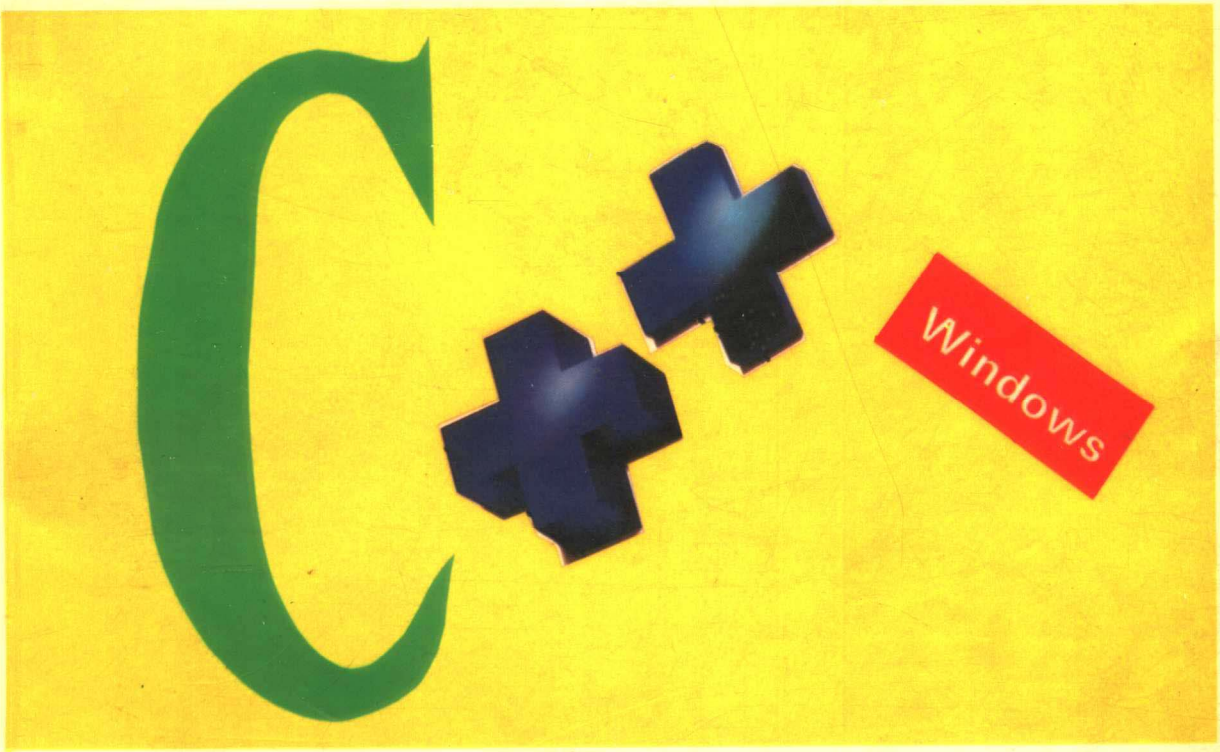
计算机C/C++语言系列丛书

蔡明志　　编著



BORLAND C++ for Windows

高级Windows编程

学苑出版社

计算机 C/C++系列丛书

Borland C++ for Windows

高级 Windows 编程

蔡明志　编著
施小龙
葛玉宝　改编
邓明辉
万　博　审校

学苑出版社
1994 年. 北京

(京)新登字 151 号

内 容 简 介

本书主要是针对利用 Borland C++ 3.1 进行 Windows 程序设计的人编写的，介绍了进行 Windows 程序设计的高级技术。本书共分六章，前二章触及内存的问题，并以实例配合深入探讨，使读者对内存能够充分地了解并详加利用；接下的四章皆为数据交换的方法，分别为剪贴板、动态连接函数库和多文档界面及动态数据交换，这些皆是本书重要的部分，在本文中有详细的介绍。

欲购本书的用户可直接与北京 8721 信箱联系，电话 2562329，邮编 100080。

声 明

计算机 C/C++语言系列丛书

Borland C++ for Windows
高级 Windows 编程

原　　著：蔡明志
改　　编：施小龙　葛玉宝　邓明辉
审　　校：万　博
责任编辑：甄国宪
排　　版：万博图书创作社
出版发行：学苑出版社　邮政编码：100036
社　　址：北京市海淀区万寿路西街 11 号
印　　刷：双青印刷厂印刷
开　　本：787×1092　1/16
印　　张：24.00　　字数：611 千字
印　　数：1～5000 册
版　　次：1994 年 6 月第 1 版第 1 次
本册定价：44.00 元
ISBN7-5077-0875-6/TP・24

原　序

本书是笔者进行 Windows 程序设计系列丛书中的一本，它属于高级篇，意旨说明先有初级的 Windows 程序设计概念之后，再研读本书将会有更好的效果，而本系列书的第一本基础篇及第二本绘图篇将是导引您进入 Windows 程序设计的最佳书籍。

本书共分六章，前二章触及内存的问题，并以实例配合深入探讨，使读者对内存能够充分地了解并详加利用；接下的四章皆为数据交换的方法，分别为剪贴板，动态连接函数库，多文档界面及动态数据交换，这些皆是本书重要的部分，在本文中有详细的介绍。

