

memoryapi.h header

Article 01/24/2023

This header is used by System Services. For more information, see:

- [System Services](#)

memoryapi.h contains the following programming interfaces:

Functions

 Expand table

<p>AllocateUserPhysicalPages</p> <p>Allocates physical memory pages to be mapped and unmapped within any Address Windowing Extensions (AWE) region of a specified process.</p>
<p>AllocateUserPhysicalPages2</p> <p>Allocates physical memory pages to be mapped and unmapped within any Address Windowing Extensions (AWE) region of a specified process, with extended parameters.</p>
<p>AllocateUserPhysicalPagesNuma</p> <p>Allocates physical memory pages to be mapped and unmapped within any Address Windowing Extensions (AWE) region of a specified process and specifies the NUMA node for the physical memory.</p>
<p>CreateFileMapping2</p> <p>Creates or opens a named or unnamed file mapping object for a specified file. You can specify a preferred NUMA node for the physical memory as an extended parameter; see the <i>ExtendedParameters</i> parameter.</p>
<p>CreateFileMappingFromApp</p> <p>Creates or opens a named or unnamed file mapping object for a specified file from a Windows Store app.</p>
<p>CreateFileMappingNumaW</p> <p>Creates or opens a named or unnamed file mapping object for a specified file and specifies the NUMA node for the physical memory. (CreateFileMappingNumaW)</p>

CreateFileMappingW

Creates or opens a named or unnamed file mapping object for a specified file.
(CreateFileMappingW)

CreateMemoryResourceNotification

Creates a memory resource notification object.

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.

FlushViewOfFile

Writes to the disk a byte range within a mapped view of a file.

FreeUserPhysicalPages

Frees physical memory pages that are allocated previously by using AllocateUserPhysicalPages or AllocateUserPhysicalPagesNuma.

GetLargePageMinimum

Retrieves the minimum size of a large page.

GetMemoryErrorHandlingCapabilities

Gets the memory error handling capabilities of the system.

GetProcessWorkingSetSize

Retrieves the minimum and maximum working set sizes of the specified process.
(GetProcessWorkingSetSize)

GetProcessWorkingSetSizeEx

Retrieves the minimum and maximum working set sizes of the specified process.
(GetProcessWorkingSetSizeEx)

GetSystemFileCacheSize

Retrieves the current size limits for the working set of the system cache.

GetWriteWatch

Retrieves the addresses of the pages that are written to in a region of virtual memory.

[MapUserPhysicalPages](#)

Maps previously allocated physical memory pages at a specified address in an Address Windowing Extensions (AWE) region. (MapUserPhysicalPages)

[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. (MapViewOfFile2)

[MapViewOfFile3](#)

Maps a view of a file or a pagefile-backed section into the address space of the specified process. (MapViewOfFile3)

[MapViewOfFile3FromApp](#)

Maps a view of a file mapping into the address space of a calling Windows Store app. (MapViewOfFile3FromApp)

[MapViewOfFileEx](#)

Maps a view of a file mapping into the address space of a calling process. A caller can optionally specify a suggested base memory address for the view.

[MapViewOfFileFromApp](#)

Maps a view of a file mapping into the address space of a calling Windows Store app. (MapViewOfFileFromApp)

[MapViewOfFileNuma2](#)

Maps a view of a file or a pagefile-backed section into the address space of the specified process. (MapViewOfFileNuma2)

[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.

[OpenFileMappingFromApp](#)

Opens a named file mapping object. (OpenFileMappingFromApp)

[OpenFileMappingW](#)

Opens a named file mapping object. (OpenFileMappingW)

[PrefetchVirtualMemory](#)

Provides an efficient mechanism to bring into memory potentially discontinuous virtual address ranges in a process address space.

[QueryMemoryResourceNotification](#)

Retrieves the state of the specified memory resource object.

[QueryVirtualMemoryInformation](#)

The QueryVirtualMemoryInformation function returns information about a page or a set of pages within the virtual address space of the specified process.

[ReadProcessMemory](#)

Reads data from an area of memory in a specified process. The entire area to be read must be accessible or the operation fails.

[ReclaimVirtualMemory](#)

Reclaims a range of memory pages that were offered to the system with OfferVirtualMemory.

[RegisterBadMemoryNotification](#)

Registers a bad memory notification that is called when one or more bad memory pages are detected.

[ResetWriteWatch](#)

Resets the write-tracking state for a region of virtual memory. Subsequent calls to the GetWriteWatch function only report pages that are written to since the reset operation.

[SetProcessValidCallTargets](#)

Provides Control Flow Guard (CFG) with a list of valid indirect call targets and specifies whether they should be marked valid or not.

[SetProcessWorkingSetSize](#)

Sets the minimum and maximum working set sizes for the specified process.
(SetProcessWorkingSetSize)


[SetProcessWorkingSetSizeEx](#)

<p>Sets the minimum and maximum working set sizes for the specified process. (SetProcessWorkingSetSizeEx)</p>
<p>SetSystemFileCacheSize</p> <p>Limits the size of the working set for the file system cache.</p>
<p>UnmapViewOfFile</p> <p>Unmaps a mapped view of a file from the calling process's address space.</p>
<p>UnmapViewOfFile2</p> <p>Unmaps a previously mapped view of a file or a pagefile-backed section.</p>
<p>UnmapViewOfFileEx</p> <p>This is an extended version of UnmapViewOfFile that takes an additional flags parameter.</p>
<p>UnregisterBadMemoryNotification</p> <p>Closes the specified bad memory notification handle.</p>
<p>VirtualAlloc</p> <p>Reserves, commits, or changes the state of a region of pages in the virtual address space of the calling process. (VirtualAlloc)</p>
<p>VirtualAlloc2</p> <p>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. (VirtualAlloc2)</p>
<p>VirtualAlloc2FromApp</p> <p>Reserves, commits, or changes the state of a region of pages in the virtual address space of the calling process. (VirtualAlloc2FromApp)</p>
<p>VirtualAllocEx</p> <p>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. (VirtualAllocEx)</p>
<p>VirtualAllocExNuma</p> <p>Reserves, commits, or changes the state of a region of memory within the virtual address space of the specified process, and specifies the NUMA node for the physical memory.</p>
<p>VirtualAllocFromApp</p>

Reserves, commits, or changes the state of a region of pages in the virtual address space of the calling process. (VirtualAllocFromApp)
VirtualFree Releases, decommits, or releases and decommits a region of pages within the virtual address space of the calling process.
VirtualFreeEx Releases, decommits, or releases and 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, ensuring that subsequent access to the region will not incur a page fault.
VirtualProtect Changes the protection on a region of committed pages in the virtual address space of the calling process. (VirtualProtect)
VirtualProtectEx Changes the protection on a region of committed pages in the virtual address space of a specified process.
VirtualProtectFromApp Changes the protection on a region of committed pages in the virtual address space of the calling process. (VirtualProtectFromApp)
VirtualQuery Retrieves information about a range of pages in the virtual address space of the calling process.
VirtualQueryEx Retrieves information about a range of pages within the virtual address space of a specified process.
VirtualUnlock Unlocks a specified range of pages in the virtual address space of a process, enabling the system to swap the pages out to the paging file if necessary.
WriteProcessMemory

Writes data to an area of memory in a specified process. The entire area to be written to must be accessible or the operation fails.

Structures

 Expand table

WIN32_MEMORY_RANGE_ENTRY
Specifies a range of memory.
WIN32_MEMORY_REGION_INFORMATION
Contains information about a memory region.

Feedback

Was this page helpful?

 Yes

 No

AllocateUserPhysicalPages function (memoryapi.h)

Article 12/13/2023

Allocates physical memory pages to be mapped and unmapped within any [Address Windowing Extensions](#) (AWE) region of a specified process.

64-bit Windows on Itanium-based systems: Due to the difference in page sizes, **AllocateUserPhysicalPages** is not supported for 32-bit applications.

Syntax

C++

```
BOOL AllocateUserPhysicalPages(  
    [in]      HANDLE      hProcess,  
    [in, out] PULONG_PTR  NumberOfPages,  
    [out]     PULONG_PTR  PageArray  
);
```

Parameters

[in] hProcess

A handle to a process.

The function allocates memory that can later be mapped within the virtual address space of this process. The handle must have the **PROCESS_VM_OPERATION** access right. For more information, see [Process Security and Access Rights](#).

[in, out] NumberOfPages

The size of the physical memory to allocate, in pages.

To determine the page size of the computer, use the [GetSystemInfo](#) function. On output, this parameter receives the number of pages that are actually allocated, which might be less than the number requested.

[out] PageArray

A pointer to an array to store the page frame numbers of the allocated memory.

The size of the array that is allocated should be at least the *NumberOfPages* times the size of the **ULONG_PTR** data type.

Do not attempt to modify this buffer. It contains operating system data, and corruption could be catastrophic. The information in the buffer is not useful to an application.

Return value

If the function succeeds, the return value is **TRUE**.

Fewer pages than requested can be allocated. The caller must check the value of the *NumberOfPages* parameter on return to see how many pages are allocated. All allocated page frame numbers are sequentially placed in the memory pointed to by the *UserPfnArray* parameter.

If the function fails, the return value is **FALSE**, and no frames are allocated. To get extended error information, call [GetLastError](#).

Remarks

The **AllocateUserPhysicalPages** function is used to allocate physical memory that can later be mapped within the virtual address space of the process. The **SeLockMemoryPrivilege** privilege must be enabled in the caller's token or the function will fail with **ERROR_PRIVILEGE_NOT_HELD**. For more information, see [Privilege Constants](#).

Memory allocated by this function must be physically present in the system. After the memory is allocated, it is locked down and unavailable to the rest of the virtual memory management system.

Physical pages cannot be simultaneously mapped at more than one virtual address.

Physical pages can reside at any physical address. You should make no assumptions about the contiguity of the physical pages.


To compile an application that uses this function, define the `_WIN32_WINNT` macro as `0x0500` or later. For more information, see [Using the Windows Headers](#).

[AllocateUserPhysicalPages2](#), added to the SDK in a later release, is the same as **AllocateUserPhysicalPages** but it adds the *ExtendedParameters* and *ExtendedParameterCount* parameters.

Examples

For an example, see [AWE Example](#).

Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	memoryapi.h (include Windows.h, Memoryapi.h)
Library	oncore.lib
DLL	Kernel32.dll

See also

[Address Windowing Extensions](#)

[AllocateUserPhysicalPagesNuma](#)

[FreeUserPhysicalPages](#)

[MapUserPhysicalPages](#)

[MapUserPhysicalPagesScatter](#)

[Memory Management Functions](#)

[AllocateUserPhysicalPages2](#)

AllocateUserPhysicalPages2 function (memoryapi.h)

Article 12/13/2023

Allocates physical memory pages to be mapped and unmapped within any [Address Windowing Extensions \(AWE\)](#) region of a specified process, with extended parameters.

Syntax

C++

```
BOOL AllocateUserPhysicalPages2(  
    [in]      HANDLE          ObjectHandle,  
    [in,out] PULONG_PTR      NumberOfPages,  
    [out]     PULONG_PTR      PageArray,  
    [in,out] PMEM_EXTENDED_PARAMETER ExtendedParameters,  
    [in]      ULONG           ExtendedParameterCount  
);
```

Parameters

[in] ObjectHandle

A handle to a process.

