目录

第 3 章 主存储器及数据在计算机内的表示形式

主存储器也称为内存，用来存储当前运行的程序和数据。本章介绍存储单元中数据的存放形式、存储单元的物理地址形成、数值数据、字符数据在计算机内的表示形式、以及存储单元的数据类型和相互转换。这些内容是对数据进行存储和访问的基础。

3．1 主存储器

3.1.1 数据存储的基本形式

内存是是用来存储程序和数据的装置。它由许多存储位构成，每一位上能记住一个二进制数码 0 或者 1。每 8 个存储位组合成一个字节（Byte），换句话说，一个字节由 8 个二进制位组成。相邻的 2 个字节组成一个字（Word），即一个字由 16 个二进制位组成。相邻的 2 个字（连续的四个字节）组成一个双字（Double Word），连续的 8 个字节组成一个四字数据。在 C 语言中，char、short、int、long、double 就分别对应 1 个、2 个、4 个、8 个字节。

为了让 CPU 能够访问到指定的存储单元，就必须给各个存储单元赋予一个唯一的无歧义的编号。x86 微机的内存是按字节编址的，即以字节作为最小寻址单位，每次对内存中单元的存取访问至少是一个字节，而不能是一个字节中的某一位。对于内存中的每一个字节存储单元，都被指定一个唯一的编号，称为此单元的物理地址(Physical Address,PA)。字节的编址可以看成是从 0 开始，按自然数顺序编址的。在 x86-32 中，地址总线有 32 根，用 32 个二进制位进行编码，地址从 00000000H 开始直到 FFFFFFFFH，内存大小可达到 2^{32} 个字节，即 4GByte。在 x86-64 中，地址总线有 64 根，用 64 个二进制位进行编码，内存大小可达到 2^{64} 个字节。

除了字节单元数据外，还可以一次存取访问 2 个字节（字，16 位数据）、4 个字节（双字，32 位数据），或者 8 个字节的数据（四字，64 位数据）。字是由 2 个相邻字节组成的，字的地址是 2 个字节地址中较小那个地址。类似的，双字的地址是四个字节地址中的最小值。四字的地址是连续 8 个字节中最小的字节地址。

存放一个字数据需要 2 个字节。例如字数据 1234H，其高字节内容是 12H，

低字节内容是 34H。假设要将该数据存放到地址为 0040FF08H 的单元中，这两个字节的内容在内存中怎么存放呢？存放双字数据 12345678H 到地址为 0040FF08H 的单元，要占 4 个字节的存储字节，数据有如何摆放呢？

数据存储方法有两种，一种是小端存储（Little Endian），另一种是大端存储（Big Endian）。在小端存储方式中，最低地址字节中存放数据的最低字节，最高地址字节中存放数据的最高字节。按照数据由低字节到高字节的顺序依次存放在从低地址到高地址的单元中。大端存储方式正好相反。 不同的 CPU 采用的存储模式并不相同。Intel x86 系列等采用的小端存储方式。

例如，存放双字数据 12345678H 到地址为 0040FF08H 的单元的结果如图 3.1 所示。图的右边是各个字节的地址，左边框中是各字节中存放的内容。

78H	0040FF08H
56H	0040FF09H
34H	0040FF0AH
12H	0040FF0BH

图 3.1 主存储器中数据存储示意图

值得注意的是，Intel x86 系列采用小端存储方式是指数据在内存中的存放形式。在 x86 计算机上，可以运行 Windows、Linux、Solaris、Dos 等多种操作系统，它们在读写数据文件时，数据文件中采用的数据存放字节顺序可以是不同的。这是数据文件的读写问题。数据在网络上传输时，也会出现类似的问题，不同的存储格式需要进行转换。

3.1.2 数据地址的类型及转换

按照图 3.1，从地址 0040FF08H 开始处取一个字节的内容是 78H,即（0040FF08H）=78H；取一个字的内容是（0040FF08H）=5678H；取一个双字的内容是（0040FF08H）=12345678H。从这一段描述中可以看到，在访问一个存储单元时，不但要给出单元的地址而且还要指明类型，即是取一个字节、一个字、还是一个双字等。

在 C 语言中，同样有类型转换的问题。下面的程序段给出了一个地址类型转换的示例。

```
char a[10]="1234567";
printf("%c %x\n", *(char *)a, *(char *)a);
printf("%d %x\n", *(short *)a, *(short *)a);
printf("%ld %x\n", *(int *)a, *(int *)a);
```

执行该段程序，显示的结果为：

```
1    31
12849   3231
```

875770417　34333231

其中，3231H 转换为十进制数为 12849；34333231H 转换为十进制数为 875770417。

运行结果分析：对于字符数组 a，它是按照字符串“1234567”从左到右的顺序，从内存的低端到高端依次存放各个字符的 ASCII,即 31H、32H、…、37H，字符串以数字 0 结束。程序中的三条打印语句，都是从 a 的首地址开始读取要打印数据，只不过读取的数据类型不同，分别读取 1 个字节的数据、2 个字节的短整型数据和 4 个字节的长整型数据。a 本身是字符类型的地址，采用(short *)a 将其转换成了短整型的地址。

在实际的系统开发中，经常会遇到地址类型转换问题。例如，开发一个学生信息管理系统，管理信息包括学号、姓名、年龄、身份证号等固定长度的数据类型，也可以含有奖惩记录等长度变化较大的信息。如果有固定长度的空间来存放奖惩记录，空间开销会较大，因为有学生的记录内容多，但对于大多数人来讲，这些空间会被浪费。为了节约空间，包括存放在文件或内存中的空间，我们可以申请一个大块空间，即一个大的字符数组来紧凑的存放这些信息。其中有些数据占固定长度，如一个短整型数 2 个字节、一个单精度浮点数 4 个字节等，有些用变长，如字符串以 0 结束。根据数据存放约定的规则和实际的存放情况，可以得到各学生的信息的起始地址以及其中各个项的起始地址，此时就可以用地址类型转换来读取各种类型的数据。

