

## 第15章 在应用程序中使用虚拟内存

Windows提供了3种进行内存管理的方法，它们是：

- 虚拟内存，最适合用来管理大型对象或结构数组。
- 内存映射文件，最适合用来管理大型数据流（通常来自文件）以及在单个计算机上运行的多个进程之间共享数据。
- 内存堆栈，最适合用来管理大量的小对象。

本章将要介绍第一种方法，即虚拟内存。内存映射文件和堆栈分别在第 17章和第18章介绍。

用于管理虚拟内存的函数可以用来直接保留一个地址空间区域，将物理存储器（来自页文件）提交给该区域，并且可以设置你自己的保护属性。

### 15.1 在地址空间中保留一个区域

通过调用VirtualAlloc函数，可以在进程的地址空间中保留一个区域：

```
PVOID VirtualAlloc(  
    PVOID pvAddress,  
    SIZE_T dwSize,  
    DWORD fdwAllocationType,  
    DWORD fdwProtect);
```

第一个参数pvAddress包含一个内存地址，用于设定想让系统将地址空间保留在什么地方。在大多数情况下，你为该参数传递NULL。它告诉VirtualAlloc，保存着一个空闲地址区域的记录的系统应该将该区域保留在它认为合适的任何地方。系统可以从进程的地址空间的任何位置来保留一个区域，因为不能保证系统可以从地址空间的底部向上或者从上面向底部来分配各个区域。可以使用MEM\_TOP\_DOWN标志来说明该分配方式。这个标志将在本章的后面加以介绍。

对大多数程序员来说，能够选择一个特定的内存地址，并在该地址保留一个区域，这是个非同寻常的想法。当你在过去分配内存时，操作系统只是寻找一个其大小足以满足需要的内存块，并分配该内存块，然后返回它的地址。但是，由于每个进程有它自己的地址空间，因此可以设定一个基本内存地址，在这个地址上让操作系统保留地址空间区域。

例如，你想将一个从50 MB开始的区域保留在进程的地址空间中。这时，可以传递2 428 800（ $50 \times 1024 \times 1024$ ）作为pvAddress参数。如果该内存地址有一个足够大的空闲区域满足你的要求，那么系统就保留这个区域并返回。如果在特定的地址上不存在空闲区域，或者如果空闲区域不够大，那么系统就不能满足你的要求，VirtualAlloc函数返回NULL。注意，为pvAddress参数传递的任何地址必须始终位于进程的用户方式分区中，否则对VirtualAlloc函数的调用就会失败，导致它返回NULL。

第13章讲过，地址空间区域总是按照分配粒度的边界来保留的(迄今为止在所有的Windows环境下均是64KB)。因此，如果试图在进程地址空间中保留一个从19 668 992( $300 \times 65\,536 + 8192$ )这个地址开始的区域，系统就会将这个地址圆整为64KB的倍数，然后保留从19 660 800（ $300 \times 65\,536$ ）这个地址开始的区域。

如果VirtualAlloc函数能够满足你的要求，那么它就返回一个值，指明保留区域的基地址。

如果传递一个特定的地址作为 VirtualAlloc 的 pvAddress 参数, 那么该返回值与传递给 VirtualAlloc 的值相同, 并被圆整为 ( 如果需要的话 ) 64KB 边界值。

VirtualAlloc 函数的第二个参数是 dwSize, 用于设定想保留的区域的大小 ( 以字节为计量单位 )。由于系统保留的区域始终必须是 CPU 页面大小的倍数, 因此, 如果试图保留一个跨越 62KB 的区域, 结果就会在使用 4 KB、8 KB 或 16 KB 页面的计算机上产生一个跨越 64KB 的区域。

VirtualAlloc 函数的第三个参数是 fdwAllocationType, 它能够告诉系统你想保留一个区域还是提交物理存储器 ( 这样的区分是必要的, 因为 VirtualAlloc 函数也可以用来提交物理存储器 )。若要保留一个地址空间区域, 必须传递 MEM\_RESERVE 标识符作为 FdwAllocationType 参数的值。

如果保留的区域预计在很长时间内不会被释放, 那么可以在尽可能高的内存地址上保留该区域。这样, 该区域就不会从进程地址空间的中间位置上进行保留。因为在这个位置上它可能导致区域分成碎片。如果想让系统在最高内存地址上保留一个区域, 必须为 pvAddress 参数和 fdwAllocationType 参数传递 NULL, 还必须逐位使用 OR 将 MEM\_TOP\_DOWN 标志和 MEM\_RESERVE 标志连接起来。

注意 在 Windows 98 下, MEM\_TOP\_DOWN 标志将被忽略。

最后一个参数是 fdwProtect, 用于指明应该赋予该地址空间区域的保护属性。与该区域相关联的保护属性对映射到该区域的已提交内存没有影响。无论赋予区域的保护属性是什么, 如果没有提交任何物理存储器, 那么访问该范围中的内存地址的任何企图都将导致该线程引发一个访问违规。

当保留一个区域时, 应该为该区域赋予一个已提交内存最常用的保护属性。例如, 如果打算提交的物理存储器的保护属性是 PAGE\_READWRITE ( 这是最常用的保护属性 ), 那么应该用 PAGE\_READWRITE 保护属性来保留该区域。当区域的保护属性与已提交内存的保护属性相匹配时, 系统保存的内部记录的运行效率最高。

可以使用下列保护属性中的任何一个: PAGE\_NOACCESS、PAGE\_READWRITE、PAGE\_READONLY、PAGE\_EXECUTE、PAGE\_EXECUTE\_READ 或 PAGE\_EXECUTE\_READWRITE。但是, 既不能设定 PAGE\_WRITECOPY 属性, 也不能设定 PAGE\_EXECUTE\_WRITECOPY 属性。如果设定了这些属性, VirtualAlloc 函数将不保留该区域, 并且返回 NULL。另外, 当保留地址空间区域时, 不能使用保护属性标志 PAGE\_GUARD, PAGE\_NOCACHE 或 PAGE\_WRITECOMBINE, 这些标志只能用于已提交的内存。

注意 Windows 98 只支持 PAGE\_NOACCESS、PAGE\_READONLY 和 PAGE\_READWRITE 保护属性。如果试图保留使用 PAGE\_EXECUTE 或 PAGE\_EXECUTE\_READ 两个保护属性的区域, 将会产生一个带有 PAGE\_READONLY 保护属性的区域。同样, 如果保留一个使用 PAGE\_EXECUTE\_READWRITE 保护属性的区域, 就会产生一个带有 PAGE\_READWRITE 保护属性的区域。

## 15.2 在保留区域中的提交存储器

当保留一个区域后, 必须将物理存储器提交给该区域, 然后才能访问该区域中包含的内存地址。系统从它的页文件中将已提交的物理存储器分配给一个区域。物理存储器总是按页面边界和页面大小的块来提交的。

若要提交物理存储器，必须再次调用 VirtualAlloc 函数。不过这次为 fdwAllocationType 参数传递的是 MEM\_COMMIT 标志，而不是 MEM\_RESERVE 标志。传递的页面保护属性通常与调用 VirtualAlloc 来保留区域时使用的保护属性相同（大多数情况下是 PAGE\_READWRITE），不过也可以设定一个不同的保护属性。

在已保留的区域中，你必须告诉 VirtualAlloc 函数，你想将物理存储器提交到何处，以及要提交多少物理存储器。为了做到这一点，可以在 pvAddress 参数中设定你需要的内存地址，并在 dwSize 参数中设定物理存储器的数量（以字节为计量单位）。注意，不必立即将物理存储器提交给整个区域。

下面让我们来看一个如何提交物理存储器。比如说，你的应用程序是在 x86 CPU 上运行的，该应用程序保留了一个从地址 5 242 880 开始的 512 KB 的区域。你想让应用程序将物理存储器提交给已保留区域的 6 KB 部分，从 2 KB 的地方开始，直到已保留区域的地址空间。为此，可以调用带有 MEM\_COMMIT 标志的 VirtualAlloc 函数，如下所示：

```
VirtualAlloc((PVOID) (5242880 + (2 * 1024)), 6 * 1024,  
MEM_COMMIT, PAGE_READWRITE);
```

在这个例子中，系统必须提交 8 KB 的物理存储器，地址范围从 5 242 880 到 5 251 071 (5 242 880 + 8 KB - 1 字节)。这两个提交的页面都拥有 PAGE\_READWRITE 保护属性。保护属性只以整个页面为单位来赋予。同一个内存页面的不同部分不能使用不同的保护属性。然而，区域中的一个页面可以使用一种保护属性（比如 PAGE\_READWRITE），而同一个区域中的另一个页面可以使用不同的保护属性（比如 PAGE\_READONLY）。

### 15.3 同时进行区域的保留和内存的提交

有时你可能想要在保留区域的同时，将物理存储器提交给它。只需要一次调用 VirtualAlloc 函数就能进行这样的操作，如下所示：

```
PVOID pvMem = VirtualAlloc(NULL, 99 * 1024,  
MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);
```

这个函数调用请求保留一个 99 KB 的区域，并且将 99 KB 的物理存储器提交给它。当系统处理这个函数调用时，它首先要搜索你的进程的地址空间，找出未保留的地址空间中一个地址连续的区域，它必须足够大，能够存放 100 KB（在 4 KB 页面的计算机上）或 104 KB（在 8 KB 页面的计算机上）。

系统之所以要搜索地址空间，原因是已将 pvAddress 参数设定为 NULL。如果为 pvAddress 设定了内存地址，系统就要查看在该内存地址上是否存在足够大的未保留地址空间。如果系统找不到足够大的未保留地址空间，VirtualAlloc 将返回 NULL，