The function allocates memory that can later be mapped within the virtual address space of this process. The handle must have the **PROCESS_VM_OPERATION** access right. For more information, see [Process Security and Access Rights](#).

[in,out] NumberOfPages

The size of the physical memory to allocate, in pages.

To determine the page size of the computer, use the [GetSystemInfo](#) function. On output, this parameter receives the number of pages that are actually allocated, which might be less than the number requested.

[out] PageArray

A pointer to an array to store the page frame numbers of the allocated memory.

The size of the array that is allocated should be at least the *NumberOfPages* times the size of the **ULONG_PTR** data type.

Do not attempt to modify this buffer. It contains operating system data, and corruption could be catastrophic. The information in the buffer is not useful to an application.

[in,out] *ExtendedParameters*

Pointer to an array of [MEM_EXTENDED_PARAMETER](#) structures.

[in] *ExtendedParameterCount*

The number of **MEM_EXTENDED_PARAMETER** in the *ExtendedParameters* array.

Return value

If the function succeeds, the return value is **TRUE**.

Fewer pages than requested can be allocated. The caller must check the value of the *NumberOfPages* parameter on return to see how many pages are allocated. All allocated page frame numbers are sequentially placed in the memory pointed to by the *UserPfnArray* parameter.

If the function fails, the return value is **FALSE**, and no frames are allocated. To get extended error information, call [GetLastError](#).

Remarks

AllocateUserPhysicalPages2 is the same as [AllocateUserPhysicalPages](#) but it adds the *ExtendedParameters* and *ExtendedParameterCount* parameters.

The **AllocateUserPhysicalPages2** function is used to allocate physical memory that can later be mapped within the virtual address space of the process. The **SeLockMemoryPrivilege** privilege must be enabled in the caller's token or the function will fail with **ERROR_PRIVILEGE_NOT_HELD**. For more information, see [Privilege Constants](#).

Memory allocated by this function must be physically present in the system. After the memory is allocated, it is locked down and unavailable to the rest of the virtual memory management system.

Physical pages cannot be simultaneously mapped at more than one virtual address.

Physical pages can reside at any physical address. You should make no assumptions about the contiguity of the physical pages.

Requirements

 Expand table

Minimum supported client	Windows 11, Build 20348
Minimum supported server	Windows Server, Build 20348
Header	memoryapi.h
Library	onecore.lib
DLL	kernelbase.dll

See also

[AllocateUserPhysicalPages](#)

[Address Windowing Extensions](#)

[AllocateUserPhysicalPagesNuma](#)

[FreeUserPhysicalPages](#)


[MapUserPhysicalPages](#)


[MapUserPhysicalPagesScatter](#)

[Memory Management Functions](#)

Feedback

Was this page helpful?

 Yes

 No

AllocateUserPhysicalPagesNuma function (memoryapi.h)

Article 05/14/2022

Allocates physical memory pages to be mapped and unmapped within any [Address Windowing Extensions](#) (AWE) region of a specified process and specifies the NUMA node for the physical memory.

Syntax

C++

```
BOOL AllocateUserPhysicalPagesNuma(  
    [in]      HANDLE      hProcess,  
    [in, out] PULONG_PTR  NumberOfPages,  
    [out]     PULONG_PTR  PageArray,  
    [in]      DWORD       nndPreferred  
);
```

Parameters

[in] hProcess

A handle to a process.

The function allocates memory that can later be mapped within the virtual address space of this process. The handle must have the **PROCESS_VM_OPERATION** access right. For more information, see [Process Security and Access Rights](#).

[in, out] NumberOfPages

The size of the physical memory to allocate, in pages.

To determine the page size of the computer, use the [GetSystemInfo](#) function. On output, this parameter receives the number of pages that are actually allocated, which might be less than the number requested.

[out] PageArray

A pointer to an array to store the page frame numbers of the allocated memory.

The size of the array that is allocated should be at least the *NumberOfPages* times the size of the **ULONG_PTR** data type.

Caution Do not attempt to modify this buffer. It contains operating system data, and corruption could be catastrophic. The information in the buffer is not useful to an application.

[in] `nndPreferred`

The NUMA node where the physical memory should reside.

Return value

If the function succeeds, the return value is **TRUE**.

Fewer pages than requested can be allocated. The caller must check the value of the *NumberOfPages* parameter on return to see how many pages are allocated. All allocated page frame numbers are sequentially placed in the memory pointed to by the *PageArray* parameter.

If the function fails, the return value is **FALSE** and no frames are allocated. To get extended error information, call the [GetLastError](#) function.

Remarks

The **AllocateUserPhysicalPagesNuma** function is used to allocate physical memory within a NUMA node that can later be mapped within the virtual address space of the process. The **SeLockMemoryPrivilege** privilege must be enabled in the caller's token or the function will fail with **ERROR_PRIVILEGE_NOT_HELD**. For more information, see [Privilege Constants](#).

Memory allocated by this function must be physically present in the system. After the memory is allocated, it is locked down and unavailable to the rest of the virtual memory management system.

Physical pages cannot be simultaneously mapped at more than one virtual address.

Physical pages can reside at any physical address. You should make no assumptions about the contiguity of the physical pages.

To compile an application that uses this function, define **_WIN32_WINNT** as 0x0600 or later.

Requirements

Requirement	Value
Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	memoryapi.h (include Windows.h, Memoryapi.h)
Library	oncore.lib
DLL	Kernel32.dll

See also

[Address Windowing Extensions](#)

[AllocateUserPhysicalPages](#)

[FreeUserPhysicalPages](#)

[Memory Management Functions](#)

[NUMA Support](#)

CreateFileMapping2 function (memoryapi.h)

Article 05/14/2022

Creates or opens a named or unnamed file mapping object for a specified file. You can specify a preferred NUMA node for the physical memory as an extended parameter; see the *ExtendedParameters* parameter.

Syntax

C++

```
HANDLE CreateFileMapping2(  
    HANDLE                File,  
    SECURITY_ATTRIBUTES   *SecurityAttributes,  
    ULONG                 DesiredAccess,  
    ULONG                 PageProtection,  
    ULONG                 AllocationAttributes,  
    ULONG64               MaximumSize,  
    PCWSTR                Name,  
    MEM_EXTENDED_PARAMETER *ExtendedParameters,  
    ULONG                 ParameterCount  
);
```

Parameters

File

Type: `_In_ HANDLE`

A handle to the file from which to create a file mapping object.

The file must be opened with access rights that are compatible with the protection flags that the *flProtect* parameter specifies. It is not required, but it is recommended that files you intend to map be opened for exclusive access. For more information, see [File security and access rights](#).

If *hFile* is `INVALID_HANDLE_VALUE`, the calling process must also specify a size for the file mapping object in the *dwMaximumSizeHigh* and *dwMaximumSizeLow* parameters. In this scenario, **CreateFileMapping** creates a file mapping object of a specified size that is backed by the system paging file instead of by a file in the file system.

SecurityAttributes

Type: `_In_opt_ SECURITY_ATTRIBUTES*`

A pointer to a `SECURITY_ATTRIBUTES` structure that determines whether a returned handle can be inherited by child processes. The `lpSecurityDescriptor` member of the `SECURITY_ATTRIBUTES` structure specifies a security descriptor for a new file mapping object.

If `lpAttributes` is `NULL`, the handle cannot be inherited and the file mapping object gets a default security descriptor. The access control lists (ACL) in the default security descriptor for a file mapping object come from the primary or impersonation token of the creator. For more information, see [File Mapping Security and Access Rights](#).

DesiredAccess

Type: `_In_ ULONG`

The desired access mask for the returned file mapping handle. For a list of access rights, see [File-mapping security and access rights](#).

PageProtection

Type: `_In_ ULONG`

Specifies the page protection of the file mapping object. All mapped views of the object must be compatible with this protection.

This parameter can be one of the following values.

Value	Meaning
PAGE_EXECUTE_READ 0x20	Allows views to be mapped for read-only, copy-on-write, or execute access. The file handle specified by the <i>hFile</i> parameter must be created with the GENERIC_READ and GENERIC_EXECUTE access rights. Windows Server 2003 and Windows XP: This value is not available until Windows XP with SP2 and Windows Server 2003 with SP1.
PAGE_EXECUTE_READWRITE 0x40	Allows views to be mapped for read-only, copy-on-write, read/write, or execute access. The file handle that the <i>hFile</i> parameter specifies must be created with the GENERIC_READ , GENERIC_WRITE , and GENERIC_EXECUTE access rights.

	<p>Windows Server 2003 and Windows XP: This value is not available until Windows XP with SP2 and Windows Server 2003 with SP1.</p>
PAGE_EXECUTE_WRITECOPY 0x80	<p>Allows views to be mapped for read-only, copy-on-write, or execute access. This value is equivalent to PAGE_EXECUTE_READ.</p> <p>The file handle that the <i>hFile</i> parameter specifies must be created with the GENERIC_READ and GENERIC_EXECUTE access rights.</p> <p>Windows Vista: This value is not available until Windows Vista with SP1.</p> <p>Windows Server 2003 and Windows XP: This value is not supported.</p>
PAGE_READONLY 0x02	<p>Allows views to be mapped for read-only or copy-on-write access. An attempt to write to a specific region results in an access violation.</p> <p>The file handle that the <i>hFile</i> parameter specifies must be created with the GENERIC_READ access right.</p>
PAGE_READWRITE 0x04	<p>Allows views to be mapped for read-only, copy-on-write, or read/write access.</p> <p>The file handle that the <i>hFile</i> parameter specifies must be created with the GENERIC_READ and GENERIC_WRITE access rights.</p>
PAGE_WRITECOPY 0x08	<p>Allows views to be mapped for read-only or copy-on-write access. This value is equivalent to PAGE_READONLY.</p> <p>The file handle that the <i>hFile</i> parameter specifies must be created with the GENERIC_READ access right.</p>

AllocationAttributes

Type: `_In_ ULONG`

You can specify one or more of the following attributes for the file mapping object. Also see the *PageProtection* parameter.