笔者深深觉得电脑进步实在太快了，如您不加以努力充实自己，可能在某些人的谈话中，您是一位无声者。Windows 的应用程序绝对是您现在必需学习的事情，朋友们，何不放下一切，加把劲，让我们共创美好的前程。

蔡明志

目　录

第一章　Windows 的内存管理结构与规则

1.1　80x86 微处理器的内存规则与支持 …… 1
1.1.1　物理地址空间 …… 2
1.1.2　段式的内存管理方法 …… 2
1.1.3　逻辑地址空间 …… 3
1.1.4　保护模式与内存寻址 …… 5
1.1.5　虚拟内存的使用 …… 6
1.2　Windows 基本的内存结构 …… 7
1.2.1　固定的段(FIXED) …… 8
1.2.2　可移动的段(MOVEABLE) …… 8
1.2.3　可抛弃的段(DISCARDABLE) …… 9
1.3　Windows 的动态数据分配空间 …… 10
1.3.1　全局堆 …… 10
1.3.2　局部堆 …… 11
1.4　Windows 的三种操作模式 …… 11
1.4.1　实模式(real mode) …… 12
1.4.2　标准模式(standard mode) …… 12
1.4.3　386 增强模式(386 enhanced mode) …… 13
1.5　程序段与数据段 …… 14
1.5.1　编译程序使用的内存模型(MEMORY MODEL) …… 15
1.5.2　程序段的基本概念 …… 16
1.5.3　数据段的使用 …… 16
1.5.4　数据段的内容 …… 17
1.5.5　多重代码段 …… 20
1.5.6　各种段的属性设置 …… 21
1.6　Windows 提供的系统模块 …… 22
1.6.1　KERNEL.EXE 的内存需求 …… 22
1.6.2　GDI.EXE 内存需求 …… 23
1.6.3　USER.EXE 的内存需求 …… 23

第二章　灵活运用 Windows 提供的内存空间

2.1　全局堆的动态分配 …… 26
2.1.1　动态分配全局堆的策略 …… 26
2.1.2　全局堆相关的 API 函数与分配块标志的设置 …… 26

2.1.3 如何动态分配 Windows 的全局堆 …… 28
2.1.4 锁住与开启全局堆中的内存块 …… 30
2.1.5 重新分配全局堆中内存块的大小 …… 31
2.1.6 释放与抛弃全局堆中的内存块 …… 32
2.1.7 如何运用其他的全局堆 API 函数 …… 33
2.1.8 全局堆的高层应用 …… 36
2.2 局部堆的动态分配 …… 50
2.2.1 动态分配局部堆的策略 …… 50
2.2.2 局部堆相关的 API 函数与分配块标志的设置 …… 51
2.2.3 如何动态分配数据段内的局部堆 …… 51
2.2.4 重新分配局部堆中内存块的大小 …… 52
2.2.5 使用其他重要的局部堆 API 函数 …… 53
2.2.6 使用额外局部堆的方法 …… 54
2.3 资源与内存空间使用的关系 …… 83
2.3.1 系统定义的资源对象 …… 83
2.3.2 用户自定义的资源 …… 84
2.4 额外字节 …… 97
2.5 特性列表 …… 110
2.6 原子表的应用 …… 122
2.7 本章综述 …… 148

第三章 剪贴板的使用方法

3.1 剪贴板使用的数据格式 …… 151
3.2 文本数据与剪贴板 …… 151
3.2.1 剪下或复制文本数据到剪贴板 …… 152
3.2.2 粘贴文本数据到应用程序 …… 153
3.3 图形数据与剪贴板 …… 162
3.4 虚拟文件图与剪贴板 …… 172
3.4.1 剪下或复制虚拟文件图到剪贴板 …… 173
3.4.2 粘贴虚拟文件图到应用程序 …… 174
3.5 剪贴板的高层使用技巧 …… 182
3.5.1 数据格式多样化 …… 183
3.5.2 延迟提供数据给剪贴板的时间 …… 184
3.5.3 使用自定义的剪贴板格式 …… 185
3.5.4 剪贴板内容显示程序 …… 187

第四章 动态连接函数库

4.1 什么是动态连接函数库? …… 196
4.1.1 静态连接与动态连接 …… 196

4.1.2 动态连接函数库与一般应用程序间的差异 …… 198
4.1.3 目标函数库与导入函数库 …… 199
4.1.4 Windows 如何找到动态连接函数库 …… 199
4.1.5 使用动态连接函数库的优点 …… 200
4.1.6 查看动态连接函数库的内容 …… 200
4.2 自行设计动态连接函数库 …… 201
4.2.1 动态连接函数库的入口 …… 202
4.2.2 LibMain()函数 …… 203
4.2.3 WEP()函数 …… 204
4.2.4 动态连接函数库的模块定义文件 …… 205
4.3 动态连接函数库的调用方式 …… 212
4.3.1 输入动态连接函数库中的函数的方法 …… 212
4.3.2 远程函数调用 …… 213
4.3.3 DS! =SS 的衍生问题 …… 214
4.3.4 动态连接函数库的重要限制 …… 216
4.4 动态连接的高层技巧 …… 230
4.4.1 回调函数的运用 …… 230
4.4.2 应用程序指定链结的方法 …… 232
4.4.3 善用输入函数库 …… 233
4.5 动态连接函数库的使用扩展 …… 249
4.5.1 无需事先输入函数的动态连接方式 …… 249
4.5.2 仅含资源的动态连接函数库 …… 250
4.5.3 内存对象的归属权 …… 251
4.6 其他需要运用动态连接函数库的时机 …… 257
4.6.1 挂接与动态连接函数库 …… 257
4.6.2 驱动程序与动态连接函数库 …… 258
4.6.3 用户自定义的子控制 …… 259