如果能够保留一个合适的区域，系统就将物理存储器提交给整个区域。无论是该区域还是提交的内存，都将被赋予 PAGE\_READWRITE 保护属性。

最后需要说明的是，VirtualAlloc 将返回保留区域和提交区域的虚拟地址，然后该虚拟地址被保存在 pvMem 变量中。如果系统无法找到足够大的地址空间，或者不能提交该物理存储器，VirtualAlloc 将返回 NULL。

当用这种方式来保留一个区域和提交物理存储器时，将特定的地址作为 pvAddress 参数传递给 VirtualAlloc 当然是可能的。否则就必须用 OR 将 MEM\_TOP\_DOWN 标志与 fdwAllocationType 参数连接起来，并为 pvAddress 参数传递 NULL，让系统在进程的地址空间的顶部选定一个适当的区域。

## 15.4 何时提交物理存储器

假设想实现一个电子表格应用程序，这个电子表格为 200 行 x 256 列。对于每一个单元格，都需要一个 CELLDATA 结构来描述单元格的内容。若要处理这种二维单元格矩阵，最容易的方法是在应用程序中声明下面的变量：

```
CELLDATA CellData[200][256];
```

如果 CELLDATA 结构的大小是 128 字节，那么这个二维矩阵将需要 6 553 600 (200 x 256 x 128) 个字节的物理存储器。对于电子表格来说，如果直接用页文件来分配物理存储器，那么这是个不小的数目了，尤其是考虑到大多数用户只是将信息放入少数的单元格中，而大部分单元格却空闲不用，因此显得有些浪费。内存的利用率非常低。

传统上，电子表格一直是用其他数据结构技术来实现的，比如链接表等。使用链接表，只需要为电子表格中实际包含数据的单元格创建 CELLDATA 结构。由于电子表格中的大多数单元格都是不用的，因此这种方法可以节省大量的内存。但是这种方法使得你很难获得单元格的内容。如果想知道第 5 行第 10 列的单元格的内容，必须遍历链接表，才能找到需要的单元格，因此使用链接表方法比明确声明的矩阵方法速度要慢。

虚拟内存为我们提供了一种兼顾预先声明二维矩阵和实现链接表的两全其美的方法。运用虚拟内存，既可以使用已声明的矩阵技术进行快速而方便的访问，又可以利用链接表技术大大节省内存的使用量。

如果想利用虚拟内存技术的优点，你的程序必须按照下列步骤来编写：

- 1) 保留一个足够大的地址空间区域，用来存放 CELLDATA 结构的整个数组。保留一个根本不使用任何物理存储器的区域。

- 2) 当用户将数据输入一个单元格时，找出 CELLDATA 结构应该进入的保留区域中的内存地址。当然，这时尚未有任何物理存储器被映射到该地址，因此，访问该地址的内存的任何企图都会引发访问违规。

- 3) 就 CELLDATA 结构来说，只将足够的物理存储器提交给第二步中找到的内存地址（你可以告诉系统将物理存储器提交给保留区域的特定部分，这个区域既可以包含映射到物理存储器的各个部分，也可以包含没有映射到物理存储器的各个部分）。

- 4) 设置新的 CELLDATA 结构的成员。

现在物理存储器已经映射到相应的位置，你的程序能够访问内存，而不会引发访问违规。这个虚拟内存技术非常出色，因为只有在用户将数据输入电子表格的单元格时，才会提交物理存储器。由于电子表格中的大多数单元格是空的，因此大部分保留区域没有提交给它的物理存储器。

虚拟内存技术存在的一个问题是，必须确定物理存储器在何时提交。如果用户将数据输入一个单元格，然后只是编辑或修改该数据，那么就没有必要提交物理存储器，因为该单元格的 CELLDATA 结构的内存存在数据初次输入时就已经提交了。

另外，系统总是按页面的分配粒度来提交物理存储器的。因此，当试图为单个 CELLDATA 结构提交物理存储器时（像上面的第二步那样），系统实际上提交的是内存的一个完整的页面。这并不像它听起来那样十分浪费：为单个 CELLDATA 结构提交物理存储器的结果是，也要为附近的其他 CELLDATA 结构提交内存。如果这时用户将数据输入邻近的单元格（这是经常出现的情况），就不需要提交更多的物理存储器。

有 4 种方法可以用来确定是否要将物理存储器提交给区域的一个部分：



- 始终设法进行物理存储器的提交。每次调用 VirtualAlloc 函数的时候，不要查看物理存储器是否已经映射到地址空间区域的一个部分，而是让你的程序设法进行内存的提交。系统首先查看内存是否已经被提交，如果已经提交，那么就不要再提交更多的物理存储器。这种方法最容易操作，但是它的缺点是每次改变 CELLDATA 结构时要多进行一次函数的调用，这会使程序运行得比较慢。
- （使用 VirtualQuery 函数）确定物理存储器是否已经提交给包含 CELLDATA 结构的地址空间。如果已经提交了，那么就不要再进行任何别的操作。如果尚未提交，则可以调用 VirtualAlloc 函数以便提交内存。这种方法实际上比第一种方法差，它既会增加代码的长度，又会降低程序运行的速度（因为增加了对 VirtualAlloc 函数的调用）。
- 保留一个关于哪些页面已经提交和哪些页面尚未提交的记录。这样做可以使你的应用程序运行得更快，因为不必调用 VirtualAlloc 函数，你的代码能够比系统更快地确定内存是否已经被提交。它的缺点是，必须不断跟踪页面提交的信息，这可能非常简单，也可能非常困难，要根据你的情况而定。
- 使用结构化异常处理（SEH）方法，这是最好的方法。SEH 是一个操作系统特性，它使系统能够在发生某种情况时将此情况通知你的应用程序。实际上可以创建一个带有异常处理程序的应用程序，然后，每当试图访问未提交的内存时，系统就将这个问题通知应用程序。然后你的应用程序便进行内存的提交，并告诉系统重新运行导致异常条件的指令。这时对内存的访问就能成功地进行了，程序将继续运行，仿佛从未发生过问题一样。这种方法是优点最多的方法，因为需要做的工作最少（也就是说要你编写的代码比较少），同时，你的程序可以全速运行。关于 SEH 的全面介绍，请参见第 23、24 和 25 章。第 25 章中的电子表格示例应用程序说明了如何按照上面介绍的方法来使用虚拟内存。

## 15.5 回收虚拟内存和释放地址空间区域

若要回收映射到一个区域的物理存储器，或者释放这个地址空间区域，可调用 VirtualFree 函数：

```
BOOL VirtualFree(  
    LPVOID pvAddress,  
    SIZE_T dwSize,  
    DWORD fdwFreeType);
```

首先让我们观察一下调用 VirtualFree 函数来释放一个已保留区域的简单例子。当你的进程不再访问区域中的物理存储器时，就可以释放整个保留的区域和所有提交给该区域的物理存储器，方法是一次调用 VirtualFree 函数。

就这个函数的调用来说，pvAddress 参数必须是该区域的基地址。此地址与该区域被保留时 VirtualAlloc 函数返回的地址相同。系统知道在特定内存地址上的该区域的大小，因此可以为 dwSize 参数传递 0。实际上，必须为 dwSize 参数传递 0，否则对 VirtualFree 的调用就会失败。对于第三个参数 fdwFreeType，必须传递 MEM\_RELEASE，以告诉系统将所有映射的物理存储器提交给该区域并释放该区域。当释放一个区域时，必须释放该区域保留的所有地址空间。例如不能保留一个 128 KB 的区域，然后决定只释放它的 64 KB。必须释放所有的 128 KB。

当想要从一个区域回收某些物理存储器，但是却不释放该区域时，也可以调用 VirtualFree 函数，若要回收某些物理存储器，必须在 VirtualFree 函数的 pvAddress 参数中传递用于标识要回收的第一个页面的内存地址，还必须在 dwSize 参数中设定要释放的字节数，并在 fdwFreeType

参数中传递MEM\_DECOMMIT标志。

与提交物理存储器的情况一样，回收时也必须按照页面的分配粒度来进行。这就是说，设定页面中间的一个内存地址就可以回收整个页面。当然，如果  $pvAddress + dwSize$  的值位于一个页面的中间，那么包含该地址的整个页面将被回收。因此位于  $pvAddress$  至  $pvAddress + dwSize$  范围内的所有页面均被回收。

如果  $dwSize$  是 0， $pvAddress$  是已分配区域的基地址，那么 `VirtualFree` 将回收全部范围内的已分配页面。当物理存储器的页面已经回收之后，已释放的物理存储器就可以供系统中的所有其他进程使用，如果试图访问未回收的内存，将会造成访问违规。

### 15.5.1 何时回收物理存储器

在实践中，知道何时回收内存是非常困难的。让我们再以电子表格为例。如果你的应用程序是在 x86 计算机上运行，每个内存页面是 4 KB，它可以存放 32 个 (4096/128) `CELLDATA` 结构。如果用户删除了单元格 `CellData[0][1]` 的内容，那么只要单元格 `CellData[0][0]` 至 `CellData[0][31]` 也不被使用，就可以回收它的内存页面。那么怎么能够知道这个情况呢？可以用下面 3 种方法来解决这个问题。