另外，需要注意的是地址类型转换与数据类型转换的差别。地址类型转换，是指相同的单元中的内容用不同的类型来解读，例如，相同的 4 个字节的数据，若以整型来解读会对应出一个整数，若以单精度浮点数来解读，同样的 4 个字节的内容，解读出来又是另一个结果。数据类型转换，是指相同的数据，用不同的形式表达。例如，一个 int 类型的数转换为 float 类型的数，虽然都用 4 个字节来存储，但字节中的内容是不同的。若将一个短整型数变成一个长整型数，或者一个单精度浮点数转换成一个双精度浮点数，所占的单元字节数会发生变化，存储的结果也不相同。

3．2 数值数据在计算机内的表示形式

在计算机中，数值数据有两种表示法：定点表示法和浮点表示法。浮点表示法比定点表示法所表示的数的范围大，精度高。早期的 x86 CPU 是通用微处理器，它处理的数据小数点位置是固定的，属定点数，对浮点数的数值运算是由与其配套的浮点部件 x87 FPU 实现的。在后面出现的 CPU 中，所有支持 MMX、SSE 扩展和 Intel AVX 扩展的系统中，都支持单/双精度（32 位/64 位）的浮点数数据类型。浮点数的表示形式在第 14 章 x87 FPU 程序设计中介绍。本节仅介绍整型数据的表示方法，细分为有符号整数和无符号整数的表示方法、BCD

（Binary Coded Decimal，二进制编码的十进制）码表示法。

3.2.1 有符号和无符号整数的表示形式

无符号整数就是一个普通的二进制数值，其取值范围是 0 至操作数各二进制位全为 1 的最大整数。设一个字节的二进制数为 $b_7b_6b_5b_4b_3b_2b_1b_0$，其中 $b_i(0 \le i \le 7)$为 0 或者 1，b_7为最高二进制位，b_0为最低二进制位，其对应的无符号十进制数是$\sum_{i=0}^{7}(b_i * 2^i)$。对于其他长度的数据，如一个字（16 个二进制位）、一个双字（32 个二进制位），计算方法是类似的。

表 3.1 无符号数的表示范围

n	十进制数表示范围		十六进制数表示范围	
	最大值	最小值	最大值	最小值
8 位	255	0	0FFH	0
16 位	65535	0	0FFFFH	0
32 位	4294967295	0	0FFFFFFFFH	0

有符号整数表示有正有负的二进制数值。在计算机中，使用补码来表示一个有符号数。设一个字节的二进制数补码表示为 $b_7b_6b_5b_4b_3b_2b_1b_0$，若b_7为 0，则该数为正数，其值为$\sum_{i=0}^{6}(b_i * 2^i)$。由此也可知，最大的正数是 01111111B，即+127。当b_7为 1，表示为负数，其值是$-(2^8 - \sum_{i=0}^{7}(b_i * 2^i))$。因此，当数据为 10000000B，对应最小的负数-128。当数据为 11111111B 时，对应最大负数-1。此外，对于负数的值，也可以先对各二进制位都求反然后加 1。

$$2^8 - \sum_{i=0}^{7}(b_i * 2^i) = \{\sum_{i=0}^{7}(1 - b_i) * 2^i\} + 1$$

对于一个十进制的负数，求其补码表示时，可以先求其相反数（即不理睬负号）对应的二进制数，并且按给定的字长表示，高位不足的部分补为 0，然后对各二进制位求反，最后再加 1 即可。当然一个正数的补码是其本身的二进制表

示。

下面将字节、字、双字所能表示有符号数的范围列在表 3.2 中。

表 3.2 有符号数的表示范围

n	十进制数表示范围		二进制数表示范围		补码表示范围	
	最大值	最小值	最大值	最小值	最大值	最小值
8 位	+127	-128	2^7-1	-2^7	7FH	80H
16 位	+32767	-32768	$2^{15}-1$	-2^{15}	7FFFH	8000H
32 位	+2147483647	-2147483648	$2^{31}-1$	-2^{31}	7FFFFFFFH	80000000H

【例 3.1】 设有符号数 M＝33，求 M 的 8 位、16 位、32 补码表示

【解】 M 的 8 位补码表示为：[M]$_{补}$＝21H

M 的 16 位补码表示为：[M]$_{补}$＝0021H

M 的 32 位补码表示为：[M]$_{补}$＝00000021H

【例 3.2】 设有符号数 M＝－33，求 M 的 8 位、16 位和 32 位补码表示

【解】 -33 = -21H，其反码表示为 0DEH。一个数和它的反码之和为 0FFH。

M 的 8 位补码表示为：[M]$_{补}$＝0DFH

M 的 16 位补码表示为：[M]$_{补}$＝0FFDFH

M 的 32 位补码表示为：[M]$_{补}$＝0FFFFFFDFH

从以上两例均可看出，M 的 16 位补码实际上是其 8 位补码的符号扩展；M 的 32 位补码是其 16 补码或 8 位补码的符号扩展。

由此可得出一个很重要的结论：一个二进制数的补码表示中的最高位(即符号位)向左扩展若干位(即符号扩展)后，所得到的仍是该数的补码。

注意：对于无符号数类型的变量，在定义在 C 语言程序中，可以通过在类型前增加关键字 unsigned 来定义，如 unsigned char、unsigned short，unsigned int。按理而言，给无符号整型变量赋值时，其右的表达式不应出现负号。但在实际程序中，可以出现负号，其存放的结果该负数的补码表示。例如 unsigned short x=-1，x 中存放的结果是 FFFF，这与 short y=-1 的存放结果是一样的。但是 printf("%d %d\n",x,y); 显示的结果是不同的，前者是 65535，后者是-1，即对 0FFFFH 分别解释成无符号数 65535 和右符号数-1。这就表明相同存储单元中的内容会根据定义的类型不同而解释成不同的结果。

3.2.2 BCD 码

BCD(Binary Coded Decimal)码的特点是利用二进制形式来表示十进制数，即用 4 位二进制数(0000B～1001B)表示一位十进制数(0～9)，而每 4 位二进制数之间的进位又是十进制的形式。因此，BCD 码既具有二进制的特点又具有十进制的特点。

