内存管理器提供的服务

锁定到内存

页面可通过如下两种方式锁定到内存中。

1. Windows应用程序调用 VirtualLock 函数锁定自己进程工作集中的页面。使用这种机制锁定的页面会始终驻留在物理内存中，直到被明确解除锁定或锁定它的进程被终止。一个进程可以锁定的页面数量最大值无法超过“其最小工作集大小减去 8个页面”如果进程需要锁定更多页面，可以通过 SetProcessWorkingSetSizeEx 函数增加自己的工作集最小值。
2. 设备驱动程序可以调用 MmProbeAndLockPages、MmLockPagableCodeSection、MmLockPagableDataSection或MmLockPagableSectionByHandle 这几个内核模式函数来实现锁定。使用这种机制锁定的页面会始终驻留在物理内存中，直到被明确解除锁定。上述后3个API 可以对能够锁定到内存中的页面数量不加限制，因为在驱动程序首次加载时就已经获得了可用的驻留页面数量。这也可以确保不会因为超量锁定导致系统崩溃。对于第一个 API，必须首先获取用量配额，否则该 API会返回失败状态。

共享内存和映射文件

共享内存可以被定义为一种能被超过一个进程可见，或存在于超过一个进程虚拟地址空间中的内存。举例来说，如果两个进程使用了同一个 DLL，那么此时更合理的做法是将引用该 DLL 的代码页面加载到物理内存中，但只加载一次，并在所有映射该DLL的进程之间共享这些页面。

每个进程依然可以通过自己的私有内存区域存储私有数据，但 DLL代码和未修改的数据页面依然可以放心共享。正如下文将要解释的那样，此类共享可以自动进行，因为可执行映像[EXE 和DLL 文件，以及诸如屏幕保护程序(Screen Saver，SCR)等其他多种类型的内容，虽然名称不同，但本质上依然是 DLL] 的代码页面将以“仅执行”的方式映射，可写页面则会映射为写入时复制

内存管理器中用于实现共享内存的底层原语名为区域对象 (section object)，在 Window API中它是通过文件映射对象的形式暴露的。一个区域可被一个或多个进程打开

若要创建区域对象，请调用 Windows 的 CreateFileMapping、CreateFileMappingFromApp或CreateFileMappingNuma(Ex)函数，指定要映射到的、之前打开的文件句柄 (对于页面文件支撑的区域，可指定 INVALID\_HANDLE\_VALUE，并提供可选的名称和安全描述符。如果该区域有名称，其他进程即可使用 OpenFileMapping 或CreateFileMapping\*函数打开它。或者可以通过句柄的继承(打开或创建句柄时指定该句柄是可继承的)，或通过句柄的复制(使用 DuplicateHandle) 允许对区域对象的访问。设备驱动程序也可以通过ZwOpenSection、ZwMapViewOfSection和 ZwUnmapViewOfSection函数操作区域对象。

区域对象可以引用比进程地址空间大很多的文件。(如果区域对象由页面文件支撑，那么页面文件/物理内存中必须有足够的空间来包含该对象。) 若要访问一个非常大的区域对象，进程只能映射该区域对象中自己实际需要的部分(这个“部分”也可以称为区域视图).为此需要调用MapViewOfFile(Ex)、MapViewOfFileFromApp 或MapViewOfFileExNuma函数，并指定要映射的范围。映射的视图可以帮助进程节约地址空间，因为只需要将区域对象中实际需要的视图映射至内存。

数据执行保护

数据执行保护(Data Execution Prevention，DEP)即不可执行(No-execute，NX)页面保护，会在试图将控制权传递给标记为“不可执行”的页面中所包含的指令时产生访问错误。借此可防止某些类型的恶意软件利用系统 bug 执行位于数据页面(例如栈)中的代码。DEP 还能发现编写不够完善的软件，这类软件可能未给想要执行的代码所在页面设置正确的权限。如果是在内核模式下试图执行“不可执行”页面中包含的代码，系统会期溃(BSOD)，此时的错误检查代码为ATTEMPTED\_EXECUTE\_OF\_NOEXECUTE\_MEMORY(0xFC)。如果用户模式下出现这种情况，试图执行非法引用的线程会收到STATUS\_ACCESS\_VIOLATION(0xC0000005)异常。