Value	Meaning
SEC_COMMIT 0x80000000	<p>If the file mapping object is backed by the operating system paging file (the <i>hfile</i> parameter is INVALID_HANDLE_VALUE), specifies that when a view of the file is mapped into a process address space, the entire range of pages is committed rather than reserved.</p>

	<p>The system must have enough committable pages to hold the entire mapping. Otherwise, CreateFileMapping fails.</p> <p>This attribute has no effect for file mapping objects that are backed by executable image files or data files (the <i>hfile</i> parameter is a handle to a file).</p> <p>SEC_COMMIT cannot be combined with SEC_RESERVE.</p> <p>If no attribute is specified, SEC_COMMIT is assumed.</p>
SEC_IMAGE 0x1000000	<p>Specifies that the file that the <i>hFile</i> parameter specifies is an executable image file.</p> <p>The SEC_IMAGE attribute must be combined with a page protection value such as PAGE_READONLY. However, this page protection value has no effect on views of the executable image file. Page protection for views of an executable image file is determined by the executable file itself.</p> <p>No other attributes are valid with SEC_IMAGE.</p>
SEC_IMAGE_NO_EXECUTE 0x11000000	<p>Specifies that the file that the <i>hFile</i> parameter specifies is an executable image file that will not be executed and the loaded image file will have no forced integrity checks run. Additionally, mapping a view of a file mapping object created with the SEC_IMAGE_NO_EXECUTE attribute will not invoke driver callbacks registered using the PsSetLoadImageNotifyRoutine kernel API.</p> <p>The SEC_IMAGE_NO_EXECUTE attribute must be combined with the PAGE_READONLY page protection value. No other attributes are valid with SEC_IMAGE_NO_EXECUTE.</p> <p>Windows Server 2008 R2, Windows 7, Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP: This value is not supported before Windows Server 2012 and Windows 8.</p>
SEC_LARGE_PAGES 0x80000000	<p>Enables large pages to be used for file mapping objects that are backed by the operating system paging file (the <i>hfile</i> parameter is INVALID_HANDLE_VALUE). This attribute is not supported for file mapping objects that are backed by executable image files or data files (the <i>hFile</i> parameter is a handle to an executable image or data file).</p> <p>The maximum size of the file mapping object must be a multiple of the minimum size of a large page returned by</p>

	<p>the GetLargePageMinimum function. If it is not, CreateFileMapping fails. When mapping a view of a file mapping object created with SEC_LARGE_PAGES, the base address and view size must also be multiples of the minimum large page size.</p> <p>SEC_LARGE_PAGES requires the SeLockMemoryPrivilege privilege to be enabled in the caller's token.</p> <p>If SEC_LARGE_PAGES is specified, SEC_COMMIT must also be specified.</p> <p>Windows Server 2003: This value is not supported until Windows Server 2003 with SP1.</p> <p>Windows XP: This value is not supported.</p>
SEC_NOCACHE 0x10000000	<p>Sets all pages to be non-cacheable.</p> <p>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.</p> <p>SEC_NOCACHE requires either the SEC_RESERVE or SEC_COMMIT attribute to be set.</p>
SEC_RESERVE 0x40000000	<p>If the file mapping object is backed by the operating system paging file (the <i>hfile</i> parameter is INVALID_HANDLE_VALUE), specifies that when a view of the file is mapped into a process address space, the entire range of pages is reserved for later use by the process rather than committed.</p> <p>Reserved pages can be committed in subsequent calls to the VirtualAlloc function. After the pages are committed, they cannot be freed or decommitted with the VirtualFree function.</p> <p>This attribute has no effect for file mapping objects that are backed by executable image files or data files (the <i>hfile</i> parameter is a handle to a file).</p> <p>SEC_RESERVE cannot be combined with SEC_COMMIT.</p>
SEC_WRITECOMBINE 0x40000000	<p>Sets all pages to be write-combined.</p> <p>Applications should not use this attribute except when explicitly required for a device. Using the interlocked functions with memory that is mapped with SEC_WRITECOMBINE can result in an EXCEPTION_ILLEGAL_INSTRUCTION exception.</p>

SEC_WRITECOMBINE requires either the **SEC_RESERVE** or **SEC_COMMIT** attribute to be set.

Windows Server 2003 and Windows XP: This flag is not supported until Windows Vista.

MaximumSize

Type: `_In_` [ULONG64](#)

The maximum size of the file mapping object.

If this parameter is 0 (zero), then the maximum size of the file mapping object is equal to the current size of the file that *hFile* identifies.

An attempt to map a file with a length of 0 (zero) fails with an error code of **ERROR_FILE_INVALID**. You should test for files with a length of 0 (zero), and reject those files.

Name

Type: `_In_opt_` [PCWSTR](#)

The name of the file mapping object.

If this parameter matches the name of an existing mapping object, then the function requests access to the object with the protection that *flProtect* specifies.

If this parameter is **NULL**, then the file mapping object is created without a name.

If *lpName* matches the name of an existing event, semaphore, mutex, waitable timer, or job object, the function fails, and the [GetLastError](#) function returns **ERROR_INVALID_HANDLE**. This occurs because these objects share the same namespace.

The name can have a "Global" or "Local" prefix to explicitly create the object in the global or session namespace. The remainder of the name can contain any character except the backslash character (\). Creating a file mapping object in the global namespace from a session other than session zero requires the [SeCreateGlobalPrivilege](#) privilege. For more information, see [Kernel Object Namespaces](#).

Fast user switching is implemented by using Terminal Services sessions. The first user to log on uses session 0 (zero), the next user to log on uses session 1 (one), and so on. Kernel object names must follow the guidelines that are outlined for Terminal Services so that applications can support multiple users.

ExtendedParameters

Type: `_Inout_updates_opt_(ParameterCount) MEM_EXTENDED_PARAMETER*`

An optional pointer to one or more extended parameters of type [MEM_EXTENDED_PARAMETER](#). Each of those extended parameter values can itself have a *Type* field of either **MemExtendedParameterAddressRequirements** or **MemExtendedParameterNumaNode**. If no **MemExtendedParameterNumaNode** extended parameter is provided, then the behavior is the same as for the [VirtualAlloc/MapViewOfFile](#) functions (that is, the preferred NUMA node for the physical pages is determined based on the ideal processor of the thread that first accesses the memory).

ParameterCount

In ULONG ParameterCount

The number of extended parameters pointed to by *ExtendedParameters*.

Return value

If the function succeeds, the return value is a handle to the newly created file mapping object.

If the object exists before the function call, the function returns a handle to the existing object (with its current size, not the specified size), and [GetLastError](#) returns **ERROR_ALREADY_EXISTS**.

If the function fails, the return value is **NULL**. To get extended error information, call [GetLastError](#).

Remarks

See the **Remarks** for [CreateFileMapping](#).

Examples

For an example, see [Creating named shared memory](#), or [Creating a file mapping using large pages](#).

Requirements

Minimum supported client	Windows 10 Build 20348
Minimum supported server	Windows 10 Build 20348
Target Platform	Windows
Header	memoryapi.h (include Windows.h, Memoryapi.h)
Library	oncore.lib
DLL	Kernel32.dll

See also

[CloseHandle](#)

[CreateFileMappingNuma](#)

[Creating a file mapping object](#)

[DuplicateHandle](#)

[MapViewOfFile](#)

[MapViewOfFileEx](#)

[Memory management functions](#)

[OpenFileMapping](#)

[ReadFile](#)

[SECURITY_ATTRIBUTES](#)

[UnmapViewOfFile](#)

[VirtualAlloc](#)

[WriteFile](#)

Feedback

Was this page helpful?

Get help at [Microsoft Q&A](#)

CreateFileMappingFromApp function (memoryapi.h)

Article 05/14/2022

Creates or opens a named or unnamed file mapping object for a specified file from a Windows Store app.

Syntax

C++

```
HANDLE CreateFileMappingFromApp(  
    [in] HANDLE hFile,  
    [in, optional] PSECURITY_ATTRIBUTES SecurityAttributes,  
    [in] ULONG PageProtection,  
    [in] ULONG64 MaximumSize,  
    [in, optional] PCWSTR Name  
);
```

Parameters

[in] `hFile`

A handle to the file from which to create a file mapping object.

The file must be opened with access rights that are compatible with the protection flags that the *flProtect* parameter specifies. It is not required, but it is recommended that files you intend to map be opened for exclusive access. For more information, see [File Security and Access Rights](#).

If *hFile* is `INVALID_HANDLE_VALUE`, the calling process must also specify a size for the file mapping object in the *dwMaximumSizeHigh* and *dwMaximumSizeLow* parameters. In this scenario, **CreateFileMappingFromApp** creates a file mapping object of a specified size that is backed by the system paging file instead of by a file in the file system.

[in, optional] `SecurityAttributes`

A pointer to a [SECURITY_ATTRIBUTES](#) structure that determines whether a returned handle can be inherited by child processes. The `lpSecurityDescriptor` member of the `SECURITY_ATTRIBUTES` structure specifies a security descriptor for a new file mapping object.

If *SecurityAttributes* is **NULL**, the handle cannot be inherited and the file mapping object gets a default security descriptor. The access control lists (ACL) in the default security descriptor for a file mapping object come from the primary or impersonation token of the creator. For more information, see [File Mapping Security and Access Rights](#).

[in] PageProtection

Specifies the page protection of the file mapping object. All mapped views of the object must be compatible with this protection.

This parameter can be one of the following values.

 Expand table

Value	Meaning
PAGE_READONLY 0x02	Allows views to be mapped for read-only or copy-on-write access. An attempt to write to a specific region results in an access violation. The file handle that the <i>hFile</i> parameter specifies must be created with the GENERIC_READ access right.
PAGE_READWRITE 0x04	Allows views to be mapped for read-only, copy-on-write, or read/write access. The file handle that the <i>hFile</i> parameter specifies must be created with the GENERIC_READ and GENERIC_WRITE access rights.
PAGE_WRITECOPY 0x08	Allows views to be mapped for read-only or copy-on-write access. This value is equivalent to PAGE_READONLY . The file handle that the <i>hFile</i> parameter specifies must be created with the GENERIC_READ access right.

An application can specify one or more of the following attributes for the file mapping object by combining them with one of the preceding page protection values.

 Expand table

Value	Meaning
SEC_COMMIT 0x80000000	If the file mapping object is backed by the operating system paging file (the <i>hfile</i> parameter is INVALID_HANDLE_VALUE), specifies that when a view of the file is mapped into a process address space, the entire range of pages is committed rather than reserved. The system must have enough committable

pages to hold the entire mapping. Otherwise, **CreateFileMappingFromApp** fails.

This attribute has no effect for file mapping objects that are backed by executable image files or data files (the *hfile* parameter is a handle to a file).

SEC_COMMIT cannot be combined with **SEC_RESERVE**.

If no attribute is specified, **SEC_COMMIT** is assumed.

SEC_IMAGE_NO_EXECUTE
0x11000000

Specifies that the file that the *hFile* parameter specifies is an executable image file that will not be executed and the loaded image file will have no forced integrity checks run. Additionally, mapping a view of a file mapping object created with the **SEC_IMAGE_NO_EXECUTE** attribute will not invoke driver callbacks registered using the [PsSetLoadImageNotifyRoutine](#) kernel API.

The **SEC_IMAGE_NO_EXECUTE** attribute must be combined with the **PAGE_READONLY** page protection value. No other attributes are valid with **SEC_IMAGE_NO_EXECUTE**.

SEC_LARGE_PAGES
0x80000000

Enables large pages to be used for file mapping objects that are backed by the operating system paging file (the *hfile* parameter is **INVALID_HANDLE_VALUE**). This attribute is not supported for file mapping objects that are backed by executable image files or data files (the *hFile* parameter is a handle to an executable image or data file).

The maximum size of the file mapping object must be a multiple of the minimum size of a large page returned by the [GetLargePageMinimum](#) function. If it is not, **CreateFileMappingFromApp** fails. When mapping a view of a file mapping object created with **SEC_LARGE_PAGES**, the base address and view size must also be multiples of the minimum large page size.

SEC_LARGE_PAGES requires the [SeLockMemoryPrivilege](#) privilege to be enabled in the caller's token.