- 毫无疑问，最容易的方法是设计一个 `CELLDATA` 结构，它的大小只有一个页面。这时，由于始终都是每个页面使用一个结构，因此当不再需要该结构中的数据时，就可以回收该页面的物理存储器。即使你的数据结构是 x86 CPU 上的 8 KB 或 12 KB 页面的倍数（通常这是非常大的数据结构），回收内存仍然是非常容易的。当然，如果要使用这种方法，必须定义你的数据结构，使之符合你针对的 CPU 的页面大小而不是我们通常编写程序所用的结构。
- 更为实用的方法是保留一个正在使用的结构的记录。为了节省内存，可以使用一个位图。这样，如果有一个 100 个结构的数组，你也可以维护一个 100 位的数组。开始时，所有的位均设置为 0，表示这些结构都没有使用。当使用这些结构时，可以将对应的位设置为 1。然后，每当不需要某个结构，并将它的位重新改为 0 时，你可以检查属于同一个内存页面的相邻结构的位。如果没有相邻的结构正在使用，就可以回收该页面。
- 最后一个方法是实现一个无用单元收集函数。这个方案依赖于这样一种情况，即当物理存储器初次提交时，系统将一个页面中的所有字节设置为 0。若要使用该方案，首先必须要在你的结构中设置一个 `BOOL`（也许称为 `fInUse`）。然后，每次你将一个结构放入已提交的内存中，必须确保该 `fInUse` 被置于 `TRUE`。

当你的应用程序运行时，必须定期调用无用单元收集函数。该函数应该遍历所有潜在的数据结构。对于每个数据结构，该函数首先要确定是否已经为该结构提交内存。如果已经提交，该函数将检查 `fInUse` 成员，以确定它是否是 0。如果该值是 0，则表示该结构没有被使用。如果该值是 `TRUE`，则表示该结构正在使用。当无用单元函数检查了属于既定页面的所有结构后，如果所有结构都没有被使用，它将调用 `VirtualFree` 函数，回收该内存。

当一个结构不再被视为“在用”（In Use）后，就可以立即调用无用单元收集函数，不过这项操作需要的时间比你想像的要长，因为该函数要循环通过所有可能的结构。实现该函数的一个出色方法是让它作为低优先级线程的一部分来运行。这样，就不必占用执行主应用程序的线程的时间。每当主应用程序运行空闲时，或者主应用程序的线程执行文件的 I/O 操作时，系统就可以给无用单元收集函数安排运行时间。

在上面列出的所有方法中，前面的两种方法是我个人喜欢使用的方法。不过，如果你的结

构比较小（小于一个页面），那么建议你使用最后一种方法。

### 15.5.2 虚拟内存分配的示例应用程序

清单15-1中列出的VMAlloc应用程序（“15 VMAlloc.exe”）显示了如何使用虚拟内存技术来处理一个结构数组。该应用程序的源代码和资源文件均位于本书所附光盘上的15-VMAlloc目录下。当启动该应用程序时，将出现图15-1所示的窗口。

开始时，没有为该结构数组保留任何地址空间的区域，准备为它保留的所有地址空间都是空闲的，如内存表所示。当你点击 Reserve Region（50,2 KB结构）按钮时，VMAlloc便调用VirtualAlloc函数，保留该区域，同时内存表被更新，以反映该区域已经被保留。当VirtualAlloc函数保留该区域后，其余按钮变均为活动按钮。

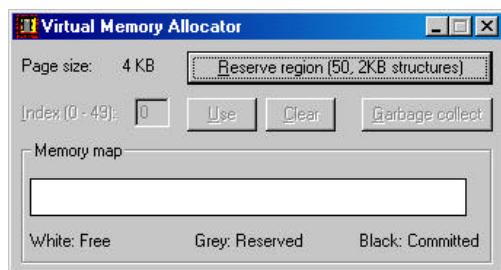


图15-1 运时VMAlloc 程序时出现的窗口

现在可以将一个索引键入编辑控件，以便选定一个索引，然后单击Use按钮。它的作用是将物理存储器提交给用于放置数组元素的内存地址。当一个内存页面被提交时，内存表被刷新，以反映整个数组的保留区域的状态。因此，如果该区域被保留后，你用Use按钮将数组元素7和46标记为“在用”，那么该窗口就显示出图15-2所示的样子（当在4 KB计算机上运行该程序时）。

单击Clear按钮，清除带有“在用”标记的任何元素。但是这样做并不回收映射到数组元素的物理存储器，因为每个页面都包含多个结构的空间，清除一个元素并不意味着其他元素也被清除。如果内存被回收，那么其他结构中的数据就会丢失。由于单击 Clear并不影响区域的物理存储器，因此当数组元素被清除时，内存表不会被更新。

但是，当一个结构被清除时，它的 fInUse成员将被设置为FALSE。这样的设置是必要的，因为这使得无用单元收集函数能够运行通过所有的结构，并回收不再使用的内存。如果你现在尚未想到这一点，那么 Garbage Collect按钮会告诉 VMAlloc执行它的无用单元收集例程。为了简化操作，我没有在各个线程上实现无用单元收集例程。

若要展示无用单元收集函数，清除索引46上的数组元素。注意，内存表并没有改变。现在单击Garbage Collect按钮。该程序回收包含元素46的内存页面，同时内存表被更新，以反映这个变化，如图15-3所示。注意，GarbageCollect函数可以很容易地用于你自己的应用程序。我将它用于对任意大小的数据结构数组的操作，这些结构不必完全等于一个页面的大小。唯一的要求是，结构的第一个成员的值必须是BOOL，这表示该结构是否处于在用状态。

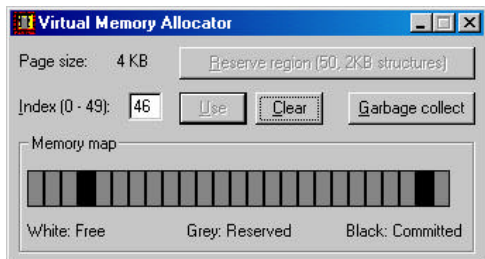


图15-2 标记数组元素后显示的窗口

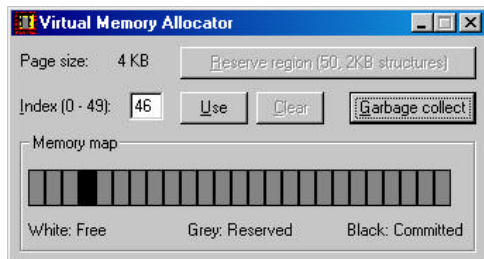


图15-3 内存表更新后显示的窗口

最后要说明的是，尽管没有直观的图形为你提供必要的信息，但是，当窗口关闭时，所有已提交的内存均被回收，保留的区域被释放。

该程序还包含另一个元素尚未加以说明。该程序必须在 3 个位置上确定区域的地址空间中的内存状态：

- 当改变索引后，该程序必须激活 Use 按钮，并停用 Clear 按钮，或者激活 Use 按钮，停用 Clear 按钮。
- 在无用单元收集函数中，在实际查看是否已经设置 fInUse 标志之前，该程序必须查看内存是否已经提交。
- 当更新内存表时，该程序必须知道哪个页面是空闲的，哪个页面已经被保留，哪个页面已经提交。

VMAAlloc 通过调用 VirtualQuery 函数来执行所有这些测试。

清单 15-1 VMAAlloc 示例应用程序



## VMAAlloc.cpp

```

/*****
Module: VMAAlloc.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
*****/

#include " ..\CmnHdr.h"      /* See Appendix A. */
#include <WindowsX.h>
#include <tchar.h>
#include "Resource.h"

////////////////////////////////////

// The number of bytes in a page on this host machine.
UINT g_uPageSize = 0;

// A dummy data structure used for the array.
typedef struct {
    BOOL fInUse;
    BYTE bOtherData[2048 - sizeof(BOOL)];
} SOMEDATA, *PSOMEDATA;

// The number of structures in the array
#define MAX_SOMEDATA      (50)

// Pointer to an array of data structures
PSOMEDATA g_pSomeData = NULL;

// The rectangular area in the window occupied by the memory map
RECT g_rcMemMap;

////////////////////////////////////

BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {

