Contents

CreateMemoryResourceNotification

DiscardVirtualMemory

FillMemory

FlushViewOfFile

FreeUserPhysicalPages

GetLargePageMinimum

GetMappedFileName

GetMemoryErrorHandlingCapabilities

GetPhysicallyInstalledSystemMemory

GetProcessDEPPolicy

GetProcessHeap

GetProcessHeaps

GetSystemDEPPolicy

GetSystemFileCacheSize

GetWriteWatch

GlobalAlloc

GlobalDiscard

GlobalFlags

GlobalFree

GlobalHandle

GlobalLock

GlobalMemoryStatus

GlobalMemoryStatusEx

GlobalReAlloc

GlobalSize

GlobalUnlock

HeapAlloc

HeapCompact

HeapCreate

HeapDestroy

HeapFree

HeapLock

HeapQueryInformation

HeapReAlloc

HeapSetInformation

HeapSize

HeapUnlock

HeapValidate

HeapWalk

InitializeEnclave

IsBadCodePtr

IsBadReadPtr

IsBadStringPtr

IsBadWritePtr

IsEnclaveTypeSupported

LoadEnclaveData

LocalAlloc

LocalDiscard

LocalFlags

LocalFree

LocalHandle

LocalLock

LocalReAlloc

LocalSize

LocalUnlock

MapViewOfFile

MapViewOfFile2

MapViewOfFile3

MapViewOfFile3FromApp

MapViewOfFileEx

MapViewOfFileExNuma

MapViewOfFileFromApp

MapViewOfFileNuma2

MapUserPhysicalPages

MapUserPhysicalPagesScatter

MoveMemory

OpenFileMapping

OpenFileMappingFromApp

OfferVirtualMemory

PrefetchVirtualMemory

QueryMemoryResourceNotification

QueryVirtualMemoryInformation

ReadProcessMemory

ReclaimVirtualMemory

RegisterBadMemoryNotification

RemoveSecureMemoryCacheCallback

ResetWriteWatch

SecureMemoryCacheCallback

SecureZeroMemory

SetProcessDEPPolicy

SetProcessValidCallTargets

SetSystemFileCacheSize

UnmapViewOfFile

UnmapViewOfFile2

UnmapViewOfFileEx

UnregisterBadMemoryNotification

VirtualAlloc

VirtualAlloc2

VirtualAlloc2FromApp

VirtualAllocEx

VirtualAllocExNuma

VirtualAllocFromApp

VirtualFree

VirtualFreeEx

VirtualLock

VirtualProtect

Memory Management (Memory Management)

3/5/2021 • 2 minutes to read • Edit Online

The memory manager implements virtual memory, provides a core set of services such as memory mapped files, copy-on-write memory, large memory support, and underlying support for the cache manager.

In This Section

About Memory Management

3/5/2021 • 2 minutes to read • Edit Online

Each process on 32-bit Microsoft Windows has its own virtual address space that enables addressing up to 4 gigabytes of memory. Each process on 64-bit Windows has a virtual address space of 8 terabytes. All threads of a process can access its virtual address space. However, threads cannot access memory that belongs to another process, which protects a process from being corrupted by another process.

For information on the virtual address space and the memory management functions, see the following topics.

- Virtual Address Space
- Memory Pools
- Memory Performance Information
- Virtual Memory Functions
- Heap Functions
- File Mapping
- Large Memory Support
- Global and Local Functions
- Standard C Library Functions
- Comparing Memory Allocation Methods

Virtual Address Space (Memory Management)

3/5/2021 • 2 minutes to read • Edit Online

The virtual address space for a process is the set of virtual memory addresses that it can use. The address space for each process is private and cannot be accessed by other processes unless it is shared.

A virtual address does not represent the actual physical location of an object in memory; instead, the system maintains a *page table* for each process, which is an internal data structure used to translate virtual addresses into their corresponding physical addresses. Each time a thread references an address, the system translates the virtual address to a physical address.

The virtual address space for 32-bit Windows is 4 gigabytes (GB) in size and divided into two partitions: one for use by the process and the other reserved for use by the system. For more information about the virtual address space in 64-bit Windows, see Virtual Address Space in 64-bit Windows.

For more information about virtual memory, see the following topics:

- Virtual Address Space and Physical Storage
- Working Set
- Page State
- Scope of Allocated Memory
- Data Execution Prevention
- Memory Protection
- Memory Limits for Windows Releases

Default Virtual Address Space for 32-bit Windows

The following table shows the default memory range for each partition.

MEMORY RANGE	USAGE
Low 2GB (0x00000000 through 0x7FFFFFFF)	Used by the process.
High 2GB (0x80000000 through 0xFFFFFFFF)	Used by the system.

Virtual Address Space for 32-bit Windows with 4GT

If 4-gigabyte tuning (4GT) is enabled, the memory range for each partition is as follows.

MEMORY RANGE	USAGE
Low 3GB (0x00000000 through 0xBFFFFFFF)	Used by the process.
High 1GB (0xC0000000 through 0xFFFFFFFF)	Used by the system.

After 4GT is enabled, a process that has the **IMAGE_FILE_LARGE_ADDRESS_AWARE** flag set in its image header will have access to the additional 1 GB of memory above the low 2 GB.

Adjusting the Virtual Address Space for 32-bit Windows

You can use the following command to set a boot entry option that configures the size of the partition that is available for use by the process to a value between 2048 (2 GB) and 3072 (3 GB):

BCDEdit /set **increaseuserva** *Megabytes*

After the boot entry option is set, the memory range for each partition is as follows.

MEMORY RANGE	USAGE
Low (0x00000000 through *Megabytes*)	Used by the process.
High (*Megabytes*+1 through 0xFFFFFFFF)	Used by the system.

Windows Server 2003: Set the **/USERVA** switch in boot.ini to a value between 2048 and 3072.

Virtual Address Space and Physical Storage

3/5/2021 • 2 minutes to read • Edit Online

The maximum amount of physical memory supported by Microsoft Windows ranges from 2 GB to 24 TB, depending on the version of Windows. For more information, see Memory Limits for Windows Releases. The virtual address space of each process can be smaller or larger than the total physical memory available on the computer. The subset of the virtual address space of a process that resides in physical memory is known as the *working set*. If the threads of a process attempt to use more physical memory than is currently available, the system pages some of the memory contents to disk. The total amount of virtual address space available to a process is limited by physical memory and the free space on disk available for the paging file.

Physical storage and the virtual address space of each process are organized into *pages*, units of memory, whose size depends on the host computer. For example, on x86 computers the host page size is 4 kilobytes.

To maximize its flexibility in managing memory, the system can move pages of physical memory to and from a paging file on disk. When a page is moved in physical memory, the system updates the page maps of the affected processes. When the system needs space in physical memory, it moves the least recently used pages of physical memory to the paging file. Manipulation of physical memory by the system is completely transparent to applications, which operate only in their virtual address spaces.

Working Set

6/4/2021 • 2 minutes to read • Edit Online

The working set of a process is the set of pages in the virtual address space of the process that are currently resident in physical memory. The working set contains only pageable memory allocations; nonpageable memory allocations such as Address Windowing Extensions (AWE) or large page allocations are not included in the working set.

When a process references pageable memory that is not currently in its working set, a *page fault* occurs. The system page fault handler attempts to resolve the page fault and, if it succeeds, the page is added to the working set. (Accessing AWE or large page allocations never causes a page fault, because these allocations are not pageable .)

A *hard page fault* must be resolved by reading page contents from the page's *backing store*, which is either the system paging file or a memory-mapped file created by the process. A *soft page fault* can be resolved without accessing the backing store. A soft page fault occurs when:

- The page is in the working set of some other process, so it is already resident in memory.
- The page is in transition, because it either has been removed from the working sets of all processes that were using the page and has not yet been repurposed, or it is already resident as a result of a memory manager prefetch operation.
- A process references an allocated virtual page for the first time (sometimes called a *demand-zero fault*).

Pages can be removed from a process working set as a result of the following actions:

- The process reduces or empties the working set by calling the **SetProcessWorkingSetSize**, **SetProcessWorkingSetSizeEx** or **EmptyWorkingSet** function.
- The process calls the **VirtualUnlock** function on a memory range that is not locked.
- The process unmaps a mapped view of a file using the **UnmapViewOfFile** function.
- The memory manager trims pages from the working set to create more available memory.
- The memory manager must remove a page from the working set to make room for a new page (for example, because the working set is at its maximum size).

If several processes share a page, removing the page from the working set of one process does not affect other processes. After a page is removed from the working sets of all processes that were using it, the page becomes a *transition page*. Transition pages remain cached in RAM until the page is either referenced again by some process or repurposed (for example, filled with zeros and given to another process). If a transition page has been modified since it was last written to disk (that is, if the page is "dirty"), then the page must be written to its backing store before it can be repurposed. The system may start writing dirty transition pages to their backing store as soon as such pages become available.

Each process has a minimum and maximum working set size that affect the virtual memory paging behavior of the process. To obtain the current size of the working set of a specified process, use the **GetProcessMemoryInfo** function. To obtain or change the minimum and maximum working set sizes, use the **GetProcessWorkingSetSizeEx** and **SetProcessWorkingSetSizeEx** functions.

The process status application programming interface (PSAPI) provides a number of functions that return detailed information about the working set of a process. For details, see Working Set Information.

Page State

3/5/2021 • 2 minutes to read • Edit Online

The pages of a process's virtual address space can be in one of the following states.

STATE	DESCRIPTION
Free	The page is neither committed nor reserved. The page is not accessible to the process. It is available to be reserved, committed, or simultaneously reserved and committed. Attempting to read from or write to a free page results in an access violation exception. A process can use the **VirtualFree** or **VirtualFreeEx** function to release reserved or committed pages of its address space, returning them to the free state.
Reserved	The page has been reserved for future use. The range of addresses cannot be used by other allocation functions. The page is not accessible and has no physical storage associated with it. It is available to be committed. A process can use the **VirtualAlloc** or **VirtualAllocEx** function to reserve pages of its address space and later to commit the reserved pages. It can use **VirtualFree** or **VirtualFreeEx** to decommit committed pages and return them to the reserved state.
Committed	Memory charges have been allocated from the overall size of RAM and paging files on disk. The page is accessible and access is controlled by one of the memory protection constants. The system initializes and loads each committed page into physical memory only during the first attempt to read or write to that page. When the process terminates, the system releases the storage for committed pages. A process can use **VirtualAlloc** or **VirtualAllocEx** to commit physical pages from a reserved region. They can also simultaneously reserve and commit pages. The **GlobalAlloc** and **LocalAlloc** functions allocate committed pages with read/write access.

Scope of Allocated Memory

3/5/2021 • 2 minutes to read • Edit Online

All memory a process allocates by using the memory allocation functions (**HeapAlloc**, **VirtualAlloc**, **GlobalAlloc**, or **LocalAlloc**) is accessible only to the process. However, memory allocated by a DLL is allocated in the address space of the process that called the DLL and is not accessible to other processes using the same DLL. To create shared memory, you must use file mapping.

Named file mapping provides an easy way to create a block of shared memory. A process can specify a name when it uses the **CreateFileMapping** function to create a file-mapping object. Other processes can specify the same name to either the **CreateFileMapping** or **OpenFileMapping** function to obtain a handle to the mapping object.

Each process specifies its handle to the file-mapping object in the **MapViewOfFile** function to map a view of the file into its own address space. The views of all processes for a single file-mapping object are mapped into the same sharable pages of physical storage. However, the virtual addresses of the mapped views can vary from one process to another, unless the **MapViewOfFileEx** function is used to map the view at a specified address. Although sharable, the pages of physical storage used for a mapped file view are not global; they are not accessible to processes that have not mapped a view of the file.

Any pages committed by mapping a view of a file are released when the last process with a view of the mapping object either terminates or unmaps its view by calling the **UnmapViewOfFile** function. At this time, the specified file (if any) associated with the mapping object is updated. A specified file can also be forced to update by calling the **FlushViewOfFile** function.

For more information, see File Mapping. For an example of shared memory in a DLL, see Using Shared Memory in a Dynamic-Link Library.

If multiple processes have write access to shared memory, you must synchronize access to the memory. For more information, see Synchronization.

Data Execution Prevention

3/5/2021 • 3 minutes to read • Edit Online

Data Execution Prevention (DEP) is a system-level memory protection feature that is built into the operating system starting with Windows XP and Windows Server 2003. DEP enables the system to mark one or more pages of memory as non-executable. Marking memory regions as non-executable means that code cannot be run from that region of memory, which makes it harder for the exploitation of buffer overruns.

DEP prevents code from being run from data pages such as the default heap, stacks, and memory pools. If an application attempts to run code from a data page that is protected, a memory access violation exception occurs, and if the exception is not handled, the calling process is terminated.

DEP is not intended to be a comprehensive defense against all exploits; it is intended to be another tool that you can use to secure your application.

How Data Execution Prevention Works

If an application attempts to run code from a protected page, the application receives an exception with the status code **STATUS_ACCESS_VIOLATION**. If your application must run code from a memory page, it must allocate and set the proper virtual memory protection attributes. The allocated memory must be marked **PAGE_EXECUTE**, **PAGE_EXECUTE_READ**, **PAGE_EXECUTE_READWRITE**, or **PAGE_EXECUTE_WRITECOPY** when allocating memory. Heap allocations made by calling the **malloc** and **HeapAlloc** functions are non-executable.

Applications cannot run code from the default process heap or the stack.

DEP is configured at system boot according to the no-execute page protection policy setting in the boot configuration data. An application can get the current policy setting by calling the **GetSystemDEPPolicy** function. Depending on the policy setting, an application can change the DEP setting for the current process by calling the **SetProcessDEPPolicy** function.

Programming Considerations

An application can use the **VirtualAlloc** function to allocate executable memory with the appropriate memory protection options. It is suggested that an application set, at a minimum, the **PAGE_EXECUTE** memory protection option. After the executable code is generated, it is recommended that the application set memory protections to disallow write access to the allocated memory. Applications can disallow write access to allocated memory by using the **VirtualProtect** function. Disallowing write access ensures maximum protection for executable regions of process address space. You should attempt to create applications that use the smallest executable address space possible, which minimizes the amount of memory that is exposed to memory exploitation.

You should also attempt to control the layout of your application's virtual memory and create executable regions. These executable regions should be located in a lower memory space than non-executable regions. By locating executable regions below non-executable regions, you can help prevent a buffer overflow from overflowing into the executable area of memory.

Application Compatibility

Some application functionality is incompatible with DEP. Applications that perform dynamic code generation (such as Just-In-Time code generation) and do not explicitly mark generated code with execute permission may

have compatibility issues on computers that are using DEP. Applications written to the Active Template Library (ATL) version 7.1 and earlier can attempt to execute code on pages marked as non-executable, which triggers an NX fault and terminates the application; for more information, see **SetProcessDEPPolicy**. Most applications that perform actions incompatible with DEP must be updated to function properly.

A small number of executable files and libraries may contain executable code in the data section of an image file. In some cases, applications may place small segments of code (commonly referred to as thunks) in the data sections. However, DEP marks sections of the image file that is loaded in memory as non-executable unless the section has the executable attribute applied.

Therefore, executable code in data sections should be migrated to a code section, or the data section that contains the executable code should be explicitly marked as executable. The executable attribute, **IMAGE_SCN_MEM_EXECUTE**, should be added to the **Characteristics** field of the corresponding section header for sections that contain executable code. For more information about adding attributes to a section, see the documentation included with your linker.

Related topics

Data Execution Prevention (TechNet)

How to Configure Memory Protection

KB Article 875352

Memory Protection

3/5/2021 • 2 minutes to read • Edit Online

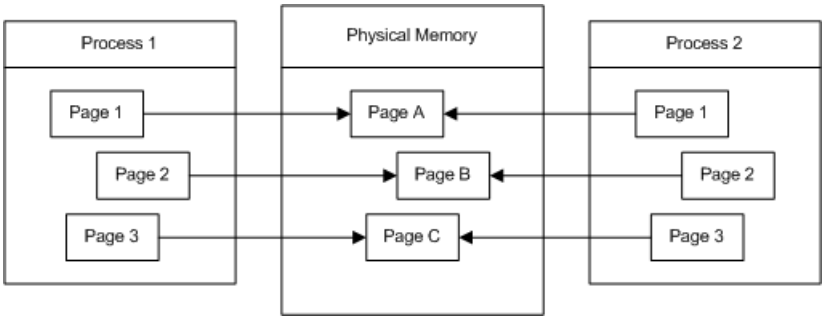
Memory that belongs to a process is implicitly protected by its private virtual address space. In addition, Windows provides memory protection by using the virtual memory hardware. The implementation of this protection varies with the processor, for example, code pages in the address space of a process can be marked read-only and protected from modification by user-mode threads.

For the complete list of attributes, see Memory Protection Constants.

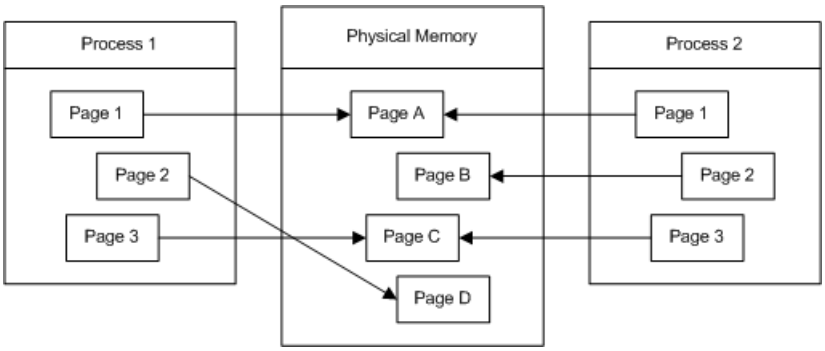
Copy-on-Write Protection

Copy-on-write protection is an optimization that allows multiple processes to map their virtual address spaces such that they share a physical page until one of the processes modifies the page. This is part of a technique called *lazy evaluation*, which allows the system to conserve physical memory and time by not performing an operation until absolutely necessary.

For example, suppose two processes load pages from the same DLL into their virtual memory spaces. These virtual memory pages are mapped to the same physical memory pages for both processes. As long as neither process writes to these pages, they can map to and share, the same physical pages, as shown in the following diagram.



If Process 1 writes to one of these pages, the contents of the physical page are copied to another physical page and the virtual memory map is updated for Process 1. Both processes now have their own instance of the page in physical memory. Therefore, it is not possible for one process to write to a shared physical page and for the other process to see the changes.



Loading Applications and DLLs

When multiple instances of the same Windows-based application are loaded, each instance is run in its own protected virtual address space. However, their instance handles (*hInstance*) typically have the same value. This value represents the base address of the application in its virtual address space. If each instance can be loaded

into its default base address, it can map to and share the same physical pages with the other instances, using copy-on-write protection. The system allows these instances to share the same physical pages until one of them modifies a page. If for some reason one of these instances cannot be loaded in the desired base address, it receives its own physical pages.

DLLs are created with a default base address. Every process that uses a DLL will try to load the DLL within its own address space at the default virtual address for the DLL. If multiple applications can load a DLL at its default virtual address, they can share the same physical pages for the DLL. If for some reason a process cannot load the DLL at the default address, it loads the DLL elsewhere. Copy-on-write protection forces some of the DLL's pages to be copied into different physical pages for this process, because the fixes for jump instructions are written within the DLL's pages, and they will be different for this process. If the code section contains many references to the data section, this can cause the entire code section to be copied to new physical pages.

Memory Limits for Windows and Windows Server Releases

3/5/2021 • 9 minutes to read • Edit Online

This topic describes the memory limits for supported Windows and Windows Server releases.

Limits on memory and address space vary by platform, operating system, and by whether the **IMAGE_FILE_LARGE_ADDRESS_AWARE** value of the **LOADED_IMAGE** structure and 4-gigabyte tuning (4GT) are in use. **IMAGE_FILE_LARGE_ADDRESS_AWARE** is set or cleared by using the /LARGEADDRESSAWARE linker option.

4-gigabyte tuning (4GT), also known as application memory tuning, or the /3GB switch, is a technology (only applicable to 32 bit systems) that alters the amount of virtual address space available to user mode applications. Enabling this technology reduces the overall size of the system virtual address space and therefore system resource maximums. For more information, see What is 4GT.

Limits on physical memory for 32-bit platforms also depend on the Physical Address Extension (PAE), which allows 32-bit Windows systems to use more than 4 GB of physical memory.

Memory and Address Space Limits

The following table specifies the limits on memory and address space for supported releases of Windows. Unless otherwise noted, the limits in this table apply to all supported releases.