If **SEC_LARGE_PAGES** is specified, **SEC_COMMIT** must also be specified.

SEC_NOCACHE
0x10000000

Sets all pages to be non-cacheable. 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.

SEC_NOCACHE requires either the **SEC_RESERVE** or **SEC_COMMIT** attribute to be set.

SEC_RESERVE

0x4000000

If the file mapping object is backed by the operating system paging file (the *hfile* parameter is **INVALID_HANDLE_VALUE**), specifies that when a view of the file is mapped into a process address space, the entire range of pages is reserved for later use by the process rather than committed.

Reserved pages can be committed in subsequent calls to the [VirtualAlloc](#) function. After the pages are committed, they cannot be freed or decommitted with the [VirtualFree](#) function.

This attribute has no effect for file mapping objects that are backed by executable image files or data files (the *hfile* parameter is a handle to a file).

SEC_RESERVE cannot be combined with **SEC_COMMIT**.

SEC_WRITECOMBINE

0x40000000

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 with **SEC_WRITECOMBINE** can result in an **EXCEPTION_ILLEGAL_INSTRUCTION** exception.

SEC_WRITECOMBINE requires either the **SEC_RESERVE** or **SEC_COMMIT** attribute to be set.

[in] `MaximumSize`

The maximum size of the file mapping object.

An attempt to map a file with a length of 0 (zero) fails with an error code of **ERROR_FILE_INVALID**. Applications should test for files with a length of 0 (zero) and reject those files.

[in, optional] `Name`

The name of the file mapping object.

If this parameter matches the name of an existing mapping object, the function requests access to the object with the protection that *flProtect* specifies.

If this parameter is **NULL**, the file mapping object is created without a name.

If *lpName* matches the name of an existing event, semaphore, mutex, waitable timer, or job object, the function fails, and the [GetLastError](#) function returns **ERROR_INVALID_HANDLE**. This occurs because these objects share the same namespace.

The name can have a "Global" or "Local" prefix to explicitly create the object in the global or session namespace. The remainder of the name can contain any character except the backslash

character (\). Creating a file mapping object in the global namespace from a session other than session zero requires the [SeCreateGlobalPrivilege](#) privilege. For more information, see [Kernel Object Namespaces](#).

Fast user switching is implemented by using Terminal Services sessions. The first user to log on uses session 0 (zero), the next user to log on uses session 1 (one), and so on. Kernel object names must follow the guidelines that are outlined for Terminal Services so that applications can support multiple users.

Return value

If the function succeeds, the return value is a handle to the newly created file mapping object.

If the object exists before the function call, the function returns a handle to the existing object (with its current size, not the specified size), and [GetLastError](#) returns **ERROR_ALREADY_EXISTS**.

If the function fails, the return value is **NULL**. To get extended error information, call [GetLastError](#).

Remarks

After a file mapping object is created, the size of the file must not exceed the size of the file mapping object; if it does, not all of the file contents are available for sharing.

If an application specifies a size for the file mapping object that is larger than the size of the actual named file on disk and if the page protection allows write access (that is, the *flProtect* parameter specifies **PAGE_READWRITE**), then the file on disk is increased to match the specified size of the file mapping object. If the file is extended, the contents of the file between the old end of the file and the new end of the file are not guaranteed to be zero; the behavior is defined by the file system. If the file on disk cannot be increased, **CreateFileMappingFromApp** fails and [GetLastError](#) returns **ERROR_DISK_FULL**.

The initial contents of the pages in a file mapping object backed by the operating system paging file are 0 (zero).

The handle that **CreateFileMappingFromApp** returns has full access to a new file mapping object, and can be used with any function that requires a handle to a file mapping object.

Multiple processes can share a view of the same file by either using a single shared file mapping object or creating separate file mapping objects backed by the same file. A single file mapping object can be shared by multiple processes through inheriting the handle at process

creation, duplicating the handle, or opening the file mapping object by name. For more information, see the [CreateProcess](#), [DuplicateHandle](#) and [OpenFileMapping](#) functions.

Creating a file mapping object does not actually map the view into a process address space. The [MapViewOfFileEx](#) function map a view of a file into a process address space.

With one important exception, file views derived from any file mapping object that is backed by the same file are coherent or identical at a specific time. Coherency is guaranteed for views within a process and for views that are mapped by different processes.

The exception is related to remote files. Although **CreateFileMappingFromApp** works with remote files, it does not keep them coherent. For example, if two computers both map a file as writable, and both change the same page, each computer only sees its own writes to the page. When the data gets updated on the disk, it is not merged.

A mapped file and a file that is accessed by using the input and output (I/O) functions ([ReadFile](#) and [WriteFile](#)) are not necessarily coherent.


Mapped views of a file mapping object maintain internal references to the object, and a file mapping object does not close until all references to it are released. Therefore, to fully close a file mapping object, an application must unmap all mapped views of the file mapping object by calling [UnmapViewOfFile](#) and close the file mapping object handle by calling [CloseHandle](#). These functions can be called in any order.

When modifying a file through a mapped view, the last modification timestamp may not be updated automatically. If required, the caller should use [SetFileTime](#) to set the timestamp.

Use structured exception handling to protect any code that writes to or reads from a file view. For more information, see [Reading and Writing From a File View](#).

You can only successfully request executable protection if your app has the **codeGeneration** capability.

Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 8 [desktop apps UWP apps]
Minimum supported server	Windows Server 2012 [desktop apps UWP apps]
Target Platform	Windows

Requirement	Value
Header	memoryapi.h (include Windows.h)
Library	oncore.lib
DLL	Kernel32.dll

See also

[CloseHandle](#)

[CreateFileMapping](#)

[Creating a File Mapping Object](#)

[DuplicateHandle](#)

[File Mapping Functions](#)

[MapViewOfFile](#)

[MapViewOfFileEx](#)

[Memory Management Functions](#)

[OpenFileMapping](#)

[ReadFile](#)

[SECURITY_ATTRIBUTES](#)

[UnmapViewOfFile](#)

[VirtualAlloc](#)

[WriteFile](#)

CreateFileMappingNumaW function (memoryapi.h)

Article 07/27/2022

Creates or opens a named or unnamed file mapping object for a specified file and specifies the NUMA node for the physical memory.

Syntax

C++

```
HANDLE CreateFileMappingNumaW(  
    [in] HANDLE hFile,  
    [in, optional] LPSECURITY_ATTRIBUTES lpFileMappingAttributes,  
    [in] DWORD flProtect,  
    [in] DWORD dwMaximumSizeHigh,  
    [in] DWORD dwMaximumSizeLow,  
    [in, optional] LPCWSTR lpName,  
    [in] DWORD nndPreferred  
);
```

Parameters

[in] hFile

A handle to the file from which to create a file mapping object.

The file must be opened with access rights that are compatible with the protection flags that the *flProtect* parameter specifies. It is not required, but it is recommended that files you intend to map be opened for exclusive access. For more information, see [File Security and Access Rights](#).

If *hFile* is **INVALID_HANDLE_VALUE**, the calling process must also specify a size for the file mapping object in the *dwMaximumSizeHigh* and *dwMaximumSizeLow* parameters. In this scenario, **CreateFileMappingNuma** creates a file mapping object of a specified size that is backed by the system paging file instead of by a file in the file system.

[in, optional] lpFileMappingAttributes

A pointer to a [SECURITY_ATTRIBUTES](#) structure that determines whether a returned handle can be inherited by child processes. The **lpSecurityDescriptor** member of the

SECURITY_ATTRIBUTES structure specifies a security descriptor for a new file mapping object.

If *lpFileMappingAttributes* is **NULL**, the handle cannot be inherited and the file mapping object gets a default security descriptor. The access control lists (ACL) in the default security descriptor for a file mapping object come from the primary or impersonation token of the creator. For more information, see [File Mapping Security and Access Rights](#).

[in] **flProtect**

Specifies the page protection of the file mapping object. All mapped views of the object must be compatible with this protection.

This parameter can be one of the following values.

Value	Meaning
PAGE_EXECUTE_READ 0x20	Allows views to be mapped for read-only, copy-on-write, or execute access. The file handle that the <i>hFile</i> parameter specifies must be created with the GENERIC_READ and GENERIC_EXECUTE access rights.
PAGE_EXECUTE_READWRITE 0x40	Allows views to be mapped for read-only, copy-on-write, read/write or execute access. The file handle that the <i>hFile</i> parameter specifies must be created with the GENERIC_READ , GENERIC_WRITE , and GENERIC_EXECUTE access rights.
PAGE_EXECUTE_WRITECOPY 0x80	Allows views to be mapped for read-only, copy-on-write, or execute access. This value is equivalent to PAGE_EXECUTE_READ . The file handle that the <i>hFile</i> parameter specifies must be created with the GENERIC_READ and GENERIC_EXECUTE access rights. Windows Vista: This value is not available until Windows Vista with SP1.
PAGE_READONLY 0x02	Allows views to be mapped for read-only or copy-on-write access. An attempt to write to a specific region results in an access violation. The file handle that the <i>hFile</i> parameter specifies must be created with the GENERIC_READ access right.
PAGE_READWRITE 0x04	Allows views to be mapped for read-only, copy-on-write, or read/write access.

	The file handle that the <i>hFile</i> parameter specifies must be created with the GENERIC_READ and GENERIC_WRITE access rights.
PAGE_WRITECOPY 0x08	<p>Allows views to be mapped for read-only or copy-on-write access. This value is equivalent to PAGE_READONLY.</p> <p>The file handle that the <i>hFile</i> parameter specifies must be created with the GENERIC_READ access right.</p>

An application can specify one or more of the following attributes for the file mapping object by combining them with one of the preceding page protection values.