第五章 多重文件界面

5.1 MDI 应用程序的基本结构 …… 273
5.2 MDI 应用程序的操作标准 …… 274
5.3 MDI 应用程序启动时的初始化过程 …… 276
5.3.1 登记窗口类 …… 276
5.3.2 创建窗口 …… 276
5.4 如何编写 MDI 程序的消息循环 …… 277
5.5 编写窗口函数的要领 …… 278
5.5.1 框架窗口函数 …… 278
5.5.2 子窗口函数 …… 278
5.6 子窗口与文件数据的连接 …… 278

5.6.1 窗口额外字节与 MDI 子窗口 …… 279
5.6.2 特性串列与 MDI 子窗口 …… 279
5.7 MDI 子窗口 …… 279
5.7.1 创建 MDI 子窗口 …… 280
5.7.2 破坏 MDI 子窗口 …… 281
5.7.3 MDI 子窗口工作状态的切换 …… 281

第六章 动态数据交换

6.1 DDE 概念简介 …… 307
6.2 DDE 的组成元素 …… 310
6.2.1 创建会话时的三大元素 …… 310
6.2.2 组成消息的基本要素 …… 310
6.3 DDE 的通讯类型 …… 312
6.3.1 冷式链结 …… 312
6.3.2 温式链结 …… 313
6.3.3 热式链结 …… 314
6.4 DDE 消息汇总 …… 316
6.5 创建会话 …… 316
6.6 如何向服务器程序请求数据 …… 319
6.6.1 WM_DDE_REQUEST 消息 …… 319
6.6.2 WM_DDE_DATA 消息 …… 319
6.6.3 WM_DDE_ACK 消息 …… 324
6.7 传送项目数据给服务器程序 …… 325
6.8 传送命令消息给服务器程序 …… 327

第一章 Windows 的内存管理结构与规则

从习以为常的 DOS 操作系统,“移植”到 Windwos 这个全然陌生的用户接口开发程序,想必各位程序员都有很充分的理由。然而人同此心,我想所持的理由不外乎下面这几点:

1. 老板的要求与上司的期望。
2. 时势所趋。
3. 在图形用户接口下运行的程序对用户比较具有吸引力。

.
.
.

可能的原因有很多,我想 PC 上的用户大概不会漏掉一项最重要的理由:DOS 640KB 的内存的限制!

或许是时代进步得太快,当初 IBM 公司设计 PC 时,本以为 640KB 的内存已够使用,却没有想到仅在短短的数年间,640KB 的内存限制已经变成程序设计者在 MS - DOS 操作系统下工作时挥之不去的恶梦。但是 Windows 3.0 这个成熟版本的推出,却指引出另外一条道路!虽然它在内存方面的支持并非毫无限制,但是比起过去的 DOS,真可说有天壤之别。现代程序的发展趋势,渐渐走向规模扩大、功能多样化的方向。因此,本书在介绍提供的各项进阶性的功能以前,有必要请大家先行了解 PC 及 Windows 上内存功能管理的来龙去脉,日后在 Windows 操作系统下设计和开发系统时,得以妥善运用内存这个重要的资源。

本章的内容大致可以区分为下列这几个主题:

- 80x86 微处理器的内存设计与支持
- Windows 的基本内存结构
- Windows 的动态数据空间分配
- Windows 的三种工作模式
- 程序段与数据段
- Windows 提供的系统模块

下面我们会分成六个小节分别讨论,有了这些观念作为基础,下一章即可将这些内容实际运用在程序上。

1.1 80x86 微处理器的内存规则与支持

Intel 公司生产的 80x86 家族系列处理器,第一项得到市场认可的产品应该是 8088,然后随着电子科技的突飞猛进,连续推出功能更强大的微处理器。例如在 80286 上面添加了保护模式(protected mode),提供另外一种较具扩充性的方法来处理内存寻址的问题。接下来的 80386 微处理器,更在其中添加了分页寄存器(page register)等硬件,使得 80386 以上的微处理器更进一步支持操作系统进行虚拟内存的管理任务,将磁盘空间视为物理内存(Physical memory)的延伸!这正是 Intel 公司的发展策略:设计更新一代的微处理器时,都保

留了某种方法延续上一代的设计，将过去的功能移植到另具新功能的微处理器上。到目前为止，即使 Intel 推出功能更新更强的微处理器，仍然会保有原先的功能，使得过去开发的程序依然能够在新的微处理器上面照常运行。

本小节会从各种角度讨论 Intel 家族微处理器与内存的关系，以及保护模式和虚拟内存的使用。这些背景知识对于开发大型的程序颇有益处。

1.1.1 物理地址空间

Intel 80x86 家族系列的微处理器，随着产品系列号的增加，从 8088，80286，80386 到 80486 等等，除了运行速度加快，寄存器位数加大，其寻址能力亦有所区别。什么叫做寻址能力呢？就是微处理器所能够访问的最大内存容量，也就是可供程序使用地址空间的大小。例如 8088 有 20 条地址线，即表示代有 $2^{20}=1$ Mega 字节的地址空间可供程序使用。同样的道理 80386 这个拥有 32 条地址线的微处理器，则最多可以寻址达 $2^{32}=4$Giga 字节的内存空间！