MEMORY TYPE	LIMIT ON X86	LIMIT IN 64-BIT WINDOWS

MEMORY TYPE	LIMIT ON X86	LIMIT IN 64-BIT WINDOWS
User-mode virtual address space for each 32-bit process	2 GB Up to 3 GB with **IMAGE_FILE_LARGE_ADDRESS_AWARE** and 4GT	2 GB with **IMAGE_FILE_LARGE_ADDRESS_AWARE** cleared (default) 4 GB with **IMAGE_FILE_LARGE_ADDRESS_AWARE** set
User-mode virtual address space for each 64-bit process	Not applicable	**With IMAGE_FILE_LARGE_ADDRESS_AWARE** set (default): **x64: Windows 8.1 and Windows Server 2012 R2 or later:** 128 TB **x64: Windows 8 and Windows Server 2012 or earlier** 8 TB **Intel Itanium-based systems:** 7 TB 2 GB with **IMAGE_FILE_LARGE_ADDRESS_AWARE** cleared
Kernel-mode virtual address space	2 GB From 1 GB to a maximum of 2 GB with 4GT	**Windows 8.1 and Windows Server 2012 R2 or later:** 128 TB **Windows 8 and Windows Server 2012 or earlier** 8 TB
Paged pool	384 GB or system commit limit, whichever is smaller. **Windows 8.1 and Windows Server 2012 R2:** 15.5 TB or system commit limit, whichever is smaller. **Windows Server 2008 R2, Windows 7, Windows Server 2008 and Windows Vista:** Limited by available kernel-mode virtual address space. Starting with Windows Vista with Service Pack 1 (SP1), the paged pool can also be limited by the PagedPoolLimit registry key value. **Windows Home Server and Windows Server 2003:** 530 MB **Windows XP:** 490 MB	384 GB or system commit limit, whichever is smaller **Windows 8.1 and Windows Server 2012 R2:** 15.5 TB or system commit limit, whichever is smaller. **Windows Server 2008 R2, Windows 7, Windows Server 2008 and Windows Vista:** 128 GB or system commit limit, whichever is smaller **Windows Server 2003 and Windows XP:** Up to 128 GB depending on configuration and RAM.

MEMORY TYPE	LIMIT ON X86	LIMIT IN 64-BIT WINDOWS
Nonpaged pool	75% of RAM or 2 GB, whichever is smaller. **Windows 8.1 and Windows Server 2012 R2:** RAM or 16 TB, whichever is smaller (address space is limited to 2 x RAM). **Windows Vista:** Limited only by kernel mode virtual address space and physical memory. Starting with Windows Vista with SP1, the nonpaged pool can also be limited by the NonPagedPoolLimit registry key value. **Windows Home Server, Windows Server 2003 and Windows XP:** 256 MB, or 128 MB with 4GT.	RAM or 128 GB, whichever is smaller (address space is limited to 2 x RAM) **Windows 8.1 and Windows Server 2012 R2:** RAM or 16 TB, whichever is smaller (address space is limited to 2 x RAM). **Windows Server 2008 R2, Windows 7 and Windows Server 2008:** 75% of RAM up to a maximum of 128 GB **Windows Vista:** 40% of RAM up to a maximum of 128 GB. **Windows Server 2003 and Windows XP:** Up to 128 GB depending on configuration and RAM.
System cache virtual address space (physical size limited only by physical memory)	Limited by available kernel-mode virtual address space or the SystemCacheLimit registry key value. **Windows 8.1 and Windows Server 2012 R2:** 16 TB. **Windows Vista:** Limited only by kernel mode virtual address space. Starting with Windows Vista with SP1, system cache virtual address space can also be limited by the SystemCacheLimit registry key value. **Windows Home Server, Windows Server 2003 and Windows XP:** 860 MB with LargeSystemCache registry key set and without 4GT; up to 448 MB with 4GT.	Always 1 TB regardless of physical RAM **Windows 8.1 and Windows Server 2012 R2:** 16 TB. **Windows Server 2003 and Windows XP:** Up to 1 TB depending on configuration and RAM.

Physical Memory Limits: Windows 10

The following table specifies the limits on physical memory for Windows 10.

VERSION	LIMIT ON X86	LIMIT ON X64
Windows 10 Enterprise	4 GB	6 TB
Windows 10 Education	4 GB	2 TB
Windows 10 Pro for Workstations	4 GB	6 TB
Windows 10 Pro	4 GB	2 TB
Windows 10 Home	4 GB	128 GB

Physical Memory Limits: Windows Server 2016

The following table specifies the limits on physical memory for Windows Server 2016.

VERSION	LIMIT ON X64
Windows Server 2016 Datacenter	24 TB
Windows Server 2016 Standard	24 TB

Physical Memory Limits: Windows 8

The following table specifies the limits on physical memory for Windows 8.

VERSION	LIMIT ON X86	LIMIT ON X64
Windows 8 Enterprise	4 GB	512 GB
Windows 8 Professional	4 GB	512 GB
Windows 8	4 GB	128 GB

Physical Memory Limits: Windows Server 2012

The following table specifies the limits on physical memory for Windows Server 2012. Windows Server 2012 is available only in X64 editions.

VERSION	LIMIT ON X64
Windows Server 2012 Datacenter	4 TB
Windows Server 2012 Standard	4 TB
Windows Server 2012 Essentials	64 GB
Windows Server 2012 Foundation	32 GB
Windows Storage Server 2012 Workgroup	32 GB
Windows Storage Server 2012 Standard	4 TB
Hyper-V Server 2012	4 TB

Physical Memory Limits: Windows 7

The following table specifies the limits on physical memory for Windows 7.

VERSION	LIMIT ON X86	LIMIT ON X64
Windows 7 Ultimate	4 GB	192 GB
Windows 7 Enterprise	4 GB	192 GB
Windows 7 Professional	4 GB	192 GB
Windows 7 Home Premium	4 GB	16 GB
Windows 7 Home Basic	4 GB	8 GB
Windows 7 Starter	2 GB	N/A

Physical Memory Limits: Windows Server 2008 R2

The following table specifies the limits on physical memory for Windows Server 2008 R2. Windows Server 2008 R2 is available only in 64-bit editions.

VERSION	LIMIT ON X64	LIMIT ON IA64
Windows Server 2008 R2 Datacenter	2 TB	
Windows Server 2008 R2 Enterprise	2 TB	
Windows Server 2008 R2 for Itanium-Based Systems		2 TB
Windows Server 2008 R2 Foundation	8 GB	
Windows Server 2008 R2 Standard	32 GB	
Windows HPC Server 2008 R2	128 GB	
Windows Web Server 2008 R2	32 GB	

Physical Memory Limits: Windows Server 2008

The following table specifies the limits on physical memory for Windows Server 2008. Limits greater than 4 GB for 32-bit Windows assume that PAE is enabled.

VERSION	LIMIT ON X86	LIMIT ON X64	LIMIT ON IA64
Windows Server 2008 Datacenter	64 GB	1 TB	
Windows Server 2008 Enterprise	64 GB	1 TB	

VERSION	LIMIT ON X86	LIMIT ON X64	LIMIT ON IA64
Windows Server 2008 HPC Edition		128 GB	
Windows Server 2008 Standard	4 GB	32 GB	
Windows Server 2008 for Itanium-Based Systems			2 TB
Windows Small Business Server 2008	4 GB	32 GB	
Windows Web Server 2008	4 GB	32 GB	

Physical Memory Limits: Windows Vista

The following table specifies the limits on physical memory for Windows Vista.

VERSION	LIMIT ON X86	LIMIT ON X64
Windows Vista Ultimate	4 GB	128 GB
Windows Vista Enterprise	4 GB	128 GB
Windows Vista Business	4 GB	128 GB
Windows Vista Home Premium	4 GB	16 GB
Windows Vista Home Basic	4 GB	8 GB
Windows Vista Starter	1 GB	

Physical Memory Limits: Windows Home Server

Windows Home Server is available only in a 32-bit edition. The physical memory limit is 4 GB.

Physical Memory Limits: Windows Server 2003 R2

The following table specifies the limits on physical memory for Windows Server 2003 R2. Limits over 4 GB for 32-bit Windows assume that PAE is enabled.

VERSION	LIMIT ON X86	LIMIT ON X64
Windows Server 2003 R2 Datacenter Edition	64 GB (16 GB with 4GT)	1 TB
Windows Server 2003 R2 Enterprise Edition	64 GB (16 GB with 4GT)	1 TB

VERSION	LIMIT ON X86	LIMIT ON X64
Windows Server 2003 R2 Standard Edition	4 GB	32 GB

Physical Memory Limits: Windows Server 2003 with Service Pack 2 (SP2)

The following table specifies the limits on physical memory for Windows Server 2003 with Service Pack 2 (SP2). Limits over 4 GB for 32-bit Windows assume that PAE is enabled.

VERSION	LIMIT ON X86	LIMIT ON X64	LIMIT ON IA64
Windows Server 2003 with Service Pack 2 (SP2), Datacenter Edition	64 GB (16 GB with 4GT)	1 TB	2 TB
Windows Server 2003 with Service Pack 2 (SP2), Enterprise Edition	64 GB (16 GB with 4GT)	1 TB	2 TB
Windows Server 2003 with Service Pack 2 (SP2), Standard Edition	4 GB	32 GB	

Physical Memory Limits: Windows Server 2003 with Service Pack 1 (SP1)

The following table specifies the limits on physical memory for Windows Server 2003 with Service Pack 1 (SP1). Limits over 4 GB for 32-bit Windows assume that PAE is enabled.

VERSION	LIMIT ON X86	LIMIT ON X64	LIMIT ON IA64
Windows Server 2003 with Service Pack 1 (SP1), Datacenter Edition	64 GB (16 GB with 4GT)	X64 1 TB	1 TB
Windows Server 2003 with Service Pack 1 (SP1), Enterprise Edition	64 GB (16 GB with 4GT)	X64 1 TB	1 TB
Windows Server 2003 with Service Pack 1 (SP1), Standard Edition	4 GB	32 GB	

Physical Memory Limits: Windows Server 2003

The following table specifies the limits on physical memory for Windows Server 2003. Limits over 4 GB for 32-bit Windows assume that PAE is enabled.

VERSION	LIMIT ON X86	LIMIT ON IA64
Windows Server 2003, Datacenter Edition	64 GB (16 GB with 4GT)	512 GB
Windows Server 2003, Enterprise Edition	64 GB (16 GB with 4GT)	512 GB
Windows Server 2003, Standard Edition	4 GB	
Windows Server 2003, Web Edition	2 GB	
Windows Small Business Server 2003	4 GB	
Windows Compute Cluster Server 2003		32 GB
Windows Storage Server 2003, Enterprise Edition	8 GB	
Windows Storage Server 2003	4 GB	

Physical Memory Limits: Windows XP

The following table specifies the limits on physical memory for Windows XP.

VERSION	LIMIT ON X86	LIMIT ON X64	LIMIT ON IA64
Windows XP	4 GB	128 GB	128 GB (not supported)
Windows XP Starter Edition	512 MB	N/A	N/A

Physical Memory Limits: Windows Embedded

The following table specifies the limits on physical memory for Windows Embedded.

VERSION	LIMIT ON X86	LIMIT ON X64
Windows XP Embedded	4 GB	
Windows Embedded Standard 2009	4 GB	
Windows Embedded Standard 7	4 GB	192 GB

How graphics cards and other devices affect memory limits

Devices have to map their memory below 4 GB for compatibility with non-PAE-aware Windows releases.

Therefore, if the system has 4GB of RAM, some of it is either disabled or is remapped above 4GB by the BIOS. If the memory is remapped, X64 Windows can use this memory. X86 client versions of Windows don't support physical memory above the 4GB mark, so they can't access these remapped regions. Any X64 Windows or X86 Server release can.

X86 client versions with PAE enabled do have a usable 37-bit (128 GB) physical address space. The limit that these versions impose is the highest permitted physical RAM address, not the size of the IO space. That means PAE-aware drivers can actually use physical space above 4 GB if they want. For example, drivers could map the "lost" memory regions located above 4 GB and expose this memory as a RAM disk.

Related topics

4-Gigabyte Tuning

IMAGE_FILE_LARGE_ADDRESS_AWARE

Physical Address Extension

Memory Pools

3/5/2021 • 2 minutes to read • Edit Online

The memory manager creates the following memory pools that the system uses to allocate memory: nonpaged pool and paged pool. Both memory pools are located in the region of the address space that is reserved for the system and mapped into the virtual address space of each process. The nonpaged pool consists of virtual memory addresses that are guaranteed to reside in physical memory as long as the corresponding kernel objects are allocated. The paged pool consists of virtual memory that can be paged in and out of the system. To improve performance, systems with a single processor have three paged pools, and multiprocessor systems have five paged pools.

The handles for kernel objects are stored in the paged pool, so the number of handles you can create is based on available memory.

The system records the limits and current values for its nonpaged pool, paged pool, and page file usage. For more information, see Memory Performance Information.

Memory Performance Information

3/5/2021 • 3 minutes to read • Edit Online

Memory performance information is available from the memory manager through the system performance counters and through functions such as **GetPerformanceInfo**, **GetProcessMemoryInfo**, and **GlobalMemoryStatusEx**. Applications such as the Windows Task Manager, the Reliability and Performance Monitor, and the Process Explorer tool use performance counters to display memory information for the system and for individual processes.

This topic associates performance counters with the data returned by memory performance functions and the Windows Task Manager:

System Memory Performance Information

The following table associates memory object performance counters with the data returned by the memory performance functions in the **MEMORYSTATUSEX**, **PERFORMANCE_INFORMATION**, and **PROCESS_MEMORY_COUNTERS_EX** structures, and with the corresponding information displayed by Task Manager.

MEMORY OBJECT COUNTER (UNLESS OTHERWISE NOTED)	STRUCTURE	TASK MANAGER PERFORMANCE TAB FOR WINDOWS SERVER 2008 AND WINDOWS VISTA	TASK MANAGER PERFORMANCE TAB FOR WINDOWS SERVER 2003 AND WINDOWS XP
Available KB	**MEMORYSTATUSEX**.**ullAvailPhys** and **PERFORMANCE_INFORMATION**.**PhysicalAvailable**	Subtract usage value shown in **Memory** graph from **Physical Memory (MB): Total**	**Physical Memory: Available**
None	**MEMORYSTATUSEX**.**ullTotalPhys** and **PERFORMANCE_INFORMATION**.**PhysicalTotal**	**Physical Memory (MB): Total**	**Physical Memory: Total**
Committed Bytes	**PERFORMANCE_INFORMATION**.**CommitTotal**	**System: Page File** first value (in MB)	**Commit Charge: Total**
Commit Limit	**MEMORYSTATUSEX**.**ullTotalPageFile** and **PERFORMANCE_INFORMATION**.**CommitLimit**	**System: Page File** second value (in MB)	**Commit Charge: Limit**
Free & Zero Page List Bytes **Windows Server 2003 and Windows XP:** This performance counter is not supported.	None	**Physical Memory (MB): Free**	Not applicable

MEMORY OBJECT COUNTER (UNLESS OTHERWISE NOTED)	STRUCTURE	TASK MANAGER PERFORMANCE TAB FOR WINDOWS SERVER 2008 AND WINDOWS VISTA	TASK MANAGER PERFORMANCE TAB FOR WINDOWS SERVER 2003 AND WINDOWS XP
None	PERFORMANCE_INFORMATION.CommitPeak	None	**Commit Charge: Peak**
None	PERFORMANCE_INFORMATION.HandleCount	**System: Handles**	**Totals: Handles**
None	MEMORYSTATUSEX.ullAvailPageFile	None	None
Pool Nonpaged Bytes	PERFORMANCE_INFORMATION.KernelNonpaged	**Kernel Memory: Nonpaged**	**Kernel Memory: Nonpaged**
Pool Paged Bytes	PERFORMANCE_INFORMATION.KernelPaged	**Kernel Memory: Paged**	**Kernel Memory: Paged**
Pool Paged Bytes + Pool Nonpaged Bytes	PERFORMANCE_INFORMATION.KernelTotal	**Kernel Memory: Total**	**Kernel Memory: Total**
Processes (Objects object)	PERFORMANCE_INFORMATION.ProcessCount	**System: Processes**	**Totals: Processes**
Thread Count (Process(_Total) object)	PERFORMANCE_INFORMATION.ThreadCount	**System: Threads**	**Totals: Threads**
Cache Bytes + Sharable pages on the standby and modified lists	PERFORMANCE_INFORMATION.SystemCache	None	**System Cache**
Cache Bytes + Modified Page List Bytes + Standby Cache Reserve Bytes + Standby Cache Normal Priority Bytes + Standby Cache Code Bytes **Windows Server 2003 and Windows XP:** Except for Cache Bytes, these performance counters are not supported.	None	**Physical Memory (MB): Cached**	Not applicable

Process Memory Performance Information

The following table associates process object performance counters with the data returned by the memory performance functions in the MEMORYSTATUSEX, PERFORMANCE_INFORMATION, and PROCESS_MEMORY_COUNTERS_EX structures, and with the corresponding information displayed by Task Manager.

PROCESS OBJECT COUNTER	STRUCTURE	TASK MANAGER PROCESSES TAB FOR WINDOWS SERVER 2008 AND WINDOWS VISTA	TASK MANAGER PROCESSES TAB FOR WINDOWS SERVER 2003 AND WINDOWS XP
Handle Count	None	**Handles**	**Handles**
Page File Bytes	PROCESS_MEMORY_COUNTERS_EX.**PagefileUsage**	**Commit Size** for all processes except the System process. For the System process, Page File Bytes is always 0.	**VM Size** for all processes except the System process. For the System process, Page File Bytes is always 0.
Page File Bytes Peak	PROCESS_MEMORY_COUNTERS_EX.**PeakPagefileUsage**	None	None
Pool Nonpaged Bytes	PROCESS_MEMORY_COUNTERS_EX.**QuotaNonPagedPoolUsage**	**NP Pool**	**NP Pool**
Pool Paged Bytes	PROCESS_MEMORY_COUNTERS_EX.**QuotaPagedPoolUsage**	**Paged Pool**	**Paged Pool**
Private Bytes	PROCESS_MEMORY_COUNTERS_EX.**PrivateUsage**	**Commit Size**	**VM Size**
Thread Count (Process() for the specified image)	None	**Threads**	**Threads**
Virtual Bytes	MEMORYSTATUSEX.**ullTotalVirtual** – MEMORYSTATUSEX.**ullAvailVirtual**	None	None
Virtual Bytes Peak	None	None	None
Working Set	PROCESS_MEMORY_COUNTERS_EX.**WorkingSetSize**	**Working Set (Memory)**	**Mem Usage**
Working Set Peak	PROCESS_MEMORY_COUNTERS_EX.**PeakWorkingSetSize**	**Peak Working Set (Memory)**	**Peak Mem Usage**
Working Set - Private **Windows Server 2003 and Windows XP:** This performance counter is not supported.	None	**Private Working Set**	Not applicable
None	PROCESS_MEMORY_COUNTERS_EX.**QuotaPeakNonPagedPoolUsage**	None	None

PROCESS OBJECT COUNTER	STRUCTURE	TASK MANAGER PROCESSES TAB FOR WINDOWS SERVER 2008 AND WINDOWS VISTA	TASK MANAGER PROCESSES TAB FOR WINDOWS SERVER 2003 AND WINDOWS XP
None	PROCESS_MEMORY_COUNTERS_EX.QuotaPeakPagedPoolUsage	None	None
None	MEMORYSTATUSEX.ullAvailPageFile	None	None
None	MEMORYSTATUSEX.ullTotalPageFile	None	None

Related topics

Memory Object

Objects Object

Process Object

Process Explorer tool

Virtual Memory Functions

3/5/2021 • 2 minutes to read • Edit Online

The virtual memory functions enable a process to manipulate or determine the status of pages in its virtual address space. They can perform the following operations:

- Reserve a range of a process's virtual address space. Reserving address space does not allocate any physical storage, but it prevents other allocation operations from using the specified range. It does not affect the virtual address spaces of other processes. Reserving pages prevents needless consumption of physical storage, while enabling a process to reserve a range of its address space into which a dynamic data structure can grow. The process can allocate physical storage for this space, as needed.
- Commit a range of reserved pages in a process's virtual address space so that physical storage (either in RAM or on disk) is accessible only to the allocating process.
- Specify read/write, read-only, or no access for a range of committed pages. This differs from the standard allocation functions that always allocate pages with read/write access.
- Free a range of reserved pages, making the range of virtual addresses available for subsequent allocation operations by the calling process.
- Decommit a range of committed pages, releasing their physical storage and making it available for subsequent allocation by any process.
- Lock one or more pages of committed memory into physical memory (RAM) so that the system cannot swap the pages out to the paging file.
- Obtain information about a range of pages in the virtual address space of the calling process or a specified process.
- Change the access protection for a specified range of committed pages in the virtual address space of the calling process or a specified process.

For more information, see the following topics.

- Allocating Virtual Memory
- Comparing Memory Allocation Methods
- Freeing Virtual Memory
- Working With Pages
- Memory Management Functions

Allocating Virtual Memory

3/5/2021 • 2 minutes to read • Edit Online

The virtual memory functions manipulate pages of memory. The functions use the size of a page on the current computer to round off specified sizes and addresses.

The **VirtualAlloc** function performs one of the following operations:

- Reserves one or more free pages.
- Commits one or more reserved pages.
- Reserves and commits one or more free pages.

You can specify the starting address of the pages to be reserved or committed, or you can allow the system to determine the address. The function rounds the specified address to the appropriate page boundary. Reserved pages are not accessible, but committed pages can be allocated with **PAGE_READWRITE**, **PAGE_READONLY**, or **PAGE_NOACCESS** access. When pages are committed, memory charges are allocated from the overall size of RAM and paging files on disk, but each page is initialized and loaded into physical memory only at the first attempt to read from or write to that page. You can use normal pointer references to access memory committed by the **VirtualAlloc** function.