例如：1098＝0001000010011000$_{BCD}$

$79=01111001_{BCD}$

BCD 码的使用为十进制数在计算机内的表示提供了一种简单而实用的手段，特别是 x86 具有直接处理 BCD 码的指令，更给我们带来了方便。

在 x86 中，根据 BCD 码在存储器中的不同存放方式，又分为未压缩的 BCD 码和压缩的 BCD 码。未压缩的 BCD 码每个字节只存放一个十进制数字位，而压缩的 BCD 码是在一个字节中存放两个十进制数字位。

例如，将十进制数 9781 用压缩的 BCD 码表示为：

1001011110000001

在主存中的存放形式为：

1 0 0 0 0 0 0 1
1 0 0 1 0 1 1 1

而用未压缩的 BCD 码表示为：00001001000001110000100000000001

在主存中的存放形式为：

0 0 0 0 0 0 0 1
0 0 0 0 1 0 0 0
0 0 0 0 0 1 1 1
0 0 0 0 1 0 0 1

3．3 字符数据在机内的表示形式

在 x86 中，字符数据的二进制数值表示遵循 ASCII 码标准，用一个字节存放一个字符。为了区别数值数据，程序中的字符数据全部以单引号或双引号括起来。

字符数据的使用给人和计算机交换信息带来了很大的方便。例如，当从键盘上敲入字符串'12ABCD'时，从输入设备处接受的首先是键盘扫描码，当一个键按下时，它产生一个唯一的数值。当然，当一个键被释放时，它也会产生一个唯一的数值，键盘上的每一个键都有两个唯一的数值进行标志。将这些数值都保存在一张表里面，通过查表就可以知道是哪一个键被敲击，并且可以知道是它是被按下还是被释放了（如大小写切换键 CapsLock 的状态）。这些数值在系统中被称为键盘扫描码。在读键盘扫描码后，由键盘中断处理程序或者用户自己编写程序等，将其转换成与之对应的 ASCII 码 31H、32H、41H、42H、43H、44H。

3．4 数据段定义

在汇编语言程序中，可以包含有数据段，在数据段中定义变量或者分配一些待用的空间。数据段以 “.data”开头，到一个段定义结束。

在数据段中，可以由多个数据定义伪指令组成。语句格式如下：

[变量名] 数据定义伪指令 表达式 1 [，表达式 2，…]

其中，在方括号中的内容是可省的。如果没有出现变量名，就只分配了一片存储空间，数据定义伪指令后，至少出现一个表达式，若有多个表达式，表达式之间以西文的逗号分隔。

1、数据定义伪指令

数据定义伪指令有 db、dw、dd、dq、dt、real4、real8、byte、word、dword、qword、sbyte、sword、sdword、sqword、tbyte 等。

db、byte、sbyte，这三条数据定义伪指令都是定义字节类型的变量，每个表达式占一个字节。db 和 byte 定义的变量等同于 C 语言中的 unsigned char 类型，而 sbyte 定义的变量等同于 C 语言中的 char 类型。

dw、word、sword，定义字类型的变量，每个表达式占二个字节。dw 和 word 是等价的，它们与 sword 之间的异同点与 byte 和 sbyte 是一样的。sword 定义的是有符号字类型变量。

dd、dword、sdword，定义双字类型的变量，每个表达式占 4 个字节。

dq、qword、sqword 定义四字类型的变量，每个表达式占 8 个字节。

dt 与 tbyte 等价，定义 10 字节类型的变量，每个表达式占 10 个字节。

real4 定义单精度浮点数类型的变量，每个表达式占 4 个字节。

real8 定义双精度浮点数类型的变量，每个表达式占 8 个字节。

提示：在定义有符号和无符号类型的变量时，其后的表达式相同时变量对应的单元中存储的内容相同。但是在使用控制流伪指令编写程序（参见分支程序设计），由编译器来完成“类似 C 语言”的程序编译时，会生成不同的机器指令。

2、表达式

表达式确定了变量的初值，所使用的表达式可以是以下几种形式。

(1) 数值表达式

所谓数值表达式是由数值常量、符号常量和一些运算符组成的有意义的式子。单个数值常量，如 10、20H、1010B 等都是一个数值表达式。

所谓符号常量，就是用“=”或“equ”定义的一个符号。

例如：num = 10 或者 num quu 10。

这等价于 C 语言中的 #define num 10。

注意，符号常量不是变量，不会为其分配空间，它可以出现在程序的任何位置，在编译的时候，编译器发现了一个符号常量就会用其值代替。

运算包括算术运算、逻辑运算和关系运算。算术运算有加(+)、减(-)、乘(*)、除(/)、模除(mod)、右移(shr)、左移(shl)；逻辑运算有逻辑乘(and)、逻辑加(or)、按位加(xor)和逻辑非(not)；关系运算包括相等(eq)、不等(ne)、小于(lt)、大于(gt)、小于等于(le)及大于等于(ge)。

注意，数值表达式在编译后就变成了一个值。在分配的空间中存放的就是编译后的值。例如 x dw 3*3+4*4 完全等价于 x dw 25。

(2) ASCII 字符串

一般只跟在 db 后面。s db 'ABCD' 等同于

s db 'A', 'B', 'C', 'D'

也等同于 s db 41H, 42H, 43H, 44H

当字符串长度等于 2 时，可以跟在 dw 后面。

当字符串长度等于 4 时，可以跟在 dd 后面。

(3) 地址表达式

单个变量是一个地址表达式。在程序中出现的标号、定义的子程序名，都是地址表达式。它们是存储空间中某个单元的地址的符号表达式。

一个变量名加或减一个数值表达式，得到是仍是地址表达式。它是取变量的地址参与运算的，最后得到了另一个单元的地址。

在 32 位段中，变量的地址是 32 位的，因此要出现在 dd 后面。

注意：若表达式是两个变量的差，例如 z dw x – y，z 单元中的内容是变量 x 的地址与变量 y 的地址之差，是两个变量起始地址之间的字节距离。但是，若定义 z dw x+y 则有语法错误。为什么编译器要做这样的规定？可以在我们的生活找到类似的例子。一个日期可以减去另一个日期，得到两个日期之间的时间间隔，但两个日期相加则没有意义。一个日期加上或者减去一个时间间隔，可以得到一个新日期。一个地址加或减一个长度，可以得到一个新的地址，两个地址之差表示了长度。