Value	Meaning
SEC_COMMIT 0x80000000	<p>Allocates physical storage in memory or the paging file for all pages.</p> <p>This is the default setting.</p>
SEC_IMAGE 0x10000000	<p>Sets the file that is specified to be an executable image file.</p> <p>The SEC_IMAGE attribute must be combined with a page protection value such as PAGE_READONLY. However, this page protection value has no effect on views of the executable image file. Page protection for views of an executable image file is determined by the executable file itself.</p> <p>No other attributes are valid with SEC_IMAGE.</p>
SEC_IMAGE_NO_EXECUTE 0x11000000	<p>Specifies that the file that the <i>hFile</i> parameter specifies is an executable image file that will not be executed and the loaded image file will have no forced integrity checks run. Additionally, mapping a view of a file mapping object created with the SEC_IMAGE_NO_EXECUTE attribute will not invoke driver callbacks registered using the PsSetLoadImageNotifyRoutine kernel API.</p> <p>The SEC_IMAGE_NO_EXECUTE attribute must be combined with the PAGE_READONLY page protection value. No other attributes are valid with SEC_IMAGE_NO_EXECUTE.</p> <p>Windows Server 2008 R2, Windows 7, Windows Server 2008 and Windows Vista: This value is not supported before Windows Server 2012 and Windows 8.</p>
SEC_LARGE_PAGES 0x80000000	Enables large pages to be used when mapping images or backing from the pagefile, but not when mapping data

	for regular files. Be sure to specify the maximum size of the file mapping object as the minimum size of a large page reported by the GetLargePageMinimum function and to enable the SeLockMemoryPrivilege privilege.
SEC_NOCACHE 0x10000000	<p>Sets all pages to noncachable.</p> <p>Applications should not use this flag except when explicitly required for a device. Using the interlocked functions with memory mapped with SEC_NOCACHE can result in an EXCEPTION_ILLEGAL_INSTRUCTION exception.</p> <p>SEC_NOCACHE requires either SEC_RESERVE or SEC_COMMIT to be set.</p>
SEC_RESERVE 0x40000000	<p>Reserves all pages without allocating physical storage. The reserved range of pages cannot be used by any other allocation operations until the range of pages is released.</p> <p>Reserved pages can be identified in subsequent calls to the VirtualAllocExNuma function. This attribute is valid only if the <i>hFile</i> parameter is INVALID_HANDLE_VALUE (that is, a file mapping object that is backed by the system paging file).</p>
SEC_WRITECOMBINE 0x40000000	<p>Sets all pages to be write-combined.</p> <p>Applications should not use this attribute except when explicitly required for a device. Using the interlocked functions with memory that is mapped with SEC_WRITECOMBINE can result in an EXCEPTION_ILLEGAL_INSTRUCTION exception.</p> <p>SEC_WRITECOMBINE requires either the SEC_RESERVE or SEC_COMMIT attribute to be set.</p>

[in] `dwMaximumSizeHigh`

The high-order **DWORD** of the maximum size of the file mapping object.

[in] `dwMaximumSizeLow`

The low-order **DWORD** of the maximum size of the file mapping object.

If this parameter and the *dwMaximumSizeHigh* parameter are 0 (zero), the maximum size of the file mapping object is equal to the current size of the file that the *hFile* parameter identifies.

An attempt to map a file with a length of 0 (zero) fails with an error code of **ERROR_FILE_INVALID**. Applications should test for files with a length of 0 (zero) and

reject those files.

[in, optional] lpName

The name of the file mapping object.

If this parameter matches the name of an existing file mapping object, the function requests access to the object with the protection that the *flProtect* parameter specifies.

If this parameter is **NULL**, the file mapping object is created without a name.

If the *lpName* parameter matches the name of an existing event, semaphore, mutex, waitable timer, or job object, the function fails and the [GetLastError](#) function returns **ERROR_INVALID_HANDLE**. This occurs because these objects share the same namespace.

The name can have a "Global" or "Local" prefix to explicitly create the object in the global or session namespace. The remainder of the name can contain any character except the backslash character (\). Creating a file mapping object in the global namespace requires the [SeCreateGlobalPrivilege](#) privilege. For more information, see [Kernel Object Namespaces](#).

Fast user switching is implemented by using Terminal Services sessions. The first user to log on uses session 0 (zero), the next user to log on uses session 1 (one), and so on. Kernel object names must follow the guidelines so that applications can support multiple users.

[in] nndPreferred

The NUMA node where the physical memory should reside.

Value	Meaning
NUMA_NO_PREFERRED_NODE 0xffffffff	No NUMA node is preferred. This is the same as calling the CreateFileMapping function.

Return value

If the function succeeds, the return value is a handle to the file mapping object.

If the object exists before the function call, the function returns a handle to the existing object (with its current size, not the specified size) and the [GetLastError](#) function returns **ERROR_ALREADY_EXISTS**.

If the function fails, the return value is **NULL**. To get extended error information, call the [GetLastError](#) function.

Remarks

After a file mapping object is created, the size of the file must not exceed the size of the file mapping object; if it does, not all of the file contents are available for sharing.

The file mapping object can be shared by duplication, inheritance, or by name. The initial contents of the pages in a file mapping object backed by the page file are 0 (zero).

If an application specifies a size for the file mapping object that is larger than the size of the actual named file on disk and if the page protection allows write access (that is, the *flProtect* parameter specifies **PAGE_READWRITE** or **PAGE_EXECUTE_READWRITE**), then the file on disk is increased to match the specified size of the file mapping object. If the file is extended, the contents of the file between the old end of the file and the new end of the file are not guaranteed to be zero; the behavior is defined by the file system.

If the file cannot be increased, the result is a failure to create the file mapping object and the [GetLastError](#) function returns **ERROR_DISK_FULL**.

The handle that the **CreateFileMappingNuma** function returns has full access to a new file mapping object and can be used with any function that requires a handle to a file mapping object. A file mapping object can be shared through process creation, handle duplication, or by name. For more information, see the [DuplicateHandle](#) and [OpenFileMapping](#) functions.

Creating a file mapping object creates the potential for mapping a view of the file but does not map the view. The [MapViewOfFileExNuma](#) function maps a view of a file into a process address space.

With one important exception, file views derived from a single file mapping object are coherent or identical at a specific time. If multiple processes have handles of the same file mapping object, they see a coherent view of the data when they map a view of the file.

The exception is related to remote files. Although the **CreateFileMappingNuma** function works with remote files, it does not keep them coherent. For example, if two computers both map a file as writable and both change the same page, each computer sees only its own writes to the page. When the data gets updated on the disk, the page is not merged.

A mapped file and a file that is accessed by using the input and output (I/O) functions ([ReadFile](#) and [WriteFile](#)) are not necessarily coherent.

To fully close a file mapping object, an application must unmap all mapped views of the file mapping object by calling the [UnmapViewOfFile](#) function and then close the file mapping object handle by calling the [CloseHandle](#) function.

These functions can be called in any order. The call to the [UnmapViewOfFile](#) function is necessary, because mapped views of a file mapping object maintain internal open handles to the object, and a file mapping object does not close until all open handles to it are closed.

When modifying a file through a mapped view, the last modification timestamp may not be updated automatically. If required, the caller should use [SetFileTime](#) to set the timestamp.

Creating a file-mapping object from a session other than session zero requires the [SeCreateGlobalPrivilege](#) privilege. Note that this privilege check is limited to the creation of file mapping objects and does not apply to opening existing ones. For example, if a service or the system creates a file mapping object, any process running in any session can access that file mapping object provided that the caller has the required access rights.

Use structured exception handling to protect any code that writes to or reads from a memory mapped view. For more information, see [Reading and Writing From a File View](#).

To have a mapping with executable permissions, an application must call the [CreateFileMappingNuma](#) function with either **PAGE_EXECUTE_READWRITE** or **PAGE_EXECUTE_READ** and then call the [MapViewOfFileExNuma](#) function with `FILE_MAP_EXECUTE | FILE_MAP_WRITE` or `FILE_MAP_EXECUTE | FILE_MAP_READ`.

In Windows Server 2012, this function is supported by the following technologies.

Technology	Supported
Server Message Block (SMB) 3.0 protocol	Yes
SMB 3.0 Transparent Failover (TFO)	Yes
SMB 3.0 with Scale-out File Shares (SO)	Yes
Cluster Shared Volume File System (CsvFS)	Yes
Resilient File System (ReFS)	Yes

Requirements

Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	memoryapi.h (include Windows.h, Memoryapi.h)
Library	oncore.lib
DLL	Kernel32.dll

See also

[CloseHandle](#)

[CreateFileMapping](#)

[DuplicateHandle](#)

[File Mapping Functions](#)

[MapViewOfFileExNuma](#)

[NUMA Support](#)

[OpenFileMapping](#)

[ReadFile](#)

[SECURITY_ATTRIBUTES](#)

[UnmapViewOfFile](#)

[VirtualAllocExNuma](#)

[WriteFile](#)

Feedback

Was this page helpful?

Get help at [Microsoft Q&A](#)

CreateFileMappingW function (memoryapi.h)

Article 07/27/2022

Creates or opens a named or unnamed file mapping object for a specified file.

To specify the NUMA node for the physical memory, see [CreateFileMappingNuma](#).

Syntax

C++

```
HANDLE CreateFileMapping(  
    [in] HANDLE hFile,  
    [in, optional] LPSECURITY_ATTRIBUTES lpFileMappingAttributes,  
    [in] DWORD flProtect,  
    [in] DWORD dwMaximumSizeHigh,  
    [in] DWORD dwMaximumSizeLow,  
    [in, optional] LPCWSTR lpName  
);
```

Parameters

[in] `hFile`

A handle to the file from which to create a file mapping object.

The file must be opened with access rights that are compatible with the protection flags that the *flProtect* parameter specifies. It is not required, but it is recommended that files you intend to map be opened for exclusive access. For more information, see [File Security and Access Rights](#).

If *hFile* is `INVALID_HANDLE_VALUE`, the calling process must also specify a size for the file mapping object in the *dwMaximumSizeHigh* and *dwMaximumSizeLow* parameters. In this scenario, **CreateFileMapping** creates a file mapping object of a specified size that is backed by the system paging file instead of by a file in the file system.

[in, optional] `lpFileMappingAttributes`

A pointer to a [SECURITY_ATTRIBUTES](#) structure that determines whether a returned handle can be inherited by child processes. The **lpSecurityDescriptor** member of the

SECURITY_ATTRIBUTES structure specifies a security descriptor for a new file mapping object.

If *lpAttributes* is **NULL**, the handle cannot be inherited and the file mapping object gets a default security descriptor. The access control lists (ACL) in the default security descriptor for a file mapping object come from the primary or impersonation token of the creator. For more information, see [File Mapping Security and Access Rights](#).

[in] **flProtect**

Specifies the page protection of the file mapping object. All mapped views of the object must be compatible with this protection.

This parameter can be one of the following values.

Value	Meaning
PAGE_EXECUTE_READ 0x20	<p>Allows views to be mapped for read-only, copy-on-write, or execute access.</p> <p>The file handle specified by the <i>hFile</i> parameter must be created with the GENERIC_READ and GENERIC_EXECUTE access rights.</p> <p>Windows Server 2003 and Windows XP: This value is not available until Windows XP with SP2 and Windows Server 2003 with SP1.</p>
PAGE_EXECUTE_READWRITE 0x40	<p>Allows views to be mapped for read-only, copy-on-write, read/write, or execute access.</p> <p>The file handle that the <i>hFile</i> parameter specifies must be created with the GENERIC_READ, GENERIC_WRITE, and GENERIC_EXECUTE access rights.</p> <p>Windows Server 2003 and Windows XP: This value is not available until Windows XP with SP2 and Windows Server 2003 with SP1.</p>
PAGE_EXECUTE_WRITECOPY 0x80	<p>Allows views to be mapped for read-only, copy-on-write, or execute access. This value is equivalent to PAGE_EXECUTE_READ.</p> <p>The file handle that the <i>hFile</i> parameter specifies must be created with the GENERIC_READ and GENERIC_EXECUTE access rights.</p> <p>Windows Vista: This value is not available until Windows Vista with SP1.</p> <p>Windows Server 2003 and Windows XP: This value is not supported.</p>

PAGE_READONLY 0x02	<p>Allows views to be mapped for read-only or copy-on-write access. An attempt to write to a specific region results in an access violation.</p> <p>The file handle that the <i>hFile</i> parameter specifies must be created with the GENERIC_READ access right.</p>
PAGE_READWRITE 0x04	<p>Allows views to be mapped for read-only, copy-on-write, or read/write access.</p> <p>The file handle that the <i>hFile</i> parameter specifies must be created with the GENERIC_READ and GENERIC_WRITE access rights.</p>
PAGE_WRITECOPY 0x08	<p>Allows views to be mapped for read-only or copy-on-write access. This value is equivalent to PAGE_READONLY.</p> <p>The file handle that the <i>hFile</i> parameter specifies must be created with the GENERIC_READ access right.</p>

An application can specify one or more of the following attributes for the file mapping object by combining them with one of the preceding page protection values.