或许大家看到这里想到一个挥之不去的疑问：既然可以寻址的空间那么大，到底这 640KB 的空间限制从何而来呢？这都是向下兼容惹的祸！由于过去的应用程序在 DOS 操作系统下开展，如果考虑到要让后期开展的应用系统能够在早期的旧型机器上运行，受限于 8088 微处理器只支持 640KB 的内存空间供程序运用，问题随之而来：操作系统的供应厂商到底要不要照顾过去的客户，还是任其自生自灭？这就牵涉到 Microsoft 公司的行销策略。不过 Windows 却以另一种方式，不仅照顾到过去的用户，在硬件环境的支持存在时，也能够以较高效率的方式来运行程序。因此，Windows 的重要性与日俱增是不言自明的。

1.1.2 段式的内存管理方法

Intel 公司制造的 80x86 家族系列微处理器，他们管理内存是采用段式的分配方法，这也是计算机结构直接影响到系统设计方向的实例。在 PC 上编写过汇编语言程序的人一定都晓得，无论程序取用数据段中的数据或者微处理器读取内存中的指令，其寻址方式皆为

```
        CX  :  offset
或      DX  :  offset
        ↑      ↑
    段地址    偏移量(又称"位移")
```

这就是所谓的"段式"内存管理访问内存值的标准方法。如果各位并不熟悉，在此我们会做一番详细的介绍，并且告诉各位 Windows 如何运用这种内存管理方式来分配内存。

何谓段式的内存管理呢？其实在日常生活中这种两段式的寻址方法运用极广。假如你家住在忠孝西路 385 号，那么可以写做

```
    忠孝西路  :  385
      ↑          ↑
    段地址      偏移量
```

忠孝西路就好比是段地址，而 385 就好像是先前所提到的偏移量，两者组合后形成完整的地址，表示内存内一个固定的地方，这样的概念应该不难理解吧！

那么实际上 PC 的内存管理程序如何运用这种结构，又有什么特殊的限制呢？请看下面

这张图。

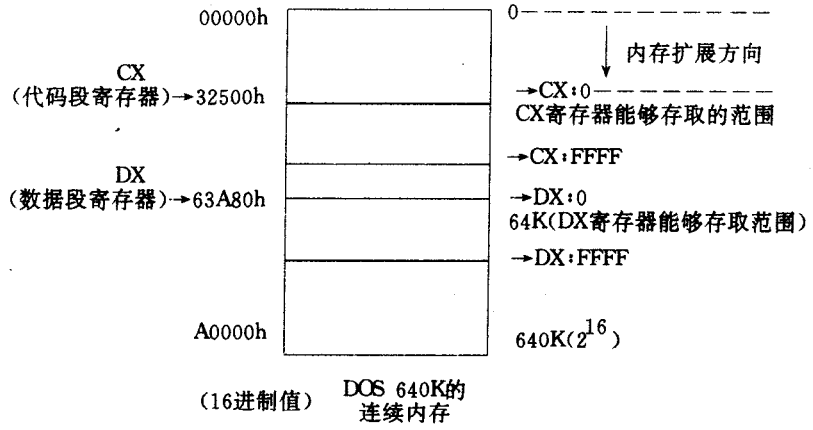


图 1.1 段式内存的寻址方法

PC 上 640KB 的连续内存,利用段式内存管理方法,系统可以将内存分割为若干段,每个段都拥有 64K = 65536 个"小房子",或者称之为字节。如此一来,程序就可以利用此种较为便捷的寻址方法来处理繁琐的物理地址寻址问题,而不需要在写程序的时候,为地址的运算而大伤脑筋。

段式内存管理方式最大的一项优点,就是运行程序的可移植性。这种设计概念使得程序只需要从段的观点来看待内存,而不必在乎运行的机器到底是 8 位,16 位,还是 32 位。如果从另外一个角度来看,只要操作系统支持段式内存管理方法,那么无论是 Windows 还是 OS/2,都能够运行原先在 DOS 下运行的应用程序!

段式内存管理方法带来的另外一项好处,就是将运行程序与数据段模块化,有助于保持程序内部结构的独立性,减少发生错误的机会。怎么说呢?因为程序可以分割成许多个程序段以及数据段,就好象是在内存中书划分若干段落将他们隔离开来,所以程序误用或者修改到其他段中内容之类的错误就可以避免,使得调试的任务变得更为容易。80286 以上的微处理器提供保护模式的意义,就是微处理器不允许某个程序去读取或写超过预先定义的内存段结束处以外的部分,或者产生了不合法的段引用(segment reference),进而达到保证其他程序的目的。

最后一个与段式内存管理相关课题,和管理内存有着非常密切的关系。Windows 具备在运行程序时允许程序段可以在需要时才装入到内存里面的功能(load on call),因而并不需要把整个程序都放到内存内,如此可以使得内存的运用更高效。尤其当剩余可用的内存空间不够大时,这种段式的内存管理更明显地发挥了效率。只要程序员将程序代码段适当予以调整,把同一时段可能运行到的程序代码尽量放在相同的段内,如此内存与磁盘间的切换(swap)动作就会减少,使得程序的运行效率不会降低到让人无法忍受的地步。