```



```
chSETDLGICONS(hwnd, IDI_VMALLOC);
```

```
// Initialize the dialog box by disabling all the nonsetup controls.
EnableWindow(GetDlgItem(hwnd, IDC_INDEXTEXT), FALSE);
EnableWindow(GetDlgItem(hwnd, IDC_INDEX), FALSE);
EnableWindow(GetDlgItem(hwnd, IDC_USE), FALSE);
EnableWindow(GetDlgItem(hwnd, IDC_CLEAR), FALSE);
EnableWindow(GetDlgItem(hwnd, IDC_GARBAGECOLLECT), FALSE);
```

```
// Get the coordinates of the memory map display.
GetWindowRect(GetDlgItem(hwnd, IDC_MEMMAP), &g_rcMemMap);
MapWindowPoints(NULL, hwnd, (LPPOINT) &g_rcMemMap, 2);
```

```
// Destroy the window that identifies the location of the memory map
DestroyWindow(GetDlgItem(hwnd, IDC_MEMMAP));

// Put the page size in the dialog box just for the user's information.
TCHAR szBuf[10];
wsprintf(szBuf, TEXT("(%d KB)"), g_uPageSize / 1024);
SetDlgItemText(hwnd, IDC_PAGESIZE, szBuf);
```

```
// Initialize the edit control.
SetDlgItemInt(hwnd, IDC_INDEX, 0, FALSE);
```

```
return(TRUE);
```

```
}
```

```
////////////////////////////////////////////////////////////////
```

```
void Dlg_OnDestroy(HWND hwnd) {
```

```
    if (g_pSomeData != NULL)
        VirtualFree(g_pSomeData, 0, MEM_RELEASE);
```

```
}
```

```
////////////////////////////////////////////////////////////////
```

```
VOID GarbageCollect(PVOID pvBase, DWORD dwNum, DWORD dwStructSize) {
```

```
    static DWORD s_uPageSize = 0;
```

```
    if (s_uPageSize == 0) {
        // Get the page size used on this CPU.
        SYSTEM_INFO si;
        GetSystemInfo(&si);
        s_uPageSize = si.dwPageSize;
```

```
}
```

```
UINT uMaxPages = dwNum * dwStructSize / g_uPageSize;
```

```

for (UINT uPage = 0; uPage < uMaxPages; uPage++) {
    BOOL fAnyAllocsInThisPage = FALSE;
    UINT uIndex      = uPage * g_uPageSize / dwStructSize;
    UINT uIndexLast = uIndex + g_uPageSize / dwStructSize;

    for (; uIndex < uIndexLast; uIndex++) {
        MEMORY_BASIC_INFORMATION mbi;
        VirtualQuery(&g_pSomeData[uIndex], &mbi, sizeof(mbi));
        fAnyAllocsInThisPage = ((mbi.State == MEM_COMMIT) &&
            * (PBOOL) ((PBYTE) pvBase + dwStructSize * uIndex));

        // Stop checking this page, we know we can't decommit it.
        if (fAnyAllocsInThisPage) break;
    }
    if (!fAnyAllocsInThisPage) {
        // No allocated structures in this page; decommit it.
        VirtualFree(&g_pSomeData[uIndexLast - 1], dwStructSize, MEM_DECOMMIT);
    }
}

}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

    UINT uIndex = 0;

    switch (id) {
        case IDCANCEL:
            EndDialog(hwnd, id);
            break;

        case IDC_RESERVE:
            // Reserve enough address space to hold the array of structures.
            g_pSomeData = (PSOMEDATA) VirtualAlloc(NULL,
                MAX_SOMEDATA * sizeof(SOMEDATA), MEM_RESERVE, PAGE_READWRITE);

            // Disable the Reserve button and enable all the other controls.
            EnableWindow(GetDlgItem(hwnd, IDC_RESERVE), FALSE);
            EnableWindow(GetDlgItem(hwnd, IDC_INDEXTEXT), TRUE);
            EnableWindow(GetDlgItem(hwnd, IDC_INDEX), TRUE);
            EnableWindow(GetDlgItem(hwnd, IDC_USE), TRUE);
            EnableWindow(GetDlgItem(hwnd, IDC_GARBAGECOLLECT), TRUE);

            // Force the index edit control to have the focus.
            SetFocus(GetDlgItem(hwnd, IDC_INDEX));

            // Force the memory map to update
            InvalidateRect(hwnd, &g_rcMemMap, FALSE);
            break;

        case IDC_INDEX:
            if (codeNotify != EN_CHANGE)
                break;
    }
}

```

```

uIndex = GetDlgItemInt(hwnd, id, NULL, FALSE);
if ((g_pSomeData != NULL) && chINRANGE(0, uIndex, MAX_SOMEDATA - 1)) {
    MEMORY_BASIC_INFORMATION mbi;
    VirtualQuery(&g_pSomeData[uIndex], &mbi, sizeof(mbi));
    BOOL fOk = (mbi.State == MEM_COMMIT);
    if (fOk)
        fOk = g_pSomeData[uIndex].fInUse;

    EnableWindow(GetDlgItem(hwnd, IDC_USE), !fOk);
    EnableWindow(GetDlgItem(hwnd, IDC_CLEAR), fOk);

} else {
    EnableWindow(GetDlgItem(hwnd, IDC_USE), FALSE);
    EnableWindow(GetDlgItem(hwnd, IDC_CLEAR), FALSE);
}
break;

case IDC_USE:
    uIndex = GetDlgItemInt(hwnd, IDC_INDEX, NULL, FALSE);
    if (chINRANGE(0, uIndex, MAX_SOMEDATA - 1)) {

        // NOTE: New pages are always zeroed by the system
        VirtualAlloc(&g_pSomeData[uIndex], sizeof(SOMEDATA),
            MEM_COMMIT, PAGE_READWRITE);

        g_pSomeData[uIndex].fInUse = TRUE;

        EnableWindow(GetDlgItem(hwnd, IDC_USE), FALSE);
        EnableWindow(GetDlgItem(hwnd, IDC_CLEAR), TRUE);

        // Force the Clear button control to have the focus.
        SetFocus(GetDlgItem(hwnd, IDC_CLEAR));

        // Force the memory map to update
        InvalidateRect(hwnd, &g_rcMemMap, FALSE);
    }
    break;

case IDC_CLEAR:
    uIndex = GetDlgItemInt(hwnd, IDC_INDEX, NULL, FALSE);
    if (chINRANGE(0, uIndex, MAX_SOMEDATA - 1)) {
        g_pSomeData[uIndex].fInUse = FALSE;
        EnableWindow(GetDlgItem(hwnd, IDC_USE), TRUE);
        EnableWindow(GetDlgItem(hwnd, IDC_CLEAR), FALSE);

        // Force the Use button control to have the focus.
        SetFocus(GetDlgItem(hwnd, IDC_USE));
    }
    break;

case IDC_GARBAGECOLLECT:
    GarbageCollect(g_pSomeData, MAX_SOMEDATA, sizeof(SOMEDATA));

    // Force the memory map to update
    InvalidateRect(hwnd, &g_rcMemMap, FALSE);

```

```
break;
}
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////

void Dlg_OnPaint(HWND hwnd) {    // Update the memory map

    PAINTSTRUCT ps;
    BeginPaint(hwnd, &ps);

    UINT uMaxPages = MAX_SOMEDATA * sizeof(SOMEDATA) / g_uPageSize;
    UINT uMemMapWidth = g_rcMemMap.right - g_rcMemMap.left;

    if (g_pSomeData == NULL) {

        // The memory has yet to be reserved.
        Rectangle(ps.hdc, g_rcMemMap.left, g_rcMemMap.top,
            g_rcMemMap.right - uMemMapWidth % uMaxPages, g_rcMemMap.bottom);

    } else {

        // Walk the virtual address space, painting the memory map
        for (UINT uPage = 0; uPage < uMaxPages; uPage++) {

            UINT uIndex = uPage * g_uPageSize / sizeof(SOMEDATA);
            UINT uIndexLast = uIndex + g_uPageSize / sizeof(SOMEDATA);
            for (; uIndex < uIndexLast; uIndex++) {

                MEMORY_BASIC_INFORMATION mbi;
                VirtualQuery(&g_pSomeData[uIndex], &mbi, sizeof(mbi));

                int nBrush = 0;
                switch (mbi.State) {
                    case MEM_FREE:      nBrush = WHITE_BRUSH; break;
                    case MEM_RESERVE:   nBrush = GRAY_BRUSH; break;
                    case MEM_COMMIT:    nBrush = BLACK_BRUSH; break;
                }

                SelectObject(ps.hdc, GetStockObject(nBrush));
                Rectangle(ps.hdc,
                    g_rcMemMap.left + uMemMapWidth / uMaxPages * uPage,
                    g_rcMemMap.top,
                    g_rcMemMap.left + uMemMapWidth / uMaxPages * (uPage + 1),
                    g_rcMemMap.bottom);

            }

        }

    }

    EndPaint(hwnd, &ps);
}
```



```

INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

    switch (uMsg) {
        chHANDLE_DLGMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
        chHANDLE_DLGMSG(hwnd, WM_COMMAND,    Dlg_OnCommand);
        chHANDLE_DLGMSG(hwnd, WM_PAINT,      Dlg_OnPaint);
        chHANDLE_DLGMSG(hwnd, WM_DESTROY,    Dlg_OnDestroy);
    }
    return(FALSE);
}

/////////////////////////////////////////////////////////////////

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, LPTSTR pszCmdLine, int) {

    // Get the page size used on this CPU.
    SYSTEM_INFO si;
    GetSystemInfo(&si);
    g_uPageSize = si.dwPageSize;

    DialogBox(hinstExe, MAKEINTRESOURCE(IDD_VMALLOC), NULL, Dlg_Proc);
    return(0);
}

///////////////////////////////////////////////////////////////// End of File ///////////////////////////////////////////////////////////////////

```

## VMAlloc.rc

```

//Microsoft Developer Studio generated resource script.
//
#include "Resource.h"

#define APSTUDIO_READONLY_SYMBOLS
/////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

/////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

/////////////////////////////////////////////////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif // _WIN32

```

```

#ifdef APSTUDIO_INVOKED
////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "Resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END

#endif    // APSTUDIO_INVOKED

////////////////////////////////////
//
// Dialog

IDD_VMALLOC_DIALOG DISCARDABLE 15, 24, 224, 97
STYLE WS_MINIMIZEBOX | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Virtual Memory Allocator"
FONT 8, "MS Sans Serif"
BEGIN
    LTEXT                "Page size:", IDC_STATIC, 4, 6, 34, 8
    CONTROL               "16 KB", IDC_PAGESIZE, "Static", SS_LEFTNOWORDWRAP |
        SS_NOPREFIX | WS_GROUP, 50, 6, 20, 8
    DEFPUSHBUTTON         "&Reserve region (50, 2KB structures)", IDC_RESERVE, 80, 4,
        140, 14, WS_GROUP
    LTEXT                 "&Index (0 - 49):", IDC_INDEXTEXT, 4, 26, 45, 8
    EDITTEXT              IDC_INDEX, 56, 24, 16, 12
    PUSHBUTTON            "&Use", IDC_USE, 80, 24, 32, 14
    PUSHBUTTON            "&Clear", IDC_CLEAR, 116, 24, 32, 14
    PUSHBUTTON            "&Garbage collect", IDC_GARBAGECOLLECT, 160, 24, 60, 14
    GROUPBOX              "Memory map", IDC_STATIC, 4, 42, 216, 52
    CONTROL               "", IDC_MEMMAP, "Static", SS_BLACKRECT, 8, 58, 208, 16
    LTEXT                 "White: Free", IDC_STATIC, 8, 80, 39, 8
    CTEXT                 "Grey: Reserved", IDC_STATIC, 82, 80, 52, 8
    RTEXT                 "Black: Committed", IDC_STATIC, 155, 80, 58, 8
END