Value	Meaning
SEC_COMMIT 0x80000000	<p>If the file mapping object is backed by the operating system paging file (the <i>hfile</i> parameter is INVALID_HANDLE_VALUE), specifies that when a view of the file is mapped into a process address space, the entire range of pages is committed rather than reserved. The system must have enough committable pages to hold the entire mapping. Otherwise, CreateFileMapping fails.</p> <p>This attribute has no effect for file mapping objects that are backed by executable image files or data files (the <i>hfile</i> parameter is a handle to a file).</p> <p>SEC_COMMIT cannot be combined with SEC_RESERVE.</p> <p>If no attribute is specified, SEC_COMMIT is assumed. However, SEC_COMMIT must be explicitly specified when combining it with another SEC_ attribute that requires it.</p>
SEC_IMAGE 0x10000000	<p>Specifies that the file that the <i>hFile</i> parameter specifies is an executable image file.</p> <p>The SEC_IMAGE attribute must be combined with a page protection value such as PAGE_READONLY. However, this page protection value has no effect on views of the executable image file. Page protection for views of an</p>

	<p>executable image file is determined by the executable file itself.</p> <p>No other attributes are valid with SEC_IMAGE.</p>
SEC_IMAGE_NO_EXECUTE 0x11000000	<p>Specifies that the file that the <i>hFile</i> parameter specifies is an executable image file that will not be executed and the loaded image file will have no forced integrity checks run. Additionally, mapping a view of a file mapping object created with the SEC_IMAGE_NO_EXECUTE attribute will not invoke driver callbacks registered using the PsSetLoadImageNotifyRoutine kernel API.</p> <p>The SEC_IMAGE_NO_EXECUTE attribute must be combined with the PAGE_READONLY page protection value. No other attributes are valid with SEC_IMAGE_NO_EXECUTE.</p> <p>Windows Server 2008 R2, Windows 7, Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP: This value is not supported before Windows Server 2012 and Windows 8.</p>
SEC_LARGE_PAGES 0x80000000	<p>Enables large pages to be used for file mapping objects that are backed by the operating system paging file (the <i>hfile</i> parameter is INVALID_HANDLE_VALUE). This attribute is not supported for file mapping objects that are backed by executable image files or data files (the <i>hFile</i> parameter is a handle to an executable image or data file).</p> <p>The maximum size of the file mapping object must be a multiple of the minimum size of a large page returned by the GetLargePageMinimum function. If it is not, CreateFileMapping fails. When mapping a view of a file mapping object created with SEC_LARGE_PAGES, the base address and view size must also be multiples of the minimum large page size.</p> <p>SEC_LARGE_PAGES requires the SeLockMemoryPrivilege privilege to be enabled in the caller's token.</p> <p>If SEC_LARGE_PAGES is specified, SEC_COMMIT must also be specified.</p> <p>Windows Server 2003: This value is not supported until Windows Server 2003 with SP1.</p> <p>Windows XP: This value is not supported.</p>
SEC_NOCACHE 0x10000000	<p>Sets all pages to be non-cacheable.</p>

	<p>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.</p> <p>SEC_NOCACHE requires either the SEC_RESERVE or SEC_COMMIT attribute to be set.</p>
SEC_RESERVE 0x4000000	<p>If the file mapping object is backed by the operating system paging file (the <i>hfile</i> parameter is INVALID_HANDLE_VALUE), specifies that when a view of the file is mapped into a process address space, the entire range of pages is reserved for later use by the process rather than committed.</p> <p>Reserved pages can be committed in subsequent calls to the VirtualAlloc function. After the pages are committed, they cannot be freed or decommitted with the VirtualFree function.</p> <p>This attribute has no effect for file mapping objects that are backed by executable image files or data files (the <i>hfile</i> parameter is a handle to a file).</p> <p>SEC_RESERVE cannot be combined with SEC_COMMIT.</p>
SEC_WRITECOMBINE 0x40000000	<p>Sets all pages to be write-combined.</p> <p>Applications should not use this attribute except when explicitly required for a device. Using the interlocked functions with memory that is mapped with SEC_WRITECOMBINE can result in an EXCEPTION_ILLEGAL_INSTRUCTION exception.</p> <p>SEC_WRITECOMBINE requires either the SEC_RESERVE or SEC_COMMIT attribute to be set.</p> <p>Windows Server 2003 and Windows XP: This flag is not supported until Windows Vista.</p>

[in] `dwMaximumSizeHigh`

The high-order **DWORD** of the maximum size of the file mapping object.

[in] `dwMaximumSizeLow`

The low-order **DWORD** of the maximum size of the file mapping object.

If this parameter and *dwMaximumSizeHigh* are 0 (zero), the maximum size of the file mapping object is equal to the current size of the file that *hFile* identifies.

An attempt to map a file with a length of 0 (zero) fails with an error code of **ERROR_FILE_INVALID**. Applications should test for files with a length of 0 (zero) and reject those files.

`[in, optional] lpName`

The name of the file mapping object.

If this parameter matches the name of an existing mapping object, the function requests access to the object with the protection that *flProtect* specifies.

If this parameter is **NULL**, the file mapping object is created without a name.

If *lpName* matches the name of an existing event, semaphore, mutex, waitable timer, or job object, the function fails, and the [GetLastError](#) function returns **ERROR_INVALID_HANDLE**. This occurs because these objects share the same namespace.

The name can have a "Global" or "Local" prefix to explicitly create the object in the global or session namespace. The remainder of the name can contain any character except the backslash character (\). Creating a file mapping object in the global namespace from a session other than session zero requires the [SeCreateGlobalPrivilege](#) privilege. For more information, see [Kernel Object Namespaces](#).

Fast user switching is implemented by using Terminal Services sessions. The first user to log on uses session 0 (zero), the next user to log on uses session 1 (one), and so on. Kernel object names must follow the guidelines that are outlined for Terminal Services so that applications can support multiple users.

Return value

If the function succeeds, the return value is a handle to the newly created file mapping object.

If the object exists before the function call, the function returns a handle to the existing object (with its current size, not the specified size), and [GetLastError](#) returns **ERROR_ALREADY_EXISTS**.

If the function fails, the return value is **NULL**. To get extended error information, call [GetLastError](#).

Remarks

After a file mapping object is created, the size of the file must not exceed the size of the file mapping object; if it does, not all of the file contents are available for sharing.

If an application specifies a size for the file mapping object that is larger than the size of the actual named file on disk and if the page protection allows write access (that is, the *flProtect* parameter specifies **PAGE_READWRITE** or **PAGE_EXECUTE_READWRITE**), then the file on disk is increased to match the specified size of the file mapping object. If the file is extended, the contents of the file between the old end of the file and the new end of the file are not guaranteed to be zero; the behavior is defined by the file system. If the file on disk cannot be increased, **CreateFileMapping** fails and [GetLastError](#) returns **ERROR_DISK_FULL**.

The initial contents of the pages in a file mapping object backed by the operating system paging file are 0 (zero).

The handle that **CreateFileMapping** returns has full access to a new file mapping object, and can be used with any function that requires a handle to a file mapping object.

Multiple processes can share a view of the same file by either using a single shared file mapping object or creating separate file mapping objects backed by the same file. A single file mapping object can be shared by multiple processes through inheriting the handle at process creation, duplicating the handle, or opening the file mapping object by name. For more information, see the [CreateProcess](#), [DuplicateHandle](#) and [OpenFileMapping](#) functions.

Creating a file mapping object does not actually map the view into a process address space. The [MapViewOfFile](#) and [MapViewOfFileEx](#) functions map a view of a file into a process address space.

With one important exception, file views derived from any file mapping object that is backed by the same file are coherent or identical at a specific time. Coherency is guaranteed for views within a process and for views that are mapped by different processes.

The exception is related to remote files. Although **CreateFileMapping** works with remote files, it does not keep them coherent. For example, if two computers both map a file as writable, and both change the same page, each computer only sees its own writes to the page. When the data gets updated on the disk, it is not merged.

A mapped file and a file that is accessed by using the input and output (I/O) functions ([ReadFile](#) and [WriteFile](#)) are not necessarily coherent.

Mapped views of a file mapping object maintain internal references to the object, and a file mapping object does not close until all references to it are released. Therefore, to

fully close a file mapping object, an application must unmap all mapped views of the file mapping object by calling [UnmapViewOfFile](#) and close the file mapping object handle by calling [CloseHandle](#). These functions can be called in any order.

When modifying a file through a mapped view, the last modification timestamp may not be updated automatically. If required, the caller should use [SetFileTime](#) to set the timestamp.

Creating a file mapping object in the global namespace from a session other than session zero requires the [SeCreateGlobalPrivilege](#) privilege. Note that this privilege check is limited to the creation of file mapping objects and does not apply to opening existing ones. For example, if a service or the system creates a file mapping object in the global namespace, any process running in any session can access that file mapping object provided that the caller has the required access rights.

Windows XP: The requirement described in the previous paragraph was introduced with Windows Server 2003 and Windows XP with SP2

Use structured exception handling to protect any code that writes to or reads from a file view. For more information, see [Reading and Writing From a File View](#).

To have a mapping with executable permissions, an application must call **CreateFileMapping** with either **PAGE_EXECUTE_READWRITE** or **PAGE_EXECUTE_READ**, and then call [MapViewOfFile](#) with `FILE_MAP_EXECUTE | FILE_MAP_WRITE` or `FILE_MAP_EXECUTE | FILE_MAP_READ`.

In Windows Server 2012, this function is supported by the following technologies.

Technology	Supported
Server Message Block (SMB) 3.0 protocol	Yes
SMB 3.0 Transparent Failover (TFO)	Yes
SMB 3.0 with Scale-out File Shares (SO)	Yes
Cluster Shared Volume File System (CsvFS)	Yes
Resilient File System (ReFS)	Yes

Examples

For an example, see [Creating Named Shared Memory](#) or [Creating a File Mapping Using Large Pages](#).

Requirements

Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	memoryapi.h (include Windows.h, Memoryapi.h)
Library	oncore.lib
DLL	Kernel32.dll

See also

[CloseHandle](#)

[CreateFileMappingNuma](#)

[Creating a File Mapping Object](#)

[DuplicateHandle](#)

[File Mapping Functions](#)

[MapViewOfFile](#)

[MapViewOfFileEx](#)

[Memory Management Functions](#)

[OpenFileMapping](#)

[ReadFile](#)

[SECURITY_ATTRIBUTES](#)

[UnmapViewOfFile](#)

[VirtualAlloc](#)

[WriteFile](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

CreateMemoryResourceNotification function (memoryapi.h)

Article05/14/2022

Creates a memory resource notification object.

