Microsoft Microsoft Microsoft

Microsoft® Win 32™ 程序员参考大全（二）

——系统服务、多媒体、系统扩展、应用程序须知

[美] Microsoft Corporation 著

欣力 李莉 陈维 李志峰 译

张燕 审校

清华大学出版社

Microsoft® Win32™程序员参考大全(二)

——系统服务、多媒体、系统扩展、应用程序须知

[美] Microsoft Corporation 著
欣力 李莉 陈维 李志峰 译
张 燕 审校

清华大学出版社

(京)新登字 158 号

Microsoft Win32 程序员参考大全(二)
——系统服务、多媒体、系统扩展、应用程序须知
Microsoft Win32 Programmer's Reference, (Volume 2)
——System Services, Multimedia, Extensions, and Application Notes
Microsoft Corporation

图书在版编目(CIP)数据

Microsoft Win32 程序员参考大全(二):系统服务、多媒体、系统扩展应用程序须知/欣力等译.—北京:清华大学出版社,1994.
ISBN 7-302-01684-4

Ⅰ.M… Ⅱ.欣… Ⅲ.操作系统(软件)-基本知识 Ⅳ.TP316

中国版本图书馆 CIP 数据核字(94)第 13727 号

出 版 者:清华大学出版社(北京清华大学校内,邮编 100084)
责任编辑:李幼哲
印 刷 者:清华大学印刷厂
发 行 者:新华书店总店北京科技发行所
开 本:787×1092 1/16 印张:44 字数:1023 千字
版 次:1995 年 4 月第 1 版 1995 年 4 月第 1 次印刷
书 号:ISBN 7-302-01684-4/TP·730
印 数:0001—4000
定 价:79.00 元

引　言

Microsoft Win32 应用程序编程接口(API)允许应用程序拓展 Microsoft Windosw 操作系统系列的 32 位功能。使用 Win32 API 的应用程序既可以在单个处理器也可以在多处理器系统上运行,并能移植到 RISC 结构上。这套手册是整个 Win32 API 的全面文档。包括窗口管理、图形、文件 I/O、线程、内存管理、安全性和网络。

本手册的组织

下面是对本手册主要部分的简明介绍。

第 3 部分"系统服务",描述应用程序用于管理 Windows 基本多任务服务的 Win32 API 部分。这部分中的章节提供了有关进程和线程、内存管理、网络、动态链接、安全性以及有关系统的其它信息。

第 4 部分"多媒体服务",描述 Win32 API 的多媒体部分。多媒体 API 支持音频、媒体控制,多媒体文件输入和输出(I/O)以及增强的定时器服务。

第 5 部分"扩展库",描述给 Win32 API 增加额外特点的库。这些特点包括公用对话框、简化动态数据交换(DDE)的管理函数、Shell 中增强的拖曳特点、文件安装函数以及数据解压缩函数。

第 6 部分"应用程序注释",描述应用程序可以用来实现某些 Windows 特点和增强功能的技术。手册的这部分解释如何建立 Control Panel 应用程序、如何建立和安装 File Manager 的扩展库,如何使用 Program Manger 的动态数据交换接口以及如何从 Win32 应用程序提供的多种进程间通信的方式中作出选择。

附录 A"错误码",列出并描述 Win32 API 函数在它们失败时返回的错误代码。

附录 B"虚拟键码",提供 Windows 虚拟键代码的符号常量名字,十六进制值和键盘对等键。

附录 C"地区标识符",列出并描述用于使应用程序适应国际市场的符号常量。

附录 D"制造厂商和产品标识符",列出当前 Microsoft Windows 操作系统中使用 Multimedia Extensions 的制造厂商和产品标识符。

附录 E"MCI 命令串语法总结",列出 Media Control Interface 命令串和它们的语法。

附录 F"光栅操作码",列出并描述由图形设备界面使用的二元和三元光栅操作。

本手册着重描述 Win32 API 目的以及介绍 API 背后的操作系统概念,并说明 Win32 函数是如何协作来完成指定任务的,但并不包括如何写、编译和连接包含这些函数的程序。

关于《Microsoft Win32 程序员参考大全》

《Microsoft Win32 程序员参考大全》共五卷,完整描述了 Win32 API,包括函数和相关的数据类型、宏、结构以及消息。《程序员参考大全》是有关 Windows 编程方面信息的权威。