```

```

////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_VMALLOC          ICON          DISCARDABLE          "VMAlloc.Ico"
#endif              // English (U.S.) resources
////////////////////////////////////

#ifdef APSTUDIO_INVOKED
////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//

////////////////////////////////////
#endif              // not APSTUDIO_INVOKED

```

## 15.6 改变保护属性

虽然实践中很少这样做，但是可以改变已经提交的物理存储器的一个或多个页面的保护属性。例如，你编写了一个用于管理链接表的代码，将它的节点存放在一个保留区域中。可以设计一些函数，以便处理该链接表，这样，它们就可以在每个函数开始运行时将已提交内存的保护属性改为 `PAGE_READWRITE`，然后在每个函数终止运行时将保护属性重新改为 `PAGE_NOACCESS`。

通过这样的设置，就能够使链接表数据不受隐藏在程序中的其他错误的影响。如果进程中的任何其他代码存在一个迷失指针，试图访问你的链接表数据，那么就会引发访问违规。当试图寻找应用程序中难以发现的错误时，利用保护属性是极其有用的。

若要改变内存页面的保护属性，可以调用 `VirtualProtect` 函数：

```

BOOL VirtualProtect(
    PVOID pvAddress,
    SIZE_T dwSize,
    DWORD flNewProtect,
    PDWORD pflOldProtect);

```

这里的 `pvAddress` 参数指向内存的基地址（它必须位于进程的用户方式分区中），`dwSize` 参数用于指明你想要改变保护属性的字节数，而 `flNewProtect` 参数则代表 `PAGE_*` 保护属性标志中的任何一个标志，但 `PAGE_WRITECOPY` 和 `PAGE_EXECUTE_WRITECOPY` 这两个标志除外。

最后一个参数 `pflOldProtect` 是 `DWORD` 的地址，`VirtualProtect` 将用原先与 `pvAddress` 位置上的字节相关的保护属性填入该 `DWORD`。尽管许多应用程序并不需要该信息，但是必须为该参数传递一个有效地址，否则该函数的运行将会失败。

当然，保护属性是与内存的整个页面相关联的，而不是赋予内存的各个字节的。因此，如果要使用下面的代码来调用 4 KB 页面的计算机上的 `VirtualProtect` 函数，其结果是把 `PAGE_NOACCESS` 保护属性赋予内存的两个页面：

```
VirtualProtect(pvRgnBase + (3 * 1024), 2 * 1024,  
PAGE_NOACCESS, &flOldProtect);
```

Windows 98 Windows 98只支持PAGE\_READONLY和PAGE\_READWRITE两个保护属性。如果试图将页面的保护属性改为PAGE\_EXECUTE或PAGE\_EXECUTE\_READ,该页面可得到PAGE\_READONLY保护属性。同样,如果试图将页面的保护属性改为PAGE\_EXECUTE\_READWRITE,那么该页面将得到PAGE\_READWRITE保护属性。

VirtualProtect函数不能用于改变跨越不同保留区域的页面的保护属性。如果拥有相邻的保留区域并想改变这些区域中的一些页面的保护属性,那么必须多次调用VirtualProtect函数。

## 15.7 清除物理存储器的内容

Windows 98 Windows 98不支持物理存储器内容的清除。

当你修改物理存储器的各个页面的内容时,系统将尽量设法将修改的内容保存在RAM中。但是,当应用程序运行时,从.exe文件、DLL文件和/或页文件加载页面就可能需要占用系统的RAM。由于系统要查找RAM的页面,以满足当前加载页面的需求,因此系统必须将RAM的已修改页面转到系统的页文件中。

Windows 2000提供了一个特性,使得应用程序能够提高它的性能,这个特性就是对物理存储器内容进行清除。清除存储器意味着你告诉系统,内存的一个或多个页面上的数据并没有被修改。如果系统正在搜索RAM的一个页面,并且选择一个已修改的页面,系统必须将RAM的这个页面写入页文件。这个操作的速度很慢,而且会影响系统的运行性能。对于大多数应用程序来说,可以让系统将你修改了的页面保留在系统的页文件中。

然而,有些应用程序使用内存的时间很短,然后就不再要求保留该内存的内容。为了提高性能,应用程序可以告诉系统不要将内存的某些页面保存在系统的页文件中。这是应用程序告诉系统数据页面尚未修改的一个基本方法。因此,如果系统选择将RAM的页面用于别的目的,那么该页面的内容就不必保存在页文件中,从而可以提高应用程序的运行性能。若要清除内存的内容,应用程序可以调用VirtualAlloc函数,在第三个参数中传递MEM\_RESET标志。

如果在调用VirtualAlloc函数时引用的页面位于页文件中,系统将删除这些页面。下次应用程序访问内存时,便使用最初被初始化为0的RAM页面。如果清除了当前RAM中的页面内容,那么它们将被标上未修改的标记,这样它们将永远不会被写入页文件。注意,虽然RAM页面的内容没有被置0,但是不应该继续从内存的该页面读取数据。如果系统不需要RAM的这个页面,它将包含其原始内容。但是如果系统需要RAM的这个页面,系统就可以提取该页面。然后当你试图访问该页面的内容时,系统将给你一个已经删除内容的新页面。由于你无法控制这个行为特性,因此,当清除页面的内容后,你必须假定该页面的内容是无用信息。

当清除内存的内容时,有两件事情必须记住。首先,当调用VirtualAlloc函数时,基地址通常圆整为一个页面边界的值,而字节数则圆整为一个页面的整数。当清除页面的内容时,用这种办法圆整基地址和字节数是非常危险的,因此,当传递MEM\_RESET标志时,VirtualAlloc将按反方向对这些值进行圆整。例如,有下面这个代码:

```
PINT pData = (PINT) VirtualAlloc(NULL, 1024,  
MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);  
pn[0] = 100;  
pn[1] = 200;  
VirtualAlloc((PVOID) pData, sizeof(int), MEM_RESET, PAGE_RE
```



该代码提交了一个内存页面，然后指明前 4 个字节（sizeof(int)）不再需要，可以被清除。但是，与所有内存操作一样，一切操作都必须在页面边界上并按页面增量来进行。结果，对上面这个内存清除函数的调用会失败（VirtualAlloc函数返回NULL）。为什么呢？因为当将MEM\_RESET传递给VirtualAlloc时，传递给函数的基地址被圆整为页面边界的值，字节数圆整为一个页面的整数。这样做是为了确保重要的数据不会丢失。在上面的例子中，字节数圆整后得出的是0，而清除0字节是不合法的。

要记住的第二件事情是，MEM\_RESET标志始终必须自己单独使用，不能用OR将它与其他标志连接起来使用。下面这个函数调用总是失败的，它返回的是NULL。

```
PVOID pvMem = VirtualAlloc(NULL, 1024,  
    MEM_RESERVE | MEM_COMMIT | MEM_RESET, PAGE_READWRITE);
```

将MEM\_RESET标志与任何其他标志连接起来确实没有任何意义。

最后请注意，带有MEM\_RESET标志的VirtualAlloc函数要求传递一个有效的页面保护属性，即使该函数不使用这个值，也必须传递该值。

### MemReset示例应用程序

清单15-2列出的MemReset应用程序（“15 MemReset.exe”）显示了MEM\_RESET标志是如何运行的。该应用程序的源代码和资源文件位于本书所附光盘上的15-MemReset目录下。

MemReset.cpp代码执行的第一项操作是保留和提交一个物理存储器的区域。由于传递给VirtualAlloc函数的页面大小是1024，因此系统将自动把这个值圆整为系统的页面大小。这时，使用lstrcpy函数将一个字符串拷贝到该缓存，导致页面的内容被修改。如果系统后来决定它需要我们的数据页面占用的RAM页面，那么系统将首先将我们页面中的数据写入系统的页文件。当我们的应用程序后来试图访问该数据时，系统将自动把该页面从页文件重新加载到RAM的另一个页面，这样我们就能够成功地访问该数据。

当该字符串被写入内存页面后，该代码便向用户显示一个消息框，询问是否需要在早些时候访问这些数据。如果用户选定No按钮，那么该代码就迫使操作系统认为该页面中的数据没有通过调用VirtualAlloc函数并传递MEM\_RESET标志而被修改。

为了说明内存的内容已经被清除，我们必须对系统的RAM提出大量的使用需求。若要进行这项操作，可以分3步来进行：

- 1) 调用GlobalMemoryStatus函数，获取计算机中RAM的总容量。
- 2) 调用VirtualAlloc函数，提交该数量的内存。这项操作的运行速度非常快，因为在进程试图访问页面之前，系统实际上并不为该内存分配RAM。
- 3) 调用ZeroMemory函数，使新提交的页面可以被访问。这将给系统的RAM带来沉重的负担，导致当前正在RAM中的某些页面被写入页文件。

如果用户指明该数据将在以后被访问，那么该数据将不被清除，并且在以后访问该数据时将数据转入RAM。但是，如果用户指明以后将不再访问该数据，那么数据将被清除，并且系统不把数据写入页文件，这样就可以提高应用程序的运行性能。

当ZeroMemory函数返回时，代码将对数据页面的内容与原先写入页面的字符串进行比较。如果数据没有被清除，那么可以保证它们的内容是相同的。如果数据页面已经被清除，那么它们的内容可能相同，也可能不同。在MemReset程序中，它们的内容决不可能相同，因为RAM中的所有页面均被强制写入页文件中。但是，如果这个伪区域小于计算机中的RAM总容量，那么原始内容有可能仍然在RAM中。如前所述，操作时务必小心。