Syntax

```
C++

HANDLE CreateMemoryResourceNotification(
    [in] MEMORY_RESOURCE_NOTIFICATION_TYPE NotificationType
);
```

Parameters

[in] NotificationType

The memory condition under which the object is to be signaled. This parameter can be one of the following values from the **MEMORY_RESOURCE_NOTIFICATION_TYPE** enumeration.

 Expand table

Value	Meaning
LowMemoryResourceNotification 0	Available physical memory is running low.
HighMemoryResourceNotification 1	Available physical memory is high.

Return value

If the function succeeds, the return value is a handle to a memory resource notification object.

If the function fails, the return value is **NULL**. To get extended information, call [GetLastError](#).

Remarks

Applications can use memory resource notification events to scale the memory usage as appropriate. If available memory is low, the application can reduce its working set. If available


memory is high, the application can allocate more memory.

Any thread of the calling process can specify the memory resource notification handle in a call to the [QueryMemoryResourceNotification](#) function or one of the [wait functions](#). The state of the object is signaled when the specified memory condition exists. This is a system-wide event, so all applications receive notification when the object is signaled. Note that there is a range of memory availability where neither the **LowMemoryResourceNotification** or **HighMemoryResourceNotification** object is signaled. In this case, applications should attempt to keep the memory use constant.

Use the [CloseHandle](#) function to close the handle. The system closes the handle automatically when the process terminates. The memory resource notification object is destroyed when its last handle has been closed.

To compile an application that uses this function, define the `_WIN32_WINNT` macro as 0x0501 or later. For more information, see [Using the Windows Headers](#).

Requirements

 [Expand table](#)

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	memoryapi.h (include Windows.h, Memoryapi.h)
Library	oncore.lib
DLL	Kernel32.dll

See also

[CloseHandle](#)

[Memory Management Functions](#)

[QueryMemoryResourceNotification](#)

DiscardVirtualMemory function (memoryapi.h)

Article 02/22/2024

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.

Syntax

C++

```
DWORD DiscardVirtualMemory(  
    [in] PVOID VirtualAddress,  
    [in] SIZE_T Size  
);
```

Parameters

[in] VirtualAddress

Page-aligned starting address of the memory to discard.

[in] Size

Size, in bytes, of the memory region to discard. *Size* must be an integer multiple of the system page size.

Return value

ERROR_SUCCESS if successful; a [System Error Code](#) otherwise.

Remarks

If **DiscardVirtualMemory** fails, the contents of the region is not altered.

Use this function to discard memory contents that are no longer needed, while keeping the memory region itself committed. Discarding memory may give physical RAM back to the system. When the region of memory is again accessed by the application, the backing RAM is restored, and the contents of the memory is undefined.

Important Calls to `DiscardVirtualMemory` will fail if the memory protection is not `PAGE_READWRITE`.

Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 8.1 Update [desktop apps UWP apps]
Minimum supported server	Windows Server 2012 R2 Update [desktop apps UWP apps]
Target Platform	Windows
Header	memoryapi.h (include Windows.h, Memoryapi.h)
Library	oncore.lib
DLL	Kernel32.dll

See also

[Memory Management Functions](#)

[OfferVirtualMemory](#)

[ReclaimVirtualMemory](#)

[Virtual Memory Functions](#)

[VirtualAlloc](#)

[VirtualFree](#)

[VirtualLock](#)

[VirtualQuery](#)

FlushViewOfFile function (memoryapi.h)

Article 05/14/2022

Writes to the disk a byte range within a mapped view of a file.

Syntax

C++

```
BOOL FlushViewOfFile(  
    [in] LPCVOID lpBaseAddress,  
    [in] SIZE_T dwNumberOfBytesToFlush  
);
```

Parameters

[in] lpBaseAddress

A pointer to the base address of the byte range to be flushed to the disk representation of the mapped file.

[in] dwNumberOfBytesToFlush

The number of bytes to be flushed. If *dwNumberOfBytesToFlush* is zero, the file is flushed from the base address to the end of the mapping.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

Flushing a range of a mapped view initiates writing of dirty pages within that range to the disk. Dirty pages are those whose contents have changed since the file view was mapped. The **FlushViewOfFile** function does not flush the file metadata, and it does not wait to return until the changes are flushed from the underlying hardware disk cache and physically written to disk. To flush all the dirty pages plus the metadata for the file and ensure that they are physically written to disk, call **FlushViewOfFile** and then call the [FlushFileBuffers](#) function.

When flushing a memory-mapped file over a network, **FlushViewOfFile** guarantees that the data has been written from the local computer, but not that the data resides on the remote computer. The server can cache the data on the remote side. Therefore, **FlushViewOfFile** can return before the data has been physically written to disk.

When modifying a file through a mapped view, the last modification timestamp may not be updated automatically. If required, the caller should use [SetFileTime](#) to set the timestamp.

In Windows Server 2012, this function is supported by the following technologies.

[Expand table](#)

Technology	Supported
Server Message Block (SMB) 3.0 protocol	Yes
SMB 3.0 Transparent Failover (TFO)	Yes
SMB 3.0 with Scale-out File Shares (SO)	Yes
Cluster Shared Volume File System (CsvFS)	Yes
Resilient File System (ReFS)	Yes

When CsvFs is paused this call might fail with an error indicating that there is a lock conflict.

Examples

For an example, see [Reading and Writing From a File View](#).

Requirements

[Expand table](#)

Requirement	Value
Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	memoryapi.h (include Windows.h, Memoryapi.h)
Library	oncore.lib

Requirement	Value
DLL	Kernel32.dll

See also

[CreateFile](#)

[Creating a File View](#)

[File Mapping Functions](#)

[MapViewOfFile](#)

[UnmapViewOfFile](#)

FreeUserPhysicalPages function (memoryapi.h)

Article 02/22/2024

Frees physical memory pages that are allocated previously by using [AllocateUserPhysicalPages](#) or [AllocateUserPhysicalPagesNuma](#). If any of these pages are currently mapped in the [Address Windowing Extensions](#) (AWE) region, they are automatically unmapped by this call. This does not affect the virtual address space that is occupied by a specified Address Windowing Extensions (AWE) region.

64-bit Windows on Itanium-based systems: Due to the difference in page sizes, **FreeUserPhysicalPages** is not supported for 32-bit applications.

Syntax

C++

```
BOOL FreeUserPhysicalPages(  
    [in]      HANDLE      hProcess,  
    [in, out] PULONG_PTR  NumberOfPages,  
    [in]      PULONG_PTR  PageArray  
);
```

Parameters

[in] hProcess

The handle to a process.

The function frees memory within the virtual address space of this process.

[in, out] NumberOfPages

The size of the physical memory to free, in pages.

On return, if the function fails, this parameter indicates the number of pages that are freed.

[in] PageArray

A pointer to an array of page frame numbers of the allocated memory to be freed.

Return value

If the function succeeds, the return value is **TRUE**.

If the function fails, the return value is **FALSE**. In this case, the *NumberOfPages* parameter reflect how many pages have actually been released. To get extended error information, call [GetLastError](#).

Remarks

In a multiprocessor environment, this function maintains coherence of the hardware translation buffer. When this function returns, all threads on all processors are guaranteed to see the correct mapping.

To compile an application that uses this function, define the `_WIN32_WINNT` macro as 0x0500 or later. For more information, see [Using the Windows Headers](#).

Examples

For an example, see [AWE Example](#).

Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	memoryapi.h (include Windows.h, Memoryapi.h)
Library	oncore.lib
DLL	Kernel32.dll

See also

[Address Windowing Extensions](#)

[AllocateUserPhysicalPages](#)

[AllocateUserPhysicalPagesNuma](#)

MapUserPhysicalPages

MapUserPhysicalPagesScatter

Memory Management Functions

GetLargePageMinimum function (memoryapi.h)

Article 05/14/2022

Retrieves the minimum size of a large page.

Syntax

C++

```
SIZE_T GetLargePageMinimum();
```

Return value

If the processor supports large pages, the return value is the minimum size of a large page.

If the processor does not support large pages, the return value is zero.

Remarks

The minimum large page size varies, but it is typically 2 MB or greater.

Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows Vista [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	memoryapi.h (include Windows.h, Memoryapi.h)
Library	oncore.lib
DLL	Kernel32.dll

See also

[Large Page Support](#)

[Memory Management Functions](#)

GetMemoryErrorHandlingCapabilities function (memoryapi.h)

Article 05/14/2022

Gets the memory error handling capabilities of the system.

Syntax


C++

```
BOOL GetMemoryErrorHandlingCapabilities(  
    [out] PULONG Capabilities  
);
```

Parameters

[out] Capabilities

A **PULONG** that receives one or more of the following flags.

 Expand table

Value	Meaning
MEHC_PATROL_SCRUBBER_PRESENT 1	The hardware can detect and report failed memory.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

To compile an application that calls this function, define **_WIN32_WINNT** as **_WIN32_WINNT_WIN8** or higher. For more information, see [Using the Windows Headers](#).

Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]
Target Platform	Windows
Header	memoryapi.h (include Windows.h, Memoryapi.h)
Library	onecore.lib
DLL	Kernel32.dll

GetProcessWorkingSetSize function (memoryapi.h)

Article 02/22/2024

Retrieves the minimum and maximum working set sizes of the specified process.

Syntax

C++

```
BOOL GetProcessWorkingSetSize(  
    [in] HANDLE hProcess,  
    [out] PSIZE_T lpMinimumWorkingSetSize,  
    [out] PSIZE_T lpMaximumWorkingSetSize  
);
```

Parameters

[in] hProcess

A handle to the process whose working set sizes will be obtained. The handle must have the **PROCESS_QUERY_INFORMATION** or **PROCESS_QUERY_LIMITED_INFORMATION** access right. For more information, see [Process Security and Access Rights](#).

Windows Server 2003 and Windows XP: The handle must have the **PROCESS_QUERY_INFORMATION** access right.

[out] lpMinimumWorkingSetSize

A pointer to a variable that receives the minimum working set size of the specified process, in bytes. The virtual memory manager attempts to keep at least this much memory resident in the process whenever the process is active.

[out] lpMaximumWorkingSetSize

A pointer to a variable that receives the maximum working set size of the specified process, in bytes. The virtual memory manager attempts to keep no more than this much memory resident in the process whenever the process is active when memory is in short supply.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

The "working set" of a process is the set of memory pages currently visible to the process in physical RAM memory. These pages are resident and available for an application to use without triggering a page fault. The minimum and maximum working set sizes affect the virtual memory paging behavior of a process.