卷(一)和卷(二)描述 Win32 API 中函数的目的并介绍这些函数的概念和原理。这两卷是为那些刚接触 Windows 或第一次学习这部分内容的程序员写的。这两卷为理解 Windows 编程提供了基本的信息。

卷(三)和卷(四)按字母顺序列出 Win32 函数。这两卷定义了语法、参数的细节和各个函数的返回值。卷五包含按字母顺序排列的 Win32 数据类型、宏、消息和结构。卷(三)到卷(五)是为那些已了解 Windows 编程只需要特定函数特定内容的人编写的。

Microsoft Windows 与 C 编程语言

C 编程语言是基于 Windows 应用程序的首选开发语言,Windows 的许多编程特性都是与 C 有关的。虽然基于 Windows 的应用程序也可以用其它语言开发,但 C 是最直接和最简单的访问 Windows 函数的语言。由于这个原因,所有的语法和程序范例都是用 C 编程语言写成的。

Win32 API 使用了许多不是标准 C 语言部分的类型、宏和结构。这些类型、宏和结构用来使建立基于 Windows 应用程序的任务更简单并使应用程序源代码更清楚,更易理解。本手册中所有的类型,宏和结构都在 Win32 C 语言头文件中定义。

卷(一)和卷(二)的许多章节中都包括代码举例。这些举例说明如何使用 Win32 函数完成任务。几乎在所有情况中,举例都是代码片段,而不是完整的程序。各个例子都是为了解释函数应用的上下文;一般来说,例子都假定变量、结构和常量已定义或初始化或两者都有。举例还常常使用备注来描述任务而没有列出相应的语句。尽管举例不完整,但可以用下面的步骤在你的应用程序中使用它们:

- 在你的程序中包括 WINDOWS.H 文件。
- 定义恰当的包含在函数、结构内的常量和用于举例中的常量。
- 定义并初始化所有变量。
- 用适当语句替换代表任务的备注。
- 检查返回的错误值并采取相应的操作。

本手册中有些举例结合了 Win32 和 C 运行时函数来完成它们的任务。

文档约定

下面的约定在本手册中用于定义语法:

约 定	含　义
斜体字	表示一个占位符或变量:必须提供实际值。例如,语句 SetCursorPos(X, Y)要求用数值替代 X 和 Y 参数。
〔〕	可选参数。
|	分隔一个选择项。
…	指明前面的项目可以重复。
⋮	代表示范应用程序的略去的部分。

第三部分

系统服务

内存管理

第42章

42.1 关于内存管理

在 Microsoft Win32 应用程序编程接口(API)中,每个进程都有它自己的 32 位虚拟地址区,使编址能力达到 4GB 的内存。在低端内存(0x00 到 0x7FFFFFFF)中的 2GB 提供给用户,而在高内存(0x80000000 到 0xFFFFFFFF)中的 2GB 为核心保留。由进程使用的虚拟地址并不代表对象在内存中的真实物理位置。相反,对各个进程而言,核心维护一个页面图,一个用于把虚拟地址翻译成对应物理地址的数据结构。

本章介绍 Win32 的内存管理,它还描述如何分配和使用内存。

42.1.1 虚拟地址空间和物理存储

各个进程的虚拟地址空间要比总的物理内存、随机访问内存((RAM)大得多,并能提供给所有进程。为了增加物理存储的大小,内核使用磁盘作为外存储。能提供给所有执行进程的存储总量是物理内存、RAM 和磁盘上提供给页面文件(用于提高物理存储量的一个磁盘文件)自由空间的总和。各个进程的物理存储和虚拟(逻辑)地址空间组织成页面(Page),它是内存的单位,其大小取决于主计算机。例如,在 x86 计算机上主机页面是 4KB。

要使内存管理中的灵活性最大,内核可以把物理内存的页面从磁盘上的页面文件中移出和移进。当一个页面移进物理内存时,内核更新涉及进程的页面映射。当内核需要物理内存空间时,它把物理内存中的最近最少使用的页面移到页面文件中。由 Kernel 操作的物理内存对应用程序完全透明,只在它们的虚拟地址空间中进行。

进程虚拟地址空间的页面可以处于下列状态之一:

