段，描述符中的界限值加 1，就是该段有多少个 4KB。因此，其实际使用的段界限为

```
（描述符中的段界限值＋1）×0x1000－1
```

将其展开后，即

```
描述符中的段界限值×0x1000＋0x1000－1
```

因此，在换算成实际使用的段界限时，其公式为

```
描述符中的段界限值×0x1000＋0xFFF
```

这就是说，实际使用的段界限是

```
0xFFFFF×0x1000＋0xFFF＝0xFFFFFFFF
```

也就是 4GB。就 32 位处理器来说，这个地址范围已经最大了。一旦使用这个段，就可以访问 0 到 4GB 空间内的任意一个单元，这是本书开篇以来，从来没有过的事情。

第 29、30 行，安装保护模式下的代码段描述符。该段是 32 位的代码，线性基地址为 0x00007C00；段界限为 0x001FF，粒度为字节。对于向上扩展的段来说，段界限在数值上等于段的长度减去 1，因此该段的长度是 0x200，即 512 字节。

根据上一章的经验，该段实际上就是当前程序所在的段（正在安装该描述符呢），也就是主引导程序所在的区域。尽管在描述符中把它定义成 32 位的段，但它实际上既包含 16 位代码，也包含 32 位代码。[bits 32]之前的代码是 16 位的，之后的代码是 32 位的。不过，在该描述符生效的时候，处理器的执行流已经位于 32 位代码中了。

第 33、34 行，安装保护模式下的数据段描述符。该段是 32 位的数据段，线性基地址为 0x00007C00；段界限为 0x001FF，粒度为字节。可以看出，该描述符和前面的代码段描述符，描述和指向的是同一个段。你可能很想知道，这样做的用意何在？

参见上一章的表 11-1，我们都已经知道，在保护模式下，代码段是不可写入的。所谓不可写入，并非是说改变了内存的物理性质，使得内存写不进去，而是说，通过该段的描述符来访问这个区域时，处理器不允许向里面写入数据或者更改数据。

但是，很多时候，又需要对代码段做一些修改。比如在调试程序时，需要加入断点指令 int3。不管怎么样，如果需要访问代码段内的数据，只能重新为该段安装一个新的描述符，并将其定义为可读可写的数据段。这样，当需要修改代码段内的数据时，可以通过这个新的描述符来进行。

像这样，当两个以上的描述符都描述和指向同一个段时，把另外的描述符称为别名（alias）。注意，别名技术并非仅仅用于读写代码段，如果两个程序想共享同一个内存区域，可以分别为每个程序都创建一个描述符，而且它们都指向同一个内存段，这也是别名应用的例子。

第 36、37 行，安装保护模式下的栈段描述符。该段的线性基地址是 0x00007C00，段界限为 0xFFFFE，粒度为 4KB。

尽管该段和代码段使用同一个线性基地址，但这不会有什么问题，代码段是向上（高地址方向）扩展的，而栈段是向下（低地址方向）扩展的。至于段界限为 0xFFFFE，粒度为 4KB，我知道你可能会有某些疑问，这些事情马上就会讲到。

第 40 行，设置 GDT 的界限值为 39，因为这里共有 5 个描述符，总大小为 40 字节，界限值为 39。后面的代码用于进入保护模式，差不多和上一章相同，不再赘述。GDT 和 GDT 内的描述符，以及本章程序，它们在内存中的映象如图 12-1 所示。