(4) 重复子句

重复子句的格式为：n dup(表达式)

其中，n 是重复因子（只能取正整数），它表示 dup 后面圆括号中的表达式重复出现 n 次。表达式可以由一个，也可以由多个，还可以嵌套出现重复子句。

例如： x db 3 dup (5) 等价于 x db 5, 5, 5

y db 3 dup (1, 2) 等价于 y db 1,2 , 1,2, 1,2

z db 3 dup(1, 2 dup(5)) 等价于 z db 1,5,5,1,5,5,1,5,5

在一个数据定义伪指令后面出现多个表达式时，要按从左到右的顺序，依次给各个表达式分配空间。先出现的表达式分配在地址小的空间上，后出现的表达式分配在地址大的空间上。

对于数据段中多个数据定义伪指令，也按照从上到下的顺序，依次给各个数据定义伪指令分配空间，先出现的伪指令，分配在地址小的空间上。

注意，对于字、双字、四字数据，按照小端存储方式存放。

3、汇编地址计数器

在编译器对程序进行翻译的时候，利用了汇编地址计数器来记录当前拟使用的存储单元的地址。存储单元可以用来存储数据，也可以用来存储指令。汇编地址计数器的符号记为$，标示了汇编程序当前的工作位置。汇编地址计数器符号$可出现在表达式中。

设有如下数据段定义

```
x   db   'ABCD'
y   dw   $-x
z   dw   $-x,   $-x
```

在 y 定义中的表达式出现了$，即表示 y 的地址。假设 x 的地址是 0x009e4000，则 y 的地址是 0x009e4004，$-x 的值为 4，是从$出现的位置到变量 x 出现的位置之间的字节距离。

在 z 定义中出现了两个$，第一个表达式中出现的$ 表示 z 的地址，即准备分配给 z 的第一个表达式的存储空间的地址，此时$=0x009e4006，故第一个 $-x 的值为 6。在分配第一个表达式的空间后，$要自增分配单元的长度，此时$=0x009e4008，故第二个表达式的值为 8。

注意，两个地址之间的差，是一个数值表达式，不再具有变量的属性，它可以跟在任何数据定义伪指令之后，只要它的值不超出该数据定义伪指令所要求的数据范围。

利用汇编地址计数器，可以很容易的由编译器自动计算当前位置到指定变量之间的距离，例如 y 中的值就是变量 x 中字符串的长度。

```
x   dw   20,   30,   40
len = ($-x)/2     ; 该值为 x 中字数据的个数。
```

4、数据段定义举例

设有如下程序，在反汇编窗口观察数据存放结果，如图 3.2 所示。

```
.686P
.model flat, stdcall
   ExitProcess proto    :dword
   includelib    kernel32.lib
.data
   x    db    10, 20, 30
   y    dw    10, 20, 30
   z    dd    10, 20, 30
   u    db    '12345'
   p    dd    x,    y
   q    db    2 dup (5), 3 dup (4)
.stack 200
.code
```

```
start:
    invoke ExitProcess, 0
end start
```

在调试程序时，在监视窗口输入&x，可以看到 x 的地址为 0x00fb4000。打开内存窗口，输入地址 0x00fb4000，可看到从该地址开始的一片内存单元的值，如图 3.2 所示。窗口中最左端的一列是单元的地址。右边的部分依次是各个字节单元中的内容。

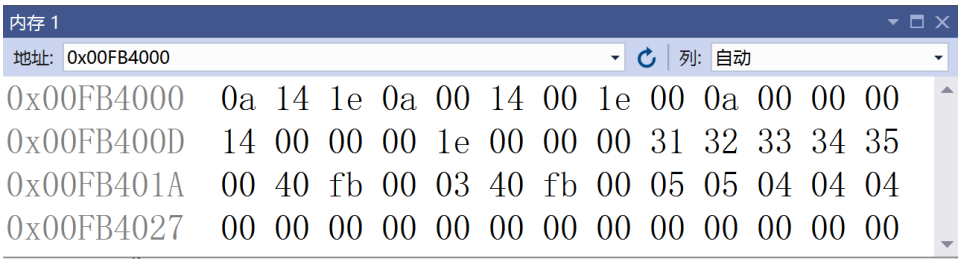


图 3.2 数据段数据存放结果截图

图 3.3 给出了等价的数据段内存示意图。

对于变量 p，其中存放的变量 x 和 y 的地址。在存放地址时，等同于存放一个双字数据，数据的高位字节存放在大地址对应的单元中。

```
.data
    x   db   10, 20, 30
    y   dw    10, 20, 30
    z   dd   10, 20, 30
    u   db   '12345'
    p   dd   x,   y
    q   db   2 dup (5), 3 dup (4)
```

x	0aH	00FB4000H		0eH	00FB4011H
	14H			00H	
	1eH			00H	
	0aH	00FB4003H		00H	
	00H		u	31H	00FB4015H
	14H			32H	
	00H			33H	
	1EH			34H	
	00H			35H	
z	0aH	00FB4009H	p	00H	00FB401AH
	00H			40H	
	00H			0fbH	
	00H			00H	
	14H	00FB400DH		03H	00FB401EH
	00H			40H	
	00H			0fbH	
	00H			00H	

图 3.3 数据存储示意图

说明：上面的完整的程序与第 1 章中给出的程序例子的写法有些不同。在本例中，没有调用 C 语言的标准库函数，也就不存在加载相应的库函数的问题，不需要让程序的入口点为 main，可以自己在程序中设定入口点，用“end 标号/子程序名”均可。当然，也可以套用第 1 章的程序例子来改写程序。