Freeing Virtual Memory

3/5/2021 • 2 minutes to read • Edit Online

The **VirtualFree** function decommits and releases pages according to the following rules:

- Decommits one or more committed pages, changing the state of the pages to reserved. Decommitting pages releases the physical storage associated with the pages, making it available to be allocated by any process. Any block of committed pages can be decommitted.
- Releases a block of one or more reserved pages, changing the state of the pages to free. Releasing a block of pages makes the range of reserved addresses available to be allocated by the process. Reserved pages can be released only by freeing the entire block that was initially reserved by **VirtualAlloc**.
- Decommits and releases a block of one or more committed pages simultaneously, changing the state of the pages to free. The specified block must include the entire block initially reserved by **VirtualAlloc**, and all of the pages must be currently committed.

After a memory block is released or decommitted, you can never refer to it again. Any information that may have been in that memory is gone forever. Attempting to read from or write to a free page results in an access violation exception. If you require information, do not decommit or free memory containing that information.

To specify that the data in a memory range is no longer of interest, call **VirtualAlloc** with **MEM_RESET**. The pages will not be read from or written to the paging file. However, the memory block can be used again later.

Working with Pages

3/5/2021 • 2 minutes to read • Edit Online

To determine the size of a page on the current computer, use the **GetSystemInfo** function.

The **VirtualQuery** and **VirtualQueryEx** functions return information about a region of consecutive pages beginning at a specified address in the address space of a process. **VirtualQuery** returns information about memory in the calling process. **VirtualQueryEx** returns information about memory in a specified process and is used to support debuggers that need information about a process being debugged. The region of pages is bounded by the specified address rounded down to the nearest page boundary. It extends through all subsequent pages with the following attributes in common:

- The state of all pages is the same: either committed, reserved, or free.
- If the initial page is not free, all pages in the region are part of the same initial allocation of pages that were reserved by a call to **VirtualAlloc**.
- The access protection of all pages is the same (that is, **PAGE_READONLY**, **PAGE_READWRITE**, or **PAGE_NOACCESS**).

The **VirtualLock** function enables a process to lock one or more pages of committed memory into physical memory (RAM), preventing the system from swapping the pages out to the paging file. It can be used to ensure that critical data is accessible without disk access. Locking pages into memory is dangerous because it restricts the system's ability to manage memory. Excessive use of **VirtualLock** can degrade system performance by causing executable code to be swapped out to the paging file. The **VirtualUnlock** function unlocks memory locked by **VirtualLock**.

The **VirtualProtect** function enables a process to modify the access protection of any committed page in the address space of a process. For example, a process can allocate read/write pages to store sensitive data, and then it can change the access to read only or no access to protect against accidental overwriting. **VirtualProtect** is typically used with pages allocated by **VirtualAlloc**, but it also works with pages committed by any of the other allocation functions. However, **VirtualProtect** changes the protection of entire pages, and pointers returned by the other functions are not necessarily aligned on page boundaries. The **VirtualProtectEx** function is similar to **VirtualProtect**, except it changes the protection of memory in a specified process. Changing the protection is useful to debuggers in accessing the memory of a process being debugged.

Heap Functions

3/5/2021 • 3 minutes to read • Edit Online

Each process has a default heap provided by the system. Applications that make frequent allocations from the heap can improve performance by using private heaps.

A private heap is a block of one or more pages in the address space of the calling process. After creating the private heap, the process uses functions such as **HeapAlloc** and **HeapFree** to manage the memory in that heap.

The heap functions can also be used to manage memory in the process's default heap, using the handle returned by the **GetProcessHeap** function. New applications should use the heap functions instead of the global and local functions for this purpose.

There is no difference between memory allocated from a private heap and that allocated by using the other memory allocation functions. For a complete list of functions, see the table in Memory Management Functions.

> **NOTE**
> A thread should call heap functions only for the process's default heap and private heaps that the thread creates and manages, using handles returned by the **GetProcessHeap** or **HeapCreate** function.

The **HeapCreate** function creates a private heap object from which the calling process can allocate memory blocks by using the **HeapAlloc** function. **HeapCreate** specifies both an initial size and a maximum size for the heap. The initial size determines the number of committed, read/write pages initially allocated for the heap. The maximum size determines the total number of reserved pages. These pages create a contiguous block in the virtual address space of a process into which the heap can grow. Additional pages are automatically committed from this reserved space if requests by **HeapAlloc** exceed the current size of committed pages, assuming that the physical storage for it is available. Once the pages are committed, they are not decommitted until the process is terminated or until the heap is destroyed by calling the **HeapDestroy** function.

The memory of a private heap object is accessible only to the process that created it. If a dynamic-link library (DLL) creates a private heap, it does so in the address space of the process that called the DLL. It is accessible only to that process.

The **HeapAlloc** function allocates a specified number of bytes from a private heap and returns a pointer to the allocated block. This pointer can be used in the **HeapFree**, **HeapReAlloc**, **HeapSize**, and **HeapValidate** functions.

Memory allocated by **HeapAlloc** is not movable. The address returned by **HeapAlloc** is valid until the memory block is freed or reallocated; the memory block does not need to be locked.

Because the system cannot compact a private heap, it can become fragmented. Applications that allocate large amounts of memory in various allocation sizes can use the low-fragmentation heap to reduce heap fragmentation.

A possible use for the heap functions is to create a private heap when a process starts up, specifying an initial size sufficient to satisfy the memory requirements of the process. If the call to the **HeapCreate** function fails, the process can terminate or notify the user of the memory shortage; if it succeeds, however, the process is assured of having the memory it needs.

Memory requested by **HeapCreate** may or may not be contiguous. Memory allocated within a heap by **HeapAlloc** is contiguous. You should not write to or read from memory in a heap except that allocated by **HeapAlloc**, nor should you assume any relationship between two areas of memory allocated by **HeapAlloc**.

You should not refer in any way to memory that has been freed by **HeapFree**. After the memory is freed, any information that may have been in it is gone forever. If you require information, do not free memory containing the information. Function calls that return information about memory (such as **HeapSize**) may not be used with freed memory, as they may return bogus data.

The **HeapDestroy** function destroys a private heap object. It decommits and releases all the pages of the heap object, and it invalidates the handle to the heap.

Related topics

Comparing Memory Allocation Methods

Low-fragmentation Heap

3/5/2021 • 2 minutes to read • Edit Online

[The information in this topic applies to Windows Server 2003 and Windows XP. Starting with Windows Vista, the system uses the low-fragmentation heap (LFH) as needed to service memory allocation requests. Applications do not need to enable the LFH for their heaps.]

Heap fragmentation is a state in which available memory is broken into small, noncontiguous blocks. When a heap is fragmented, memory allocation can fail even when the total available memory in the heap is enough to satisfy a request, because no single block of memory is large enough. The low-fragmentation heap (LFH) helps to reduce heap fragmentation.

The LFH is not a separate heap. Instead, it is a policy that applications can enable for their heaps. When the LFH is enabled, the system allocates memory in certain predetermined sizes. When an application requests a memory allocation from a heap that has the LFH enabled, the system allocates the smallest block of memory that is large enough to contain the requested size. The system does not use the LFH for allocations larger than 16 KB, whether or not the LFH is enabled.

An application should enable the LFH only for the default heap of the calling process or for private heaps that the application has created. To enable the LFH for a heap, use the **GetProcessHeap** function to obtain a handle to the default heap of the calling process, or use the handle to a private heap created by the **HeapCreate** function. Then call the **HeapSetInformation** function with the handle.

The LFH cannot be enabled for heaps created with **HEAP_NO_SERIALIZE** or for heaps created with a fixed size. The LFH also cannot be enabled if you are using the heap debugging tools in Debugging Tools for Windows or Microsoft Application Verifier.

After the LFH has been enabled for a heap, it cannot be disabled.

Applications that benefit most from the LFH are multithreaded applications that allocate memory frequently and use a variety of allocation sizes under 16 KB. However, not all applications benefit from the LFH. To assess the effects of enabling the LFH in your application, use performance profiling data.

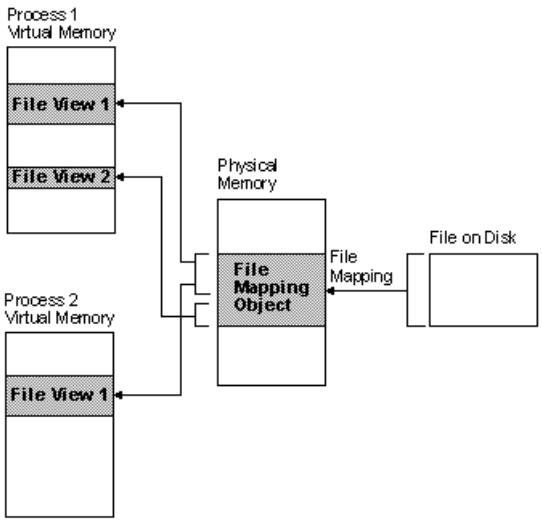
File Mapping

3/5/2021 • 2 minutes to read • Edit Online

File mapping is the association of a file's contents with a portion of the virtual address space of a process. The system creates a *file mapping object* (also known as a *section object*) to maintain this association. A *file view* is the portion of virtual address space that a process uses to access the file's contents. File mapping allows the process to use both random input and output (I/O) and sequential I/O. It also allows the process to work efficiently with a large data file, such as a database, without having to map the whole file into memory. Multiple processes can also use memory-mapped files to share data.

Processes read from and write to the file view using pointers, just as they would with dynamically allocated memory. The use of file mapping improves efficiency because the file resides on disk, but the file view resides in memory. Processes can also manipulate the file view with the **VirtualProtect** function.

The following illustration shows the relationship between the file on disk, a file mapping object, and a file view.



The file on disk can be any file that you want to map into memory, or it can be the system page file. The file mapping object can consist of all or only part of the file. It is backed by the file on disk. This means that when the system swaps out pages of the file mapping object, any changes made to the file mapping object are written to the file. When the pages of the file mapping object are swapped back in, they are restored from the file.

A file view can consist of all or only part of the file mapping object. A process manipulates the file through the file views. A process can create multiple views for a file mapping object. The file views created by each process reside in the virtual address space of that process. When the process needs data from a portion of the file other than what is in the current file view, it can unmap the current file view, then create a new file view.

When multiple processes use the same file mapping object to create views for a local file, the data is coherent. That is, the views contain identical copies of the file on disk. The file cannot reside on a remote computer if you want to share memory between multiple processes.

For more information, see the following topics:

- Creating a File Mapping Object
- Creating a File View
- Sharing Files and Memory
- Reading and Writing From a File View
- Closing a File Mapping Object

- File Mapping Security and Access Rights
- Using File Mapping

Creating a File Mapping Object

6/6/2021 • 3 minutes to read • Edit Online

The first step in mapping a file is to open the file by calling the **CreateFile** function. To ensure that other processes cannot write to the portion of the file that is mapped, you should open the file with exclusive access. In addition, the file handle should remain open until the process no longer needs the file mapping object. An easy way to obtain exclusive access is to specify zero in the *fdwShareMode* parameter of **CreateFile**. The handle returned by **CreateFile** is used by the **CreateFileMapping** function to create a file mapping object.

The **CreateFileMapping** function returns a handle to the file mapping object. This handle will be used when creating a file view so that you can access the shared memory. When you call **CreateFileMapping**, you specify an object name, the number of bytes to be mapped from the file, and the read/write permission for the mapped memory. The first process that calls **CreateFileMapping** creates the file mapping object. Processes calling **CreateFileMapping** for an existing object receive a handle to the existing object. You can tell whether or not a successful call to **CreateFileMapping** created or opened the file mapping object by calling the **GetLastError** function. **GetLastError** returns **NO_ERROR** to the creating process and **ERROR_ALREADY_EXISTS** to subsequent processes.

The **CreateFileMapping** function fails if the access flags conflict with those specified when the **CreateFile** function opened the file. For example, to read and write to the file:

- Specify the **GENERIC_READ** and **GENERIC_WRITE** values in the *fdwAccess* parameter of **CreateFile**.
- Specify the **PAGE_READWRITE** value in the *fdwProtect* parameter of **CreateFileMapping**.

Creating a file mapping object does not commit physical memory, it only reserves it.

File Mapping Size

The size of the file mapping object is independent of the size of the file being mapped. However, if the file mapping object is larger than the file, the system expands the file before **CreateFileMapping** returns. If the file mapping object is smaller than the file, the system maps only the specified number of bytes from the file.

The *dwMaximumSizeHigh* and *dwMaximumSizeLow* parameters of **CreateFileMapping** allow you to specify the number of bytes to be mapped from the file:

- When you do not want the size of the file to change (for example, when mapping read-only files), call **CreateFileMapping** and specify zero for both *dwMaximumSizeHigh* and *dwMaximumSizeLow*. Doing this creates a file mapping object that is exactly the same size as the file. Otherwise, you must calculate or estimate the size of the finished file because file mapping objects are static in size; once created, their size cannot be increased or decreased. An attempt to map a file with a length of zero in this manner fails with an error code of **ERROR_FILE_INVALID**. Programs should test for files with a length of zero and reject such files.

- The size of a file mapping object that is backed by a named file is limited by disk space. The size of a file view is limited to the largest available contiguous block of unreserved virtual memory. This is at most 2 GB minus the virtual memory already reserved by the process.

The size of the file mapping object that you select controls how far into the file you can "see" with memory mapping. If you create a file mapping object that is 500 Kb in size, you have access only to the first 500 Kb of the file, regardless of the size of the file. Since it does not cost you any system resources to create a larger file mapping object, create a file mapping object that is the size of the file (set the *dwMaximumSizeHigh* and *dwMaximumSizeLow* parameters of **CreateFileMapping** both to zero) even if you do not expect to view the

entire file. The cost in system resources comes in creating the views and accessing them.

You can view a portion of the file that does not start at the beginning of the file. For more information, see Creating a View Within a File.

Related topics

Creating a File View

Creating a View Within a File

Creating a File View

3/5/2021 • 2 minutes to read • Edit Online

To map the data from a file to the virtual memory of a process, you must create a view of the file. The **MapViewOfFile** and **MapViewOfFileEx** functions use the file mapping object handle returned by **CreateFileMapping** to create a view of the file or a portion of the file in the process's virtual address space. These functions fail if the access flags conflict with those specified when **CreateFileMapping** created the file mapping object.

The **MapViewOfFile** function returns a pointer to the file view. By dereferencing a pointer in the range of addresses specified in **MapViewOfFile**, an application can read data from the file and write data to the file. Writing to the file view results in changes to the file mapping object. The actual writing to the file on disk is handled by the system. Data is not actually transferred at the time the file mapping object is written to. Instead, much of the file input and output (I/O) is cached to improve general system performance. Applications can override this behavior by calling the **FlushViewOfFile** function to force the system to perform disk transactions immediately.

The **MapViewOfFileEx** function works exactly like the **MapViewOfFile** function except that it allows a process to specify the base address of the view of the file in the process's virtual address space in the *lpvBase* parameter. If there is not enough space at the specified address, the call fails. Therefore, if you must map a file to the same address in multiple processes, the processes should negotiate an appropriate address: The *lpvBase* parameter must be an integral multiple of the system memory allocation granularity or the call fails. To obtain the system's memory allocation granularity, use the **GetSystemInfo** function, which fills in the members of a **SYSTEM_INFO** structure.

An application can create multiple file views from the same file mapping object. A file view can be a different size than the file mapping object from which it is derived, but it must be smaller than the file mapping object. The offset specified by the *dwOffsetHigh* and *dwOffsetLow* parameters of **MapViewOfFile** must be a multiple of the allocation granularity of the system.

Related topics

Creating a View Within a File

Sharing Files and Memory

3/5/2021 • 2 minutes to read • Edit Online

File mapping can be used to share a file or memory between two or more processes. To share a file or memory, all of the processes must use the name or the handle of the same file mapping object.

To share a file, the first process creates or opens a file by using the **CreateFile** function. Next, it creates a file mapping object by using the **CreateFileMapping** function, specifying the file handle and a name for the file mapping object. The names of event, semaphore, mutex, waitable timer, job, and file mapping objects share the same namespace. Therefore, the **CreateFileMapping** and **OpenFileMapping** functions fail if they specify a name that is in use by an object of another type.

To share memory that is not associated with a file, a process must use the **CreateFileMapping** function and specify INVALID_HANDLE_VALUE as the *hFile* parameter instead of an existing file handle. The corresponding file mapping object accesses memory backed by the system paging file. You must specify a size greater than zero when you specify an *hFile* of INVALID_HANDLE_VALUE in a call to **CreateFileMapping**.

The easiest way for other processes to obtain a handle of the file mapping object created by the first process is to use the **OpenFileMapping** function and specify the object's name. This is referred to as *named shared memory*. If the file mapping object does not have a name, the process must obtain a handle to it through inheritance or duplication. For more information on inheritance and duplication, see Inheritance.

Processes that share files or memory must create file views by using the **MapViewOfFile** or **MapViewOfFileEx** function. They must coordinate their access using semaphores, mutexes, events, or some other mutual exclusion technique. For more information, see Synchronization.

A shared file mapping object will not be destroyed until all processes that use it close their handles to it by using the **CloseHandle** function.

For information about file mapping object security, see File Mapping Security and Access Rights.

Related topics

Creating Named Shared Memory

Reading and Writing From a File View

3/5/2021 • 2 minutes to read • Edit Online

To read from a file view, dereference the pointer returned by the **MapViewOfFile** function as shown in the examples below.

Reading from or writing to a file view of a file other than the page file can cause an **EXCEPTION_IN_PAGE_ERROR** exception. For example, accessing a mapped file that resides on a remote server can generate an exception if the connection to the server is lost. Exceptions can also occur because of a full disk, an underlying device failure, or a memory allocation failure. When writing to a file view, exceptions can also occur because the file is shared and a different process has locked a byte range. To guard against exceptions due to input and output (I/O) errors, all attempts to access memory mapped files should be wrapped in structured exception handlers. When you receive **EXCEPTION_IN_PAGE_ERROR** in your **__except** filter, make sure that the address is within the mapping you are currently accessing. If so, recover or fail gracefully; otherwise, do not handle the exception.

The following example uses the pointer returned by **MapViewOfFile** to read from the file view:

```
DWORD dwLength;

__try
{
  dwLength = *((LPDWORD) lpMapAddress);
}
__except(GetExceptionCode()==EXCEPTION_IN_PAGE_ERROR ?
  EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
{
  // Failed to read from the view.
}
```

The following example uses the pointer returned by **MapViewOfFile** to write to the file view:

```
DWORD dwLength;

__try
{
  *((LPDWORD) lpMapAddress) = dwLength;
}
__except (GetExceptionCode() == EXCEPTION_IN_PAGE_ERROR ?
  EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH)
{
  // Failed to write to the view.
}
```

The **FlushViewOfFile** function copies the specified number of bytes of the file view to the physical file, without waiting for the cached write operation to occur:

```
if (!FlushViewOfFile(lpMapAddress, dwBytesToFlush))
{
  printf("Could not flush memory to disk (%d).\n", GetLastError());
}
```

If you are mapping a compressed or sparse file on an NTFS partition, there is additional potential for an I/O error when paging in a portion of the file. In this case, the address space mapped by **MapViewOfFile** may not

be backed by allocated disk space. This is because a sparse file can have regions of zeros for which NTFS does not allocate disk space, and a compressed file can take up less disk space than the actual data that it represents. If you read from or write to a portion of a sparse or compressed file that is not backed by disk space, the operating system may try to allocate disk space. If the disk is full, this can result in an exception indicating an I/O error.

Related topics

Structured Exception Handling

Closing a File Mapping Object

3/5/2021 • 2 minutes to read • Edit Online

When a process has finished with the file mapping object, it should destroy all file views in its address space by using the **UnmapViewOfFile** function for each file view.

Unmapping a mapped view of a file invalidates the range occupied by the view in the address space of the process and makes the range available for other allocations. It removes the working set entry for each unmapped virtual page that was part of the working set of the process and reduces the working set size of the process. It also decrements the share count of the corresponding physical page.

Modified pages in the unmapped view are not written to disk until their share count reaches zero, or in other words, until they are unmapped or trimmed from the working sets of all processes that share the pages. Even then, the modified pages are written "lazily" to disk; that is, modifications may be cached in memory and written to disk at a later time. To minimize the risk of data loss in the event of a power failure or a system crash, applications should explicitly flush modified pages using the **FlushViewOfFile** function.

When each process finishes using the file mapping object and has unmapped all views, it must close the file mapping object's handle and the file on disk by calling **CloseHandle**. These calls to **CloseHandle** succeed even when there are file views that are still open. However, leaving file views mapped causes memory leaks.

File-mapping security and access rights

3/5/2021 • 2 minutes to read • Edit Online

The Windows security model lets you control access to file-mapping objects. For more information, see Access-Control Model.

You can specify a security descriptor for a file-mapping object when you call the **CreateFileMapping** function. If you specify **NULL**, the object gets a default security descriptor. The ACLs in the default security descriptor for a file-mapping object come from the primary or impersonation token of the creator.

To retrieve the security descriptor of a file-mapping object, call the **GetNamedSecurityInfo** or **GetSecurityInfo** function. To set the security descriptor of a file-mapping object, call the **SetNamedSecurityInfo** or **SetSecurityInfo** function.