1.1.3 逻辑地址空间

物理地址空间固然限制住机器运行时真正可取用内存的大小,不过,程序设计者只需要考虑微处理器定义的逻辑地址空间,就可以将物理地址空间置于脑后。如此不仅比较方便,便于程序员的运行,同时也增加了编程的弹性。至于如何将逻辑地址转换为实际地址则是微

处理器的任务，与程序员无关。下面以图例来说明这种地址转换过程。

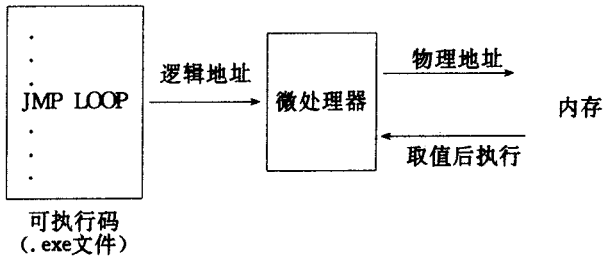


图 1.2 逻辑地址与物理地址的转换

目前 Windwos 提供三种工作模式，对内存的管理则因为所限定的硬件要求层次不同而有所差别歧视，稍后我们会逐一讨论。但是不论是那一种模式，其基本结构完全相同：处理由一对段地址及偏移量组面的“两段式”逻辑地址。

段地址标识出程序要处理的内存段，其大小为 16 位。至于该值应于实体地址空间的哪一段，程序设计者并不需要担心，完全由操作系统负责。在实模式(real mode)下，它一定位于实体地址范围之内；而在 386 增强模式(386 enhancde mode)下，却有可能指向虚拟内存中已经被移到磁盘上的段，这就是操作系统的功能：将另外一种截然不同的虚拟机器展现在用户的眼前。

究竟逻辑地址如何转换成物理地址呢？我们拿最基本的实模式为例来说明。在实模式下，逻辑地址其实就是物理地址。程序运行时或者要取用内存上某个地址值时，便将段地址以及段地址同时送给微处理器，微处理器会将段地址向左移(shift)四位后，再和段地址相加，求得转换后的物理地址。图 1.3 便说明了这种过程。

上面这是地址转换方法中最简单的例子。实际上微处理器内的地址转换过程可能相当地繁琐，如前所述，这和工作模式息息相关。例如在保护模式下，段地址不再直接和物理地址相对应，通过查询段转换表(segment table)才能决定对应的物理地址。在 386 以上的微处理器中，又添加了特殊的硬件，除了支持内存段的功能以外，同时还可以运行分页(paging)的功能，便于虚拟内存的管理。最后要提醒大家，不管在那种模式下执行，程序员所写的程序和使用到的数据，运行时在内存中是分成一个又一个各自独立的段，这是程序员必须谨记在心的。

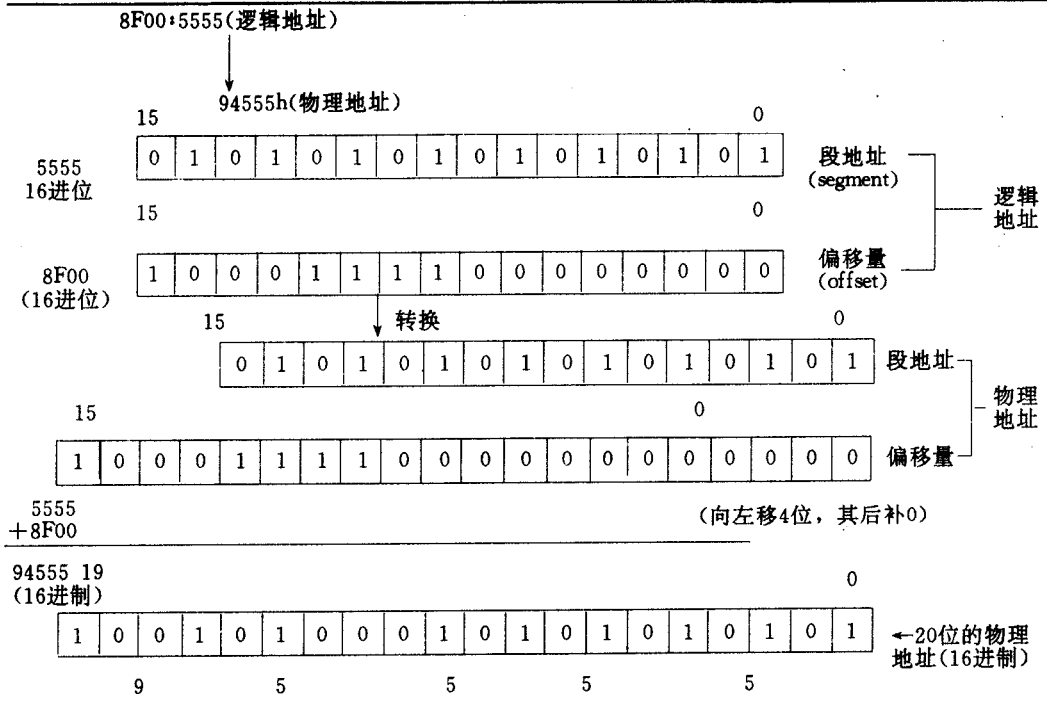


图 1.3 真实模式下逻辑地址转换为实体地址的过程

1.1.4 保护模式与内存寻址

保护模式指的是 80286 以上微处理器的一种特殊的运行模式，这种模式规定了内存寻址的顺序。如果程序企图使用不属于它自己的数据段，系统会予以阻止。由于在实模式下 DOS 仅采用单任务执行，所以允许驻留程序等拦截正常中断矢量的程序保存在内存中。但是保护模式仍是针对多任务执行而设计，为了避免行为不端的程序导致系统死机，系统提供适当的保护机制是必要的。