在支持DEP的32位x86 系统中，会使用页表项(Page Table Entry，PTE)中的第63位将页面标记为不可执行。因此 DEP 功能只能在处理器运行于物理地址扩展(Physical Address Extension，PAE)模式时才能使用，原因在于不使用PAE 时页表项只有32位宽。进而为了在32位系统上支持硬件DEP，需要加载PAE内核(%SystemRoot%System32\Ntkrnlpa.exe)，这也是x86系统上唯一支持的内核。

ARM系统中，DEP 已被设置为AlwaysOn(始终启用)。

在64位版Windows 中，这种执行保护会始终应用于所有64 位进程和设备驱动程序，只能通过将 nxBCD选项设置为AlwaysOff的方式将其禁用。在 64 位Windows 中，执行保护会应用给线程栈(用户模式和内核模式)、未明确标记为可执行的用户模式页面、内核换页池以及内核会话池。不过在 32位 Windows 中，执行保护将仅应用于线程栈和用户模式页面，而不会应用于内核换页池和内核会话池。

是否为 32 位进程应用执行保护，这取决于 BCD nx选项的值。



上图为BCD nx选项的设置页面，下表列出了该选项不同的值，以及它们在 数据执行保护 选项卡下对应的设置。注册表 HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AppCompatFlags\Layers键下的值列出了不应用执行保护的 32位应用程序，此处值的名称是可执行文件的完整路径，值的数据需要设置为 DisableNXShowUI。

|  |  |  |
| --- | --- | --- |
| **BCD nx值** | **对应的数据执行保护选项卡设置** | **解释** |
| OptIn | 仅为基本的Windows程序和服务启动DEP | 可为核心Windows系统映像启动DEP，可以让32为进程运行期间动态配置DEP |
| OptOut | 为除下列选定程序之外的所有程序和服务启动DEP | 可以为除了选定内容外的所有可执行文件启用 DEP。可以让32位进程在运行期间动态配置 DEP。可为DEP启用系统兼容性修复 |
| AlwaysOn | 该设置没有对应的图形界面选项 | 可为所有组件启用 DEP，但无法排除特定应用程序。可禁止32位进程动态配置 DEP 并禁止系统兼容性修复 |
| AlwaysOff | 该设置没有对应的图形界面选项 | 可彻底禁用 DEP(不推荐)。此外还可禁止 32位进程动态配置DEP |

就算强制启用DEP，应用程序依然可以通过其他方式为自己的映像禁用DEP。例如无论启用哪种执行保护选项，映像加载器都会向已知的复制保护机制(如 SafeDisc和SecuROM)验证可执行文件的签名，并通过禁用执行保护提供与电脑游戏等较老的复制保护软件的兼容性。

此外，为兼容老版本Active Template Library (ATL)库(7.1或更老版本)，Windows内核还提供了一种 ATL形式转换 (thunk)模拟环境。该环境可检测曾导致 DEP异常的ATL形式转换代码序列并模拟预期操作。应用程序开发者可以使用最新的Microsoft C++编译器指定NX\_COMPA标志(借此可在PE 头中设置IMAGE\_DLLCHARACTERISTICS\_NX\_COMPAT标志)，借此请求不应用这种ATL形式转换模拟，进而告诉系统该可执行文件完全支持DEP。注意，如果设置了 AlwaysOn值，ATL形式转换模拟将被永远禁用。