The valid access rights for file-mapping objects include the **DELETE**, **READ_CONTROL**, **WRITE_DAC**, and **WRITE_OWNER** standard access rights. File mapping objects do not support the **SYNCHRONIZE** standard access right. The following table lists the access rights that are specific to file-mapping objects.

ACCESS RIGHT	MEANING
FILE_MAP_ALL_ACCESS	Includes all access rights to a file-mapping object except **FILE_MAP_EXECUTE**. The **MapViewOfFile** and **MapViewOfFileEx** functions treat this the same as specifying **FILE_MAP_WRITE**.
FILE_MAP_EXECUTE	Allows mapping of executable views of the file-mapping object. The object must have been created with page protection that allows execute access, such as **PAGE_EXECUTE_READ**, **PAGE_EXECUTE_WRITECOPY**, or **PAGE_EXECUTE_READWRITE** protection.
FILE_MAP_READ	Allows mapping of read-only or copy-on-write views of the file-mapping object.
FILE_MAP_WRITE	Allows mapping of read-only, copy-on-write, or read/write views of a file-mapping object. The object must have been created with page protection that allows write access, such as **PAGE_READWRITE** or **PAGE_EXECUTE_READWRITE** protection.

Mapping a copy-on-write view of a file-mapping object requires the same access as mapping a read-only view. The **FILE_MAP_COPY** flag is not an access right, and it should not be specified as part of a DACL in a security descriptor. This value can be used only with functions that map a view of a file-mapping object, such as the **MapViewOfFile** and **MapViewOfFileEx** functions, or with the **OpenFileMapping** function, which treats **FILE_MAP_COPY** the same way it treats **FILE_MAP_READ**.

You can request the **ACCESS_SYSTEM_SECURITY** access right to a file-mapping object if you want to read or write the object's SACL. For more information, see Access-Control Lists (ACLs) and SACL Access Right.

Large Memory Support

3/5/2021 • 2 minutes to read • Edit Online

The following features enable applications to access more memory:

- 4-Gigabyte Tuning
- Physical Address Extension
- Address Windowing Extensions
- Large-Page Support

4-Gigabyte Tuning: BCDEdit and Boot.ini

3/5/2021 • 3 minutes to read • Edit Online

On 32-bit editions of Windows, applications have 4 gigabyte (GB) of virtual address space available. The virtual address space is divided so that 2 GB is available to the application and the other 2 GB is available only to the system. The 4-gigabyte tuning (4GT or 4GT RAM Tuning) feature, enabled with the *BCDEdit /set increaseuserva* command, increases the virtual address space that is available to the application up to 3 GB, and reduces the amount available to the system to between 1 and 2 GB.

For applications that are memory-intensive, such as database management systems (DBMS), the use of a larger virtual address space can provide considerable performance and scalability benefits. However, the file cache, paged pool, and nonpaged pool are smaller, which can adversely affect applications with heavy networking or I/O. Therefore, you might want to test your application under load, and examine the performance counters to determine whether your application benefits from the larger address space.

To enable 4GT, use the BCDEdit /set command to set the **increaseuserva** boot entry option to a value between 2048 (2 GB) and 3072 (3 GB).

Windows Server 2003 and earlier: To enable 4GT, add the **/3GB** switch to the Boot.ini file. The **/3GB** switch is supported on the following systems:

- Windows Server 2003
- Windows XP Professional

The **/3GB** switch makes a full 3 GB of virtual address space available to applications and reduces the amount available to the system to 1 GB. On Windows Server 2003, the amount of address space available to applications can be adjusted by setting the **/USERVA** switch in Boot.ini to a value between 2048 and 3072, which increases the amount of address space available to the system. This can help maintain overall system performance when the application requires more than 2 GB but less than 3 GB of address space.

To enable an application to use the larger address space, set the **IMAGE_FILE_LARGE_ADDRESS_AWARE** flag in the image header. The linker included with Microsoft Visual C++ supports the **/LARGEADDRESSAWARE** switch to set this flag. Setting this flag and then running the application on a system that does not have 4GT support should not affect the application.

On 64-bit editions of Windows, 32-bit applications marked with the **IMAGE_FILE_LARGE_ADDRESS_AWARE** flag have 4 GB of address space available.

Itanium editions of Windows Server 2003: Prior to SP1, 32-bit processes have only 2 GB of address space available.

Use the following guidelines to support 4GT in applications:

- Addresses near the 2-GB boundary are typically used by various system DLLs. Therefore, a 32-bit process cannot allocate more than 2 GB of contiguous memory, even if the entire 4-GB address space is available.
- To retrieve the amount of total user virtual space, use the **GlobalMemoryStatusEx** function. To retrieve the highest possible user address, use the **GetSystemInfo** function. Always detect the real value at runtime, and avoid using hard-wired constant definitions such as: `#define HIGHEST_USER_ADDRESS 0xC0000000`.
- Avoid signed comparisons with pointers, because they might cause applications to crash on a 4GT-enabled system. A condition such as the following is false for a pointer that is above 2 GB: `if (pointer > 40000000)`.
- Code that uses the highest bit of a pointer for an application-defined purpose will fail when 4GT is enabled. For example, a 32-bit word might be considered a user-mode address if it is below 0x80000000, and an error

code if above. This is not true with 4GT.

VirtualAlloc usually returns low addresses before high addresses. Therefore, your process may not use very high addresses unless it allocates a lot of memory or has a fragmented virtual address space. To force allocations to allocate from higher addresses before lower addresses for testing purposes, specify **MEM_TOP_DOWN** when calling **VirtualAlloc** or set the following registry value to 0x100000:

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Session Manager\Memory Management\AllocationPreference

Related topics

Memory Limits for Windows Releases

Physical Address Extension

4GT Technical Reference

Physical Address Extension

3/5/2021 • 3 minutes to read • Edit Online

Physical Address Extension (PAE) is a processor feature that enables x86 processors to access more than 4 GB of physical memory on capable versions of Windows. Certain 32-bit versions of Windows Server running on x86-based systems can use PAE to access up to 64 GB or 128 GB of physical memory, depending on the physical address size of the processor. For details, see Memory Limits for Windows Releases.

The Intel Itanium and x64 processor architectures can access more than 4 GB of physical memory natively and therefore do not provide the equivalent of PAE. PAE is used only by 32-bit versions of Windows running on x86-based systems.

With PAE, the operating system moves from two-level linear address translation to three-level address translation. Instead of a linear address being split into three separate fields for indexing into memory tables, it is split into four separate fields: a 2-bit bitfield, two 9-bit bitfields, and a 12-bit bitfield that corresponds to the page size implemented by Intel architecture (4 KB). The size of page table entries (PTEs) and page directory entries (PDEs) in PAE mode is increased from 32 to 64 bits. The additional bits allow an operating system PTE or PDE to reference physical memory above 4 GB.

In 32-bit Windows running on x64-based systems, PAE also enables several advanced system and processor features, including hardware-enabled Data Execution Prevention (DEP), non-uniform memory access (NUMA), and the ability to add memory to a system while it is running (hot-add memory).

PAE does not change the amount of virtual address space available to a process. Each process running in 32-bit Windows is still limited to a 4 GB virtual address space.

System Support for PAE

PAE is supported only on the following 32-bit versions of Windows running on x86-based systems:

- Windows 7 (32 bit only)
- Windows Server 2008 (32-bit only)
- Windows Vista (32-bit only)
- Windows Server 2003 (32-bit only)
- Windows XP (32-bit only)

Enabling PAE

Windows automatically enables PAE if DEP is enabled on a computer that supports hardware-enabled DEP, or if the computer is configured for hot-add memory devices in memory ranges beyond 4 GB. If the computer does not support hardware-enabled DEP or is not configured for hot-add memory devices in memory ranges beyond 4 GB, PAE must be explicitly enabled.

To explicitly enable PAE, use the following **BCDEdit /set** command to set the **pae** boot entry option:

```
bcdedit /set [{ID}] pae ForceEnable
```

IF DEP is enabled, PAE cannot be disabled. Use the following **BCDEdit /set** commands to disable both DEP and PAE:

```
bcdedit /set [{ID}] nx AlwaysOff
bcdedit /set [{ID}] pae ForceDisable
```

Windows Server 2003 and Windows XP: To enable PAE, use the **/PAE** switch in the boot.ini file. To disable PAE, use the **/NOPAE** switch. To disable DEP, use the **/EXECUTE** switch.

Comparing PAE and other Large Memory Support

PAE, 4-gigabyte tuning (4GT), and Address Windowing Extensions (AWE) serve different purposes and can be used independently of each other:

- PAE allows the operating system to access and use more than 4 GB of physical memory.
- 4GT increases the portion of the virtual address space that is available to a process from 2 GB to up to 3 GB.
- AWE is a set of APIs that allows a process to allocate nonpaged physical memory and then dynamically map portions of this memory into the virtual address space of the process.

When neither 4GT nor AWE are being used, the amount of physical memory that a single 32-bit process can use is limited by the size of its address space (2 GB). In this case, a PAE-enabled system can still make use of more than 4 GB of RAM to run multiple processes at the same time or to cache file data in memory.

4GT can be used with or without PAE. However, some versions of Windows limit the maximum amount of physical memory that can be supported when 4GT is used. On such systems, booting with 4GT enabled causes the operating system to ignore any memory in excess of the limit.

AWE does not require PAE or 4GT but is often used together with PAE to allocate more than 4 GB of physical memory from a single 32-bit process.

Related topics

IsProcessorFeaturePresent

PAE X86 Technical Reference

Address Windowing Extensions

3/5/2021 • 3 minutes to read • Edit Online

Address Windowing Extensions (AWE) is a set of extensions that allows an application to quickly manipulate physical memory greater than 4GB. Certain data-intensive applications, such as database management systems and scientific and engineering software, need access to very large caches of data. In the case of very large data sets, restricting the cache to fit within an application's 2GB of user address space is a severe restriction. In these situations, the cache is too small to properly support the application.

AWE solves this problem by allowing applications to directly address huge amounts of memory while continuing to use 32-bit pointers. AWE allows applications to have data caches larger than 4GB (where sufficient physical memory is present). AWE uses physical nonpaged memory and window views of various portions of this physical memory within a 32-bit virtual address space.

AWE places a few restrictions on how this memory may be used, primarily because these restrictions allow extremely fast mapping, remapping, and freeing. Fast memory management is important for these potentially enormous address spaces.

- Virtual address ranges allocated for the AWE are not sharable with other processes (and therefore not inheritable). In fact, two different AWE virtual addresses within the same process are not allowed to map the same physical page. These restrictions provide fast remapping and cleanup when memory is freed.
- The physical pages that can be allocated for an AWE region are limited by the number of physical pages present in the machine, since this memory is never paged – it is locked down until the application explicitly frees it or exits. The physical pages allocated for a given process can be mapped into any AWE virtual region within the same process. Applications that use AWE must be careful not to take so much physical memory that they cause other applications to page excessively or prevent creation of new processes or threads due to lack of resources. Use the **GlobalMemoryStatusEx** function to monitor physical memory use.
- AWE virtual addresses are always read/write and cannot be protected via calls to **VirtualProtect** (that is, no read-only memory, noaccess memory, guard pages, and the like can be specified).
- AWE address ranges cannot be used to buffer data for graphics or video calls.
- An AWE memory range cannot be split, nor can pieces of it be deleted. Instead, the entire virtual address range must be deleted as a unit when deletion is required. This means you must specify **MEM_RELEASE** when calling **VirtualFree**.
- Applications can map multiple regions simultaneously, provided they do not overlap.
- Applications that use AWE are not supported in emulation mode. That is, an x86 application that uses AWE functions must be recompiled to run on another processor, whereas most applications can run without recompiling under an emulator on other platforms.

This solution addresses the physical memory issues in a very general, widely applicable manner. Some of the benefits of AWE are:

- A small group of new functions is defined to manipulate AWE memory.
- AWE provides a very fast remapping capability. Remapping is done by manipulating virtual memory tables, not by moving data in physical memory.
- AWE provides page size granularity appropriate to the processor (for example, 4 KB on x86), which is more useful to applications than large pages (for example, 2MB or 4MB on x86).

An application must have the Lock Pages in Memory privilege to use AWE. To obtain this privilege, an administrator must add **Lock Pages in Memory** to the user's **User Rights Assignments**. For more information on how to do this, see "User Rights" in the operating system help.

The following functions make up the Address Windowing Extensions (AWE) API.

FUNCTION	DESCRIPTION
VirtualAlloc and VirtualAllocEx	Reserve a portion of virtual address space to use for AWE, using **MEM_PHYSICAL**.
AllocateUserPhysicalPages	Allocate physical memory for use with AWE.
MapUserPhysicalPages	Map (or invalidate) AWE virtual addresses onto any set of physical pages obtained with **AllocateUserPhysicalPages**.
MapUserPhysicalPagesScatter	Map (or invalidate) AWE virtual addresses onto any set of physical pages obtained with **AllocateUserPhysicalPages**, but with finer control than that provided by **MapUserPhysicalPages**.
FreeUserPhysicalPages	Free physical memory that was used for AWE.

Large-Page Support

3/5/2021 • 2 minutes to read • Edit Online

Large-page support enables server applications to establish large-page memory regions, which is particularly useful on 64-bit Windows. Each large-page translation uses a single translation buffer inside the CPU. The size of this buffer is typically three orders of magnitude larger than the native page size; this increases the efficiency of the translation buffer, which can increase performance for frequently accessed memory.

The following procedure describes how to use large-page support.

To use large-page support

1. Obtain the **SeLockMemoryPrivilege** privilege by calling the **AdjustTokenPrivileges** function. For more information, see Assigning Privileges to an Account and Changing Privileges in a Token.
2. Retrieve the minimum large-page size by calling the **GetLargePageMinimum** function.
3. Include the **MEM_LARGE_PAGES** value when calling the **VirtualAlloc** function. The size and alignment must be a multiple of the large-page minimum.

When writing applications that use large-page memory, keep the following considerations in mind:

- Large-page memory regions may be difficult to obtain after the system has been running for a long time because the physical space for each large page must be contiguous, but the memory may have become fragmented. Allocating large pages under these conditions can significantly affect system performance. Therefore, applications should avoid making repeated large-page allocations and instead allocate all large pages one time, at startup.
- The memory is always read/write and nonpageable (always resident in physical memory).
- The memory is part of the process private bytes but not part of the working set, because the working set by definition contains only pageable memory.
- Large-page allocations are not subject to job limits.
- Large-page memory must be reserved and committed as a single operation. In other words, large pages cannot be used to commit a previously reserved range of memory.
- WOW64 on Intel Itanium-based systems does not support 32-bit applications that use this feature. The applications should be recompiled as native 64-bit applications.

Global and Local Functions

3/5/2021 • 2 minutes to read • Edit Online

The global and local functions are supported for porting from 16-bit code, or for maintaining source code compatibility with 16-bit Windows. Starting with 32-bit Windows, the global and local functions are implemented as wrapper functions that call the corresponding heap functions using a handle to the process's default heap. Therefore, the global and local functions have greater overhead than other memory management functions.

The heap functions provide more features and control than the global and local functions. New applications should use the heap functions unless documentation specifically states that a global or local function should be used. For example, some Windows functions allocate memory that must be freed with **LocalFree**, and the global functions are still used with Dynamic Data Exchange (DDE), the clipboard functions, and OLE data objects. For a complete list of global and local functions, see the table in Memory Management Functions.

Windows memory management does not provide a separate local heap and global heap, as 16-bit Windows does. As a result, the global and local families of functions are equivalent and choosing between them is a matter of personal preference. Note that the change from a 16-bit segmented memory model to a 32-bit virtual memory model has made some of the related global and local functions and their options unnecessary or meaningless. For example, there are no longer near and far pointers, because both local and global allocations return 32-bit virtual addresses.

Memory objects allocated by **GlobalAlloc** and **LocalAlloc** are in private, committed pages with read/write access that cannot be accessed by other processes. Memory allocated by using **GlobalAlloc** with **GMEM_DDESHARE** is not actually shared globally as it is in 16-bit Windows. This value has no effect and is available only for compatibility. Applications requiring shared memory for other purposes must use file-mapping objects. Multiple processes can map a view of the same file-mapping object to provide named shared memory. For more information, see File Mapping.

Memory allocations are limited only by the available physical memory, including storage in the paging file on disk. When you allocate fixed memory, **GlobalAlloc** and **LocalAlloc** return a pointer that the calling process can immediately use to access the memory. When you allocate moveable memory, the return value is a handle. To get a pointer to a movable memory object, use the **GlobalLock** and **LocalLock** functions.

The actual size of the memory allocated can be larger than the requested size. To determine the actual number of bytes allocated, use the **GlobalSize** or **LocalSize** function. If the amount allocated is greater than the amount requested, the process can use the entire amount.

The **GlobalReAlloc** and **LocalReAlloc** functions change the size or the attributes of a memory object allocated by **GlobalAlloc** and **LocalAlloc**. The size can increase or decrease.

The **GlobalFree** and **LocalFree** functions release memory allocated by **GlobalAlloc**, **LocalAlloc**, **GlobalReAlloc**, or **LocalReAlloc**. To discard the specified memory object without invalidating the handle, use the **GlobalDiscard** or **LocalDiscard** function. The handle can be used later by **GlobalReAlloc** or **LocalReAlloc** to allocate a new block of memory associated with the same handle.

To return information about a specified memory object, use the **GlobalFlags** or **LocalFlags** function. The information includes the object's lock count and indicates whether the object is discardable or has already been discarded. To return a handle to the memory object associated with a specified pointer, use the **GlobalHandle** or **LocalHandle** function.

Related topics

Comparing Memory Allocation Methods

Standard C Library Functions

3/5/2021 • 2 minutes to read • Edit Online

Applications can safely use the memory management features of the C run-time library (**malloc**, **free**, and so on) and C++ (**new**, **delete**, and so on). The C run-time library functions do not have the potential problems they have under 16-bit Windows. Memory management is no longer a problem because the system is free to manage memory by moving pages of physical memory without affecting the virtual addresses. Similarly, the distinction between near and far pointers is no longer relevant. Therefore, you can use the standard C library functions for memory management. However, the memory management functions do provide functionality that is unavailable in the C run-time library.

Comparing Memory Allocation Methods

3/10/2021 • 2 minutes to read • Edit Online

The following is a brief comparison of the various memory allocation methods:

- CoTaskMemAlloc
- GlobalAlloc
- HeapAlloc
- LocalAlloc
- **malloc**
- **new**
- VirtualAlloc

Although the GlobalAlloc, LocalAlloc, and HeapAlloc functions ultimately allocate memory from the same heap, each provides a slightly different set of functionality. For example, **HeapAlloc** can be instructed to raise an exception if memory could not be allocated, a capability not available with **LocalAlloc**. **LocalAlloc** supports allocation of handles which permit the underlying memory to be moved by a reallocation without changing the handle value, a capability not available with **HeapAlloc**.

Starting with 32-bit Windows, GlobalAlloc and LocalAlloc are implemented as wrapper functions that call HeapAlloc using a handle to the process's default heap. Therefore, **GlobalAlloc** and **LocalAlloc** have greater overhead than **HeapAlloc**.

Because the different heap allocators provide distinctive functionality by using different mechanisms, you must free memory with the correct function. For example, memory allocated with HeapAlloc must be freed with HeapFree and not LocalFree or GlobalFree. Memory allocated with GlobalAlloc or LocalAlloc must be queried, validated, and released with the corresponding global or local function.

The VirtualAlloc function allows you to specify additional options for memory allocation. However, its allocations use a page granularity, so using **VirtualAlloc** can result in higher memory usage.

The **malloc** function has the disadvantage of being run-time dependent. The **new** operator has the disadvantage of being compiler dependent and language dependent.

The CoTaskMemAlloc function has the advantage of working well in either C, C++, or Visual Basic. It is also the only way to share memory in a COM-based application, since MIDL uses **CoTaskMemAlloc** and CoTaskMemFree to marshal memory.

Examples

- Reserving and Committing Memory
- AWE Example

Related topics

Global and Local Functions

Heap Functions

Virtual Memory Functions

Using the Memory Management Functions

3/5/2021 • 2 minutes to read • Edit Online

The following examples demonstrate the use of the memory management functions:

- Reserving and Committing Memory
- Creating Guard Pages
- Enumerating a Heap
- Getting Process Heaps
- Using File Mapping
- AWE Example
- Allocating Memory from a NUMA Node

Reserving and Committing Memory

3/5/2021 • 3 minutes to read • Edit Online

The following example illustrates the use of the **VirtualAlloc** and **VirtualFree** functions in reserving and committing memory as needed for a dynamic array. First, **VirtualAlloc** is called to reserve a block of pages with **NULL** specified as the base address parameter, forcing the system to determine the location of the block. Later, **VirtualAlloc** is called whenever it is necessary to commit a page from this reserved region, and the base address of the next page to be committed is specified.

The example uses structured exception-handling syntax to commit pages from the reserved region. Whenever a page fault exception occurs during the execution of the **__try** block, the filter function in the expression preceding the **__except** block is executed. If the filter function can allocate another page, execution continues in the **__try** block at the point where the exception occurred. Otherwise, the exception handler in the **__except** block is executed. For more information, see Structured Exception Handling.