保护模式下的寻址，依然保存了实模式中段地址加上偏移量的两段式寻址法，但地址的计算并不只是单纯的移位后再相加而已。微处理器提供了寻址时使用的描述表(descriptor table)，这个表由操作系统创建及维护。描述表可以看作由特殊定义的数据结构所组成的数组。这个结构中的字段主要就是用来描述所指向段与寻址相关的信息。图 1.4 说明了保护模式下微处理器究竟如何处理内存寻址。

(1) 将地址送入微处理器。

(2) 微处理器利用段地址值做为索引，到描述表中查出对应的段消息。

(3) 取出消息字段中此段在内存中实际的起始地址做为基地址。

(4) 最后加上偏移量这个位移值，寻址的过完成。

所以，在保护模式中，段地址并不是代表真正的内存地址，而是一个索引值，根据该值间接查到真正的地址。不过请注意，若要取用 640KB 以上的内存，请务必要在 CONFIG. SYS 配置文件中加上 HIMEM. SYS 这个驱动程序，因为保护模式接口(DOS Protected Mode InterfaceDPMI)的支持代码都放在其中。

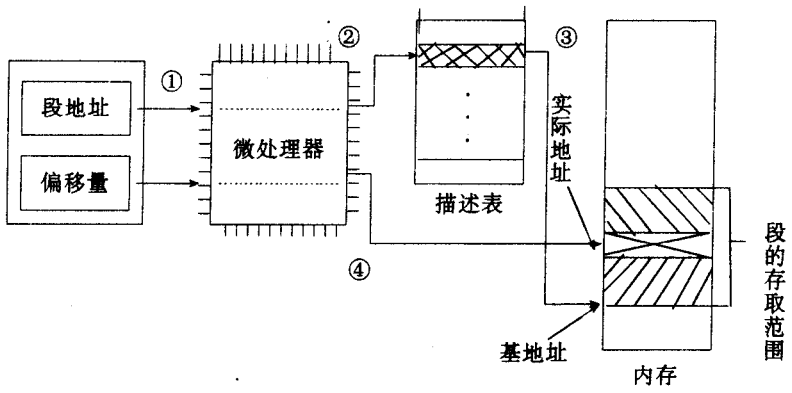


图 1.4 保护模式内存寻址的步骤

那么保护模式这种寻址方式为 Windows 的执行带来什么样的好处呢?

1. 微处理器提供内存管理的硬件,运行时比较有效率。

2. 由于程序在保护模式中运行时所访问的地址并非真实的地址,所以可以依其需求更改描述表中字段的值,达到段迁移的目的。因为程序访问的逻辑地址虽然没有改变,但是描述表中对应逻辑地址的段地址的字段值却是可以调整的。只要改变描述表内的值,就可以将段在内存中存放的位置进行移动"。

3. 最值得注意的是,关于固定式的内存对象,在实模式下是绝对无法移动的,但是在保护模式中却非如此。为什么呢? 因为在实模式中,逻辑地址就是物理地址,自然没有什么花样可变;而保护模式中,虽然段的逻辑地址不可改变,但是只要修改描述表中的对应值,就能够达到移动的目的。

1.1.5 虚拟内存的使用

先前讨论有关保护模式下的各种操作特性也适用于标准模式与 386 增强模式。唯独虚拟内存不然,这项重要的功能也是这两种模式执行的差别,关键即在于 386 以上的微处理器多了分页(paging)的能力。增加了这种能力以后,微处理器解释段地址(基址)加上偏移量(位移)的方法便有所不同。在一般的保护模式中,经由查表后所得到的值就是实际的内存地址;但是在 386 增强模式中,联机于微处理器的分页寄存器得以发挥功能之后,我们却要将此值视为虚拟内存的地址。究竟微处理器怎么做到的呢? 用图例配合说明仍是最好的办法。

与图 1.4 相比较,大家应该很快就可以看出两种寻址方式的差别。在虚拟内存的寻址过程中,逻辑地址经由描述表的对应运算后所得到的值只是虚拟内存中的虚拟地址,然后微处理器再利用分页寄存器中的值查出该地址所在页对应于物理内存的位置。如果此时该页已经切换到磁盘上,则硬件会产生一读页错误(page fault)的中断,请操作系统到物理内存中找出一页的空间,将已经被切换到磁盘中的页重新读入物理内存,寻址的任务才能够进行,这就是虚拟内存寻址的流程。