3．5 主存储器分段管理

在 2.6 节分段部件和分页部件中介绍了为什么内存要实行分段管理，本节将介绍如何实现分段管理。

内存管理有两种模型，扁平内存模型和分段内存模型。

在扁平内存模型中，代码、数据、堆栈等全部放在同一个 4GB 的空间中，如图 3.4 所示。值得注意的是，虽然它们放在同一个空间上，但仍然是分离的，分布在不同的子空间中。各个段寄存器会被加载成指向重叠段的段选择子，指向线性地址空间中的一个重叠段，该段从线性地址空间的地址 0 开始。

注意：不同的段寄存器指向了相同的内存起始地址，但是代码部分、数据部分在段中的偏移地址是不同的。所谓偏移地址，也称为有效地址，是指一个存储单元与段开始单元之间的字节距离，也等价于该存储单元的物理地址减去段首单元的物理地址。

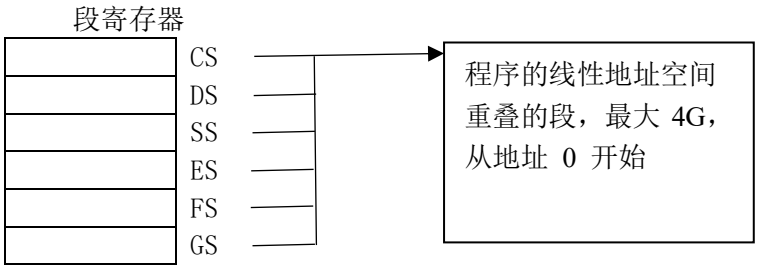


图 3.4 扁平内存模型的示意图

在分段存储器模型中，每个分段寄存器通常加载不同的分段选择器，以便每个段寄存器指向线性地址空间内的不同段，如图 3.5 所示。因此，程序可以随时访问线性地址空间中的六个段。访问段寄存器未指向的段时，程序必须首先加载要被访问到段寄存器中的段。

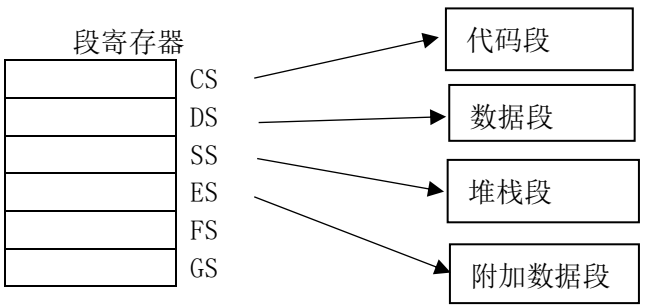


图 3.5 分段内存模型的示意图

段寄存器的使用方式取决于操作系统或执行部件正在使用的内存管理模型。

3．6 主存储器物理地址的形成

3.6.1 8086 和 x86-32 实方式下物理地址的形成

在 x86 系列机中，最低档 CPU 是 8086，它只有 20 根地址线，直接寻址能力为 2^{20}B，也就是说，主存容量可达 1MB，物理地址编号从 0～0FFFFFH。这样一来，CPU 与存储器交换信息必须使用 20 位的物理地址。但是，8086 内部却是 16 位结构，它里面与地址有关的寄存器全部都是 16 位的，例如，sp、bp、si、di、ip 等。因此，它只能进行 16 位地址运算，表示 16 位地址，寻找操作数的范围最多也只能是 64K 字节。为了能表示 20 位物理地址，8086 的设计人员提出了地址由段寄存器和段内偏移地址组成的方案。设置四个段寄存器 CS、DS、SS、ES，保存当前可使用段的段首址。如果使各段的段首址都从能被 16 整除的地址开始，那么，这些段首址的最低 4 位总是 0，若暂时忽略这些 0，则段首址的高 16 位正好装入一个段寄存器中。访问存储单元时，CPU 可以根据操作的性质和要求，选择某一适当的段寄存器，将它里面的内容左移 4 位，即在最低位后面补入了 4 个二进制 0，恢复了段首址原来的值，再与本段中某一待访问存储单元的偏移地址相加，则得到该单元的 20 位物理地址，如图 3.6 所示。这样一来，寻找操作数的范围就可达到 1MB。

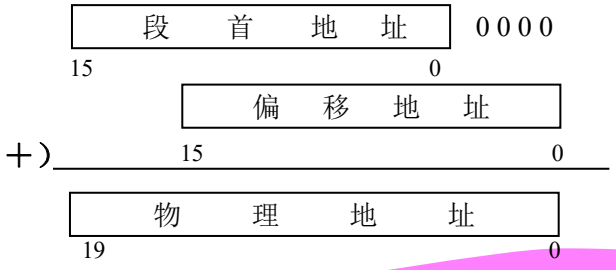


图 3.6　16 位 CPU 中物理地址的形成

由此可以看出，段中某一存储单元的地址是用两部分来表示的，即“段首地址:偏移地址”，我们称它为二维的逻辑地址。在编写程序时，采用这种表示逻辑地址表示法具有很好的优点，程序员不用关心程序运行时，代码和数据存放的实际物理地址；而代码段中的指令之间的位置关系、数据段中变量之间的位置在编译时可以确定。不论程序放在何处，内部的相对关系保持不变。由运行时确定的段首址以及编译时确定的偏移地址，就能容易的得到要访问单元的物理地址。

一个单元的物理地址是唯一的，但是它的二维逻辑地址可以很多。假设一个单元的物理地址是 12345H，我们可以将其表示为 1234H:05H，即段首地址为

1234H，它对应的物理地址为 12340H，05 是该单元在 1234H 这个段中的偏移地址。也可以将该物理地址表示为 1230H:45H，或者 1233H:15H 等等。

主存的分段使用技术可以让系统很方便地将程序中的代码段、数据段、堆栈段经重定位后，分开放在不同的存储区域里。尽管 CPU 在某一时刻最多只能同时访问 4 个段，但它并不限制程序中也只能定义 4 个段，用户完全可以根据自己的要求定义多个代码段、多个数据段和多个堆栈段，如果 CPU 需要访问 4 个段以外的存储区，只要改变相应段寄存器的内容即可。