As an alternative to dynamic allocation, the process can simply commit the entire region instead of only reserving it. Both methods result in the same physical memory usage because committed pages do not consume any physical storage until they are first accessed. The advantage of dynamic allocation is that it minimizes the total number of committed pages on the system. For very large allocations, pre-committing an entire allocation can cause the system to run out of committable pages, resulting in virtual memory allocation failures.

The **ExitProcess** function in the **__except** block automatically releases virtual memory allocations, so it is not necessary to explicitly free the pages when the program terminates through this execution path. The **VirtualFree** function frees the reserved and committed pages if the program is built with exception handling disabled. This function uses **MEM_RELEASE** to decommit and release the entire region of reserved and committed pages.

The following C++ example demonstrates dynamic memory allocation using a structured exception handler.

```
// A short program to demonstrate dynamic memory allocation
// using a structured exception handler.

#include <windows.h>
#include <tchar.h>
#include <stdio.h>
#include <stdlib.h>             // For exit

#define PAGELIMIT 80            // Number of pages to ask for

LPTSTR lpNxtPage;               // Address of the next page to ask for
DWORD dwPages = 0;              // Count of pages gotten so far
DWORD dwPageSize;               // Page size on this computer

INT PageFaultExceptionFilter(DWORD dwCode)
{
    LPVOID lpvResult;

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

    if (dwCode != EXCEPTION_ACCESS_VIOLATION)
    {
        _tprintf(TEXT("Exception code = %d.\n"), dwCode);
        return EXCEPTION_EXECUTE_HANDLER;
    }

     tprintf(TEXT("Exception is a page fault.\n"));
```

```
    _tprintf(TEXT("Exception is a page fault.\n"));

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

    if (dwPages >= PAGELIMIT)
    {
        _tprintf(TEXT("Exception: out of pages.\n"));
        return EXCEPTION_EXECUTE_HANDLER;
    }

    // Otherwise, commit another page.

    lpvResult = VirtualAlloc(
                    (LPVOID) lpNxtPage, // Next page to commit
                    dwPageSize,         // Page size, in bytes
                    MEM_COMMIT,         // Allocate a committed page
                    PAGE_READWRITE);    // Read/write access
    if (lpvResult == NULL )
    {
        _tprintf(TEXT("VirtualAlloc failed.\n"));
        return EXCEPTION_EXECUTE_HANDLER;
    }
    else
    {
        _tprintf(TEXT("Allocating another page.\n"));
    }

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

    dwPages++;
    lpNxtPage = (LPTSTR) ((PCHAR) lpNxtPage + dwPageSize);

    // Continue execution where the page fault occurred.

    return EXCEPTION_CONTINUE_EXECUTION;
}

VOID ErrorExit(LPTSTR lpMsg)
{
    _tprintf(TEXT("Error! %s with error code of %ld.\n"),
             lpMsg, GetLastError ());
    exit (0);
}

VOID _tmain(VOID)
{
    LPVOID lpvBase;               // Base address of the test memory
    LPTSTR lpPtr;                 // Generic character pointer
    BOOL bSuccess;                // Flag
    DWORD i;                      // Generic counter
    SYSTEM_INFO sSysInfo;         // Useful information about the system

    GetSystemInfo(&sSysInfo);     // Initialize the structure.

    _tprintf (TEXT("This computer has page size %d.\n"), sSysInfo.dwPageSize);

    dwPageSize = sSysInfo.dwPageSize;

    // Reserve pages in the virtual address space of the process.

    lpvBase = VirtualAlloc(
                    NULL,                 // System selects address
                    PAGELIMIT*dwPageSize, // Size of allocation
                    MEM_RESERVE,          // Allocate reserved pages
                    PAGE_NOACCESS);       // Protection = no access
    if (lpvBase == NULL )
        ErrorExit(TEXT("VirtualAlloc reserve failed."));

    lpPtr = lpNxtPage = (LPTSTR) lpvBase;
```

```
    // Use 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*dwPageSize; i++)
    {
        __try
        {
            // Write to memory.

            lpPtr[i] = 'a';
        }

        // If there's a page fault, commit another page and try again.

        __except ( PageFaultExceptionFilter( GetExceptionCode() ) )
        {

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

            _tprintf (TEXT("Exiting process.\n"));

            ExitProcess( GetLastError() );

        }

    }

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

    bSuccess = VirtualFree(
                       lpvBase,       // Base address of block
                       0,             // Bytes of committed pages
                       MEM_RELEASE);  // Decommit the pages

    _tprintf (TEXT("Release %s.\n"), bSuccess ? TEXT("succeeded") : TEXT("failed") );

}
```

Creating Guard Pages

3/5/2021 • 2 minutes to read • Edit Online

A guard page provides a one-shot alarm for memory page access. This can be useful for an application that needs to monitor the growth of large dynamic data structures. For example, there are operating systems that use guard pages to implement automatic stack checking.

To create a guard page, set the **PAGE_GUARD** page protection modifier for the page. This value can be specified, along with other page protection modifiers, in the **VirtualAlloc**, **VirtualAllocEx**, **VirtualProtect**, and **VirtualProtectEx** functions. The **PAGE_GUARD** modifier can be used with any other page protection modifiers, except **PAGE_NOACCESS**.

If a program attempts to access an address within a guard page, the system raises a **STATUS_GUARD_PAGE_VIOLATION** (0x80000001) exception. The system also clears the **PAGE_GUARD** modifier, removing the memory page's guard page status. The system will not stop the next attempt to access the memory page with a **STATUS_GUARD_PAGE_VIOLATION** exception.

If a guard page exception occurs during a system service, the service fails and typically returns some failure status indicator. Since the system also removes the relevant memory page's guard page status, the next invocation of the same system service won't fail due to a **STATUS_GUARD_PAGE_VIOLATION** exception (unless, of course, someone reestablishes the guard page).

The following short program illustrates the behavior of guard page protection.

```
/* A program to demonstrate the use of guard pages of memory. Allocate
   a page of memory as a guard page, then try to access the page. That
   will fail, but doing so releases the lock on the guard page, so the
   next access works correctly.

   The output will look like this. The actual address may vary.

   This computer has a page size of 4096.
   Committed 4096 bytes at address 0x00520000
   Cannot lock at 00520000, error = 0x80000001
   2nd Lock Achieved at 00520000

   This sample does not show how to use the guard page fault to
   "grow" a dynamic array, such as a stack. */

#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <tchar.h>

int main()
{
  LPVOID lpvAddr;               // address of the test memory
  DWORD dwPageSize;             // amount of memory to allocate.
  BOOL bLocked;                 // address of the guarded memory
  SYSTEM_INFO sSysInfo;         // useful information about the system

  GetSystemInfo(&sSysInfo);     // initialize the structure

  _tprintf(TEXT("This computer has page size %d.\n"), sSysInfo.dwPageSize);

  dwPageSize = sSysInfo.dwPageSize;

  // Try to allocate the memory.
```

```
  lpvAddr = VirtualAlloc(NULL, dwPageSize,
                         MEM_RESERVE | MEM_COMMIT,
                         PAGE_READONLY | PAGE_GUARD);

  if(lpvAddr == NULL) {
    _tprintf(TEXT("VirtualAlloc failed. Error: %ld\n"),
             GetLastError());
    return 1;

  } else {
    _ftprintf(stderr, TEXT("Committed %lu bytes at address 0x%lp\n"),
              dwPageSize, lpvAddr);
  }

  // Try to lock the committed memory. This fails the first time
  // because of the guard page.

  bLocked = VirtualLock(lpvAddr, dwPageSize);
  if (!bLocked) {
    _ftprintf(stderr, TEXT("Cannot lock at %lp, error = 0x%lx\n"),
              lpvAddr, GetLastError());
  } else {
    _ftprintf(stderr, TEXT("Lock Achieved at %lp\n"), lpvAddr);
  }

  // Try to lock the committed memory again. This succeeds the second
  // time because the guard page status was removed by the first
  // access attempt.

  bLocked = VirtualLock(lpvAddr, dwPageSize);

  if (!bLocked) {
    _ftprintf(stderr, TEXT("Cannot get 2nd lock at %lp, error = %lx\n"),
              lpvAddr, GetLastError());
  } else {
    _ftprintf(stderr, TEXT("2nd Lock Achieved at %lp\n"), lpvAddr);
  }

  return 0;
}
```

The first attempt to lock the memory block fails, raising a **STATUS_GUARD_PAGE_VIOLATION** exception. The second attempt succeeds, because the memory block's guard page protection has been toggled off by the first attempt.

Enumerating a Heap

3/5/2021 • 2 minutes to read • Edit Online

The following example illustrates the use of the HeapWalk function to enumerate a heap.

First, the example creates a private heap with the HeapCreate function. Then it uses HeapLock to lock the heap so other threads cannot access the heap while it is being enumerated. The example then calls HeapWalk with a pointer to a PROCESS_HEAP_ENTRY structure and iterates through the heap, printing each entry to the console.

After enumeration is finished, the example uses HeapUnlock to unlock the heap so that other threads can access it. Finally, the example calls HeapDestroy to destroy the private heap.

```
#include <windows.h>
#include <tchar.h>
#include <stdio.h>

int __cdecl _tmain()
{
    DWORD LastError;
    HANDLE hHeap;
    PROCESS_HEAP_ENTRY Entry;

    //
    // Create a new heap with default parameters.
    //
    hHeap = HeapCreate(0, 0, 0);
    if (hHeap == NULL) {
        _tprintf(TEXT("Failed to create a new heap with LastError %d.\n"),
                 GetLastError());
        return 1;
    }

    //
    // Lock the heap to prevent other threads from accessing the heap
    // during enumeration.
    //
    if (HeapLock(hHeap) == FALSE) {
        _tprintf(TEXT("Failed to lock heap with LastError %d.\n"),
                 GetLastError());
        return 1;
    }

    _tprintf(TEXT("Walking heap %#p...\n\n"), hHeap);

    Entry.lpData = NULL;
    while (HeapWalk(hHeap, &Entry) != FALSE) {
        if ((Entry.wFlags & PROCESS_HEAP_ENTRY_BUSY) != 0) {
            _tprintf(TEXT("Allocated block"));

            if ((Entry.wFlags & PROCESS_HEAP_ENTRY_MOVEABLE) != 0) {
                _tprintf(TEXT(", movable with HANDLE %#p"), Entry.Block.hMem);
            }

            if ((Entry.wFlags & PROCESS_HEAP_ENTRY_DDESHARE) != 0) {
                _tprintf(TEXT(", DDESHARE"));
            }
        }
        else if ((Entry.wFlags & PROCESS_HEAP_REGION) != 0) {
            _tprintf(TEXT("Region\n  %d bytes committed\n") \
                     TEXT("  %d bytes uncommitted\n  First block address: %#p\n") \
```

```
                    TEXT("  Last block address: %#p\n"),
                    Entry.Region.dwCommittedSize,
                    Entry.Region.dwUnCommittedSize,
                    Entry.Region.lpFirstBlock,
                    Entry.Region.lpLastBlock);
        }
        else if ((Entry.wFlags & PROCESS_HEAP_UNCOMMITTED_RANGE) != 0) {
            _tprintf(TEXT("Uncommitted range\n"));
        }
        else {
            _tprintf(TEXT("Block\n"));
        }

        _tprintf(TEXT("  Data portion begins at: %#p\n  Size: %d bytes\n") \
                 TEXT("  Overhead: %d bytes\n  Region index: %d\n\n"),
                 Entry.lpData,
                 Entry.cbData,
                 Entry.cbOverhead,
                 Entry.iRegionIndex);
    }
    LastError = GetLastError();
    if (LastError != ERROR_NO_MORE_ITEMS) {
        _tprintf(TEXT("HeapWalk failed with LastError %d.\n"), LastError);
    }

    //
    // Unlock the heap to allow other threads to access the heap after
    // enumeration has completed.
    //
    if (HeapUnlock(hHeap) == FALSE) {
        _tprintf(TEXT("Failed to unlock heap with LastError %d.\n"),
                 GetLastError());
    }

    //
    // When a process terminates, allocated memory is reclaimed by the operating
    // system so it is not really necessary to call HeapDestroy in this example.
    // However, it may be advisable to call HeapDestroy in a longer running
    // application.
    //
    if (HeapDestroy(hHeap) == FALSE) {
        _tprintf(TEXT("Failed to destroy heap with LastError %d.\n"),
                 GetLastError());
    }

    return 0;
}
```

The following output shows typical results for a newly created heap.

```
Walking heap 0X00530000...

Region
  4096 bytes committed
  258048 bytes uncommitted
  First block address: 0X00530598
  Last block address: 0X00570000
  Data portion begins at: 0X00530000
  Size: 1416 bytes
  Overhead: 0 bytes
  Region index: 0

Block
  Data portion begins at: 0X005307D8
  Size: 2056 bytes
  Overhead: 16 bytes
  Region index: 0

Uncommitted range
  Data portion begins at: 0X00531000
  Size: 258048 bytes
  Overhead: 0 bytes
  Region index: 0
```

Getting Process Heaps

3/5/2021 • 2 minutes to read • Edit Online

This example illustrates the use of the GetProcessHeaps function to retrieve handles to the default process heap and any private heaps that are active for the current process.

The example calls GetProcessHeaps twice, first to calculate the size of the buffer needed and again to retrieve handles into the buffer. The buffer is allocated from the default process heap, using the handle returned by GetProcessHeap. The example prints the starting address of each heap to the console. It then uses the HeapFree function to free memory allocated for the buffer.

The number of heaps in a process may vary. A process always has at least one heap—the default process heap—and it may have one or more private heaps created by the application or by DLLs that are loaded into the address space of the process.

Note that an application should call heap functions only on its default process heap or on private heaps that the application has created; calling heap functions on a private heap owned by another component may cause undefined behavior.

```
#include <windows.h>
#include <tchar.h>
#include <stdio.h>
#include <intsafe.h>

int __cdecl _tmain()
{
    DWORD NumberOfHeaps;
    DWORD HeapsIndex;
    DWORD HeapsLength;
    HANDLE hDefaultProcessHeap;
    HRESULT Result;
    PHANDLE aHeaps;
    SIZE_T BytesToAllocate;

    //
    // Retrieve the number of active heaps for the current process
    // so we can calculate the buffer size needed for the heap handles.
    //
    NumberOfHeaps = GetProcessHeaps(0, NULL);
    if (NumberOfHeaps == 0) {
        _tprintf(TEXT("Failed to retrieve the number of heaps with LastError %d.\n"),
                 GetLastError());
        return 1;
    }

    //
    // Calculate the buffer size.
    //
    Result = SIZETMult(NumberOfHeaps, sizeof(*aHeaps), &BytesToAllocate);
    if (Result != S_OK) {
        _tprintf(TEXT("SIZETMult failed with HR %d.\n"), Result);
        return 1;
    }

    //
    // Get a handle to the default process heap.
    //
    hDefaultProcessHeap = GetProcessHeap();
    if (hDefaultProcessHeap == NULL) {
        _tprintf(TEXT("Failed to retrieve the default process heap with LastError %d.\n")
```

```
        _tprintf(TEXT("Failed to retrieve the default process heap with LastError %d.\n"),
                 GetLastError());
        return 1;
    }

    //
    // Allocate the buffer from the default process heap.
    //
    aHeaps = (PHANDLE)HeapAlloc(hDefaultProcessHeap, 0, BytesToAllocate);
    if (aHeaps == NULL) {
        _tprintf(TEXT("HeapAlloc failed to allocate %d bytes.\n"),
                 BytesToAllocate);
        return 1;
    }

    //
    // Save the original number of heaps because we are going to compare it
    // to the return value of the next GetProcessHeaps call.
    //
    HeapsLength = NumberOfHeaps;

    //
    // Retrieve handles to the process heaps and print them to stdout.
    // Note that heap functions should be called only on the default heap of the process
    // or on private heaps that your component creates by calling HeapCreate.
    //
    NumberOfHeaps = GetProcessHeaps(HeapsLength, aHeaps);
    if (NumberOfHeaps == 0) {
        _tprintf(TEXT("Failed to retrieve heaps with LastError %d.\n"),
                 GetLastError());
        return 1;
    }
    else if (NumberOfHeaps > HeapsLength) {

        //
        // Compare the latest number of heaps with the original number of heaps.
        // If the latest number is larger than the original number, another
        // component has created a new heap and the buffer is too small.
        //
        _tprintf(TEXT("Another component created a heap between calls. ") \
                 TEXT("Please try again.\n"));
        return 1;
    }

    _tprintf(TEXT("Process has %d heaps.\n"), HeapsLength);
    for (HeapsIndex = 0; HeapsIndex < HeapsLength; ++HeapsIndex) {
        _tprintf(TEXT("Heap %d at address: %#p.\n"),
                 HeapsIndex,
                 aHeaps[HeapsIndex]);
    }

    //
    // Release memory allocated from default process heap.
    //
    if (HeapFree(hDefaultProcessHeap, 0, aHeaps) == FALSE) {
        _tprintf(TEXT("Failed to free allocation from default process heap.\n"));
    }

    return 0;
}
```

Using File Mapping

3/5/2021 • 2 minutes to read • Edit Online

The following examples demonstrate how two processes might access an existing file as named shared memory:

- Creating a View Within a File
- Creating Named Shared Memory
- Creating a File Mapping Using Large Pages
- Obtaining a File Name From a File Handle

The processes must synchronize their access to the memory. For more information, see Synchronization.

Creating a View Within a File

3/5/2021 • 5 minutes to read • Edit Online

If you want to view a portion of the file that does not start at the beginning of the file, you must create a file mapping object. This object is the size of the portion of the file you want to view plus the offset into the file. For example, if you want to view the 1 kilobyte (1K) that begins 131,072 bytes (128K) into the file, you must create a file mapping object of at least 132,096 bytes (129K) in size. The view starts 131,072 bytes (128K) into the file and extend for at least 1,024 bytes. This example assumes a file allocation granularity of 64K.

File allocation granularity affects where a map view can start. A map view must start at an offset into the file that is a multiple of the file allocation granularity. So the data you want to view may be the file offset modulo the allocation granularity into the view. The size of the view is the offset of the data modulo the allocation granularity, plus the size of the data that you want to examine.

For example, suppose that the **GetSystemInfo** function indicates an allocation granularity of 64K. To examine 1K of data that is 138,240 bytes (135K) into the file, do the following:

1. Create a file mapping object of at least 139,264 bytes (136K) in size.
2. Create a file view that starts at a file offset that is the largest multiple of the file allocation granularity less than the offset you require. In this case, the file view starts at offset 131,072 (128K) into the file. The view is 139264 bytes (136K) minus 131,072 bytes (128K), or 8,192 bytes (8K), in size.
3. Create a pointer offset 7K into the view to access the 1K in which you are interested.

If the data you want straddles a file allocation granularity boundary, you could make the view larger than the file allocation granularity. This avoids breaking the data into pieces.