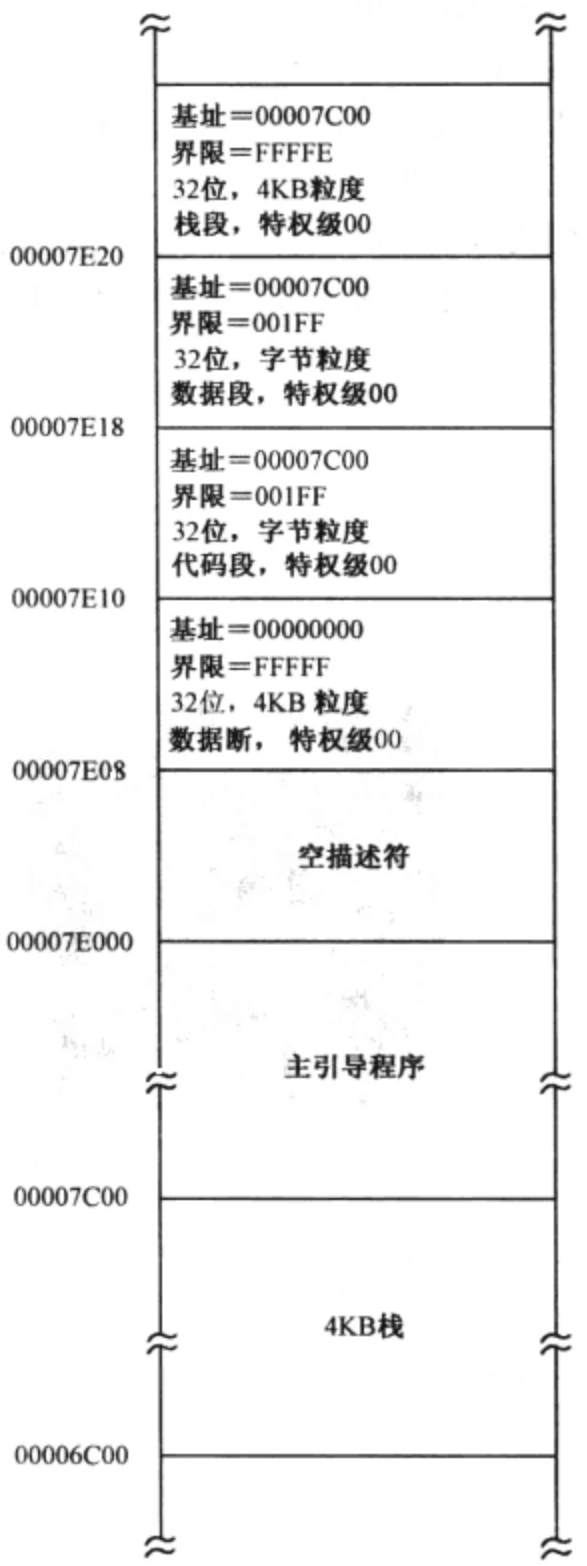

图 12-1 本章程序中各个部分在内存中的映象

12.3 修改段寄存器时的保护

随着程序的执行，经常要对段寄存器进行修改。此时，处理器在变更段寄存器以及隐藏的描述符高速缓存器的内容时，要检查其代入值的合法性。

代码清单 12-1 第 55 行，这是一条直接远转移指令：

```
jmp dword 0x0010:flush
```

这条指令会隐式地修改段寄存器 CS。

同样要修改段寄存器的指令还出现在第 59～68 行（以下粗体部分）：

```
mov eax,0x0018
mov ds,eax

mov eax,0x0008      ;加载数据段（0:4GB）选择子
mov es,eax
mov fs,eax
mov gs,eax
```

```
    mov eax,0x0020      ;0000 0000 0010 0000
    mov ss,eax
```

以上的指令涉及所有段寄存器，当这些指令执行时，处理器把指令中给出的选择子传送到段寄存器的选择器部分。但是，处理器的固件在完成传送之前，要确认选择子是正确的，并且该选择子选择的描述符也是正确的。

在当前程序中，选择子的 TI 位都是 0，故所有的描述符都在 GDT 中。如图 12-2 所示，GDT 的基地址和界限，都在寄存器 GDTR 中。描述符在内存中的地址，是用索引号乘以 8，再和描述符表的线性基地址相加得到的，而这个地址必须在描述符表的地址范围内。换句话说，索引号乘以 8 得到的数值，必须位于描述符表的边界范围之内。换句话说，处理器从 GDT 中取某个描述符时，就要求描述符的 8 个字节都在 GDT 边界之内，也就是索引号× 8 +7 小于等于边界。