随着 x86 系列机的发展，在 Intel 推出 32 位 CPU 后，它已能全面支持 32 位的数据、指令和寻址方式，可访问 4GB 的存储空间。但为了与低档的 8086 兼容，一直保留了主存的分段使用技术并提供了实工作方式。

在实方式下，32 位 CPU 与 8086 一样，但是扩充了 2 个附加数据段寄存器 FS 和 GS。在每一给定时刻 CPU 可同时访问 6 个段。这些当前能被 CPU 访问段的首地址由分段部件中的 6 个专用段寄存器来给出：

CS：给出当前代码段首地址 （取指令指针为 IP）

SS：给出当前堆栈段首地址 （堆栈指针为 SP）

DS：给出当前数据段首地址

ES、FS、GS：给出当前附加数据段首地址

代码段是程序代码的存储区。指令指示器（IP）为下一条将要取出指令的在代码段中的偏移地址。因此，指令物理地址 PA=（CS）左移四位+（IP）。

堆栈段是程序的临时数据存储区。程序中一般都需要用户自己建立堆栈段。在子程序调用、系统功能调用、中断处理等操作时，堆栈段是必不可少的。堆栈栈顶数据的偏移地址存放在 sp 中。

栈顶物理地址 pa=（ss）左移四位 +（sp）

数据段和附加数据段是程序中所使用的数据存储区。在一般情况下，程序中不需要定义附加数据段，如果必须定义附加数据段，在数据量不是太大时，最简单方法是让附加数据段和数据段重合，即将它们设置成一个段。需要在数据段中读/写数据时

数据的物理地址 PA=（DS 或 ES、FS、GS）左移四位 +16 位偏移地址

其中，数据的偏移地址由寻址方式确定。

值得注意的问题：

① 程序中每段的大小可根据实际需要而定，但必须≤64KB，每个段在主存中的具体位置由操作系统进行分配。

② 分段并不是唯一的，对于一片具体的存储单元来说，它可以属于一个段，也可同时属于几个段。换言之，DS、SS、CS 等段寄存器中的值可以相等。

③ 在汇编源程序中，通过一些质量将数据段首址置入 DS 或 ES、FS 和 GS 中才可使用，而 CS、SS 的初值由操作系统设置。

④ 段寄存器中的值可以在程序运行的过程中发生改变。

⑤ 段寄存器只指明了段从何处开始，并未指明到何处结束。一个存储单元

有多个逻辑地址，即它同时属于多个段，只是在各个段中的偏移地址不同。

3.6.2 保护方式下物理地址的形成

在保护方式下，由于使用了 32 位地址线，可寻址 4GB 的物理存储空间，程序段的大小也可达 4GB，段基址和段内偏移地址也是 32 位的。表面上看，由于有了 32 位的寄存器，不再需要像 8086 那样，由两个 16 位寄存器来“合成”一个 20 位的地址，而直接由一个 32 位的寄存器来确定地址。但是在保护模式下，出现了新的需要考虑的问题，即系统中有多个程序（任务）在同时运行，需要实施执行环境的隔离和保护，并进行调用权限的检查，防止不同程序之间的干扰和破坏，否则，在一个任务中越界修改另一个任务中的内容，对第二个任务就会造成不可预知的后果。因此，在保护方式下依然采用分段技术，将各个分段作为保护对象，采用一个描述符来记录一个分段的信息。保护方式下的物理地址形成的方式却与 16 位 CPU 完全不同。为方便大家的理解，在介绍保护方式下物理地址的形成方式之前，先介绍相关的知识。

1．特权级

x86 在保护方式下建立了 4 个特权级，特权级由高到低分别为 0 级、1 级、2 级和 3 级。程序中的每个段都有一个特权级。在任何时候，CPU 总是在一个特权级上运行，称当前特权级，该特权级一般与 CPU 正在执行的代码段的特权级相同。x86 的保护规则是：被访问段的特权级应等于或低于当前特权级。例如，某操作系统的数据在特权级为 0 的段中，那么在特权级 3 上运行的应用程序就不能访问该数据，否则系统会出现一个保护异常，防止了用户程序对操作系统的非法访问和干扰破坏，使操作系统受到保护。系统中不同程序的特权级分配如图 3.7 所示。

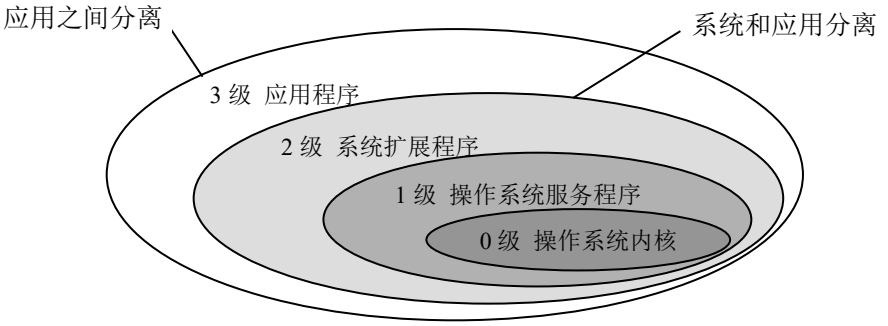


图 3.7　系统特权级分配示意图

2．描述符

在保护方式下，有关段的信息远比实地址方式要复杂得多。一个分段作为被保护的对象，需要知道段从什么位置开始的（段基地址）、段到什么位置结束，

换言之段有多大（段界限），此外还有段的类型、特权级、是否被执行过、是否能被读写等。x86 将它们集中在一起，用 4 个字来记录，这就是描述符。

根据段的用途可以分成存储段、系统段和控制段。存储段就是一般应用程序的代码段、数据段；系统段用于实现存储管理机制；控制段主要用于任务切换、特权变换、中断异常处理等。在保护方式下每一个段都有一个描述符。按不同的描述对象可分为：存储段描述符、系统段描述符和控制描述符。描述符的通用结构如图 3.8 所示。