The following program illustrates the second example above.

```
/*
   This program demonstrates file mapping, especially how to align a
   view with the system file allocation granularity.
*/

#include <windows.h>
#include <stdio.h>
#include <tchar.h>

#define BUFFSIZE 1024 // size of the memory to examine at any one time

#define FILE_MAP_START 138240 // starting point within the file of
                              // the data to examine (135K)

/* The test file. The code below creates the file and populates it,
   so there is no need to supply it in advance. */

TCHAR * lpcTheFile = TEXT("fmtest.txt"); // the file to be manipulated

int main(void)
{
  HANDLE hMapFile;      // handle for the file's memory-mapped region
  HANDLE hFile;         // the file handle
  BOOL bFlag;           // a result holder
  DWORD dBytesWritten;  // number of bytes written
  DWORD dwFileSize;     // temporary storage for file sizes
  DWORD dwFileMapSize;  // size of the file mapping
  DWORD dwMapViewSize;  // the size of the view
  DWORD dwFileMapStart; // where to start the file map view
```

```
DWORD dwSysGran;      // system allocation granularity
SYSTEM_INFO SysInfo;  // system information; used to get granularity
LPVOID lpMapAddress;  // pointer to the base address of the
                      // memory-mapped region
char * pData;         // pointer to the data
int i;                // loop counter
int iData;            // on success contains the first int of data
int iViewDelta;       // the offset into the view where the data
                      //shows up

// Create the test file. Open it "Create Always" to overwrite any
// existing file. The data is re-created below
hFile = CreateFile(lpcTheFile,
                   GENERIC_READ | GENERIC_WRITE,
                   0,
                   NULL,
                   CREATE_ALWAYS,
                   FILE_ATTRIBUTE_NORMAL,
                   NULL);

if (hFile == INVALID_HANDLE_VALUE)
{
  _tprintf(TEXT("hFile is NULL\n"));
  _tprintf(TEXT("Target file is %s\n"),
           lpcTheFile);
  return 4;
}

// Get the system allocation granularity.
GetSystemInfo(&SysInfo);
dwSysGran = SysInfo.dwAllocationGranularity;

// Now calculate a few variables. Calculate the file offsets as
// 64-bit values, and then get the low-order 32 bits for the
// function calls.

// To calculate where to start the file mapping, round down the
// offset of the data into the file to the nearest multiple of the
// system allocation granularity.
dwFileMapStart = (FILE_MAP_START / dwSysGran) * dwSysGran;
_tprintf (TEXT("The file map view starts at %ld bytes into the file.\n"),
        dwFileMapStart);

// Calculate the size of the file mapping view.
dwMapViewSize = (FILE_MAP_START % dwSysGran) + BUFFSIZE;
_tprintf (TEXT("The file map view is %ld bytes large.\n"),
          dwMapViewSize);

// How large will the file mapping object be?
dwFileMapSize = FILE_MAP_START + BUFFSIZE;
_tprintf (TEXT("The file mapping object is %ld bytes large.\n"),
        dwFileMapSize);

// The data of interest isn't at the beginning of the
// view, so determine how far into the view to set the pointer.
iViewDelta = FILE_MAP_START - dwFileMapStart;
_tprintf (TEXT("The data is %d bytes into the view.\n"),
          iViewDelta);

// Now write a file with data suitable for experimentation. This
// provides unique int (4-byte) offsets in the file for easy visual
// inspection. Note that this code does not check for storage
// medium overflow or other errors, which production code should
// do. Because an int is 4 bytes, the value at the pointer to the
// data should be one quarter of the desired offset into the file

for (i=0; i<(int)dwSysGran; i++)
{
  WriteFile (hFile, &i, sizeof (i), &dBytesWritten, NULL);
```

```
}

// Verify that the correct file size was written.
dwFileSize = GetFileSize(hFile,  NULL);
_tprintf(TEXT("hFile size: %10d\n"), dwFileSize);

// Create a file mapping object for the file
// Note that it is a good idea to ensure the file size is not zero
hMapFile = CreateFileMapping( hFile,          // current file handle
              NULL,           // default security
              PAGE_READWRITE, // read/write permission
              0,              // size of mapping object, high
              dwFileMapSize,  // size of mapping object, low
              NULL);          // name of mapping object

if (hMapFile == NULL)
{
  _tprintf(TEXT("hMapFile is NULL: last error: %d\n"), GetLastError() );
  return (2);
}

// Map the view and test the results.

lpMapAddress = MapViewOfFile(hMapFile,            // handle to
                                                  // mapping object
                             FILE_MAP_ALL_ACCESS, // read/write
                             0,                   // high-order 32
                                                  // bits of file
                                                  // offset
                             dwFileMapStart,      // low-order 32
                                                  // bits of file
                                                  // offset
                             dwMapViewSize);      // number of bytes
                                                  // to map
if (lpMapAddress == NULL)
{
  _tprintf(TEXT("lpMapAddress is NULL: last error: %d\n"), GetLastError());
  return 3;
}

// Calculate the pointer to the data.
pData = (char *) lpMapAddress + iViewDelta;

// Extract the data, an int. Cast the pointer pData from a "pointer
// to char" to a "pointer to int" to get the whole thing
iData = *(int *)pData;

_tprintf (TEXT("The value at the pointer is %d,\nwhich %s one quarter of the desired file offset.\n"),
          iData,
          iData*4 == FILE_MAP_START ? TEXT("is") : TEXT("is not"));

// Close the file mapping object and the open file

bFlag = UnmapViewOfFile(lpMapAddress);
bFlag = CloseHandle(hMapFile); // close the file mapping object

if(!bFlag)
{
  _tprintf(TEXT("\nError %ld occurred closing the mapping object!"),
           GetLastError());
}

bFlag = CloseHandle(hFile);   // close the file itself

if(!bFlag)
{
  _tprintf(TEXT("\nError %ld occurred closing the file!"),
         GetLastError());
}
```

```
  return 0;
}
```

Related topics

Creating a File View

Creating Named Shared Memory

3/5/2021 • 2 minutes to read • Edit Online

To share data, multiple processes can use memory-mapped files that the system paging file stores.

First Process

The first process creates the file mapping object by calling the **CreateFileMapping** function with **INVALID_HANDLE_VALUE** and a name for the object. By using the **PAGE_READWRITE** flag, the process has read/write permission to the memory through any file views that are created.

Then the process uses the file mapping object handle that **CreateFileMapping** returns in a call to **MapViewOfFile** to create a view of the file in the process address space. The **MapViewOfFile** function returns a pointer to the file view, `pBuf`. The process then uses the **CopyMemory** function to write a string to the view that can be accessed by other processes.

Prefixing the file mapping object names with "Global\" allows processes to communicate with each other even if they are in different terminal server sessions. This requires that the first process must have the **SeCreateGlobalPrivilege** privilege.

When the process no longer needs access to the file mapping object, it should call the **CloseHandle** function. When all handles are closed, the system can free the section of the paging file that the object uses.

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include <tchar.h>

#define BUF_SIZE 256
TCHAR szName[]=TEXT("Global\\MyFileMappingObject");
TCHAR szMsg[]=TEXT("Message from first process.");

int _tmain()
{
   HANDLE hMapFile;
   LPCTSTR pBuf;

   hMapFile = CreateFileMapping(
                 INVALID_HANDLE_VALUE,    // use paging file
                 NULL,                    // default security
                 PAGE_READWRITE,          // read/write access
                 0,                       // maximum object size (high-order DWORD)
                 BUF_SIZE,                // maximum object size (low-order DWORD)
                 szName);                 // name of mapping object

   if (hMapFile == NULL)
   {
      _tprintf(TEXT("Could not create file mapping object (%d).\n"),
             GetLastError());
      return 1;
   }
   pBuf = (LPTSTR) MapViewOfFile(hMapFile,   // handle to map object
                       FILE_MAP_ALL_ACCESS, // read/write permission
                       0,
                       0,
                       BUF_SIZE);

   if (pBuf == NULL)
   {
      _tprintf(TEXT("Could not map view of file (%d).\n"),
             GetLastError());

       CloseHandle(hMapFile);

      return 1;
   }


   CopyMemory((PVOID)pBuf, szMsg, (_tcslen(szMsg) * sizeof(TCHAR)));
    _getch();

   UnmapViewOfFile(pBuf);

   CloseHandle(hMapFile);

   return 0;
}
```

Second Process

A second process can access the string written to the shared memory by the first process by calling the **OpenFileMapping** function specifying the same name for the mapping object as the first process. Then it can use the **MapViewOfFile** function to obtain a pointer to the file view, `pBuf`. The process can display this string as it would any other string. In this example, the message box displayed contains the message "Message from first process" that was written by the first process.

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include <tchar.h>
#pragma comment(lib, "user32.lib")

#define BUF_SIZE 256
TCHAR szName[]=TEXT("Global\\MyFileMappingObject");

int _tmain()
{
   HANDLE hMapFile;
   LPCTSTR pBuf;

   hMapFile = OpenFileMapping(
                  FILE_MAP_ALL_ACCESS,   // read/write access
                  FALSE,                 // do not inherit the name
                  szName);               // name of mapping object

   if (hMapFile == NULL)
   {
      _tprintf(TEXT("Could not open file mapping object (%d).\n"),
             GetLastError());
      return 1;
   }

   pBuf = (LPTSTR) MapViewOfFile(hMapFile, // handle to map object
               FILE_MAP_ALL_ACCESS,  // read/write permission
               0,
               0,
               BUF_SIZE);

   if (pBuf == NULL)
   {
      _tprintf(TEXT("Could not map view of file (%d).\n"),
             GetLastError());

      CloseHandle(hMapFile);

      return 1;
   }

   MessageBox(NULL, pBuf, TEXT("Process2"), MB_OK);

   UnmapViewOfFile(pBuf);

   CloseHandle(hMapFile);

   return 0;
}
```

Related topics

Sharing Files and Memory

Creating a File Mapping Using Large Pages

3/5/2021 • 2 minutes to read • Edit Online

The following example uses the **CreateFileMapping** function with the **SEC_LARGE_PAGES** flag to use large pages. The buffer must be large enough to contain the minimum size of a large page. This value is obtained using the **GetLargePageMinimum** function. This feature also requires the "SeLockMemoryPrivilege" privilege.

> **NOTE**
>
> Starting in Windows 10, version 1703, the **MapViewOfFile** function maps a view using small pages by default, even for file mapping objects created with the **SEC_LARGE_PAGES** flag. In this and later OS versions, you must specify the **FILE_MAP_LARGE_PAGES** flag with the **MapViewOfFile** function to map large pages. This flag is ignored on OS versions before Windows 10, version 1703.

```
#include <windows.h>
#include <tchar.h>
#include <stdio.h>

#define BUF_SIZE 65536

TCHAR szName[]=TEXT("LARGEPAGE");
typedef int (*GETLARGEPAGEMINIMUM)(void);

void DisplayError(const wchar_t* pszAPI, DWORD dwError)
{
    LPVOID lpvMessageBuffer;

    FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER |
                  FORMAT_MESSAGE_FROM_SYSTEM |
                  FORMAT_MESSAGE_IGNORE_INSERTS,
                  NULL, dwError,
                  MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
                  (LPTSTR)&lpvMessageBuffer, 0, NULL);

    //... now display this string
    _tprintf(TEXT("ERROR: API        = %s\n"), pszAPI);
    _tprintf(TEXT("       error code = %d\n"), dwError);
    _tprintf(TEXT("       message    = %s\n"), lpvMessageBuffer);

    // Free the buffer allocated by the system
    LocalFree(lpvMessageBuffer);

    ExitProcess(GetLastError());
}

void Privilege(const wchar_t* pszPrivilege, BOOL bEnable)
{
    HANDLE           hToken;
    TOKEN_PRIVILEGES tp;
    BOOL             status;
    DWORD            error;

    // open process token
    if (!OpenProcessToken(GetCurrentProcess(), TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY, &hToken))
        DisplayError(TEXT("OpenProcessToken"), GetLastError());

    // get the luid
    if (!LookupPrivilegeValue(NULL, pszPrivilege, &tp.Privileges[0].Luid))
        DisplayError(TEXT("LookupPrivilegeValue"), GetLastError());
```

```
    tp.PrivilegeCount = 1;

    // enable or disable privilege
    if (bEnable)
        tp.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;
    else
        tp.Privileges[0].Attributes = 0;

    // enable or disable privilege
    status = AdjustTokenPrivileges(hToken, FALSE, &tp, 0, (PTOKEN_PRIVILEGES)NULL, 0);

    // It is possible for AdjustTokenPrivileges to return TRUE and still not succeed.
    // So always check for the last error value.
    error = GetLastError();
    if (!status || (error != ERROR_SUCCESS))
        DisplayError(TEXT("AdjustTokenPrivileges"), GetLastError());

    // close the handle
    if (!CloseHandle(hToken))
        DisplayError(TEXT("CloseHandle"), GetLastError());
}

int _tmain(void)
{
    HANDLE hMapFile;
    LPCTSTR pBuf;
    DWORD size;
    GETLARGEPAGEMINIMUM pGetLargePageMinimum;
    HINSTANCE  hDll;

    // call succeeds only on Windows Server 2003 SP1 or later
    hDll = LoadLibrary(TEXT("kernel32.dll"));
    if (hDll == NULL)
        DisplayError(TEXT("LoadLibrary"), GetLastError());

    pGetLargePageMinimum = (GETLARGEPAGEMINIMUM)GetProcAddress(hDll,
        "GetLargePageMinimum");
    if (pGetLargePageMinimum == NULL)
        DisplayError(TEXT("GetProcAddress"), GetLastError());

    size = (*pGetLargePageMinimum)();

    FreeLibrary(hDll);

    _tprintf(TEXT("Page Size: %u\n"), size);

    Privilege(TEXT("SeLockMemoryPrivilege"), TRUE);

    hMapFile = CreateFileMapping(
         INVALID_HANDLE_VALUE,    // use paging file
         NULL,                    // default security
         PAGE_READWRITE | SEC_COMMIT | SEC_LARGE_PAGES,
         0,                       // max. object size
         size,                    // buffer size
         szName);                 // name of mapping object

    if (hMapFile == NULL)
        DisplayError(TEXT("CreateFileMapping"), GetLastError());
    else
        _tprintf(TEXT("File mapping object successfully created.\n"));

    Privilege(TEXT("SeLockMemoryPrivilege"), FALSE);

    pBuf = (LPTSTR) MapViewOfFile(hMapFile,          // handle to map object
         FILE_MAP_ALL_ACCESS | FILE_MAP_LARGE_PAGES, // read/write permission
         0,
         0,
         BUF_SIZE);
```

```
    if (pBuf == NULL)
        DisplayError(TEXT("MapViewOfFile"), GetLastError());
    else
        _tprintf(TEXT("View of file successfully mapped.\n"));

    // do nothing, clean up an exit
    UnmapViewOfFile(pBuf);
    CloseHandle(hMapFile);
}
```

Related topics

Creating a File Mapping Object

Large Page Support

Obtaining a File Name From a File Handle

3/5/2021 • 2 minutes to read • Edit Online

GetFinalPathNameByHandle, introduced in Windows Vista and Windows Server 2008, will return a path from a handle. If you need to do this on earlier releases of Windows, the following example obtains a file name from a handle to a file object using a file mapping object. It uses the CreateFileMapping and MapViewOfFile functions to create the mapping. Next, it uses the GetMappedFileName function to obtain the file name. For remote files, it prints the device path received from this function. For local files, it converts the path to use a drive letter and prints this path. To test this code, create a **main** function that opens a file using CreateFile and passes the resulting handle to `GetFileNameFromHandle`.

```
#include <windows.h>
#include <stdio.h>
#include <tchar.h>
#include <string.h>
#include <psapi.h>
#include <strsafe.h>

#define BUFSIZE 512

BOOL GetFileNameFromHandle(HANDLE hFile)
{
  BOOL bSuccess = FALSE;
  TCHAR pszFilename[MAX_PATH+1];
  HANDLE hFileMap;

  // Get the file size.
  DWORD dwFileSizeHi = 0;
  DWORD dwFileSizeLo = GetFileSize(hFile, &dwFileSizeHi);

  if( dwFileSizeLo == 0 && dwFileSizeHi == 0 )
  {
     _tprintf(TEXT("Cannot map a file with a length of zero.\n"));
     return FALSE;
  }

  // Create a file mapping object.
  hFileMap = CreateFileMapping(hFile,
                    NULL,
                    PAGE_READONLY,
                    0,
                    1,
                    NULL);

  if (hFileMap)
  {
    // Create a file mapping to get the file name.
    void* pMem = MapViewOfFile(hFileMap, FILE_MAP_READ, 0, 0, 1);

    if (pMem)
    {
      if (GetMappedFileName (GetCurrentProcess(),
                             pMem,
                             pszFilename,
                             MAX_PATH))
      {

        // Translate path with device name to drive letters.
        TCHAR szTemp[BUFSIZE];
        szTemp[0] = '\0';
```

```
        if (GetLogicalDriveStrings(BUFSIZE-1, szTemp))
        {
          TCHAR szName[MAX_PATH];
          TCHAR szDrive[3] = TEXT(" :");
          BOOL bFound = FALSE;
          TCHAR* p = szTemp;

          do
          {
            // Copy the drive letter to the template string
            *szDrive = *p;

            // Look up each device name
            if (QueryDosDevice(szDrive, szName, MAX_PATH))
            {
              size_t uNameLen = _tcslen(szName);

              if (uNameLen < MAX_PATH)
              {
                bFound = _tcsnicmp(pszFilename, szName, uNameLen) == 0
                         && *(pszFilename + uNameLen) == _T('\\');

                if (bFound)
                {
                  // Reconstruct pszFilename using szTempFile
                  // Replace device path with DOS path
                  TCHAR szTempFile[MAX_PATH];
                  StringCchPrintf(szTempFile,
                            MAX_PATH,
                            TEXT("%s%s"),
                            szDrive,
                            pszFilename+uNameLen);
                  StringCchCopyN(pszFilename, MAX_PATH+1, szTempFile, _tcslen(szTempFile));
                }
              }
            }

            // Go to the next NULL character.
            while (*p++);
          } while (!bFound && *p); // end of string
        }
      }
      bSuccess = TRUE;
      UnmapViewOfFile(pMem);
    }

    CloseHandle(hFileMap);
  }
  _tprintf(TEXT("File name is %s\n"), pszFilename);
  return(bSuccess);
}

int _tmain(int argc, TCHAR *argv[])
{
    HANDLE hFile;

    if( argc != 2 )
    {
        _tprintf(TEXT("This sample takes a file name as a parameter.\n"));
        return 0;
    }
    hFile = CreateFile(argv[1], GENERIC_READ, FILE_SHARE_READ, NULL,
        OPEN_EXISTING, 0, NULL);

    if(hFile == INVALID_HANDLE_VALUE)
    {
        _tprintf(TEXT("CreateFile failed with %d\n"), GetLastError());
        return 0;
```

```
    }
    GetFileNameFromHandle( hFile );
}
```

Related topics

Using File Mapping

GetFinalPathNameByHandle

AWE Example

3/5/2021 • 2 minutes to read • Edit Online

The following sample program illustrates the **Address Windowing Extensions**.

```
#include <windows.h>
#include <stdio.h>
#include <tchar.h>

#define MEMORY_REQUESTED 1024*1024 // request a megabyte

BOOL
LoggedSetLockPagesPrivilege ( HANDLE hProcess,
                              BOOL bEnable);

void _cdecl main()
{
  BOOL bResult;                   // generic Boolean value
  ULONG_PTR NumberOfPages;        // number of pages to request
  ULONG_PTR NumberOfPagesInitial; // initial number of pages requested
  ULONG_PTR *aPFNs;               // page info; holds opaque data
  PVOID lpMemReserved;            // AWE window
  SYSTEM_INFO sSysInfo;           // useful system information
  int PFNArraySize;               // memory to request for PFN array

  GetSystemInfo(&sSysInfo);  // fill the system information structure

  _tprintf(_T("This computer has page size %d.\n"), sSysInfo.dwPageSize);

  // Calculate the number of pages of memory to request.

  NumberOfPages = MEMORY_REQUESTED/sSysInfo.dwPageSize;
  _tprintf (_T("Requesting %d pages of memory.\n"), NumberOfPages);

  // Calculate the size of the user PFN array.

  PFNArraySize = NumberOfPages * sizeof (ULONG_PTR);

  _tprintf (_T("Requesting a PFN array of %d bytes.\n"), PFNArraySize);

  aPFNs = (ULONG_PTR *) HeapAlloc(GetProcessHeap(), 0, PFNArraySize);

  if (aPFNs == NULL)
  {
    _tprintf (_T("Failed to allocate on heap.\n"));
    return;
  }

  // Enable the privilege.

  if( ! LoggedSetLockPagesPrivilege( GetCurrentProcess(), TRUE ) )
  {
    return;
  }

  // Allocate the physical memory.

  NumberOfPagesInitial = NumberOfPages;
  bResult = AllocateUserPhysicalPages( GetCurrentProcess(),
                                       &NumberOfPages,
                                       aPFNs );
```

```
if( bResult != TRUE )
{
  _tprintf(_T("Cannot allocate physical pages (%u)\n"), GetLastError() );
  return;
}

if( NumberOfPagesInitial != NumberOfPages )
{
  _tprintf(_T("Allocated only %p pages.\n"), NumberOfPages );
  return;
}

// Reserve the virtual memory.

lpMemReserved = VirtualAlloc( NULL,
                              MEMORY_REQUESTED,
                              MEM_RESERVE | MEM_PHYSICAL,
                              PAGE_READWRITE );

if( lpMemReserved == NULL )
{
  _tprintf(_T("Cannot reserve memory.\n"));
  return;
}

// Map the physical memory into the window.

bResult = MapUserPhysicalPages( lpMemReserved,
                                NumberOfPages,
                                aPFNs );

if( bResult != TRUE )
{
  _tprintf(_T("MapUserPhysicalPages failed (%u)\n"), GetLastError() );
  return;
}

// unmap

bResult = MapUserPhysicalPages( lpMemReserved,
                                NumberOfPages,
                                NULL );

if( bResult != TRUE )
{
  _tprintf(_T("MapUserPhysicalPages failed (%u)\n"), GetLastError() );
  return;
}

// Free the physical pages.

bResult = FreeUserPhysicalPages( GetCurrentProcess(),
                                 &NumberOfPages,
                                 aPFNs );

if( bResult != TRUE )
{
  _tprintf(_T("Cannot free physical pages, error %u.\n"), GetLastError());
  return;
}

// Free virtual memory.

bResult = VirtualFree( lpMemReserved,
                       0,
                       MEM_RELEASE );

// Release the aPFNs array.
```

```
  bResult = HeapFree(GetProcessHeap(), 0, aPFNs);

  if( bResult != TRUE )
  {
      _tprintf(_T("Call to HeapFree has failed (%u)\n"), GetLastError() );
  }

}

/*****************************************************************
   LoggedSetLockPagesPrivilege: a function to obtain or
   release the privilege of locking physical pages.

   Inputs:

       HANDLE hProcess: Handle for the process for which the
       privilege is needed

       BOOL bEnable: Enable (TRUE) or disable?

   Return value: TRUE indicates success, FALSE failure.

*****************************************************************/
BOOL
LoggedSetLockPagesPrivilege ( HANDLE hProcess,
                              BOOL bEnable)
{
  struct {
    DWORD Count;
    LUID_AND_ATTRIBUTES Privilege [1];
  } Info;

  HANDLE Token;
  BOOL Result;

  // Open the token.

  Result = OpenProcessToken ( hProcess,
                              TOKEN_ADJUST_PRIVILEGES,
                              & Token);

  if( Result != TRUE )
  {
    _tprintf( _T("Cannot open process token.\n") );
    return FALSE;
  }

  // Enable or disable?

  Info.Count = 1;
  if( bEnable )
  {
    Info.Privilege[0].Attributes = SE_PRIVILEGE_ENABLED;
  }
  else
  {
    Info.Privilege[0].Attributes = 0;
  }

  // Get the LUID.

  Result = LookupPrivilegeValue ( NULL,
                                  SE_LOCK_MEMORY_NAME,
                                  &(Info.Privilege[0].Luid));

  if( Result != TRUE )
  {
    _tprintf( _T("Cannot get privilege for %s.\n"), SE_LOCK_MEMORY_NAME );
    return FALSE;
```

```
  }

  // Adjust the privilege.

  Result = AdjustTokenPrivileges ( Token, FALSE,
                                   (PTOKEN_PRIVILEGES) &Info,
                                   0, NULL, NULL);

  // Check the result.

  if( Result != TRUE )
  {
    _tprintf (_T("Cannot adjust token privileges (%u)\n"), GetLastError() );
    return FALSE;
  }
  else
  {
    if( GetLastError() != ERROR_SUCCESS )
    {
      _tprintf (_T("Cannot enable the SE_LOCK_MEMORY_NAME privilege; "));
      _tprintf (_T("please check the local policy.\n"));
      return FALSE;
    }
  }

  CloseHandle( Token );

  return TRUE;
}
```