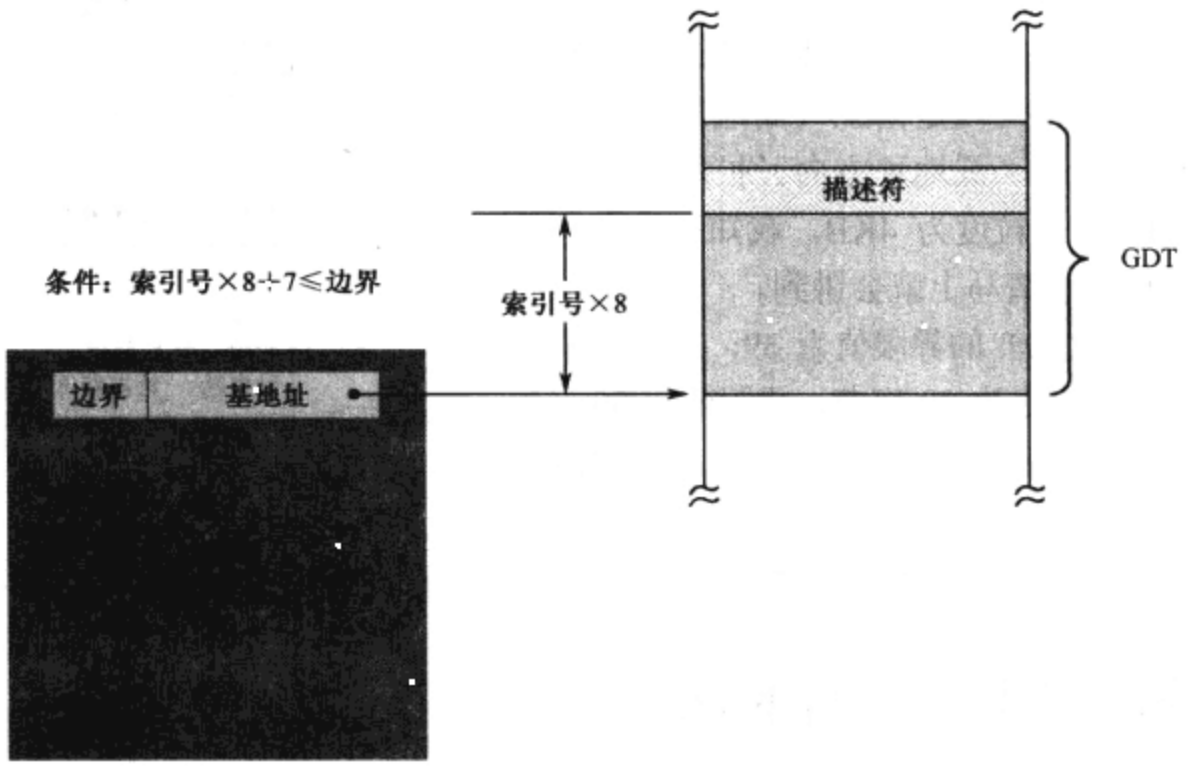

图 12-2　索引号的检查

如果检查到指定的段描述符，其位置超过表的边界时，处理器中止处理，产生异常中断 13，同时段寄存器中的原值不变。

以上仅仅是检查的第一步。要是通过了上述检查，并从表中取得描述符后，紧接着还要对描述符的类别进行确认。举个例子来说，若描述符的类别是只执行的代码段（表 11-1），则不允许加载到除 CS 之外的其他段寄存器中。

具体地说，首先，描述符的类别字段必须是有效的值，0000 是无效值的一个例子。

然后，检查描述符的类别是否和段寄存器的用途匹配。其规则如表 12-1 所示。

表 12-1　段的类别检查

段寄存器	数据段（X=0）		代码段（X=1）	
	只读（W=0）	读写（W=1）	只执行（R=0）	执行、读（R=1）
CS	N	N	Y	Y
DS	Y	Y	N	Y
ES	Y	Y	N	Y
FS	Y	Y	N	Y
GS	Y	Y	N	Y
SS	N	Y	N	N

最后，除了按表 12-1 进行段的类别检查外，还要检查描述符中的 P 位。如果 P=0，表明虽然描述符已被定义，但该段实际上并不存在于物理内存中。此时，处理器中止处理，引发异常中断 11。一般来说，应当定义一个中断处理程序，把该描述符所对应的段从硬盘等外部存储器调入内存，然后置 P 位。中断返回时，处理器将再次尝试刚才的操作。

如果 P=1，则处理器将描述符加载到段寄存器的描述符高速缓存器，同时置 A 位（仅限于当前讨论的存储器的段描述符）。

注意，如表中所指示的那样，可读的代码段类似于 ROM。可以用段超越前缀“cs:”来读其中的内容，也可以将它的描述符选择子加载到 DS、ES、FS、GS 来做为数据段访问。代码段在任何时候都是不可写的。

一旦上述规则全部验证通过，处理器就将选择子加载到段寄存器的选择器。显然，只有可以写入的数据段才能加载到 SS 的选择器，CS 寄存器只允许加载代码段描述符。另外，对于 DS、ES、FS 和 GS 的选择器，可以向其加载数值为 0 的选择子，即

```
xor eax,eax          ;eax = 0
mov ds,eax           ;ds <- 0
```

尽管在加载的时候不会有任何问题，但在，真正要用来访问内存时，就会导致一个异常中断。这是一个特殊的设计，处理器用它来保证系统安全，这在后面会讲到。不过，对于 CS 和 SS 的选择器来说，不允许向其传送为 0 的选择子。

继续回到代码清单 12-1 中来，第 55～68 行的指令执行之后，段寄存器 CS 指向 512 字节的 32 位代码段，基地址是 0x00007C00；DS 指向 512 字节的 32 位数据段，该段是上述代码段的别名，因此基地址也是 0x00007C00；ES、FS 和 GS 指向同一个段，该段是一个 4GB 的 32 位数据段，基地址为 0x00000000；SS 指向 4KB 的 32 位栈段，基地址为 0x00007C00。