## 清单15-2 MemReset示例应用程序

**MemReset.cpp**

```

/*****
Module:  MemReset.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
*****/

#include "..\CmnHdr.h"      /* See Appendix A. */
#include <tchar.h>

////////////////////////////////////

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, LPTSTR pszCmdLine, int) {

    chWindows9xNotAllowed();

    TCHAR szAppName[] = TEXT("MEM_RESET tester");
    TCHAR szTestData[] = TEXT("Some text data");

    // Commit a page of storage and modify its contents.
    LPTSTR pszData = (LPTSTR) VirtualAlloc(NULL, 1024,
        MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);
    lstrcpy(pszData, szTestData);

    if (MessageBox(NULL, TEXT("Do you want to access this data later?"),
        szAppName, MB_YESNO) == IDNO) {

        // We want this page of storage to remain in our process but the
        // contents aren't important to us anymore.
        // Tell the system that the data is not modified.

        // Note: Because MEM_RESET destroys data, VirtualAlloc rounds
        // the base address and size parameters to their safest range.
        // Here is an example:
        //     VirtualAlloc(pvData, 5000, MEM_RESET, PAGE_READWRITE)
        // resets 0 pages on CPUs where the page size is greater than 4 KB
        // and resets 1 page on CPUs with a 4-KB page. So that our call to
        // VirtualAlloc to reset memory below always succeeds, VirtualQuery
        // is called first to get the exact region size.
        MEMORY_BASIC_INFORMATION mbi;
        VirtualQuery(pszData, &mbi, sizeof(mbi));
        VirtualAlloc(pszData, mbi.RegionSize, MEM_RESET, PAGE_READWRITE);
    }

    // Commit as much storage as there is physical RAM.
    MEMORYSTATUS mst;
    GlobalMemoryStatus(&mst);
}

```

```

PVOID pvDummy = VirtualAlloc(NULL, mst.dwTotalPhys,
    MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);

// Touch all the pages in the dummy region so that any
// modified pages in RAM are written to the paging file.
ZeroMemory(pvDummy, mst.dwTotalPhys);

// Compare our data page with what we originally wrote there.
if (lstrcmp(pszData, szTestData) == 0) {

    // The data in the page matches what we originally put there.
    // ZeroMemory forced our page to be written to the paging file.
    MessageBox(NULL, TEXT("Modified data page was saved."),
        szAppName, MB_OK);
} else {

    // The data in the page does NOT match what we originally put there
    // ZeroMemory didn't cause our page to be written to the paging file
    MessageBox(NULL, TEXT("Modified data page was NOT saved."),
        szAppName, MB_OK);
}
return(0);
}

//////////////////////////////////// End of File //////////////////////////////////////

```

## 15.8 地址窗口扩展——适用于Windows 2000

随着时间的推移，应用程序需要的内存越来越多。对于服务器应用程序来说，情况更是如此。由于越来越多的客户机对服务器提出访问请求，服务器的运行性能就会降低。为了提高运行性能，服务器应用程序必须在 RAM 中保存更多的数据，并且缩小磁盘的页面。其他类别的应用程序，比如数据库、工程设计和科学应用程序，也需要具备处理大块内存的能力。对于所有这些应用程序来说，32 位地址空间是不够使用的。

为了满足这些应用程序的需要，Windows 2000 提供了一个新特性。称为地址窗口扩展（AWE）。Microsoft 创建 AWE 是出于下面两个目的：

- 允许应用程序对从来不在操作系统与磁盘之间交换的 RAM 进行分配。
- 允许应用程序访问的 RAM 大于进程的地址空间。

AWE 基本上为应用程序提供了一种分配一个或多个 RAM 块的手段。当分配 RAM 块时，在进程的地址空间中是看不见这些 RAM 块的。后来，应用程序（使用 VirtualAlloc 函数）保留一个地址空间区域，这个区域就成为地址窗口。这时应用程序调用一个函数，每次将一个 RAM 块赋予该地址窗口。将一个 RAM 块赋予地址窗口的速度是非常快的（通常只需要几个毫秒）。

显然，通过单个地址窗口，每次只能访问一个 RAM 块。这使得你的代码很难实现，因为，当需要时，必须要在你的代码中显式调用函数，才能将不同的 RAM 块赋予地址窗口。

下面的代码显示了如何使用 AWE 的方法：

```

// First, reserve a 1MB region for the address window
ULONG_PTR ulRAMBytes = 1024 * 1024
PVOID pvWindow = VirtualAlloc(NULL, ulRAMBytes,

```

```

MEM_RESERVE | MEM_PHYSICAL, PAGE_READWRITE);

// Get the number of bytes in a page for this CPU platform
SYSTEM_INFO sinfo;
GetSystemInfo(&sinfo);

// Calculate the required number of RAM pages for the
// desired number of bytes
ULONG_PTR ulRAMPages = ulRAMBytes / sinfo.dwPageSize

// Allocate array for RAM page's page frame numbers
ULONG_PTR aRAMPages[ulRAMPages];
// Allocate the pages of RAM (requires Lock Pages in Memory user right)
AllocateUserPhysicalPages(
    GetCurrentProcess(), // Allocate the storage for our process
    &ulRAMPages,          // Input: # of RAM pages, Output: # pages allocated
    aRAMPages);          // Output: Opaque array indicating pages allocated

// Assign the RAM pages to our window
MapUserPhysicalPages(pvWindow, // The address of the address window
    ulRAMPages,                // Number of entries in array
    aRAMPages);                // Array of RAM pages

// Access the RAM pages via the pvWindow virtual address
:
:

// Free the block of RAM pages
FreeUserPhysicalPages(
    GetCurrentProcess(), // Free the RAM allocated for our process
    &ulRAMPages,          // Input: # of RAM pages, Output: # pages freed
    aRAMPages);          // Input: Array indicating the RAM pages to free

// Destroy the address window
VirtualFree(pvWindow, 0, MEM_RELEASE);

```

如你所见，AWE的使用非常简单。现在我们要说明该代码的一些有意思的情况。

对VirtualAlloc函数的调用保留了一个1 MB的地址窗口。通常该地址窗口要大得多。你必须选定一个适合于应用程序需要的RAM块大小的窗口大小。当然，你创建的最大窗口取决于你的地址空间中可用的最大相邻空闲地址块。MEM\_RESERVE标志用于指明我正在保留一个地址区域。MEM\_PHYSICAL标志用于指明这个区域最终将受RAM物理存储器的支持。AWE的局限性是，映射到地址窗口的所有内存必须是可读的和可写入的，因此PAGE\_READWRITE是可以传递VirtualAlloc函数的唯一有效的保护属性。此外，不能使用VirtualProtect函数来修改这个保护属性。

RAM物理存储器的分配是非常简单的，只需要调用AllocateUserPhysicalPages：

```

BOOL AllocateUserPhysicalPages(
    HANDLE hProcess,
    PULONG_PTR pulRAMPages,
    PULONG_PTR aRAMPages);

```

该函数负责分配pulRAMPages参数指明的值设定的RAM页面的数量，并且将这些页面赋予hProcess参数标识的进程。



每个RAM页面由操作系统赋予一个页框号。当系统选择供分配用的 RAM页面时，它就将每个RAM页面的页框号填入 aRAMPages参数指向的数组。页框号本身对应用程序没有任何用处，不应该查看该数组的内容，并且肯定不应该修改该数组中的任何一个值。注意，你不知道哪些RAM页面已经被分配给该内存块，也不应该去关注这个情况。当地址窗口显示 RAM块中的页面时，它们显示为一个相邻的内存块。这使得 RAM非常便于使用，并且使你可以不必了解系统内部的运行情况。

该函数返回时，pulRAMPages参数中的值用于指明该函数成功地分配的页面数量。这个数量通常与传递给函数的值是相同的，但是它也可能是个较小的值。

只有拥有页面的进程才能使用已经分配的 RAM页面，AWE不允许RAM页面被映射到另一个进程的地址空间。因此不能在进程之间共享RAM块。

注意 当然，物理RAM是一种非常宝贵的资源，并且应用程序只能分配尚未指定用途的RAM。应该非常节省地使用 AWE，否则你的进程和其他进程将会过分地在内存与磁盘之间进行页面的交换，从而严重影响系统的运行性能。此外，如果可用 RAM的数量比较少，也会对系统创建新进程、线程和其他资源的能力产生不利的影响。应用程序可以使用GlobalMemoryStatusEx函数来监控物理存储器的使用情况。

为了保护RAM的分配，AllocateUserPhysicalPages函数要求调用者拥有Lock Pages in Memory（锁定内存中的页面）的用户权限，并且已经激活该权限，否则该函数的运行将会失败。按照默认设置，该权限不被赋予任何用户或用户组。该权限被赋予 Local System（本地系统）帐户，它通常用于服务程序。如果想要运行一个调用 AllocateUserPhysicalPages函数的交互式应用程序，那么管理员必须在你登录和运行应用程序之前为你赋予该权限。

Windows 2000 在Windows 2000中，可以执行下列步骤，打开Lock Pages in Memory用户权限：

- 1) 单击Start按钮，选定Run菜单项，打开Computer Management MMC控制台。在Run框中，键入“Compmgmt.msc/a”，再单击OK按钮。

- 2) 如果在左边的窗格中没有显示 Local Computer Policy（本地计算机政策）项，那么在控制台菜单中选定 Add/Remove Snap-ins（添加/删除咬接项（snap-in））。在Standalone选项卡上，从Snap-ins Added To（咬接项添加到）组合框中选定 Computer Management(local)。现在单击Add按钮，显示Add Standalone Snap-in（添加独立咬接项）对话框。从Available Standalone Snap-ins（可用独立咬接项）中选定Group Policy（组政策）。并单击Add按钮。在Select Group Policy Object（选定组政策对象）对话框中，保留默认值，并单击Finish按钮。单击Add Standalone Snap-in对话框上的Close按钮，再单击Add/Remove Snap-in对话框上的OK按钮。这时，在Computer Management控制台的左窗格中就可以看到Local Computer Policy项。