图 3.8　描述符的通用结构

从图 3.8 中可看出，一个段描述符指出了段的 32 位段基地址（如果不使用分页管理机制，则该基地址就是段在主存中的物理地址）和 20 位的段界限（段长度）。

段描述符中的第三个字的高字节描述了段的性质及当前使用情况。

P：　存在位（第 15 位）。P＝1，说明该描述符所对应的段已在主存中；P＝0，该描述符所对应的段还在磁盘上未读入，此时，使用该描述符转换的地址无效且会引起异常。

DPL：　即 Descriptor Privilege Level，记录了该描述符所对应的段的特权级（第 14—13 位）。

S：　记录段的类型（第 12 位）。S＝1，用户程序的代码段、数据段或堆栈段的描述符；S＝0，系统段的描述符。

TYPE：记录存储段的具体属性。共有 3 位（第 11—9 位）。其中，第 11 位 E 描述了该段是否为可执行段。E＝0 表示该段为不可执行段，是数据段或堆栈段；E＝1 表示该段为可执行段，即为代码段。在两种不同类型的段时，另两位（第 10—9 位）所描述的内容是不同的，具体内容请见表 3.3。

表 3.3 TYPE 中的各位描述的内容

TYPE	第 11 位 E		
	E＝0 时		E＝1 时
	所描述的段为数据段或堆栈段		所描述的段为代码段
第 10 位	0	所描述的段为数据段	该段不可供特权级≦PDL 的程序调用或转入
	1	所描述的段为堆栈段	该段可供特权级≦PDL 的程序调用或转入
第 9 位	0	该数据段不能写	该段只能执行不能读出
	1	该数据段或堆栈段能读写	该段既能执行又能读出

A：　已访问位（第 8 位）。A＝0 说明该段未被访问过；A＝1 说明该段已被访问过，此时，选择符已被装入段寄存器中。该位的设立方便了系统对段使用情况的监控。

第四个字中的 7－4 位所描述的信息有：

G：　粒度位（第 7 位）。G＝0 时说明段长度的计量单位为字节；G＝1 时说明段长度的计量单位为页，1 页为 4K 字节。

D：　在代码段中，D＝0 说明是使用 16 位操作数和 16 位有效地址；D＝1 说明是使用 32 位操作数和 32 位有效地址。

在数据段中，D＝0 说明堆栈用 SP 作指针，且界限值为 0FFFFH；D＝1 说明堆栈用 ESP 作指针，且界限值为 0FFFFFFFFH。

剩下的两位为保留位和系统专用位。

3. 描述符表

描述符表是描述符的集合。一个描述符表最大可为 64K，存放 8096 个描述符。x86 有三种描述符表：局部描述符表、全局描述符表和中断描述符表。

（1）局部描述符表

对于计算机每一个执行的程序，系统都为之建立一个局部描述符表（Local Descriptor Table，LDT），记录该程序中各段的有关信息。由于每个程序都有各自的 LDT，使用各自的代码和数据，加上各段特权级的检查、段长度的限制，从而实现各程序之间的隔离。

（2）全局描述符表

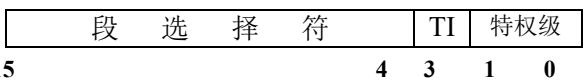
全局描述符表（Global Descriptor Table,GDT）包含着系统中所有任务使用的描述符。其中还包含着描述每一个局部描述符表的有关信息，如局部描述符表的起始基地址、表长度等。

（3）中断描述符表

中断描述符表（Interrupt Descriptor Table,IDT）含有指向多达 256 个中断服务程序位置的描述符。

4. 段选择符和描述符寄存器

在保护方式下，段寄存器保存的不再是段的开始地址，而是指出了从该任务描述符表中选择此段描述符的方式。这时，段寄存器中的内容为：



其中，TI 为该选择符所指示的描述符表的类型。如 TI＝0，表示要从全局描述符表中选择描述符；如 TI＝1，表示要从局部描述符表中选择描述符；特权级为请求访问该段的特权级别，用于特权级的检查；段选择符指出了该段描述符在描述符表中的位置，因为一个描述符表可存放 8096 个描述符，序号从 0—

8095,因此，段选择符要用 13 位二进制表示。

在保护方式下，为了真正支持多任务并运行大型程序，x86 采用了软硬件结合的虚拟存储器技术，虚拟存储空间可到 64TB，可谓海量存储空间。这时，程序员编写的程序仍然采用逻辑地址“段寄存器：偏移地址”，但在运行时，每个程序都存储在各自的虚拟存储空间中。由于只有主存中的程序和数据才能被访问，为了得到它们的物理地址，必须将它们所在的虚拟存储空间映射到物理空间，即根据段寄存器的选择符到描述符表中查找描述符。如果 CPU 每次都因此而访问主存中的描述符表，势必降低了系统的运行效率。为了解决这一问题，x86 为每一个段寄存器提供了一个对应的描述符高速缓冲寄存器。CPU 每当把一个段选择符装入到段寄存器后，就自动从描述符表中取出该段的描述符装入到对应的描述符高速缓冲寄存器中。此后，CPU 对段的访问均直接使用该寄存器中的描述符，而不用再通过总线接口部件访问主存中的描述符表，大大提高了系统的运行速度。由于描述符高速缓冲寄存器为用户不可见的寄存器，这里不再做详细介绍。

5．保护方式下物理地址的形成

在保护方式下，要形成物理地址应经过以下步骤：

1．根据段寄存器选择符值、TI 及 RPL 值，从局部描述符表中选出描述符，进行段长度是否溢出、特权级、使用合法性及各种相关属性的检查。如合格，则将描述符送入对应的描述符高速缓冲寄存器，以后对该段的访问均通过此寄存器进行；