◆ 检测点 12.1

1. 若某段描述符中的段界限是 0xFFFFC，当粒度为字节和 4KB 时，实际使用的段界限是多少？
2. 若 GDT 的界限为 0x87，寄存器 AX 的内容为 0x0088，则执行指令 mov ds,ax 时，处理器会产生异常吗？

12.4 地址变换时的保护

12.4.1 代码段执行时的保护

在 32 位模式下，尽管段的信息在描述符表中，但是，一旦相应的描述符被加载到段寄存器的描述符高速缓存器，则处理器取指令和执行指令时，将不再访问描述符表，而是直接使用段寄存器的描述符高速缓存器，从中取得线性基地址，同指令指针寄存器 EIP 的内容相加，共同形成 32 位的物理地址从内存中取得下一条指令。不过，在指令实际开始执行之前，处理器必须检验其存放地址的有效性，以防止执行超出允许范围之外的指令。

每个代码段都有自己的段界限，位于其描述符中。实际使用的段界限，其数值和粒度（G）位有关，如果 G=0，实际使用的段界限就是描述符中记载的段界限；如果 G=1，则实际使用的段界限为

```
描述符中的段界限值×0x1000+0xFFF
```

该计算公式已经在前面出现过，不再解释。

代码段是向上（高地址方向）扩展的，因此，实际使用的段界限就是当前段内最后一个允许

访问的偏移地址。当处理器在该段内取指令执行时，偏移地址由 EIP 提供。指令很有可能是跨越边界的，一部分在边界之内，一部分在边界之外，或者一条单字节指令正好位于边界上。因此，要执行的那条指令，其长度减 1 后，与 EIP 寄存器的值相加，结果必须小于等于实际使用的段界限，否则引发处理器异常。即：

```
0≤（EIP+指令长度－1）≤实际使用的段界限
```

在本章中，代码段描述符中给出的界限值是 0x001FF，粒度是字节，可以认为它就是段内最后一个允许访问的偏移地址。如图 12-3 所示，在处理器取得一条指令后，EIP 寄存器的数值加上该指令的长度减 1，得到的结果必须小于等于 0x000001FF，如果等于或者超出这个数值，必然引发异常中断。

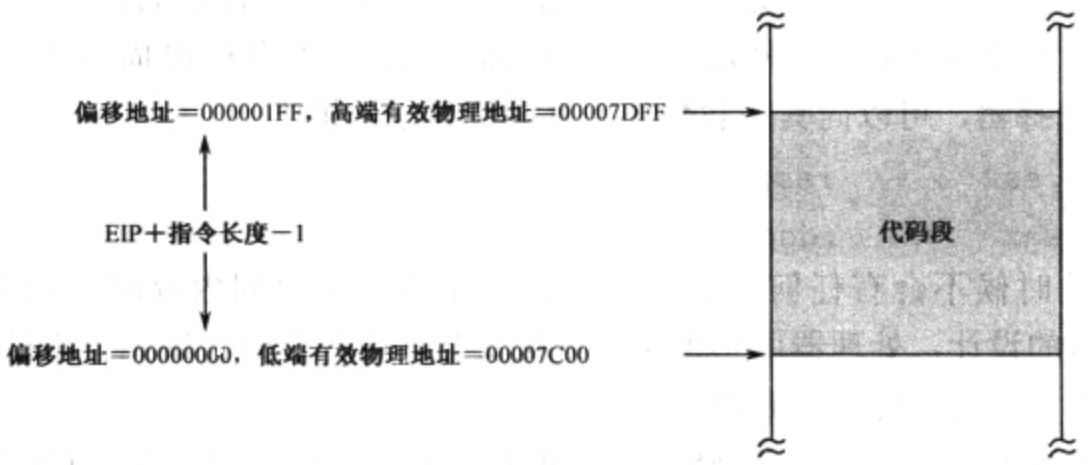

图 12-3　对代码段偏移地址的检查

做为一个额外的例子，现在，假设当前代码段的粒度是 4KB，那么，因为描述符中的段界限值是 0x001FF，故实际使用的段界限是

```
0x1FF×0x1000＋0xFFF＝0x001FFFFF
```

可以认为，此数值就是当前段内最后一个允许访问的偏移地址。任何时候，EIP 寄存器的内容加上取得的指令长度减 1，都必须小于等于 0x001FFFFF，否则将引发处理器异常中断。

任何指令都不允许，也不可能向代码段写入数据。而且，只有在代码段可读的情况下（由其描述符指定），才能由指令读取其内容。

12.4.2　栈操作时的保护

在保护模式下操作时，栈是一个容易令人感到迷惑的话题。在截止到目前的所有例子中，栈段一直是使用向下扩展的内存段，段界限的检查和向上扩展的数据段和代码段不同。当然，栈也可以使用向上扩展的段，即，把数据段用做栈段。在这种情况下，对段界限的检查按数据段的规则进行，但是无论如何，栈本身始终总是向下增长的，即，向低地址方向推进。段的扩展方向用于处理器的界限检查，而对栈的性质以及在栈上进行的操作没有关系。在第 16、17 章中，我们会接触到用向上扩展的段作为栈段的情况，现在仍然只讨论向下扩展的栈段。