状态	描　述
自由	虽然自由页面当前不能访问,但它能被提交或保留。
保留	保留页面是设定今后要用的一块进程的虚拟地址空间。进程不能访问保留页面,而且没有物理存储与它有关。保留页面保留一段虚拟地址,它不能由其后的分配操作(即 malloc、LocalAlloc 等)使用。进程可以使用 VirtualAlloc 函数保留它的地址空间页面,然后提交保留的页面。也可以使用 VirtualFree 函数释放它们。
提交	提交页面是一个物理存储(内存中或磁盘上)已分配的页面。它可以被保护,不允许访问或只读访问,或可以读和写访问。进程可以使用 VirtualAlloc 函数分配提交页面;GlobalAlloc 和 LocalAlloc 函数分配可读、写访问的提交页面。由 VirtualAlloc 分配的提交页面可以由 VirtualFree 函数解除提交,它释放页面的存储并改变页面的状态为保留。

42.1.2 全局和局部函数

进程可以使用 GlobalAlloc 和 LocalAlloc 函数分配内存。在 Win32 API 的线性 32 位环境中，局部堆和全局堆不作区分。因此，这些函数分配的内存对象之间没有什么不同。

由 GlobalAlloc 和 LocalAlloc 分配的内存对象是可读－写的，提交页面。私有内存不能被其它进程访问。通过使用 GMEM_DDESHARE 标志调用 GlobalAlloc 分配的内存并不象它在 Windows 3.x 版本中那样能够全局共享，然而，此标志由于兼容目的和有些增强动态数据交换(DDE)操作的性能需要也可以使用它。其它目的要求共享内存的应用程序必须使用文件映射对象。多个进程可以映射同一文件映射对象以提供命名共享内存。要了解有关文件映射的更多信息，参见第 47 章“文件映射”，以及第 42.1.6 节“共享内存”。

通过使用 GlobalAlloc 和 LocalAlloc，可以分配一块能用 32 位地址表示的任何大小的内存，只受可用物理内存的限制，包括磁盘上页面文件中的存储。这些函数，随同操作全局和局部内存对象的其它全局和局部函数一起包括在 Win32 API 中，与 16 位版本 Windows 兼容。但从 16 位分段内存模式到 32 位虚拟内存模式的变化已使一些函数以及它们的选项不必要或无意义。

例如，不再有近和远指针，因为局部和全局分配都返回 32 位虚拟地址。

GlobalAlloc 和 LocalAlloc 都可分配固定或可移动的内存对象。可移动对象还可以标为可丢弃的。在 Windows 的早期版本中，可移动内存对象对内存管理是重要的。它们在有必要为别的内存分配挪出空间时能让系统压缩堆。通过使用虚拟内存，系统能移动物理内存的页面而不影响使用这些页面的进程虚拟地址来管理内存，它只要把进程的虚拟页面映射到物理页面的新位置就可以了。可移动内存还对分配可丢弃内存有用，当系统需要额外的物理存储时，它可以使用“最近最少使用(Least recently used)”算法释放没有被锁定的可丢弃内存。可丢弃内存用于不经常使用并能简单地重新产生的数据。

当分配固定内存对象时，GlobalAlloc 和 LocalAlloc 返回一个调用进程能立即用来访问内存的 32 位指针。对可移动内存而言，返回值是一个句柄。要取得可移动内存对象的指针，调用进程使用 GlobalLock 和 LocalLock 函数。这些函数锁定内存使得它不能移动或丢弃。各个内存对象的内部数据结构包括一个开始值为零的锁定计数。对可移动内存对象而言，调用 GlobalLock 和 LocalLock 一次则给此计数增 1，而 GlobalUnlock 和 LocalUnLock 函数则减 1。锁定内存不能移动或丢弃，除非内存对象使用 GlobalReAlloc 或 LocalReAlloc 函数重新分配。锁定内存对象的内存块在内存中保持锁定直到它的锁定计数减为零，此时它可以被移动或丢弃。

由 GlobalAlloc 或 LocalAlloc 分配的实际内存大小可以比要求的大。要确定分配的实际字节数，使用 GlobalSize 或 LocalSize 函数。如果分配的量大于要求的量，则进程可以使用实际分配的量。

GlobalReAlloc 和 LocalReAlloc 函数用来改变大小(单位为字节)，或 GlobalAlloc 和 LocalAlloc 分配的内存对象的属性，大小可以增加或减小。