2．当需要对该段的存储单元进行访问时，则从描述符高速缓冲寄存器中取出段基址，与存放在 EIP/ESP 或某一指示器中的偏移地址相加，形成 32 位的线性地址；

3．如不选择分页部件，则上面得到的线性地址即为物理地址；如果选择分页部件，则还应该经过分页部件的映射，将线性地址转换为物理地址。

以上步骤可用图 3.9 描述。

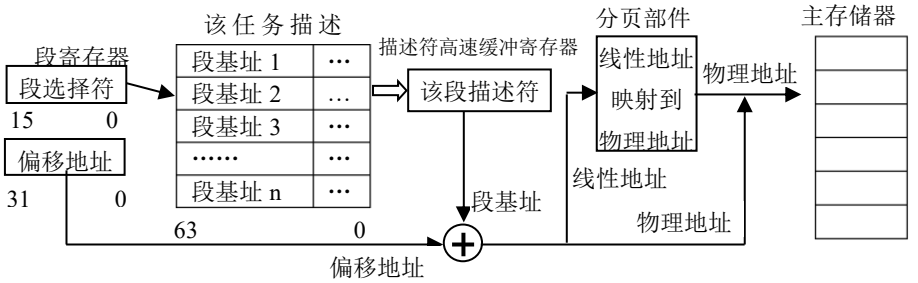


图 3.9 保护方式下物理地址的形成

但对使用汇编语言的编程者来说，实方式和保护方式的逻辑地址表达并无多大差别，因为用户既不用处理多任务，也不用关心程序映射到哪一片物理存

储区，这都是由操作系统进行管理的。因此在后面的学习中，我们都将以二维的逻辑地址“段寄存器:偏移地址”来代替存储单元的地址，而不再讨论具体的物理地址。

提示：计算机 CPU 如何对内存访问控制，如何在地址总线上加载要访问单元的物理地址，内存又是如何对地址进行译码来选中被访问的单元，以及如何将数据传送到数据总线上或者从数据总线上得到数据，都不是本门课程所关心的内容。这些功能的实现依赖于计算机中的硬件，在计算机组成原理、微机原理等课程中会进行介绍。从汇编语言学习的角度，只需要知道给定一个内存单元的物理地址就可以访问对应的存储单元中的数据，要访问一个存储单元也必须知道其物理地址即可。

习题 3

3.1 内存的最小寻址单位是什么？
3.2 内存的物理地址编址有何规律？
3.3 内存中一个字数据的物理地址是什么？双字数据的物理地址又是什么？
3.4 字数据和双字数据在内存中是如何存放的？
3.5 32 根地址总线对应的最大内存容量是多少？
3.6 实方式下，物理地址是如何形成的？
3.7 内存单元的逻辑地址由什么组成？
3.8 实方式下，设(DS)=1234H, 该段中有一个变量的偏移地址是 0012H，该单元的物理地址是多少？
3.9 实方式下，设(DS)=1234H, (SS)=1235H, (SP)=0100H。SP 指向的单元相对于数据段而言，偏移地址是多少？
3.10保护方式下，内存分段模式下，物理地址是如何形成的？
3.11访问一个存储单元时，需要明确给定哪些信息？
3.12 什么是地址类型转换？什么是数据类型转换？
3.13 设有如下数据段，画出数据在内存中的存放示意图

```
.data
str1  db  0,1,2,3,4,5
str2  db  ‘012345’
numw  dw  10H, -10H
numdw dd  1234H, 11223344H
```

3.14 设有如下数据段，画出数据在内存中的存放示意图

```
.data
x    db  10H, 20H, -1, 5
```

```
len = $ - x
y    db  len dup(0)
z    dw  $ - x
```

3.15 设有如下数据段，画出数据在内存中的存放示意图

```
.data
x  db  0AH, 0DH,  'Good', 0
   dw  10, 20
y  db  0AH, 0DH,  'Hello', 0
   dw  10H, 20H
p  dd   x,  y   ; 假设 X 的地址是 009e4000H
```

3.16 设以下各数均为有符号数的补码表示，请比较它们的大小：

345H 与 0A987H（16 位数）；80H 与 41H（8 位数）；8000H 与 0A987H（16 位数）； 71H 与 41H（8 位数）

3.17 如果将以上各对数均看作无符号数，请再比较它们的大小。

3.18 将下列十进制数分别用非压缩的 BCD 码和压缩的 BCD 码表示，并画出它们在存储单元中的存放形式。

0985； 5678； 8123

3.19 在写程序时，使用符号常量比直接使用数值常量具有什么优势？

上机实践 3

3.1 设有如下 C 语言程序段

```
char a[15]="1234567890abc";
printf("%c %x\n", *(char *)(a+2), *(char *)(a+2));
printf("%d %x\n", *(short *)(a+2), *(short *)(a+2));
printf("%ld %x\n", *(int *)(a+2), *(int *)(a+2));
printf("%s\n", a+2);
*(short *)(a+4)= 16709;
printf("%s\n", a+2);
*(int *)(a+4) = 16709;
printf("%s\n", a + 2);
```

请用所学理论分析各语句执行后，程序显示和数组 a 中的变化。实验验证观察理论分析是否正确。

3.2 将习题 3.13 中的数据段放在一个汇编语言源程序中，编译链接生成执行程序后，进行调试，记录看到的数据段的存放结果，与习题 3.8 的结果进行比较，若不一致，请找出原因。

3.3 请指出上机题 3.2，看到的各个变量的地址是多少？分析各变量的地址之间有何规律。

3.4 有如下 C 语言程序，指出程序的运行结果是什么？在调试时，在内存窗口观察变量 x 和 y 中存储的信息是什么？

```
#include <stdio.h>
int main(int argc, char* argv[])
{
    int x = -1;
    unsigned int y = -1;
    if (x > 0)  printf("x is positive\n");
    if (y > 0) printf("y is positive\n");
    return 0;
}
```

3.5 执行如下的程序，解释看到的运行结果。在内存窗口，观察变量 x,y,z 中存放的数据，指出三个变量中分别存放的具体值。

```
#include <stdio.h>
int main(int argc, char* argv[])
{
    float x = 1.25
    int y,z;
    y = *(int *)&x;
    z=x;
    printf("%d  %d   %x  %x \n",y,z,y,z);
    return 0;
}
```