对栈操作的指令一般是 push、pop、ret、iret 等。这些指令在代码段中执行，但实际操作的却是栈段。

现在只讨论 32 位的栈段，即，其描述符 B 位是 1 的栈段。处理器在这样的段上执行压栈和出栈操作时，默认使用 ESP 寄存器。

和前面刚刚讨论过的代码段一样，在栈段中，实际使用的段界限也和粒度（G）位相关，如果 G＝0，实际使用的段界限就是描述符中记载的段界限；如果 G＝1，则实际使用的段界限为

```
描述符中的段界限值×0x1000＋0xFFF
```

栈段是向下扩展的，每当往栈中压入数据时，ESP 的内容要减去操作数的长度。所以，和向高地址方向扩展的段相比，非常重要的一点就是，实际使用的段界限就是段内**不允许访问的**最低端偏移地址。至于最高端的地址，则没有限制，最大可以是 0xFFFFFFFF。也就是说，在进行栈操作时，必须符合以下规则：

```
实际使用的段界限+1≤（ESP 的内容－操作数的长度）≤0xFFFFFFFF
```

在上一章里，栈段的粒度是字节（G=0），描述符中的段界限是 0x07A00。此时，实际使用的段界限也是 0x07A00。

假设现在 ESP 的内容是 0x00007A04，那么，执行下面的指令时，会怎样呢？

```
push edx
```

因为是要压入一个双字（4 字节），故处理器在向栈中写入数据之前，先将 ESP 的内容减去 4，得到 0x7A00，这就是 ESP 寄存器在进行压栈操作时的新值。因为该值小于实际使用的段界限 0x7A00 加一（0x7A01），因此不允许执行该操作。

但是，如果执行的是这条指令：

```
push ax
```

那么，因为要压入一个字（2 字节），故实际执行压栈操作时，ESP 的内容是

```
0x7C04－2=0x7C02
```

结果大于实际使用的段界限加一，允许操作。

回到本章中，看代码清单 12-1 第 67～69 行。这三行设置栈的线性基地址为 0x00007C00，段界限为 0xFFFFE，粒度为 4KB，并设置栈指针寄存器 ESP 的初值为 0。

因为段界限的粒度是 4KB（G=1），故实际使用的段界限为

```
0xFFFFE×0x1000+0xFFF=0xFFFFEFFF
```

又因为 ESP 的最大值是 0xFFFFFFFF，因此，如图 12-4 所示，在操作该段时，处理器的检查规则是：

```
0xFFFFF000≤（ESP 的内容－操作数的长度）≤0xFFFFFFFF
```

栈指针寄存器 ESP 的内容仅仅在访问栈时提供偏移地址，操作数在压入栈时的物理地址要用段寄存器的描述符高速缓存器中的段基址和 ESP 的内容相加得到。因此，该栈最低端的有效物理地址是

```
0x00007C00+0xFFFFF000=0x00006C00
```

最高端的有效物理地址是

```
0x00007C00+0xFFFFFFFF=0x00007BFF
```

也就是说，当前程序所定义的栈空间介于地址为 0x00006C00～0x00007BFF 之间，大小是 4KB。

现在结合该栈段，用一个实例来说明处理器的检查过程。代码清单第 69 行将 ESP 的初始值设定为 0，因此，当第一次进行压栈操作时，假如压入的是一个双字（4 字节）：

```
push ecx
```