Allocating Memory from a NUMA Node

3/5/2021 • 3 minutes to read • Edit Online

The following sample code demonstrates the use of the NUMA functions **GetNumaHighestNodeNumber**, **GetNumaProcessorNode**, and **VirtualAllocExNuma**. It also demonstrates the use of the **QueryWorkingSetEx** function to retrieve the NUMA node on which pages are allocated.

```
#define _WIN32_WINNT 0x0600

#include <windows.h>
#include <stdlib.h>
#include <stdio.h>
#include <psapi.h>
#include <tchar.h>

SIZE_T AllocationSize;
DWORD  PageSize;

void DumpNumaNodeInfo (PVOID Buffer, SIZE_T Size);

void __cdecl wmain (int argc, const wchar_t* argv[])
{
    ULONG HighestNodeNumber;
    ULONG NumberOfProcessors;
    PVOID* Buffers = NULL;

    if (argc > 1)
    {
        (void)swscanf_s (argv[1], L"%Ix", &AllocationSize);
    }

    if (AllocationSize == 0)
    {
        AllocationSize = 16*1024*1024;
    }

    //
    // Get the number of processors and system page size.
    //

    SYSTEM_INFO SystemInfo;
    GetSystemInfo (&SystemInfo);
    NumberOfProcessors = SystemInfo.dwNumberOfProcessors;
    PageSize = SystemInfo.dwPageSize;

    //
    // Get the highest node number.
    //

    if (!GetNumaHighestNodeNumber (&HighestNodeNumber))
    {
        _tprintf (_T("GetNumaHighestNodeNumber failed: %d\n"), GetLastError());
        goto Exit;
    }

    if (HighestNodeNumber == 0)
    {
        _putts (_T("Not a NUMA system - exiting"));
        goto Exit;
    }

    //
```

```
    //
    // Allocate array of pointers to memory blocks.
    //

    Buffers = (PVOID*) malloc (sizeof(PVOID)*NumberOfProcessors);

    if (Buffers == NULL)
    {
        _putts (_T("Allocating array of buffers failed"));
        goto Exit;
    }

    ZeroMemory (Buffers, sizeof(PVOID)*NumberOfProcessors);

    //
    // For each processor, get its associated NUMA node and allocate some memory from it.
    //

    for (UCHAR i = 0; i < NumberOfProcessors; i++)
    {
        UCHAR NodeNumber;

        if (!GetNumaProcessorNode (i, &NodeNumber))
        {
            _tprintf (_T("GetNumaProcessorNode failed: %d\n"), GetLastError());
            goto Exit;
        }

        _tprintf (_T("CPU %u: node %u\n"), (ULONG)i, NodeNumber);

        PCHAR Buffer = (PCHAR)VirtualAllocExNuma(
            GetCurrentProcess(),
            NULL,
            AllocationSize,
            MEM_RESERVE | MEM_COMMIT,
            PAGE_READWRITE,
            NodeNumber
        );

        if (Buffer == NULL)
        {
            _tprintf (_T("VirtualAllocExNuma failed: %d, node %u\n"), GetLastError(), NodeNumber);
            goto Exit;
        }

        PCHAR BufferEnd = Buffer + AllocationSize - 1;
        SIZE_T NumPages = ((SIZE_T)BufferEnd)/PageSize - ((SIZE_T)Buffer)/PageSize + 1;

        _putts (_T("Allocated virtual memory:"));
        _tprintf (_T("%p - %p (%6Iu pages), preferred node %u\n"), Buffer, BufferEnd, NumPages, NodeNumber);

        Buffers[i] = Buffer;

        //
        // At this point, virtual pages are allocated but no valid physical
        // pages are associated with them yet.
        //
        // The FillMemory call below will touch every page in the buffer, faulting
        // them into our working set. When this happens physical pages will be allocated
        // from the preferred node we specified in VirtualAllocExNuma, or any node
        // if the preferred one is out of pages.
        //

        FillMemory (Buffer, AllocationSize, 'x');

        //
        // Check the actual node number for the physical pages that are still valid
        // (if system is low on physical memory, some pages could have been trimmed already).
        //
```

```
        DumpNumaNodeInfo (Buffer, AllocationSize);

        _putts(_T(""));
    }

Exit:
    if (Buffers != NULL)
    {
        for (UINT i = 0; i < NumberOfProcessors; i++)
        {
            if (Buffers[i] != NULL)
            {
                VirtualFree (Buffers[i], 0, MEM_RELEASE);
            }
        }

        free (Buffers);
    }
}

void DumpRegion (PVOID StartPtr, PVOID EndPtr, BOOL Valid, DWORD Node)
{
    DWORD_PTR StartPage = ((DWORD_PTR)StartPtr)/PageSize;
    DWORD_PTR EndPage   = ((DWORD_PTR)EndPtr)/PageSize;
    DWORD_PTR NumPages  = (EndPage - StartPage) + 1;

    if (!Valid)
    {
        _tprintf (_T("%p - %p (%6Iu pages): no valid pages\n"), StartPtr, EndPtr, NumPages);
    }
    else
    {
        _tprintf (_T("%p - %p (%6Iu pages): node %u\n"), StartPtr, EndPtr, NumPages, Node);
    }
}

void DumpNumaNodeInfo (PVOID Buffer, SIZE_T Size)
{
    DWORD_PTR StartPage = ((DWORD_PTR)Buffer)/PageSize;
    DWORD_PTR EndPage   = ((DWORD_PTR)Buffer + Size - 1)/PageSize;
    DWORD_PTR NumPages  = (EndPage - StartPage) + 1;

    PCHAR StartPtr = (PCHAR)(PageSize*StartPage);

    _putts (_T("Checking NUMA node:"));

    PPSAPI_WORKING_SET_EX_INFORMATION WsInfo = (PPSAPI_WORKING_SET_EX_INFORMATION)
        malloc (NumPages*sizeof(PSAPI_WORKING_SET_EX_INFORMATION));

    if (WsInfo == NULL)
    {
        _putts (_T("Could not allocate array of PSAPI_WORKING_SET_EX_INFORMATION structures"));
        return;
    }

    for (DWORD_PTR i = 0; i < NumPages; i++)
    {
        WsInfo[i].VirtualAddress = StartPtr + i*PageSize;
    }

    BOOL bResult = QueryWorkingSetEx(
        GetCurrentProcess(),
        WsInfo,
        (DWORD)NumPages*sizeof(PSAPI_WORKING_SET_EX_INFORMATION)
    );

    if (!bResult)
    {
```

```
        _tprintf (_T("QueryWorkingSetEx failed: %d\n"), GetLastError());
        free (WsInfo);
        return;
    }

    PCHAR RegionStart   = NULL;
    BOOL  RegionIsValid = false;
    DWORD RegionNode    = 0;

    for (DWORD_PTR i = 0; i < NumPages; i++)
    {
        PCHAR Address = (PCHAR)WsInfo[i].VirtualAddress;
        BOOL  IsValid = WsInfo[i].VirtualAttributes.Valid;
        DWORD Node    = WsInfo[i].VirtualAttributes.Node;

        if (i == 0)
        {
            RegionStart   = Address;
            RegionIsValid = IsValid;
            RegionNode    = Node;
        }

        if (IsValid != RegionIsValid || Node != RegionNode)
        {
            DumpRegion (RegionStart, Address - 1, RegionIsValid, RegionNode);

            RegionStart   = Address;
            RegionIsValid = IsValid;
            RegionNode    = Node;
        }

        if (i == (NumPages - 1))
        {
            DumpRegion (RegionStart, Address + PageSize - 1, IsValid, Node);
        }
    }

    free (WsInfo);
}
```

Related topics

NUMA Support

Memory Management Reference

3/5/2021 • 2 minutes to read • Edit Online

The following elements are used with memory management:

- Memory Management Enumerations
- Memory Management Functions
- Memory Management Registry Keys
- Memory Management Structures
- Memory Protection Constants
- Memory Management Tracing Events

Memory Management Enumerations

3/5/2021 • 2 minutes to read • Edit Online

The following enumerations are used with memory management:

In this section

TOPIC	DESCRIPTION
HEAP_INFORMATION_CLASS	Specifies the class of heap information to be set or retrieved.
MEM_EXTENDED_PARAMETER_TYPE	Defines values for extended parameters used for file mapping into an address space.

Memory Management Functions

11/2/2020 • 8 minutes to read • Edit Online

General memory functions

FUNCTION	DESCRIPTION
AddSecureMemoryCacheCallback	Registers a callback function to be called when a secured memory range is freed or its protections are changed.
CopyMemory	Copies a block of memory from one location to another.
CreateMemoryResourceNotification	Creates a memory resource notification object.
FillMemory	Fills a block of memory with a specified value.
GetLargePageMinimum	Retrieves the minimum size of a large page.
GetPhysicallyInstalledSystemMemory	Retrieves the amount of RAM that is physically installed on the computer.
GetSystemFileCacheSize	Retrieves the current size limits for the working set of the system cache.
GetWriteWatch	Retrieves the addresses of the pages that have been written to in a region of virtual memory.
GlobalMemoryStatusEx	Obtains information about the system's current usage of both physical and virtual memory.
MoveMemory	Moves a block of memory from one location to another.
QueryMemoryResourceNotification	Retrieves the state of the specified memory resource object.

FUNCTION	DESCRIPTION
RemoveSecureMemoryCacheCallback	Unregisters a callback function that was previously registered with the AddSecureMemoryCacheCallback function.
ResetWriteWatch	Resets the write-tracking state for a region of virtual memory.
SecureMemoryCacheCallback	An application-defined function that is called when a secured memory range is freed or its protections are changed.
SecureZeroMemory	Fills a block of memory with zeros.
SetSystemFileCacheSize	Limits the size of the working set for the file system cache.
ZeroMemory	Fills a block of memory with zeros.

Data execution prevention functions

These functions are used with Data Execution Prevention (DEP).

FUNCTION	DESCRIPTION
GetProcessDEPPolicy	Retrieves DEP settings for a process.
GetSystemDEPPolicy	Retrieves DEP settings for the system.
SetProcessDEPPolicy	Changes DEP settings for a process.

File mapping functions

These functions are used in file mapping.

FUNCTION	DESCRIPTION
CreateFileMappingA	Creates or opens a named or unnamed file-mapping object for a specified file.
CreateFileMappingW	Creates or opens a named or unnamed file-mapping object for a specified file.
CreateFileMapping2	Creates or opens a named or unnamed file mapping object for a specified file. You can specify specify a preferred NUMA node for the physical memory as an extended parameter; see the *ExtendedParameters* parameter.
CreateFileMappingFromApp	Creates or opens a named or unnamed file-mapping object for a specified file from a Windows Store app.
CreateFileMappingNuma	Creates or opens a named or unnamed file-mapping object for a specified file, and specifies the NUMA node for the physical memory.

FUNCTION	DESCRIPTION
FlushViewOfFile	Writes to the disk a byte range within a mapped view of a file.
GetMappedFileName	Checks whether the specified address is within a memory-mapped file in the address space of the specified process. If so, the function returns the name of the memory-mapped file.
MapViewOfFile	Maps a view of a file mapping into the address space of a calling process.
MapViewOfFile2	Maps a view of a file or a pagefile-backed section into the address space of the specified process.
MapViewOfFile3	Maps a view of a file or a pagefile-backed section into the address space of the specified process.
MapViewOfFile3FromApp	Maps a view of a file mapping into the address space of a calling process from a Windows Store app.
MapViewOfFileEx	Maps a view of a file mapping into the address space of a calling process. A caller can optionally specify a suggested memory address for the view.
MapViewOfFileExNuma	Maps a view of a file mapping into the address space of a calling process, and specifies the NUMA node for the physical memory.
MapViewOfFileFromApp	Maps a view of a file mapping into the address space of a calling process from a Windows Store app.
MapViewOfFileNuma2	Maps a view of a file or a pagefile-backed section into the address space of the specified process.
OpenFileMapping	Opens a named file-mapping object.
OpenFileMappingFromApp	Opens a named file-mapping object.
UnmapViewOfFile	Unmaps a mapped view of a file from the calling process's address space.
UnmapViewOfFile2	Unmaps a previously mapped view of a file or a pagefile-backed section.
UnmapViewOfFileEx	Unmaps a previously mapped view of a file or a pagefile-backed section.

AWE functions

These are the AWE functions.

FUNCTION	DESCRIPTION
AllocateUserPhysicalPages	Allocates physical memory pages to be mapped and unmapped within any AWE region of the process.
AllocateUserPhysicalPagesNuma	Allocates physical memory pages to be mapped and unmapped within any AWE region of the process, and specifies the NUMA node for the physical memory.
FreeUserPhysicalPages	Frees physical memory pages previously allocated with AllocateUserPhysicalPages.
MapUserPhysicalPages	Maps previously allocated physical memory pages at the specified address within an AWE region.
MapUserPhysicalPagesScatter	Maps previously allocated physical memory pages at the specified address within an AWE region.

Heap functions

These are the heap functions.

FUNCTION	DESCRIPTION
GetProcessHeap	Obtains a handle to the heap of the calling process.
GetProcessHeaps	Obtains handles to all of the heaps that are valid for the calling process.
HeapAlloc	Allocates a block of memory from a heap.
HeapCompact	Coalesces adjacent free blocks of memory on a heap.
HeapCreate	Creates a heap object.
HeapDestroy	Destroys the specified heap object.
HeapFree	Frees a memory block allocated from a heap.
HeapLock	Attempts to acquire the lock associated with a specified heap.
HeapQueryInformation	Retrieves information about the specified heap.
HeapReAlloc	Reallocates a block of memory from a heap.
HeapSetInformation	Sets heap information for the specified heap.
HeapSize	Retrieves the size of a memory block allocated from a heap.
HeapUnlock	Releases ownership of the lock associated with a specified heap.
HeapValidate	Attempts to validate a specified heap.

FUNCTION	DESCRIPTION
HeapWalk	Enumerates the memory blocks in a specified heap.

Virtual memory functions

These are the virtual memory functions.

FUNCTION	DESCRIPTION
DiscardVirtualMemory	Discards the memory contents of a range of memory pages, without decommitting the memory. The contents of discarded memory is undefined and must be rewritten by the application.
OfferVirtualMemory	Indicates that the data contained in a range of memory pages is no longer needed by the application and can be discarded by the system if necessary.
PrefetchVirtualMemory	Prefetches virtual address ranges into physical memory.
QueryVirtualMemoryInformation	Returns information about a page or a set of pages within the virtual address space of the specified process.
ReclaimVirtualMemory	Reclaims a range of memory pages that were offered to the system with OfferVirtualMemory.
SetProcessValidCallTargets	Provides CFG with a list of valid indirect call targets and specifies whether they should be marked valid or not.
VirtualAlloc	Reserves or commits a region of pages in the virtual address space of the calling process.
VirtualAlloc2	Reserves, commits, or changes the state of a region of memory within the virtual address space of a specified process. The function initializes the memory it allocates to zero.
VirtualAlloc2FromApp	Reserves, commits, or changes the state of a region of pages in the virtual address space of the calling process. Memory allocated by this function is automatically initialized to zero.
VirtualAllocEx	Reserves or commits a region of pages in the virtual address space of the specified process.
VirtualAllocExNuma	Reserves or commits a region of memory within the virtual address space of the specified process, and specifies the NUMA node for the physical memory.
VirtualAllocFromApp	Reserves, commits, or changes the state of a region of pages in the virtual address space of the calling process. Memory allocated by this function is automatically initialized to zero.

FUNCTION	DESCRIPTION
VirtualFree	Releases or decommits a region of pages within the virtual address space of the calling process.
VirtualFreeEx	Releases or decommits a region of memory within the virtual address space of a specified process.
VirtualLock	Locks the specified region of the process's virtual address space into physical memory.
VirtualProtect	Changes the access protection on a region of committed pages in the virtual address space of the calling process.
VirtualProtectEx	Changes the access protection on a region of committed pages in the virtual address space of the calling process.
VirtualProtectFromApp	Changes the protection on a region of committed pages in the virtual address space of the calling process.
VirtualQuery	Provides information about a range of pages in the virtual address space of the calling process.
VirtualQueryEx	Provides information about a range of pages in the virtual address space of the calling process.
VirtualUnlock	Unlocks a specified range of pages in the virtual address space of a process.

Global and local functions

Also see global and local functions. These functions are provided for compatibility with 16-bit Windows and are used with Dynamic Data Exchange (DDE), the clipboard functions, and OLE data objects. Unless documentation specifically states that a global or local function should be used, new applications should use the corresponding heap function with the handle returned by GetProcessHeap. For equivalent functionality to the global or local function, set the heap function's *dwFlags* parameter to 0.

FUNCTION	DESCRIPTION	CORRESPONDING HEAP FUNCTION
GlobalAlloc, LocalAlloc	Allocates the specified number of bytes from the heap.	HeapAlloc
GlobalDiscard, LocalDiscard	Discards the specified global memory block.	Not applicable.
GlobalFlags, LocalFlags	Returns information about the specified global memory object.	Not applicable. Use HeapValidate to validate the heap.
GlobalFree, LocalFree	Frees the specified global memory object.	HeapFree

FUNCTION	DESCRIPTION	CORRESPONDING HEAP FUNCTION
GlobalHandle, LocalHandle	Retrieves the handle associated with the specified pointer to a global memory block. This function should be used only with OLE and clipboard functions that require it.	Not applicable.
GlobalLock, LocalLock	Locks a global memory object and returns a pointer to the first byte of the object's memory block.	Not applicable.
GlobalReAlloc, LocalReAlloc	Changes the size or attributes of a specified global memory object.	HeapReAlloc
GlobalSize, LocalSize	Retrieves the current size of the specified global memory object.	HeapSize
GlobalUnlock, LocalUnlock	Decrements the lock count associated with a memory object. This function should be used only with OLE and clipboard functions that require it.	Not applicable.

Bad memory functions

FUNCTION	DESCRIPTION
BadMemoryCallbackRoutine	An application-defined function registered with the RegisterBadMemoryNotification function that is called when one or more bad memory pages are detected.
GetMemoryErrorHandlingCapabilities	Gets the memory error handling capabilities of the system.
RegisterBadMemoryNotification	Registers a bad memory notification that is called when one or more bad memory pages are detected.
UnregisterBadMemoryNotification	Closes the specified bad memory notification handle.

Enclave functions

FUNCTION	DESCRIPTION
CreateEnclave	Creates a new uninitialized enclave. An enclave is an isolated region of code and data within the address space for an application. Only code that runs within the enclave can access data within the same enclave.
InitializeEnclave	Initializes an enclave that you created and loaded with data.
IsEnclaveTypeSupported	Retrieves whether the specified type of enclave is supported.
LoadEnclaveData	Loads data into an uninitialized enclave that you created by calling CreateEnclave.

ATL thunk functions

FUNCTION	DESCRIPTION
AtlThunk_AllocateData	Allocates space in memory for an ATL thunk.
AtlThunk_DataToCode	Returns an executable function corresponding to the AtlThunkData_t parameter.
AtlThunk_FreeData	Frees memory associated with an ATL thunk.
AtlThunk_InitData	Initializes an ATL thunk.

Obsolete Functions

These functions are provided only for compatibility with 16-bit versions of Windows:

- IsBadCodePtr
- IsBadReadPtr
- IsBadStringPtr
- IsBadWritePtr

The function below can return incorrect information, and should not be used. Instead, use the GlobalMemoryStatusEx function.

- GlobalMemoryStatus

Memory Management Registry Keys

3/5/2021 • 2 minutes to read • Edit Online

System virtual address (VA) space on 32-bit systems can become exhausted due to fragmentation. Several registry keys can be used to configure memory limits on 32-bit systems that experience this issue. System VA space on 64-bit systems is not subject to exhaustion by fragmentation; therefore, these keys have no effect on 64-bit systems.

For 32-bit systems, these memory management registry keys must be explicitly created under the following registry key:

HKEY_LOCAL_MACHINE\SYSTEM\Current Control Set\Control\Session Manager\Memory Management

Windows Server 2008 and Windows Vista: These registry keys are available on 32-bit systems starting with Windows Server 2008 and Windows Vista with Service Pack 1 (SP1).