GlobalFree 和 LocalFree 函数释放由 GlobalAlloc, LocalAlloc, GlobalReAlloc 或 Local-

ReAlloc 分配的内存。

其它全局和局部函数包括 GlobalDiscard, LocalDiscard, GlobalFlags, LocalFlags, GlobalHandle 和 LocalHandle 函数。要丢弃一个指定的可丢弃内存对象而不使该句柄无效，则使用 GlobalDiscard 或 LocalDiscard。此句柄以后可被 GlobalReAlloc 或 LocalReAlloc 用来分配一块与同一句柄有关的新内存。要返回某一特定内存对象的信息，使用 GlobalFlags 或 LocalFlags。该信息包括对象的锁定计数并指明对象是否可丢弃或是否已被丢弃。要返回与指定指针有关的内存对象句柄，可以使用 GlobalHandle 或 LocalHandle。

42.1.3 标准 C 库函数

Win32 进程可以安全地使用标准 C 库函数(malloc, free 等)操作内存。当在 Windows 的早期版本中使用时，这些函数有潜在的问题，而在使用 Win32 API 的应用程序中则不再是问题。例如，malloc 只分配没有可移动优点的固定指针。由于系统通过移动物理内存页面而不影响虚拟地址来管理内存，这已不再是一个问题。与此类似，近和远指针之间的区别不再明显。因此，除非想分配可丢弃的内存，否则现在有理由使用标准 C 库函数进行内存管理。

42.1.4 虚拟内存函数

Win32 API 提供一套虚拟内存函数，能使进程操作或确定在虚拟地址空间中的页面状况。许多应用程序通过使用标准分配函数(GlobalAlloc, LocalAlloc, malloc 等)就能满足它们的内存需要。然而，虚拟内存函数可提供一些在标准分配函数中得不到的功能。它们能执行下列操作：

· 保留一段进程的虚拟地址空间。虽然保留的地址空间并不分配任何物理存储位置，但它阻止别的分配操作使用指定区段，它不影响其它进程的虚拟地址空间。保留的页面阻止物理存储的不必要消耗，同时能使进程保留一段、动态数据结构能在其中扩大的地址空间。如果需要，进程可为此空间分配物理存储。

· 提交进程虚拟地址空间中一段保留的页面使得物理存储(还在 RAM 中或在磁盘上)只让分配进程可以访问。

· 指定一段提交页面的可读－写、只读或不可访问。这不同于标准分配函数，它始终分配可读－写的页面。

· 释放一段保留页面，使虚拟地址段能提供给由调用进程进行的后续分配操作。

· 解除提交一段提交的页面，释放它们的物理存储。解除提交一段使它们的存储能提供给任何进程进行的后续分配活动。

· 把一页或多页提交内存锁进物理内存(RAM)中，因此系统不能把页面换出到页面文件。

· 查询有关调用进程或指定进程的虚拟地址空间中一段页面的信息。

· 改变调用进程或指定进程的虚拟地址空间中指定段提交页面的访问保护。

虚拟内存函数操作内存页面。函数使用当前计算机上的页面大小近似指定的大小和地址。

要了解各个函数近似这些值的方法，参见《Microsoft Win32 程序员参考大全》(三)和(四)。

要确定当前计算机上页面的大小，可使用 GetSystemInfo 函数。

VirtualAlloc 函数执行下列操作之一：

· 保留一个或多个自由页。

· 提交一个或多个保留页。

· 保留并提交一个或多个自由页。

可以指定要保留或提交页面的开始地址，也可以让系统确定该地址。函数把指定地址转成适当的页面边界。虽然保留的页面是不可访问的，但提交页面可以用 PAGE_READWRITE, PAGE_READONLY 或 PAGE_NOACCESS 标志分配。当页面提交时，虽然在页面文件中分配存储，但各个页面的初始化和装入到物理内存只发生在第一次试图从该页读出或写入时。可以使用正常的指针引用来访问由 VirtualAlloc 函数提交的内存。

VirtualFree 函数执行下列操作之一：

· 解除提交一个或多个提交的页面，把页面状态变为保留。解除提交页面释放与页面有关的物理存储，使它能被任何进程分配。任何块的提交页面都可解除提交。

· 释放一块一个或多个保留页面，把页面状态变为自由。释放一块页面使保留地址段能被进程分配。保留页面只能通过开始由 VirtualAlloc 保留的整个块的方式来释放。

· 同时解除提交和释放一块一个或多个提交的页面，把页面状态变为自由。指定的块必须包括开始由 VirtualAlloc 保留的整个块，而且所有的页面当前必须是提交的。