- 3) 在控制台的左窗格中，双击下列项目，将它们展开：Local Computer Policy(本地计算机政策)、Computer Configuration(计算机配置)、Windows Settings（窗口设置）、Security Settings（安全性设置）和Local Policy（本地政策）。然后选定User Rights Assignment（用户权限赋值）项。

- 4) 在右窗格中，选定Lock Pages in Memory属性。

- 5) 从Action（操作）菜单中选定Security，显示Lock Pages in Memory对话框，单

击Add按钮。使用 Select Users or Group对话框，添加你想为其赋予 Lock Pages in Memory用户权限的用户和/或用户组。单击OK按钮，退出每个对话框。

当用户登录时，他将被赋予相应的用户权限。如果你只是将 Lock Pages in Memory权限赋予你自己，那么必须在该权限生效前退出并重新登录。

现在我们已经创建了地址窗口并且分配了一个 RAM块，可以通过调用 MapUserPhysical Pages函数将该RAM块赋予该地址窗口：

```
BOOL MapUserPhysicalPages(  
    PVOID pvAddressWindow,  
    ULONG_PTR uIRAMPages,  
    PULONG_PTR aRAMPages);
```

第一个参数 pvAddressWindow用于指明地址窗口的虚拟地址，第二和第三个参数 uIRAMPages和aRAMPages用于指明该地址窗口中可以看到多少个RAM页面以及哪些页面可以看到。如果窗口小于你试图映射的页面数量，那么函数运行就会失败。Microsoft设置这个函数的主要目的是使它运行起来非常快。一般来说， MapUserPhysicalPages函数能够在几个微秒内映射该RAM块。

注意 也可以调用 MapUserPhysicalPages函数来取消对当前RAM块的分配，方法是为 aRAMPages参数传递NULL。下面是它的一个例子：

```
// First, reserve a 1MB region for the address window  
ULONG_PTR uIRAMBytes = 1024 * 1024
```

一旦RAM块被分配给地址窗口，只需要引用相对于地址窗口的基地址（在我的示例代码中是pvWindow）的虚拟地址，就可以很容易地访问该RAM内存。

当不再需要RAM块时，应该调用FreeUserPhysicalPages函数将它释放：

```
BOOL FreeUserPhysicalPages(  
    HANDLE hProcess,  
    PULONG_PTR puIRAMPages,  
    PULONG_PTR aRAMPages);
```

第一个参数hProcess用于指明哪个进程拥有你试图释放的RAM页面。第二和第三个参数用于指明你要释放多少个页面和这些页面的页框号。如果该 RAM块目前已经被映射到该地址窗口，那么它将被取消映射并被释放。

最后，为了彻底清除页面，我仅仅调用了 VirtualFree函数，传递窗口的虚拟基地址，为区域大小传递0，再传递MEM\_RELEASE，将地址窗口释放掉。

我的简单的示例代码创建了单个地址窗口和单个 RAM块。这使得我的应用程序能够访问没有与磁盘进行数据交换的RAM。但是，应用程序也能够创建若干个地址窗口，并且可以分配若干个RAM块。虽然这些RAM块可以分配给任何一个地址窗口，但是系统不允许单个RAM块同时出现在两个地址窗口中。

64位Windows 2000全面支持AWE。对使用AWE的32位应用程序进行移植是很容易和简单的。不过对于64位应用程序来说，AWE的用处比较小，因为进程的地址空间太大了。但是AWE仍然是有用的，因为它使得应用程序能够分配不与磁盘进行数据交换的物理RAM。

#### AWE示例应用程序

清单15-3列出的AWE应用程序（“15-AWE.exe”）显示了如何创建多个地址窗口和如何将

不同的内存块分配给这些窗口。该应用程序的源代码和资源文件位于本书所附光盘上的 15-AWE目录下。当启动该程序时，它在内部创建两个地址窗口区域，并且分配两个 RAM 块。

首先，第一个 RAM 块用字符串“Text in Storage 0”填入，第二个 RAM 块用字符串“Text in Storage 1”填入。然后，第一个 RAM 块被赋予第一个地址窗口，第二个 RAM 块被赋予第二个地址窗口。该应用程序的窗口反映了这个情况（见图 15-4）。

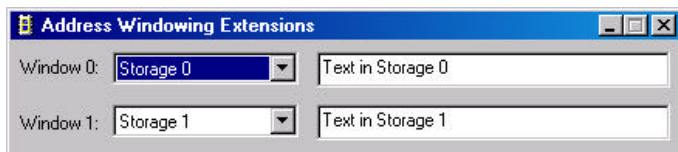


图15-4 AWE示例应用程序的窗口

使用该窗口，可以进行一些试验。首先，可以使用每个地址窗口的组合框将 RAM 块赋予各个地址窗口。该组合框也提供了一个 No Storage 选项，用于从地址窗口中撤消对任何内存的映射。其次，编辑窗口中的文本将会更新地址窗口中当前选定的 RAM 块。

如果试图将一个 RAM 块同时赋予两个地址窗口，就会出现图 15-5 所示的消息，因为 AWE 不支持这种操作。



图15-5 消息显示

该示例应用程序的源代码是非常清楚的。为了使 AWE 的操作更加容易些，我创建了 3 个 C++ 类，它们包含在 AddrWindows.h 文件中。第一个类是 CSystemInfo，它是 GetSystemInfo 函数中的一个非常简单的包装类，另外两个类均用于创建 CSystemInfo 类的实例。

第二个 C++ 类是 CAddrWindow，它用于封装一个地址窗口。Create 方法基本上用于保留一个地址窗口，Destroy 方法则用于撤消一个地址窗口，UnmapStorage 方法用于取消当前赋予地址窗口的任何 RAM 块的映像，而 PVOID 类型转换操作符方法只是用于返回地址窗口的虚拟地址。

第三个 C++ 类是 CAddrWindowStorage，它用于封装一个可以赋予 CAddrWindow 对象的 RAM 块。Allocate 方法用于激活 Lock Pages in Memory 用户权限，并设法分配 RAM 块，然后停用该用户权限。Free 方法用于释放 RAM 块。HowManyPagesAllocated 方法返回已经成功地分配的页面数量。MapStorage 和 UnmapStorage 方法则用于与 CAddrWindow 对象之间进行 RAM 块的映射和取消映射。

使用这些 C++ 类，可以使示例应用程序的实现变得容易得多。该示例应用程序创建了两个 CAddrWindow 对象和两个 CAddrWindowStorage 对象。代码的其余部分只是用于在正确的时间为正确的对象调用正确的方法而已。

清单15-3 AWE示例应用程序



AWE.cpp

/\* \*\*\*\* \*/

```

/*****
Module: AWE.cpp
Notices: Copyright (c) 2000 Jeffrey Richter
*****/

```

```

#include "..\CmnHdr.h"      /* See Appendix A. */
#include <Windowsx.h>
#include <tchar.h>
#include "AddrWindow.h"
#include "Resource.h"

////////////////////////////////////

```

```

CAddrWindow g_aw[2];           // 2 memory address windows
CAddrWindowStorage g_aws[2];    // 2 storage blocks
const ULONG_PTR g_nChars = 1024; // 1024 character buffers

```

```

////////////////////////////////////

```

```

BOOL Dlg_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {

    chSETDLGICONS(hwnd, IDI_AWE);

    // Create the 2 memory address windows
    chVERIFY(g_aw[0].Create(g_nChars * sizeof(TCHAR)));
    chVERIFY(g_aw[1].Create(g_nChars * sizeof(TCHAR)));

    // Create the 2 storage blocks
    if (!g_aws[0].Allocate(g_nChars * sizeof(TCHAR))) {
        chFAIL("Failed to allocate RAM.\nMost likely reason: "
            "you are not granted the Lock Pages in Memory user right.");
    }
    chVERIFY(g_aws[1].Allocate(g_nChars * sizeof(TCHAR)));

    // Put some default text in the 1st storage block
    g_aws[0].MapStorage(g_aw[0]);
    lstrcpy((PSTR) (PVOID) g_aw[0], TEXT("Text in Storage 0"));

    // Put some default text in the 2nd storage block
    g_aws[1].MapStorage(g_aw[0]);
    lstrcpy((PSTR) (PVOID) g_aw[0], TEXT("Text in Storage 1"));

    // Populate the dialog box controls
    for (int n = 0; n <= 1; n++) {
        // Set the combo box for each address window
        int id = ((n == 0) ? IDC_WINDOW0STORAGE : IDC_WINDOW1STORAGE);
        HWND hwndCB = GetDlgItem(hwnd, id);
        ComboBox_AddString(hwndCB, TEXT("No storage"));
        ComboBox_AddString(hwndCB, TEXT("Storage 0"));
        ComboBox_AddString(hwndCB, TEXT("Storage 1"));

        // Window 0 shows Storage 0, Window 1 shows Storage 1
        ComboBox_SetCurSel(hwndCB, n + 1);
        FORWARD_WM_COMMAND(hwnd, id, hwndCB, CBN_SELCHANGE, SendMessage);
    }
}