最后，如果设置为OptIn或OptOut 模式的系统执行的是32位进程，SetProcessDEPPolicy函数将允许进程动态地禁用 DEP 或永久启用。如果通过该 API启用，在该进程的生命周期内，DEP将无法通过编程的方式禁用。如果映像在编译时未使用/NXCOMPAT标志，该数还可用于动态地禁用ATL形式转换模拟。对于64位进程或使用AlwaysOff或AlwaysOn引导的系统，该函数将始终返回一个失败。GetProcessDEPPolicy 函数还可返回32位进程的每个进程DEP策略 DEP(在64 位系统中会失败，因为64位系统中该策略始终是启用的)

写入时复制

写入时复制(copy-on-write)页面保护是一种优化机制，内存管理器可以借此节约物理内存。当进程为区域对象映射的写入时复制视图包含读写页面时，内存管理器会对这些页面的复制延迟到向页面中写入数据时，而非在映射视图的同时创建进程的私有副本。

如果任何一个进程中的线程向页面写入数据，则会产生内存管理错误。内存管理器可以检测到该写入操作会作用于一个“写入时复制”页面，因此，此时并不会将这个错误报告为访问冲突，而是会执行下列操作。

1. 在物理内存中分配一个新的读写页面。
2. 将原页面中的内容复制到这个新建页面
3. 更新该进程中对应的页面映射信息，使其指向新位置
4. 解除异常，导致生成了上述错误的指令得以重新执行。

这次写操作将能成功进行。然而，新复制的页面对之前执行写操作的进程来说是私有的，该页面对依然共享着写入时复制页面的其他进程是不可见的。

写入时复制机制的一种应用方式是在调试器中实现对断点的支持。例如，默认情况代码页面在启动时都是“仅执行”的。然而如果程序员在调试程序时设置了断点，调试必须在代码中添加断点指令。为此，调试器首先会将页面的保护方式改为PAGE\_EXECUTE\_READWRITE，随后更改指令流。由于代码页面是已映射区域的一部分，内存管理器会设置了断点的进程创建页面的私有副本，而其他进程依然可以使用未修改的代码页面

写入时复制是惰性计算 (lazy evaluation) 技术的一个范例，内存管理器会尽可能繁地使用该技术。惰性计算算法可在绝对必要前避免执行开销昂贵的操作。如果永远不要执行这种操作，就无须为它浪费哪怕一点时间。

地址窗口扩展

为了改善性能(并提供更细化的控制)，Windows 提供了一组名为地址窗口扩展(Address Windowing Extension，AWE)的函数。这些函数可以让进程分配超过其虚拟地址空间可代表数量的物理内存。随后即可随时将自己虚拟地址空间的一部分映射到物理内存中所选择的部分，借此访问这些物理内存。

通过AWE 函数，我们可以通过下列几个步骤来分配和使用内存。

1. 启用AWE：要使用AWE，应用程序必须首先启用AWE功能。这可以通过调用InitializeAWE函数实现
2. 分配要使用的物理内存。应用程序可调用Windows函数AllocateUserPhysicalPages或AllocateUserPhysicalPagesNuma。(均需要具备 SeLockMemoryPrivilege 特权。)
3. 创建一个或多个虚拟地址空间区域，将其作为窗口来映射物理内存的不同视图。应用程序可调用Win32的VirtualAlloc、VirtualAllocEx或VirtualAllocExNuma函数并传递MEM\_PHYSICAL标志。
4. 一般来说，上述第(1)和第(2)步都是初始化步骤。为了实际使用这些内存，应用程序需要调用 MapUserPhysicalPages 或MapUserPhysicalPagesScatter，将第(1)步分配的物理区域的一部分映射到第(2)步为自己分配的一个虚拟区域(或称为“窗口”)中。

然而AWE 最适合具备超过2GB 物理内存的32位系统，因为这是32位进程能够访问超过自己虚拟地址空间数量的更多物理内存的唯一方式。该函数还可用于改善安全性。由于AWE 内存绝对不会被换出，因此AWE 内存中的数据永远不会在页面文件中留下副本，从而避免重新启动到另一个操作系统并通过访问页面文件查看到这些数据