VirtualLock 函数能使进程把一个或多个提交内存页面锁进物理内存 RAM。锁定页面防止系统把页面换出到页面文件。当有必要保证关键数据不用访盘就可以访问时它非常有用。在内存中锁定页面是危险的，因为它限制了系统管理内存的能力。过量使用 VirtualLock 会降低系统性能，引起不必要的可执行代码换出到页面文件上。VirtualUnlock 函数解锁由 VirtualLock 锁定的内存。

VirtualQuery 和 VirtualQueryEx 函数返回有关从进程地址空间任何指定地址开始的一段连续页面的信息；VirtualQuery 返回有关调用进程中内存的信息。VirtralQueryEx 返回有关指定进程中内存的信息并有助于支持需要有关正被调试进程信息的调试程序。页面区域限制在最近的页面边界上，它可超出通常有下面属性的所有后续页面：

· 所有页面的状态相同：或提交，或保留，或自由。

· 如果初始页面不是自由的，那么该区中所有页面是通过调用 VirtualAlloc 函数保留页面的初始分配的部分。

· 所有页面的访问保护是一样的(即，PAEG_READONLY, PAGE_READWRITE 或 PAGE_NOACCESS 标志)。

VirtualProtect 函数能使进程修改进程地址空间中任何提交页面的访问保护。例如，进程可以分配可读－写页面以存储敏感数据，然后可以把访问改变成只读或不可访问以防止意外覆盖。虽然 VirtualProtect 一般与用 VirtualAlloc 分配的页面使用，但也可以与由任何其它分配函数提交的面面使用。然而，VirtualProtect 改变整个页面的保护，而由

其它函数返回的指针不必一定在页面边界上。VirtualProtectEx 函数类似 VirtualProtect，只是它改变指定进程中内存的保护。改变保护对调试程序访问正被调试进程的内存很有用。

42.1.5 堆函数

堆函数能使进程建立一私有堆，在调用进程地址空间中的一块一个或多个页面。然后进程可以使用一组单独的函数管理该堆中的内存。在从私有堆分配的内存和通过使用标准分配函数(GlobalAlloc, LocalAlloc, malloc 等)分配的内存之间没有区别。

HeapCreate 函数建立私有堆对象，调用进程可以使用 HeapAlloc 函数从中分配内存块。HeapCreate 指定该堆的初始大小以及最大大小。初始大小确定开始为该堆分配的提交和可读－写页面数量。最大大小确定保留页面的总数量，这些页面建立连续块的进程虚拟地址空间，堆能在其中增大。如果 HeapAlloc 的请求超出提交页面的当前大小，则自动从保留空间提高额外的页面(假设它的物理存储可以得到)。一旦提交了页面，那么直到进程终止或通过调用 HeapDestroy 函数销毁该堆后它们才解除提交。

私有堆对象的内存只能被建立它的进程访问。如果动态链按库(DLL)建立私有堆，那么它在激活 DLL 的进程的地址空间中分配堆，它只能被该进程访问。

HeapAlloc 函数从私有堆中分配指定数量的字节，并返回该分配块的指针。指针标识 HeapFree 函数要释放的块或 HeapSize 函数确定大小的块。

由 HeapAlloc 分配的内存不可移动。由于系统不能压缩私有堆，所以堆中可能存在碎片。

堆函数的一个可能应用是在进程启动时建立一私有堆，指定足够满足进程内存要求的初始大小。如果 HeapCreate 函数的调用失败，那么进程可以终止或通知用户内存不足；如果它成功，则该进程一定会得到它需要的内存。

42.1.6 共享内存

在Win32 API 中，共享内存是通过文件映射实施的。所有由其它分配方式(GlobalAlloc, LocalAlloc、HeapAlloc 或 VirtualAlloc 函数)分配的内存只能由调用进程访问。然而，由 DLL 分配的内存在激活 DLL 的进程地址空间中，并不能被使用同一 DLL 的其它进程访问。

命名文件映射提供建立一块共享内存的简易方法。一个进程在它使用 CreateFileMapping 函数建立文件映射对象时指定名字。其它进程指定 CreateFileMapping 或 OpenFileMapping 函数的相同名字以取得映射对象的句柄。事件对象、信号灯对象，mutex 对象和文件映射对象的名字共享同一名字空间。如果指定的名字与一不同类型现有对象的名字一样，则会发生错误。当建立命名对象时，尽量使用独特的名字并检查函数返回值看是否有重名。