```

```

        Edit_LimitText(GetDlgItem(hwnd,
            (n == 0) ? IDC_WINDOW0TEXT : IDC_WINDOW1TEXT), g_nChars);
    }

    return(TRUE);
}

/////////////////////////////////////////////////////////////////

voidDlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

    switch (id) {

    case IDCANCEL:
        EndDialog(hwnd, id);
        break;

    case IDC_WINDOW0STORAGE:
    case IDC_WINDOW1STORAGE:
        if (codeNotify == CBN_SELCHANGE) {

            // Show different storage in address window
            int nWindow = ((id == IDC_WINDOW0STORAGE) ? 0 : 1);
            int nStorage = ComboBox_GetCurSel(hwndCtl) - 1;

            if (nStorage == -1) { // Show no storage in this window
                chVERIFY(g_aw[nWindow].UnmapStorage());
            } else {
                if (!g_aws[nStorage].MapStorage(g_aw[nWindow])) {
                    // Couldn't map storage in window
                    chVERIFY(g_aw[nWindow].UnmapStorage());
                    ComboBox_SetCurSel(hwndCtl, 0); // Force "No storage"
                    chMB("This storage can be mapped only once.");
                }
            }

            // Update the address window's text display
            HWND hwndText = GetDlgItem(hwnd,
                ((nWindow == 0) ? IDC_WINDOW0TEXT : IDC_WINDOW1TEXT));
            MEMORY_BASIC_INFORMATION mbi;
            VirtualQuery(g_aw[nWindow], &mbi, sizeof(mbi));
            // Note: mbi.State == MEM_RESERVE if no storage is in address window
            EnableWindow(hwndText, (mbi.State == MEM_COMMIT));
            Edit_SetText(hwndText, IsWindowEnabled(hwndText)
                ? (PCSTR) (PVOID) g_aw[nWindow] : TEXT("(No storage)"));
        }
        break;

    case IDC_WINDOW0TEXT:
    case IDC_WINDOW1TEXT:
        if (codeNotify == EN_CHANGE) {
            // Update the storage in the address window
            int nWindow = ((id == IDC_WINDOW0TEXT) ? 0 : 1);
            Edit_GetText(hwndCtl, (PSTR) (PVOID) g_aw[nWindow], g_nChars);
        }
    }
}

```

```

        break;
    }
}

////////////////////////////////////

INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

    switch (uMsg) {
        case WM_INITDIALOG: Dlg_OnInitDialog(hwnd, wParam, lParam); break;
        case WM_COMMAND:   Dlg_OnCommand(hwnd, wParam, lParam); break;
    }

    return(FALSE);
}

////////////////////////////////////

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

    chWindows2000Required();

    DialogBox(hinstExe, MAKEINTRESOURCE(IDD_AWE), NULL, Dlg_Proc);
    return(0);
}

//////////////////////////////////// End of File //////////////////////////////////////

```

## AWE.rc

```

//Microsoft Developer Studio generated resource script.
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

////////////////////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif // _WIN32

////////////////////////////////////

```



```

.....
//
// Dialog
//

IDD_AWE_DIALOG DISCARDABLE 0, 0, 288, 45
STYLE DS_SETFOREGROUND | DS_3DLOOK | DS_CENTER | WS_MINIMIZEBOX | WS_VISIBLE |
    WS_CAPTION | WS_SYSMENU
CAPTION "Address Windowing Extensions"
FONT 8, "MS Sans Serif"
BEGIN
    LTEXT                "Window 0:", IDC_STATIC, 4, 6, 35, 8
    COMBOBOX              IDC_WINDOW0STORAGE, 44, 4, 80, 58, CBS_DROPDOWNLIST |
        WS_TABSTOP
    EDITTEXT              IDC_WINDOW0TEXT, 132, 4, 152, 14, ES_AUTOHSCROLL
    LTEXT                 "Window 1:", IDC_STATIC, 4, 28, 35, 8
    COMBOBOX              IDC_WINDOW1STORAGE, 44, 25, 80, 58, CBS_DROPDOWNLIST |
        WS_TABSTOP
    EDITTEXT              IDC_WINDOW1TEXT, 132, 25, 152, 14, ES_AUTOHSCROLL
END
////////////////////////////////////
//
// DESIGNINFO
//

#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
    IDD_AWE_DIALOG
    BEGIN
        LEFTMARGIN, 7
        RIGHTMARGIN, 281
        TOPMARGIN, 7
        BOTTOMMARGIN, 38
    END
END
#endif // APSTUDIO_INVOKED

#ifdef APSTUDIO_INVOKED
////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include \"afxres.h\"\\r\\n"
    "\\0"
END

3 TEXTINCLUDE DISCARDABLE

```

```

BEGIN
    "\r\n"
    "\0"
END

#endif // APSTUDIO_INVOKED
//
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_AWE          ICON    DISCARDABLE     "AWE.ico"
#endif // English (U.S.) resources
//

#ifndef APSTUDIO_INVOKED
//
// Generated from the TEXTINCLUDE 3 resource.
//

//
#endif // not APSTUDIO_INVOKED

```

## AddrWindow.h

```

/*****
Module: AddrWindow.h
Notices: Copyright (c) 2000 Jeffrey Richter
*****/

```

```
#pragma once
```

```

//
#include "..\CmnHdr.h" /* See Appendix A. */
#include <tchar.h>

```

```

//
class CSystemInfo : public SYSTEM_INFO {
public:
    CSystemInfo() { GetSystemInfo(this); }
};
//

```

```

class CAddrWindow {
public:
    CAddrWindow() { m_pvWindow = NULL; }
    ~CAddrWindow() { Destroy(); }

    BOOL Create(SIZE_T dwBytes, PVOID pvPreferredWindowBase = NULL) {
        // Reserve address window region to view physical storage
        m_pvWindow = VirtualAlloc(pvPreferredWindowBase, dwBytes,
            MEM_RESERVE | MEM_PHYSICAL, PAGE_READWRITE);
        return(m_pvWindow != NULL);
    }

    BOOL Destroy() {
        BOOL fOk = TRUE;
        if (m_pvWindow != NULL) {
            // Destroy address window region
            fOk = VirtualFree(m_pvWindow, 0, MEM_RELEASE);
            m_pvWindow = NULL;
        }
        return(fOk);
    }

    BOOL UnmapStorage() {
        // Unmap all storage from address window region
        MEMORY_BASIC_INFORMATION mbi;
        VirtualQuery(m_pvWindow, &mbi, sizeof(mbi));
        return(MapUserPhysicalPages(m_pvWindow,
            mbi.RegionSize / sm_sinf.dwPageSize, NULL));
    }

    // Returns virtual address of address window
    operator PVOID() { return(m_pvWindow); }

private:
    PVOID m_pvWindow; // Virtual address of address window region
    static CSystemInfo sm_sinf;
};

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

CSystemInfo CAddrWindow::sm_sinf;

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

class CAddrWindowStorage {
public:
    CAddrWindowStorage() { m_ulPages = 0; m_pulUserPfnArray = NULL; }
    ~CAddrWindowStorage() { Free(); }

    BOOL Allocate(ULONG_PTR ulBytes) {
        // Allocate storage intended for an address window
    }
}

```

```

Free(); // Cleanup this object's existing address window

// Calculate number of pages from number of bytes
m_ulPages = (ulBytes + sm_sinf.dwPageSize) / sm_sinf.dwPageSize;

// Allocate array of page frame numbers
m_pulUserPfnArray = (PULONG_PTR)
    HeapAlloc(GetProcessHeap(), 0, m_ulPages * sizeof(ULONG_PTR));

BOOL fOk = (m_pulUserPfnArray != NULL);
if (fOk) {
    // The "Lock Pages in Memory" privilege must be enabled
    EnablePrivilege(SE_LOCK_MEMORY_NAME, TRUE);
    fOk = AllocateUserPhysicalPages(GetCurrentProcess(),
        &m_ulPages, m_pulUserPfnArray);
    EnablePrivilege(SE_LOCK_MEMORY_NAME, FALSE);
}
return(fOk);
}

BOOL Free() {
    BOOL fOk = TRUE;
    if (m_pulUserPfnArray != NULL) {
        fOk = FreeUserPhysicalPages(GetCurrentProcess(),
            &m_ulPages, m_pulUserPfnArray);
        if (fOk) {
            // Free the array of page frame numbers
            HeapFree(GetProcessHeap(), 0, m_pulUserPfnArray);
            m_ulPages = 0;
            m_pulUserPfnArray = NULL;
        }
    }
    return(fOk);
}

ULONG_PTR HowManyPagesAllocated() { return(m_ulPages); }

BOOL MapStorage(CAddrWindow& aw) {
    return(MapUserPhysicalPages(aw,
        HowManyPagesAllocated(), m_pulUserPfnArray));
}

BOOL UnmapStorage(CAddrWindow& aw) {
    return(MapUserPhysicalPages(aw,
        HowManyPagesAllocated(), NULL));
}

private:
    static BOOL EnablePrivilege(PCTSTR pszPrivName, BOOL fEnable = TRUE) {
        BOOL fOk = FALSE;    // Assume function fails
        HANDLE hToken;

        // Try to open this process's access token
        if (OpenProcessToken(GetCurrentProcess(),

```

```
TOKEN_ADJUST_PRIVILEGES, &hToken)) {  
  
    // Attempt to modify the "Lock pages in Memory" user right  
    TOKEN_PRIVILEGES tp = { 1 };  
    LookupPrivilegeValue(NULL, pszPrivName, &tp.Privileges[0].Luid);  
    tp.Privileges[0].Attributes = fEnable ? SE_PRIVILEGE_ENABLED : 0;  
    AdjustTokenPrivileges(hToken, FALSE, &tp, sizeof(tp), NULL, NULL);  
  
    fOk = (GetLastError() == ERROR_SUCCESS);  
    CloseHandle(hToken);  
}  
return(fOk);  
}  
  
private:  
    ULONG_PTR m_ulPages;           // Number of storage pages  
    PULONG_PTR m_pulUserPfnArray; // Page frame number array  
  
private:  
    static CSystemInfo sm_sinf;  
};  
  
/////////////////////////////////////  
  
CSystemInfo CAddrWindowStorage::sm_sinf;  
  
////////////////////////////////////// End of File //////////////////////////////////////
```