因为压栈操作是先减 ESP，然后再访问栈，故 ESP 的新值是（可以自行用 Windows 计算器算一下）

```
0－4=0xFFFFFFFC
```

这个结果符合上面的限制条件，允许操作。此时，被压入的那个双字，其线性地址为

```
0x00007C00+0xFFFFFFFC=0x00007BFC
```

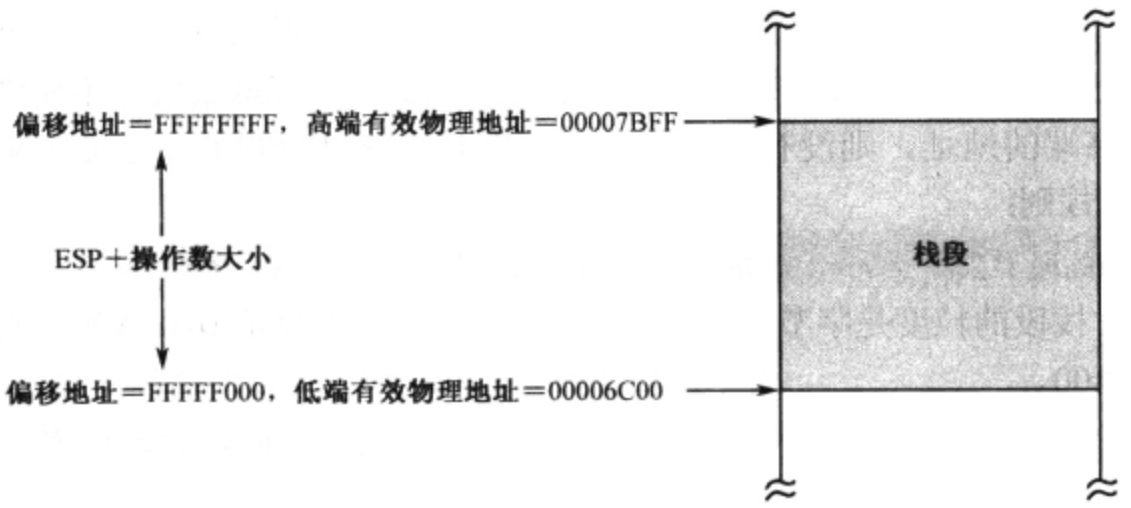

图 12-4　对栈段偏移地址的检查

尽管这里讨论的是 push 指令，但对于其他隐式操作栈的指令，比如 pop、call、ret 等，情况也没有什么不同，也要根据操作数的大小来检查是否违反了段界限的约束，以防止出现访问越界的情况。

12.4.3　数据访问时的保护

这里所说的数据段，特指向上扩展的数据段，有别于栈和向下扩展的数据段。

因为是向上扩展的，所以代码段的检查规则同样适用于数据段。不同之处仅仅在于，对于取指令来说，是否越界取决于指令的长度；而对于数据段来说，则取决于操作数的尺寸。考虑以下指令：

```
mov [0x2000],edx
```

这条指令将访问内存，并将 EDX 寄存器的内容写入当前段内偏移量为 0x2000 的双字单元。指令中给出了内存单元的有效地址 EA（0x2000），也给出了操作数的大小（4）。

很好，现在，当处理器访问数据段时，要依据以下规则进行检查：

```
0≤（EA＋操作数大小－1）≤实际使用的段界限
```

在任何时候，段界限之外的访问企图都会被阻止，并引发处理器异常中断。

在 32 位处理器上，尽管段界限的检查总在进行着，但如果段界限具有最大值，则对任何内存地址的访问都将不会违例。比如本章就定义了一个具有 4GB 长的段，段的基地址是 0x00000000，段界限是 0xFFFFF，粒度为 4KB。因此，实际使用的段界限是

```
0xFFFFF×0x1000＋0xFFF＝0xFFFFFFFF
```

在这样的段内，访问任何一个内存单元都是允许的，针对段界限的检查都会获得通过。

在 32 位模式下，处理器使用 32 位的段基地址加上 32 位的偏移量，共同形成 32 位的物理地址来访问内存。段基地址由段描述符指定，而偏移量由指令直接或者间接给出。很显然，在段最大的时候，可以自由访问 4GB 空间内的任何一个单元。

代码清单 12-1 第 71～74 行，从物理地址 0x000B8000 开始写入 16 字节的内容，用于演示 4GB 内存地址空间的访问。段寄存器 ES 当前正指向 0 到 4GB 的内存空间，其描述符高速缓存器中的基地址是 0x00000000，加上指令中提供的 32 位偏移量，所访问的地方正是显示缓冲区（显存）所在的区域。这其中的道理很简单，首先，内存的寻址依赖于段基地址和偏移地址，段基地址是 0，所以，可以把任何要访问的物理地址作为偏移量。

这 16 字节的内容是 8 个字符的 ASCII 码，以及它们各自的显示属性（颜色）。如图 12-5 所示，和往常一样，双字在内存中的写入依然是低端字节序的，这里再次展示一下，以帮助理解。