支持虚拟内存最大的好处就是可以扩大程序可以寻址的空间,由物理内存延伸到磁盘。虽然手续可能稍加繁琐,但是由于大部分的动作都是在微处理器内以极快速的硬件进行运算,所以对效率方面的影响并不大。站在程序员的观点,我们只要将虚拟内存视为程序直接面对的内存空间即可,而不需要理会它和物理内存之间的映射关系,操作系统会主动代我们

处理各种相关的琐事。因此，可供程序使用的空间变成一长条连续的内存，不再直接受到物理内存大小的限制！

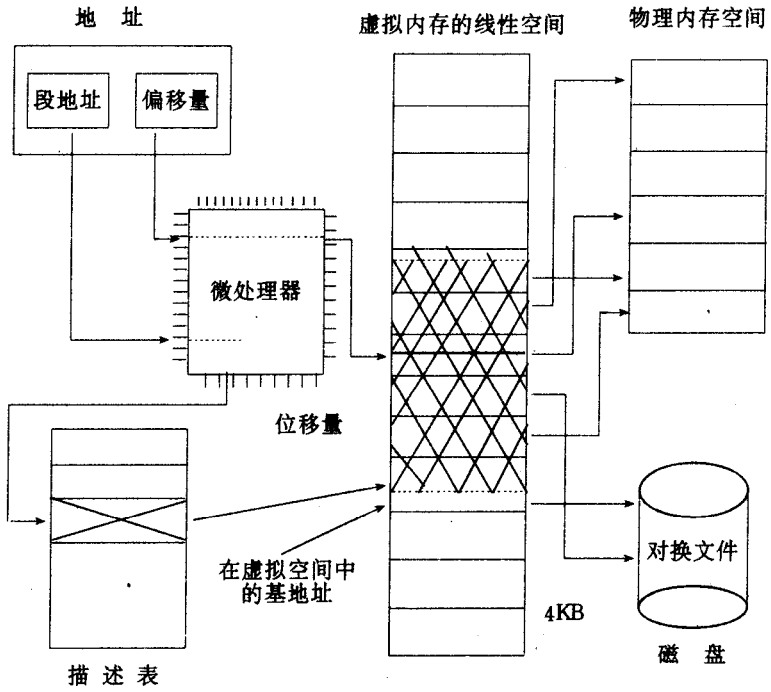


图 1.5 虚拟内存中的寻址流程

由于各位只需要面对虚拟内存这块连续的空间，因此，先前所提到的段式内存管理的对象就是虚拟内存。系统在其上自由地分配空间，移动段，各位都不必操心。只要将它想像成实际的内存空间，同一段的程序在虚拟空间的位置必须是连续的，段的各种限制依然存在。至于段中哪一页映射到实体内存或磁盘中的哪一个部分，根本不用多操心，此刻的实体内存仿佛被分割成一块块 4KB 的内存空间，由操作系统处理。由于段的概念已经使用在虚拟空间上，所以物理内存空间中已无所谓的段，而只有用来对应的“页”了。

倘若和先前 286 简单的保护模式相比较，虚拟内存的引进虽然增添了操作系统的负担，但是由于逻辑地址不再需要直接映射到物理内存，所以我们的程序也不会直接受到物理内存容量的限制。这就是现代操作系统的宗旨：以虚拟方式延伸我们的程序空间！

1.2 Windows 基本的内存结构

在 DOS 下启动 Windows 以后，其他所有尚未使用的内存空间由 Windows 管辖，这些内存称之为全局堆(global heap)。运行时，每一块分配的内存块都称之为一个段，其他尚未分配的内存空间则称为剩余空间(free memory)。站在系统的观点，全局堆就是系统唯一可以使用的内存资源。

在装入应用程序到内存之前，除了把需要的程序段和数据段分配给应用程序以外，另外还得额外从全局堆里面分配两个段提供为系统给内部使用。其中一个段用来存入.EXE 文件中的文件头信息，因为相同应用程序的所有运行实例(instance)都共用这个段，所以只需

在首次启动时分配一次即可。另外一个段则用来存入各个运行实例私有的用于运行的信息，例如工作目录、命令行字符串等。至于程序运行时装入的各种资源，例如对话框、光标、图标等，每一个资源都会在全局堆内分配到一个独立的段。

为什么程序中的窗口程序声明时都要加上 FAR PASCAL 这样的关键字呢？PASCAL 指的是系统所采用的调用协议，借以提示运行函数调用时的效率。而 FAR 则表示此函数必须提供远程调用的能力，这就和所采用的内存结构有着密不可分的关系。一般的函数则采用缺省的近程调用的方式(NEAR)，因为调用与被调用的函数位于相同的段里面，并且使用相同的 CS 程序段寄存器值，故采用近程调用即可。然而 Windows 系统本身和应用程序当然不在相同的段内，必须采用跨段的远程调用，同时提供段地址与偏移量指出函数实际的地址，所以要声明为 FAR(近程调用只需要提供内地址)。应用程序如果使用到多个程序段，函数间调用的情况也是一样的。

如果从 Windows 的观点来看待内存空间，则它是由一大块的全局堆所组成；然而从应用程序的角度出发，则主要的工作范围只有程序段与数据段。当程序开始运行的时候，微处理器内的 CS 寄存器就会指向该程序缺省程序段内的第一个可执行指令。DS 及 SS 两个寄存器则指向该程序的数据段，其中 DS 主要用来访问静态以及全局变量，SS 则标识出堆栈增减的状况，熟悉 DOS 下应用程序工作方式的人应该对此并不陌生。关于 DS 和 SS 这两个寄存器的意义，我们在动态链结函数库那一章会再次讨论。