各个进程在 MapViewOfFile 函数中指定文件映射对象的句柄以把文件的一个视图映射到它自己的地址空间。单个文件映射对象的所有进程的视图都映射到物理存储同一可共享页面上。

但是,映射视图的虚拟地址可能在不同的进程中而不同,除非用 MapViewOfFileEx 函数把该视图映射到指定地址。尽管是可共享的,但用于映射文件视图的物理存储页面不是全局的;它们不能被没有映射文件视图的进程访问。

文件映射对象与系统在映射视图换出物理内存并存入磁盘时使用的磁盘文件有关。此磁盘交换文件可以是系统的页面文件也可以是在建立文件映射对象时指定的别的文件。这种情况下,内存与文件的内容一起初始化。在文件系统中映射一指定文件对需要共享现有文件中的数据或想使用文件保存由共享进程产生数据的进程非常有用。如果映射一个指定文件,应该用独占访问打开并保持句柄打开直到用完共享内存。这样可以防止其它进程打开文件的另一个句柄来使用 ReadFile 或 WriteFile,或为同一文件建立另一映射对象,其中任何一种操作都可导致不可预料的结果。

任何通过映射文件视图提交的页面在拥有映射对象的最后一个进程终止或通过调用 UnmapViewOfFile 函数取消映射视图时释放。此时,有与映射对象有关的指定文件(如果的话)被更新。指定文件也可通过调用 FlushViewOfFile 函数强行更新。

要了解文件映射更多信息,参见第 47 章"文件映射"。有关 DLL 中共享内存的举例,参见第 50 章"动态链接库"。

如果多个进程有对共享内存的写访问,那么对该内存的访问应同步。要了解进程间同步的更多信息,参见第 44 章"同步"。

42.1.7 访问确认

Win32 API 提供一组函数,进程可以用它来确认是否对所给内存地址或地址段拥有指定类型的访问权限。访问确认函数有:

函数	描　述
IsBadCodePtr	确定调用进程是否有指定地址内存的读访问。
IsBadReadPtr	确定调用进程是否有指定地址段内存的读访问。
IsBadStringPtr	确定调用进程是否有以空字符结束的字符串指针指向内存的读访问。函数确认指定数量字符的访问或直到它迂到字符串的终止空字符。
IsBadWritePtr	确定调用进程是否有指定地址段处内存的写访问。

IsBadHugeReadPtr 和 IsBadHugeWritePtr 函数为 Windows 早期版本的兼容仍可使用,它们在正常内存分配和占多个段的巨型分配之间是有区别的。在 Win32 API 中,这些函数等同于 IsBadReadPtr 和 IsBadWritePtr。

在抢先多任务环境中,有些线程有可能改变进程对正在测试内存的访问。即使这些函数之一指示进程有指定内存要求的访问,那么在试图访问内存时应使用结构化例外处理。使用结构化例外处理能使系统在发生访问确认例外时通知进程,给进程处理例外事件的机会。要了解结构化例外处理的更多信息,参见第 64 章"结构化例外处理"。

42.2 虚拟内存函数的使用

这节介绍如何使用虚拟内存函数进行动态分配。

下例演示了保留和提交动态数组需要的内存时 VirtualAlloc 和 VirtualFree 函数的使用。首先，调用 VirtualAlloc 保留一块页面，并指定基地址参数为 NULL，迫使内核确定块的位置。然后，每当有必要从此保留区提交页面时调用 VirtualAlloc，并指定要提交的下一页的基地址。

该例使用 try－except 结构化例外处理语法从保留区中提交页面。每次在 try 块执行期间发生页面错误例外时，就执行 except 块前面表达式中的过滤函数。如果过滤函数能分配另一个页面，那么 try 块在发生例外的地方继续执行。否则，执行 except 块中的例外处理程序。要了解有关结构化例外处理的更多信息，参见第 64 章“结构化例外处理”。