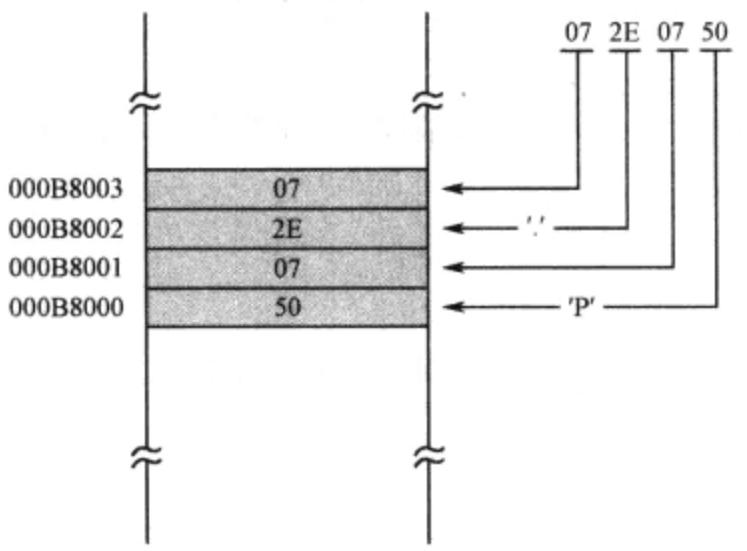

图 12-5 以低端字节序向内存中写入双字

要理解 32 位模式下的寻址，以及数据访问时的保护机制，这是一个很好的例子。

◆ **检测点 12.2**

当前栈段描述符的 B 位是 1，基地址为 0x00700000，界限值为 0xFFFFE。那么，在 32 位模式下，该栈段的有效地址范围是 0x00700000～（ ）。当 ESP 的内容为 0xFFFFF002 时，还能压入一个双字吗？为什么？

12.5 使用别名访问代码段对字符排序

接下来要做的事情是对一串散乱的字符进行排序。坦白地说，排序是假，主要目的是演示如何在保护模式下使用别名段。

字符串位于代码清单 12-1 的第 105 行，用标号 string 声明，并初始化为以下字符：

```
s0ke4or92xap3fv8giuzjcy5l1m7hd6bnqtw.
```

这串字符是主引导程序的一部分，在进入保护模式时，它就位于 32 位代码段中。代码段是用来执行的，能不能读出，取决于其描述符的类别字段。但是无论如何，它都不允许写入。

这可就难办了。我们想就地把这串字符按 ASCII 码从小到大排列，涉及原地写入数据的操作。好在前面已经建立了代码段的别名描述符，而且用段寄存器 DS 指向它。参见代码清单 12-1 第 59、60 行。

冒泡排序是比较容易理解的排序算法，但却并不是效率最高的，因此，速度自然也就很慢。如果字符串的长度（字符的数量）是 n 个，而且要从小到大排序，那么，可以将它们从头至尾两两比较，需要比较 n-1 次。但是，不要高兴太早，这一次遍历只会使最大的那个字符慢慢地、像气泡一样移动到最右边。

所以，你需要多次进行这样的遍历才能完成所有字符的排序，每一次遍历都会使一个字符冒泡到正确的位置。可以计算，共需要 n-1 次这样的遍历。有关冒泡排序算法的更多信息，请参考其他资料。

可见，这需要两个循环，一个外循环，用于控制遍历次数；一个内循环，用于控制每次遍历时的比较次数。在 32 位模式下，loop 指令所用的计数器不是 CX，而是 ECX。两个循环需要共用 ECX，这需要点技巧，那就是利用栈：

```
        mov ecx,n-1             ;控制遍历次数，内、外循环都用它
external:
        xor ebx,ebx             ;清零，从字符串开头处比较
        push ecx

internal:
        …                       ;对字符串两两比较
        inc ebx
        loop internal

        pop ecx
        loop external
```

我相信这段框架性的代码还是很好理解的。外循环总共执行 n-1 次。每执行一次外循环，内循环就会将一个数排到正确的位置，从而使下一次内循环少一次两两比对（少执行一次）。也就是说，ECX 寄存器的当前值总是内循环的次数，这就是为什么内循环的 loop 指令要使用外循环的 ECX 值。

代码清单 12-1 第 77 行，用后面的标号 pdgt 减去声明字符串的标号 string，就是字符串的长度，再减去一，就是控制循环的次数。

第 79 行，将循环次数压栈，因为内循环会改变 ECX 的内容。

第 80 行，清零 BX 寄存器。该寄存器在每次内部循环之前清零，用于从字符串的开始处进行比对。之所以没有使用 EBX，是因为要让你知道，32 位代码中也可以使用 16 位的寄存器来寻址。注意，我们知道，在 32 位模式下，如果指令的操作数是 16 位的，要加前缀 0x66。相似地，在 32 位模式下，如果要在指令中使用 16 位的有效地址，那么，必须为该指令添加前缀 0x67。因此，当指令

```
        mov eax,[bx]
```

用 bits 32 编译后，会有指令前缀 0x67；在 32 位模式下执行时，处理器会用数据段描述符中给出的 32 位数据段基地址，加上 BX 寄存器的 16 位偏移量，形成 32 位线性地址。

实际进行字符比对的代码是第 81～91 行。首先一次性读取两个字符到 AX 寄存器中。当前的数据段是由段寄存器 DS 指向的，其描述符给出的基地址为 0x00007C00，字符串的首地址就是标号 string 的汇编地址，寄存器 BX 用来指定字符串内的偏移量。