本节将就 Windows 内部对内存空间使用的三种分类方式 FIXED、MOVEABLE 以及 DISCARDABLE 做详细的介绍。

1.2.1 固定的段(FIXED)

由全局堆分配出去的每一个段都要指定使用的方式，固定的段(FIXED)是最传统的同时也是最没有效率的，因为标识为 FIXED 的段自始至终都占据着同一块内存空间。过去在 DOS 的单任务操作环境下，这种情况或许并不构成问题但是进入多任务环境后，固定的段使得 Windows 没有办法充分利用有限的空间，调整段在内存空间内的位置，满足其他程序对空间分配的需求。

当然，有些程序必须要放在固定的地址上才能任务，这也是保留 FIXED 这种段使用方式的目的。例如设备驱动程序等和中断向量密切相关，就必须要放在固定的地址。在此建议大家，如果不是其他的内存使用方式都没有办法满足程序的需求，千万不要将段设置为 FIXED。

1.2.2 可移动的段(MOVEABLE)

段的第二种使用方法是设置为可移动的。顾名思义，这种段可以任由 Windows 在全局堆内随意移动，从原来的地址搬到另外一个新的地址。所以，可移动的段地址是不固定的。

首先，我们来瞧瞧这样做对于 Windows 管理内存空间有着什么样的好处。其实道理很简单，假使我们有一个书橱零零散散地放了几本书。现在又买了一套大部头的书想放进去，但是书橱中并没有一块连续的空间可以放得下这套书。不过，如果书橱内的书能够调整摆设的位置，也许就可以挪出足够的空间摆下这套书。这就是可移动的段替 Windows 带来的好处—内存空间的运用更加游刃有余。

接下来，我们从技术观点来讨论如何解决段所造成的问题。同一段内的函数调用或者数据读取时，由于都以CS或DS寄存器作为相对位移的基址，所以并不会产生问题。至于跨段的函数调用，例如窗口程序(Windows Procedure)、回调函数(Call —— back Functions)、对话框函数等，Windows并不直接调用，而是利用MakeProcInstance()函数确定目前段的实际地址。不过只有在实模式下才一定需要这种解决方式。

段的使用方式并非绝对是一成不变的。如果有特殊的需求，可移动的段也可以暂时设置成固定的段。因为Windows分配可移动的段时，系统返回给应用程序的返回值并不是段的实际地址，而是代表该段句柄(handle)，对应于代码的段地址则会随着Windows移动段而改变。当用户要使用这个段时，必须首先将句柄转换为实际的地址。所以，锁定该数据段是首要任务，如此程序方能正确地取用其中的数据。段一经锁定以后，Windows暂时就没有办法进行搬移段的动作。等到该段的访问动作结束以后，请记住要重新开启锁住的数据段，此后请不要再继续使用任何指向该段数据段的指针。第二章我们会以程序实例来说明这种观念。

1.2.3 可抛弃的段(DISCARDABLE)

想要使得内存空间运用得更有效率，仅靠移动可移动的段是不够的。假使某些段的内容并不会改变，或者其中的内容很容易还原，那么请将这类内存块设置为可抛弃的。Windows运用最久未使用(Least Recently Used，LRU)算法来决定抛弃内存时的段选择问题。

可抛弃的段必须同时标识为MOVEABLE。Windows决定要抛弃某个段时，并非真正将它从Windows系统中拿走，而是将其内存大小重新分配成0，达到和抛弃相等的效果，当然，原先存储于其中的数据会全部毁坏，但是原来使用的句柄依然有效，所以设置段为DISCARDABLE时要特别留意。假使稍后用户企图锁住一块已经被抛弃的段，则Windows会返回NULL指针，告诉用户该段的内容已经不再有效，用户必须利用重新分配数据段的函数分配一块可用的空间，然后再读入该段被抛弃以前的内容。

什么时候Windows会打算抛弃设置成DISCARDABLE的段呢？正如前面所谈到书橱这个例子，如果把其他的书挤在一起后所挪出的空间仍不足以将大部头的书挤进去，那么只好看看有哪些书籍可以先拿出书橱外来多挪出些空间，直到挤得进去为止。另外一种状况是，有些书籍是不可以移动的，因此可能会找不出足够的空间，分配空间的动作自然变会失败。

通常哪些段需要声明为DISCARDABLE呢？第一类是只读性的段，例如程序段，因为稍后需要时可以再由磁盘中的执行文件读入，例如Windows中USER和GDI两个动态链结函数库内的函数就大半是可抛弃的。另一类可抛弃的对象就是资源(resource)，例如功能选择表、点阵图、图标等等。所以在程序中善用资源是内存运用的上上之策，因为不需要用到的资源随时都可以抛弃。

不知道大家过去是否有种疑惑，明明程序并没有进行取磁盘的动作，可是驱动器的灯亮了起来。其实这就是某些内存段随时可以被系统抛弃的缘故。程序中的某个段可能刚才在进行空间分配时被系统抛弃掉，所以现在必须重新读入，造成磁盘机的读写运转。