For default memory and address space limits on both 32-bit and 64-bit systems, see Memory Limits for Windows Releases.

The following table describes the memory management registry keys that can be used to configure memory limits on 32-bit systems. All of these keys have a REG_DWORD type and possible values that range from 0 through 2,048 MB. The default is 0, which means no limit is enforced. Values are automatically rounded up to the next system VA allocation boundary, which is 2 MB on 32-bit systems that have Physical Address Extension (PAE) enabled and 4 MB on 32-bit systems that do not have PAE enabled.

KEY	DESCRIPTION
NonPagedPoolLimit	Specifies the maximum amount of system VA space that can be used by the nonpaged pool. Under certain conditions, this limit may be exceeded by a small amount.
PagedPoolLimit	Specifies the maximum amount of system VA space that can be used by the paged pool.
SessionSpaceLimit	Specifies the maximum amount of system VA space that can be used by session space allocations.
SystemCacheLimit	Specifies the maximum amount of system VA space that can be used by the system cache. Under certain conditions, this limit may be exceeded by a small amount.
SystemPtesLimit	Specifies the maximum amount of system VA space that can be used by I/O mappings and other resources that consume system page table entries (PTEs).

Determining whether system VA space is being exhausted requires the use of a kernel debugger. For more information, see Debugging Tools for Windows.

Memory Management Structures

3/5/2021 • 2 minutes to read • Edit Online

The following structures are used with memory management.

In this section

TOPIC	DESCRIPTION
CFG_CALL_TARGET_INFO	Represents information about call targets for Control Flow Guard (CFG).
HEAP_OPTIMIZE_RESOURCES_INFORMATION	Specifies flags for a HeapOptimizeResources operation initiated with **HeapSetInformation**.
MEM_ADDRESS_REQUIREMENTS	Specifies a lowest and highest base address and alignment as part of an extended parameter to a function that manages virtual memory..
MEM_EXTENDED_PARAMETER	Represents an extended parameter for a function that manages virtual memory.
MEMORY_BASIC_INFORMATION	Contains information about a range of pages in the virtual address space of a process.
MEMORYSTATUS	Contains information about the current state of both physical and virtual memory.
MEMORYSTATUSEX	Contains information about the current state of both physical and virtual memory, including extended memory.
PROCESS_HEAP_ENTRY	Contains information about a heap element.
WIN32_MEMORY_RANGE_ENTRY	Specifies a range of memory.
WIN32_MEMORY_REGION_INFORMATION	Contains information about a memory region.
AtlThunkData_t	An opaque data structure that represents an ATL thunk.

CFG_CALL_TARGET_INFO structure

3/5/2021 • 2 minutes to read • Edit Online

Represents information about call targets for Control Flow Guard (CFG).

Syntax

```
typedef struct _CFG_CALL_TARGET_INFO {
  ULONG_PTR Offset;
  ULONG_PTR Flags;
} CFG_CALL_TARGET_INFO, *PCFG_CALL_TARGET_INFO;
```

Members

Offset

Offset relative to a provided (start) virtual address. This offset should be 16 byte aligned.

Flags

Flags describing the operation to be performed on the address. If **CFG_CALL_TARGET_VALID** is set, then the address will be marked valid for CFG. Otherwise, it will be marked an invalid call target.

Requirements

REQUIREMENT	VALUE
Minimum supported client	Windows 10 [desktop apps only]
Minimum supported server	Windows Server 2016 [desktop apps only]
Header	Ntmmapi.h

Memory Protection Constants

3/22/2021 • 5 minutes to read • Edit Online

The following are the memory-protection options; you must specify one of the following values when allocating or protecting a page in memory. Protection attributes cannot be assigned to a portion of a page; they can only be assigned to a whole page.

Example

```
STDMETHODIMP CExtBuffer::FInit
    (
    ULONG cItemMax,     //@parm IN | Maximum number of items ever
    ULONG cbItem,       //@parm IN | Size of each item, in bytes
    ULONG cbPage        //@parm IN | Size of system page size (from SysInfo)
    )
{
    BYTE  *pb;

    m_cbReserved = ((cbItem *cItemMax) / cbPage + 1) *cbPage;
    m_rgItem = (BYTE *) VirtualAlloc( NULL, m_cbReserved, MEM_RESERVE, PAGE_READWRITE );

    if (m_rgItem == NULL)
        return ResultFromScode( E_OUTOFMEMORY );

    m_cbItem  = cbItem;
    m_dbAlloc = (cbItem / cbPage + 1) *cbPage;
    pb = (BYTE *) VirtualAlloc( m_rgItem, m_dbAlloc, MEM_COMMIT, PAGE_READWRITE );
    if (pb == NULL)
        {
        VirtualFree((VOID *) m_rgItem, 0, MEM_RELEASE );
        m_rgItem = NULL;
        return ResultFromScode( E_OUTOFMEMORY );
        }

    m_cbAlloc = m_dbAlloc;
    return ResultFromScode( S_OK );
}
```

Example it from Windows classic samples on GitHub.

Constants

CONSTANT/VALUE	DESCRIPTION
PAGE_EXECUTE 0x10	Enables execute access to the committed region of pages. An attempt to write to the committed region results in an access violation. This flag is not supported by the **CreateFileMapping** function.

CONSTANT/VALUE	DESCRIPTION
PAGE_EXECUTE_READ 0x20	Enables execute or read-only access to the committed region of pages. An attempt to write to the committed region results in an access violation. **Windows Server 2003 and Windows XP:** This attribute is not supported by the **CreateFileMapping** function until Windows XP with SP2 and Windows Server 2003 with SP1.
PAGE_EXECUTE_READWRITE 0x40	Enables execute, read-only, or read/write access to the committed region of pages. **Windows Server 2003 and Windows XP:** This attribute is not supported by the **CreateFileMapping** function until Windows XP with SP2 and Windows Server 2003 with SP1.
PAGE_EXECUTE_WRITECOPY 0x80	Enables execute, read-only, or copy-on-write access to a mapped view of a file mapping object. An attempt to write to a committed copy-on-write page results in a private copy of the page being made for the process. The private page is marked as **PAGE_EXECUTE_READWRITE**, and the change is written to the new page. This flag is not supported by the **VirtualAlloc** or **VirtualAllocEx** functions. **Windows Vista, Windows Server 2003 and Windows XP:** This attribute is not supported by the **CreateFileMapping** function until Windows Vista with SP1 and Windows Server 2008.
PAGE_NOACCESS 0x01	Disables all access to the committed region of pages. An attempt to read from, write to, or execute the committed region results in an access violation. This flag is not supported by the **CreateFileMapping** function.
PAGE_READONLY 0x02	Enables read-only access to the committed region of pages. An attempt to write to the committed region results in an access violation. If Data Execution Prevention is enabled, an attempt to execute code in the committed region results in an access violation.
PAGE_READWRITE 0x04	Enables read-only or read/write access to the committed region of pages. If Data Execution Prevention is enabled, attempting to execute code in the committed region results in an access violation.
PAGE_WRITECOPY 0x08	Enables read-only or copy-on-write access to a mapped view of a file mapping object. An attempt to write to a committed copy-on-write page results in a private copy of the page being made for the process. The private page is marked as **PAGE_READWRITE**, and the change is written to the new page. If Data Execution Prevention is enabled, attempting to execute code in the committed region results in an access violation. This flag is not supported by the **VirtualAlloc** or **VirtualAllocEx** functions.

CONSTANT/VALUE	DESCRIPTION
PAGE_TARGETS_INVALID 0x40000000	Sets all locations in the pages as invalid targets for CFG. Used along with any execute page protection like **PAGE_EXECUTE**, **PAGE_EXECUTE_READ**, **PAGE_EXECUTE_READWRITE** and **PAGE_EXECUTE_WRITECOPY**. Any indirect call to locations in those pages will fail CFG checks and the process will be terminated. The default behavior for executable pages allocated is to be marked valid call targets for CFG. This flag is not supported by the **VirtualProtect** or **CreateFileMapping** functions.
PAGE_TARGETS_NO_UPDATE 0x40000000	Pages in the region will not have their CFG information updated while the protection changes for **VirtualProtect**. For example, if the pages in the region was allocated using **PAGE_TARGETS_INVALID**, then the invalid information will be maintained while the page protection changes. This flag is only valid when the protection changes to an executable type like **PAGE_EXECUTE**, **PAGE_EXECUTE_READ**, **PAGE_EXECUTE_READWRITE** and **PAGE_EXECUTE_WRITECOPY**. The default behavior for **VirtualProtect** protection change to executable is to mark all locations as valid call targets for CFG.

The following are modifiers that can be used in addition to the options provided in the previous table, except as noted.

CONSTANT/VALUE	DESCRIPTION
PAGE_GUARD 0x100	Pages in the region become guard pages. Any attempt to access a guard page causes the system to raise a **STATUS_GUARD_PAGE_VIOLATION** exception and turn off the guard page status. Guard pages thus act as a one-time access alarm. For more information, see Creating Guard Pages. When an access attempt leads the system to turn off guard page status, the underlying page protection takes over. If a guard page exception occurs during a system service, the service typically returns a failure status indicator. This value cannot be used with **PAGE_NOACCESS**. This flag is not supported by the **CreateFileMapping** function.
PAGE_NOCACHE 0x200	Sets all pages to be non-cachable. Applications should not use this attribute except when explicitly required for a device. Using the interlocked functions with memory that is mapped with **SEC_NOCACHE** can result in an **EXCEPTION_ILLEGAL_INSTRUCTION** exception. The **PAGE_NOCACHE** flag cannot be used with the **PAGE_GUARD**, **PAGE_NOACCESS**, or **PAGE_WRITECOMBINE** flags. The **PAGE_NOCACHE** flag can be used only when allocating private memory with the **VirtualAlloc**, **VirtualAllocEx**, or **VirtualAllocExNuma** functions. To enable non-cached memory access for shared memory, specify the **SEC_NOCACHE** flag when calling the **CreateFileMapping** function.

CONSTANT/VALUE	DESCRIPTION
PAGE_WRITECOMBINE 0x400	Sets all pages to be write-combined. Applications should not use this attribute except when explicitly required for a device. Using the interlocked functions with memory that is mapped as write-combined can result in an **EXCEPTION_ILLEGAL_INSTRUCTION** exception. The **PAGE_WRITECOMBINE** flag cannot be specified with the **PAGE_NOACCESS**, **PAGE_GUARD**, and **PAGE_NOCACHE** flags. The **PAGE_WRITECOMBINE** flag can be used only when allocating private memory with the **VirtualAlloc**, **VirtualAllocEx**, or **VirtualAllocExNuma** functions. To enable write-combined memory access for shared memory, specify the **SEC_WRITECOMBINE** flag when calling the **CreateFileMapping** function. **Windows Server 2003 and Windows XP:** This flag is not supported until Windows Server 2003 with SP1.

The following constants can only be used with the **LoadEnclaveData** function when you specify an enclave that has the Intel Software Guard Extensions (SGX) architecture.

CONSTANT	DESCRIPTION
PAGE_ENCLAVE_THREAD_CONTROL	The page contains a thread control structure (TCS).
PAGE_ENCLAVE_UNVALIDATED	The page contents that you supply are excluded from measurement with the EEXTEND instruction of the Intel SGX programming model.

Requirements

REQUIREMENT	VALUE
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	WinNT.h (include Windows.h)

See also

CreateFileMapping

Memory Protection

VirtualAlloc

VirtualAllocEx

LoadEnclaveData

Memory Management Tracing Events

3/5/2021 • 2 minutes to read • Edit Online

This section describes detailed information on specific Memory management tracing event details.

Memory management tracing is a troubleshooting feature that can be enabled in retail binaries to trace certain memory management events with minimal overhead. This feature allows for better diagnostic capabilities for developers and product support. Memory management event tracing supports tracing heap allocation, reallocation, and free operations.

Memory management event tracing uses Event Tracing for Windows (ETW), a general-purpose, high-speed tracing facility provided by the operating system. ETW provides a tracing mechanism for events raised by both user-mode applications and kernel-mode device drivers. ETW can enable and disable logging dynamically, making it easy to perform detailed tracing in production environments without requiring reboots or application restarts. Memory management event tracing using ETW is supported on Windows 7 , Windows Server 2008 R2, and later. For general information on ETW, see Improve Debugging And Performance Tuning With ETW.

The following list provides detailed information for each memory management tracing event. For additional information on any event, click the event name.

EVENT NAME	DESCRIPTION
ETW_HEAP_EVENT_ALLOC	Memory management tracing event for a heap allocation operation.
ETW_HEAP_EVENT_FREE	Memory management tracing event for a heap free operation.
ETW_HEAP_EVENT_REALLOC	Memory management tracing event for a heap re-allocation operation.

Related topics

Improve Debugging And Performance Tuning With ETW

ETW_HEAP_EVENT_ALLOC event

3/5/2021 • 2 minutes to read • Edit Online

The **ETW_HEAP_EVENT_ALLOC** event is a memory management tracing event for a heap allocation operation.

```
typedef struct ETW_HEAP_EVENT_ALLOC
```

Parameters

HeapHandle

The handle of the heap where the memory was allocated. This is the heap handle an app passed to the **AllocateHeap** function when the memory was allocated.

Size

The size in bytes allocated from the heap.

Address

The address of the memory that was allocated.

Source

The source of the memory that the allocator used for the heap allocation.

The following table lists the possible values for the *Source* parameter as defined in the *ntetw.h* header file:

VALUE	MEANING
MEMORY_FROM_LOOKASIDE 1	Memory from the lookaside list.
MEMORY_FROM_LOWFRAG 2	Memory from the low-fragmentation heap.
MEMORY_FROM_MAINPATH 3	Memory from main code path.
MEMORY_FROM_SLOWPATH 4	Memory from slow path.
MEMORY_FROM_INVALID 5	Memory that was not valid.

VALUE	MEANING
MEMORY_FROM_SEGMENT_HEAP 6	This value is reserved for future use and will never be returned.

Remarks

The **ETW_HEAP_EVENT_ALLOC** event is logged on all heap allocations.

Requirements

REQUIREMENT	VALUE
Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Header	Ntwmi.h

See also

Memory Management Tracing Events

ETW_HEAP_EVENT_FREE event

3/5/2021 • 2 minutes to read • Edit Online

The **ETW_HEAP_EVENT_FREE** event is a memory management tracing event for a heap free operation.

```
typedef struct ETW_HEAP_EVENT_FREE
```

Parameters

HeapHandle

The handle of the heap where the memory was allocated. This is the heap handle an app passed to the **AllocateHeap** function when the memory was allocated.

Address

The address of the memory that was freed.

Source

The source of the memory that the allocator used for the heap allocation.

The following table lists the possible values for the *Source* parameter as defined in the *ntetw.h* header file:

VALUE	MEANING
MEMORY_FROM_LOOKASIDE 1	Memory from the lookaside list.
MEMORY_FROM_LOWFRAG 2	Memory from the low-fragmentation heap.
MEMORY_FROM_MAINPATH 3	Memory from main code path.
MEMORY_FROM_SLOWPATH 4	Memory from slow c.
MEMORY_FROM_INVALID 5	Memory that was not valid.
MEMORY_FROM_SEGMENT_HEAP 6	This value is reserved for future use and will never be returned.

Remarks

The **ETW_HEAP_EVENT_FREE** event is logged on all heap free operations.

Requirements

REQUIREMENT	VALUE
Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Header	Ntwmi.h

See also

Memory Management Tracing Events

ETW_HEAP_EVENT_REALLOC event

3/5/2021 • 2 minutes to read • Edit Online

The **ETW_HEAP_EVENT_REALLOC** event is a memory management tracing event for a heap reallocation operation.

```
typedef struct ETW_HEAP_EVENT_REALLOC
```

Parameters

HeapHandle

The handle of the heap where the memory was allocated. This is the heap handle an app passed to the **AllocateHeap** function when the memory was allocated.

NewAddress

The new address of the memory that was allocated.

OldAddress

The old address of the memory that was previously allocated.

NewSize

The new size in bytes allocated from the heap.

OldSize

The old size in bytes previously allocated from the heap.

Source

The source of the memory that the allocator used for the heap allocation.

The following table lists the possible values for the *Source* parameter as defined in the *ntetw.h* header file:

VALUE	MEANING
MEMORY_FROM_LOOKASIDE 1	Memory from the lookaside list.
MEMORY_FROM_LOWFRAG 2	Memory from the low-fragmentation heap.
MEMORY_FROM_MAINPATH 3	Memory from main code path.

VALUE	MEANING
MEMORY_FROM_SLOWPATH 4	Memory from slow c.
MEMORY_FROM_INVALID 5	Memory that was not valid.
MEMORY_FROM_SEGMENT_HEAP 6	This value is reserved for future use and will never be returned.

This event has no parameters.

Remarks

The **ETW_HEAP_EVENT_REALLOC** event is logged on all heap reallocations.

Requirements

REQUIREMENT	VALUE
Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Header	Ntwmi.h

See also

Memory Management Tracing Events

Trusted Execution

3/5/2021 • 2 minutes to read • Edit Online

Enclaves are used to create trusted execution environments. An enclave is an isolated region of code and data within the address space for an application. Only code that runs within the enclave can access data within the same enclave.

In This Section

- **Trusted Execution Reference**

Trusted Execution Reference

3/5/2021 • 2 minutes to read • Edit Online

The following elements are used when working with enclaves that are used to create trusted execution environments:

- Trusted Execution Enumerations
- Trusted Execution Functions
- Trusted Execution Structures

Trusted Execution Enumerations

3/5/2021 • 2 minutes to read • Edit Online

The following enumerations are used when working with enclaves that are used to create trusted execution environments:

In this section

TOPIC	DESCRIPTION
ENCLAVE_SEALING_IDENTITY_POLICY	Defines values that specify how another enclave must be related to the enclave that calls EnclaveSealData for the enclave to unseal the data.

Trusted Execution Functions

3/5/2021 • 2 minutes to read • Edit Online

The following functions are used when working with enclaves that are used to create trusted execution environments:

In this section

TOPIC	DESCRIPTION
CallEnclave	Calls a function within an enclave.
CreateEnclave	Creates a new uninitialized enclave. An enclave is an isolated region of code and data within the address space for an application. Only code that runs within the enclave can access data within the same enclave.
DeleteEnclave	Deletes the specified enclave.
EnclaveGetAttestationReport	Gets an enclave attestation report that describes the current enclave and is signed by the authority that is responsible for the type of the enclave.
EnclaveGetEnclaveInformation	Gets information about the currently executing enclave.
EnclaveSealData	Generates an encrypted binary large object (blob) from unencypted data.
EnclaveUnsealData	Decrypts an encrypted binary large object (blob).
EnclaveVerifyAttestationReport	Verifies an attestation report that was generated on the current system.
InitializeEnclave	Initializes an enclave that you created and loaded with data.
IsEnclaveTypeSupported	Retrieves whether the specified type of enclave is supported.
LoadEnclaveData	Loads data into an uninitialized enclave that you created by calling CreateEnclave.
LoadEnclaveImage	Loads an image and all of its imports into an enclave.
TerminateEnclave	Ends the execution of the threads that are running within an enclave.

Trusted Execution Structures

3/5/2021 • 2 minutes to read • Edit Online

The following structures are used when working with enclaves that are used to create trusted execution environments:

In this section

TOPIC	DESCRIPTION
ENCLAVE_CREATE_INFO_SGX	Contains architecture-specific information to use to create an enclave when the enclave type is **ENCLAVE_TYPE_SGX**, which specifies an enclave for the Intel Software Guard Extensions (SGX) architecture extension.
ENCLAVE_CREATE_INFO_VBS	Contains architecture-specific information to use to create an enclave when the enclave type is **ENCLAVE_TYPE_VBS**, which specifies a virtualization-based security (VBS) enclave.
ENCLAVE_IDENTITY	Describes the identity of the primary module of an enclave.
ENCLAVE_INFORMATION	Contains information about the currently executing enclave.
ENCLAVE_INIT_INFO_SGX	Contains architecture-specific information to use to initialize an enclave when the enclave type is **ENCLAVE_TYPE_SGX**, which specifies an enclave for the Intel Software Guard Extensions (SGX) architecture extension.
ENCLAVE_INIT_INFO_VBS	Contains architecture-specific information to use to initialize an enclave when the enclave type is **ENCLAVE_TYPE_VBS**, which specifies a virtualization-based security (VBS) enclave.
IMAGE_ENCLAVE_CONFIG32	Defines the format of the enclave configuration for systems running 32-bit Windows.
IMAGE_ENCLAVE_CONFIG64	Defines the format of the enclave configuration for systems running 64-bit Windows.
IMAGE_ENCLAVE_IMPORT	Defines a entry in the array of images that an enclave can import.
VBS_ENCLAVE_REPORT	Describes the format of the signed statement contained in a report generated by calling the EnclaveGetAttestationReport function.
VBS_ENCLAVE_REPORT_MODULE	Describes a module loaded for the enclave.
VBS_ENCLAVE_REPORT_PKG_HEADER	Describes the contents of a report generated by calling the EnclaveGetAttestationReport function.

TOPIC	DESCRIPTION
VBS_ENCLAVE_REPORT_VARDATA_HEADER	Describes the format of a variable data block contained in a report that the **EnclaveGetAttestationReport** function generates.