接着，对寄存器 AH 和 AL 的内容进行比较。如图 12-6 所示，AL 中存放的是前一个字符，AH 中存放的是后一个字符。如果前一个字符较大，则交换 AH 和 AL 的内容，然后重新写回原来的字单元。然后，将 BX 寄存器的内容加一，以指向下一个字符。

xchg 是交换指令，用于交换两个操作数的内容，源操作数和目的操作数都可以是 8/16/32 位的寄存器，或者指向 8/16/32 位实际操作数的内存单元地址，但不允许两者同时为内存地址。其格式为

```
        xchg r/m8,r8
        xchg r/m16,r16
        xchg r/m32,r32
        xchg r8,m8
        xchg r16,m16
```

```
xchg r32,m32
```

举个例子：

```
mov ecx,0xf000f000
mov edx,0xabcdef00
xchg ecx,edx
```

以上指令执行后，寄存器 ECX 中的内容为 0xABCDEF00，EDX 中的内容为 0xF000F000。

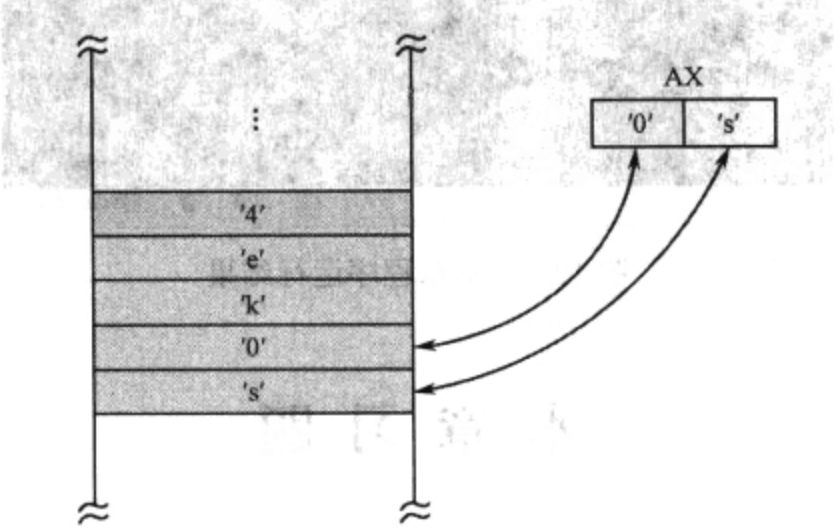

图 12-6 通过 AX 寄存器比对和排序相邻字符

第 93～100 行用于显示最终的排序结果，同样使用了循环，循环次数就是字符串的长度。和排序的时候不同，现在终于使用 EBX 了，这将提供 32 位的偏移地址。

第 96 行，向寄存器 AH 传送的是字符的显示属性（颜色），0x07 表示黑底白字，我们已经无数次重复说过了。

第 98 行是向显存中传送字符及其显示属性：

```
mov [es:0xb80a0+ebx*2],ax
```

段寄存器 ES 是在刚进入保护模式时设置的，它指向 0～4GB 内存的段。0xb80a0 等于 0xb8000 加上十进制数 160（0xa0）。在显存中，偏移量为 160 的地方对应着屏幕第 2 行第 1 列。32 位处理器提供了强大的寻址方式，可以在基址寄存器的基础上使用比例因子，这里是将 EBX 寄存器的内容乘以 2。当 EBX 的内容为 0、1、2、3、…时，计算出来的有效地址分别是 0xb80a0、0xb80a2、0xb80a4、0xb80a6、…，后面的以此类推，很容易看到使用比例因子的好处。注意，该表达式的值是在本指令执行时，由处理器来计算的。

最后，在完成了所有的工作之后，第 102 行，hlt 指令使处理器处于停机状态。

12.6 程序的编译和运行

本章代码清单 12-1 所对应的源程序文件是 c12_mbr.asm，用 Nasmide 工具将它打开并编译，生成二进制文件 c12_mbr.bin 并写入虚拟硬盘的主引导扇区。

然后，启动虚拟机 LEARN-ASM，观察运行结果。正常情况下，屏幕显示如图 12-7 所示。

图 12-7　本章程序运行结果

本 章 习 题

1．修改本章代码清单，使之可以检测 1MB 以上的内存空间（从地址 0x00100000 开始，不考虑高速缓存的影响）。要求：对内存的读写按双字的长度进行，并在检测的同时显示已检测的内存数量。建议对每个双字单元用两个花码 0x55AA55AA 和 0xAA55AA55 进行检测。

2．有一个向下扩展的段，描述符中的 B 位和 G 位都是“1”。请问，如果希望段的大小为 8KB，那么，描述符中的界限值应当是多少？