此动态分配的另一种替换方法是进程简单地提交整个区域而不是仅保留它。然而，这样会消耗不必要的物理存储，使它不能为其它进程所用。

该例在完成时使用 VirtualFree 释放保留的和提交的页面。该函数调用两次：第一次解除提交页面的提交，第二次释放保留页面的整个区域。

```
#define PAGELIMIT 80
#define PAGESIZE 0x1000

INT PageFaultExceptionFilter(DWORD);
VOID MyErrorExit(LPTSTR);

LPTSTR lpNxtPage;
DWORD dwPages = 0;

VOID UseDynamicVirtualAlloc(VOID) {
  LPVOID lpvBase;
  LPTSTR lpPtr;
  BOOL bSuccess;
  DWORD i;

  /* Reserve pages in the process's virtual address space. */

  lpvBase = VirtualAlloc(
        NULL,               /* system selects address   */
        PAGELIMIT*PAGESIZE, /* size of allocation       */
        MEM_RESERVE,        /* allocates reserved pages */
        PAGE_NOACCESS);     /* protection = no access   */
  if (lpvBase == NULL )
      MyErrorExit("VirtualAlloc reserve");

  lpPtr = lpNxtPage = (LPTSTR) lpvBase;

  /*
   * Use try-except structured exception handling when
   * accessing the pages. If a page fault occurs, the
   * exception filter is executed to commit another page
   * from the reserved block of pages.
   */

  for (i=0; i < PAGELIMIT*PAGESIZE; i++) {

      try {
```

```
            /* Write to memory. */

            lpPtr[i] = 'a';
        }

        /*
         * If there is a page fault, commit another page
         * and try again.
         */

        except ( PageFaultExceptionFilter(
                    GetExceptionCode() ) ) {

            /*
             * This is executed only if the filter function is
             * unsuccessful in committing the next page.
             */

            ExitProcess( GetLastError() );

        }

    }

    /* Release the block of pages when you are finished using them. */

    /* First, decommit the committed pages. */

    bSuccess = VirtualFree(
                lpvBase,          /* base address of block    */
                dwPages*PAGESIZE, /* bytes of committed pages */
                MEM_DECOMMIT);    /* decommit the pages       */

    /* Release the entire block. */

    if (bSuccess)
        bSuccess = VirtualFree(
                lpvBase,        /* base address of block     */
                0,              /* releases the entire block */
                MEM_RELEASE);   /* releases the pages        */

}

    INT PageFaultExceptionFilter(DWORD dwCode) {
    LPVOID lpvResult;

      /* If the exception is not a page fault, exit. */

      if (dwCode != EXCEPTION_ACCESS_VIOLATION) {
          printf("exception code = %d\n", dwCode);
          return EXCEPTION_EXECUTE_HANDLER;
      }
```

```
    printf("page fault\n");

    /* If the reserved pages are used up, exit. */

    if (dwPages >= PAGELIMIT) {
        printf("out of pages\n");
        return EXCEPTION_EXECUTE_HANDLER;
    }

    /* Otherwise, commit another page. */

    lpvResult = VirtualAlloc(
        (LPVOID) lpNxtPage, /* next page to commit  */
        PAGESIZE,           /* page size, in bytes  */
        MEM_COMMIT,         /* alloc committed page */
        PAGE_READWRITE);    /* read-write access    */
    if (lpvResult == NULL ) {
        printf("VirtualAlloc failed\n");
        return EXCEPTION_EXECUTE_HANDLER;
    }

    /*
     * Increment the page count, and advance lpNxtPage
     * to the next page.
     */

    dwPages++;
    lpNxtPage += PAGESIZE;

    /* Continue execution where the page fault occurred. */

    return EXCEPTION_CONTINUE_EXECUTION;

}
```

42.3 函数

下面是用于内存管理中的 Win32 函数：

GetProcessHeap
GlobalAlloc
GlobalDiscard
GlobalFlags
GlobalFree
GlobalHandle
GlobalLock
GlobalMemoryStatus
GlobalReAlloc
GlobalSize
GlobalUnlock
HeapAlloc
HeapCreate
HeapDestroy
HeapFree
HeapReAlloc
HeapSize
IsBadCodePtr
IsbadHugeReadPtr
IsBadHugeReadPtr

IsBadHugeWritePtr
IsBadReadPtr
IsBadStringPtr
IsBadWritePtr
LocalAlloc
LocalDiscard
LocalFlags
LocalFree
LocalHandle
LocalLock
LocalReAlloc
LocalSize
LocalUnlock
VirtualAlloc
VirtualFree
VirtualLock
VirtualProtect
VirtualProtectEx
VirtualQuery
VirtualQueryEx
VirtualUnlock