Examples

C++

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

int main(int argc, char *argv[])
{
    SIZE_T dwMin, dwMax;
    HANDLE hProcess;

    if (argc != 2)
    {
        printf("This program requires a process ID as an argument.\n");
        return 1;
    }

    // Retrieve a handle to the process.

    hProcess = OpenProcess( PROCESS_QUERY_INFORMATION,
                           FALSE, atoi(argv[1]));


    if (!hProcess)
    {
        printf( "OpenProcess failed (%d)\n", GetLastError() );
        return 1;
    }

    // Retrieve the working set size of the process.

    if (!GetProcessWorkingSetSize(hProcess, &dwMin, &dwMax))
    {
        printf("GetProcessWorkingSetSize failed (%d)\n",
               GetLastError());
        return 1;
    }
}
```

```
printf("Process ID: %d\n", atoi(argv[1]));  
printf("Minimum working set: %lu KB\n", dwMin/1024);  
printf("Maximum working set: %lu KB\n", dwMax/1024);  
  
CloseHandle(hProcess);  
  
return 0;  
}
```

Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	memoryapi.h
Library	oncore.lib
DLL	Kernel32.dll

See also

[Process Working Set](#)

[Processes](#)

[SetProcessWorkingSetSize function](#)

[SetProcessWorkingSetSizeEx function](#)

GetProcessWorkingSetSizeEx function (memoryapi.h)

Article 02/22/2024

Retrieves the minimum and maximum working set sizes of the specified process.

Syntax

C++

```
BOOL GetProcessWorkingSetSizeEx(  
    [in] HANDLE hProcess,  
    [out] PSIZE_T lpMinimumWorkingSetSize,  
    [out] PSIZE_T lpMaximumWorkingSetSize,  
    [out] PDWORD Flags  
);
```

Parameters

[in] hProcess

A handle to the process whose working set sizes will be obtained. The handle must have the **PROCESS_QUERY_INFORMATION** or **PROCESS_QUERY_LIMITED_INFORMATION** access right. For more information, see [Process Security and Access Rights](#).

Windows Server 2003: The handle must have the **PROCESS_QUERY_INFORMATION** access right.

[out] lpMinimumWorkingSetSize

A pointer to a variable that receives the minimum working set size of the specified process, in bytes. The virtual memory manager attempts to keep at least this much memory resident in the process whenever the process is active.

[out] lpMaximumWorkingSetSize

A pointer to a variable that receives the maximum working set size of the specified process, in bytes. The virtual memory manager attempts to keep no more than this much memory resident in the process whenever the process is active when memory is in short supply.

[out] Flags

The flags that control the enforcement of the minimum and maximum working set sizes.

[Expand table](#)

Value	Meaning
QUOTA_LIMITS_HARDWS_MIN_DISABLE 0x00000002	The working set may fall below the minimum working set limit if memory demands are high.
QUOTA_LIMITS_HARDWS_MIN_ENABLE 0x00000001	The working set will not fall below the minimum working set limit.
QUOTA_LIMITS_HARDWS_MAX_DISABLE 0x00000008	The working set may exceed the maximum working set limit if there is abundant memory.
QUOTA_LIMITS_HARDWS_MAX_ENABLE 0x00000004	The working set will not exceed the maximum working set limit.

Return value

None

Remarks

The "working set" of a process is the set of memory pages currently visible to the process in physical RAM memory. These pages are resident and available for an application to use without triggering a page fault. The minimum and maximum working set sizes affect the virtual memory paging behavior of a process.

Requirements

[Expand table](#)

Requirement	Value
Minimum supported client	Windows Vista [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	memoryapi.h (include Windows.h on Windows Vista, Windows 7, Windows Server 2008 Windows Server 2008 R2)
Library	oncore.lib

Requirement	Value
DLL	Kernel32.dll

See also

[Process Working Set](#)

[Processes](#)

[SetProcessWorkingSetSizeEx](#)

GetSystemFileCacheSize function (memoryapi.h)

Article02/22/2024

Retrieves the current size limits for the working set of the system cache.

Syntax

C++

```
BOOL GetSystemFileCacheSize(  
    [out] PSIZE_T lpMinimumFileCacheSize,  
    [out] PSIZE_T lpMaximumFileCacheSize,  
    [out] PDWORD lpFlags  
);
```

Parameters

[out] lpMinimumFileCacheSize


A pointer to a variable that receives the minimum size of the file cache, in bytes. The virtual memory manager attempts to keep at least this much memory resident in the system file cache, if there is a previous call to the [SetSystemFileCacheSize](#) function with the **FILE_CACHE_MIN_HARD_ENABLE** flag.

[out] lpMaximumFileCacheSize

A pointer to a variable that receives the maximum size of the file cache, in bytes. The virtual memory manager enforces this limit only if there is a previous call to [SetSystemFileCacheSize](#) with the **FILE_CACHE_MAX_HARD_ENABLE** flag.

[out] lpFlags

The flags that indicate which of the file cache limits are enabled.

 Expand table

Value	Meaning
FILE_CACHE_MAX_HARD_ENABLE 0x1	The maximum size limit is enabled. If this flag is not present, this limit is disabled.

FILE_CACHE_MIN_HARD_ENABLE
0x4

The minimum size limit is enabled. If this flag is not present, this limit is disabled.

Return value

If the function succeeds, the return value is a nonzero value.


If the function fails, the return value is 0 (zero). To get extended error information, call [GetLastError](#).

Remarks

To compile an application that uses this function, define **_WIN32_WINNT** as 0x0502 or later. For more information, see [Using the Windows Headers](#).

The **FILE_CACHE** constants will be defined in the Windows header files starting with the Windows SDK for Windows Server 2008. If you are using header files from an earlier version of the SDK, add the definitions shown in [SetSystemFileCacheSize](#) to your code.

Requirements

 [Expand table](#)

Requirement	Value
Minimum supported client	Windows Vista, Windows XP Professional x64 Edition [desktop apps only]
Minimum supported server	Windows Server 2008, Windows Server 2003 with SP1 [desktop apps only]
Target Platform	Windows
Header	memoryapi.h (include Windows.h, Memoryapi.h)
Library	oncore.lib
DLL	Kernel32.dll

See also

[Memory Management Functions](#)

[SetSystemFileCacheSize](#)

GetWriteWatch function (memoryapi.h)

Article 05/14/2022

Retrieves the addresses of the pages that are written to in a region of virtual memory.

64-bit Windows on Itanium-based systems: Due to the difference in page sizes, **GetWriteWatch** is not supported for 32-bit applications.

Syntax

C++

```
UINT GetWriteWatch(  
    [in]     DWORD      dwFlags,  
    [in]     PVOID      lpBaseAddress,  
    [in]     SIZE_T     dwRegionSize,  
    [out]    PVOID      *lpAddresses,  
    [in, out] ULONG_PTR *lpdwCount,  
    [out]    LPDWORD     lpdwGranularity  
);
```

Parameters

[in] dwFlags

Indicates whether the function resets the write-tracking state.

To reset the write-tracking state, set this parameter to **WRITE_WATCH_FLAG_RESET**. If this parameter is 0 (zero), **GetWriteWatch** does not reset the write-tracking state. For more information, see the Remarks section of this topic.

[in] lpBaseAddress

The base address of the memory region for which to retrieve write-tracking information.

This address must be in a memory region that is allocated by the [VirtualAlloc](#) function using **MEM_WRITE_WATCH**.

[in] dwRegionSize

The size of the memory region for which to retrieve write-tracking information, in bytes.

[out] lpAddresses

A pointer to a buffer that receives an array of page addresses in the memory region.

The addresses indicate the pages that have been written to since the region has been allocated or the write-tracking state has been reset.

[in, out] `lpdwCount`

On input, this variable indicates the size of the *lpAddresses* array, in array elements.

On output, the variable receives the number of page addresses that are returned in the array.

[out] `lpdwGranularity`

A pointer to a variable that receives the page size, in bytes.

Return value

If the function succeeds, the return value is 0 (zero).

If the function fails, the return value is a nonzero value.

Remarks

When you call the [VirtualAlloc](#) function to reserve or commit memory, you can specify **MEM_WRITE_WATCH**. This value causes the system to keep track of the pages that are written to in the committed memory region. You can call the **GetWriteWatch** function to retrieve the addresses of the pages that have been written to since the region has been allocated or the write-tracking state has been reset.

To reset the write-tracking state, set the **WRITE_WATCH_FLAG_RESET** value in the *dwFlags* parameter. Alternatively, you can call the [ResetWriteWatch](#) function to reset the write-tracking state. However, if you use **ResetWriteWatch**, you must ensure that no threads write to the region during the interval between the **GetWriteWatch** and **ResetWriteWatch** calls. Otherwise, there may be written pages that you do not detect.

The **GetWriteWatch** function can be useful to profilers, debugging tools, or garbage collectors.

Requirements

[Expand table](#)

Requirement	Value
Minimum supported client	Windows XP [desktop apps UWP apps]

Requirement	Value
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	memoryapi.h (include Windows.h, Memoryapi.h)
Library	oncore.lib
DLL	Kernel32.dll

See also

[Memory Management Functions](#)

[ResetWriteWatch](#)

[VirtualAlloc](#)

MapUserPhysicalPages function (memoryapi.h)

Article 07/27/2022

Maps previously allocated physical memory pages at a specified address in an [Address Windowing Extensions](#) (AWE) region.

To perform batch mapping and unmapping of multiple regions, use the [MapUserPhysicalPagesScatter](#) function.

64-bit Windows on Itanium-based systems: Due to the difference in page sizes, [MapUserPhysicalPages](#) is not supported for 32-bit applications.

Syntax

C++

```
BOOL MapUserPhysicalPages(  
    [in] PVOID      VirtualAddress,  
    [in] ULONG_PTR  NumberOfPages,  
    [in] PULONG_PTR PageArray  
);
```

Parameters

[in] VirtualAddress

A pointer to the starting address of the region of memory to remap.

The value of *lpAddress* must be within the address range that the [VirtualAlloc](#) function returns when the [Address Windowing Extensions](#) (AWE) region is allocated.

[in] NumberOfPages

The size of the physical memory and virtual address space for which to establish translations, in pages.

The virtual address range is contiguous starting at *lpAddress*. The physical frames are specified by the *UserPfnArray*.

The total number of pages cannot extend from the starting address beyond the end of the range that is specified in [AllocateUserPhysicalPages](#).

[in] `PageArray`

A pointer to an array of physical page frame numbers.

These frames are mapped by the argument *lpAddress* on return from this function. The size of the memory that is allocated should be at least the *NumberOfPages* times the size of the data type `ULONG_PTR`.

Do not attempt to modify this buffer. It contains operating system data, and corruption could be catastrophic. The information in the buffer is not useful to an application.

If this parameter is **NULL**, the specified address range is unmapped. Also, the specified physical pages are not freed, and you must call [FreeUserPhysicalPages](#) to free them.

Return value

If the function succeeds, the return value is **TRUE**.

If the function fails, the return value is **FALSE** and no mapping is done—partial or otherwise. To get extended error information, call [GetLastError](#).

Remarks

The physical pages are unmapped but they are not freed. You must call [FreeUserPhysicalPages](#) to free the physical pages.

Any number of physical memory pages can be specified, but the memory must not extend outside the virtual address space that [VirtualAlloc](#) allocates. Any existing address maps are automatically overwritten with the new translations, and the old translations are unmapped.

You cannot map physical memory pages outside the range that is specified in [AllocateUserPhysicalPages](#). You can map multiple regions simultaneously, but they cannot overlap.

Physical pages can be located at any physical address, but do not make assumptions about the contiguity of the physical pages.

To unmap the current address range, specify **NULL** as the physical memory page array parameter. Any currently mapped pages are unmapped, but are not freed. You must call [FreeUserPhysicalPages](#) to free the physical pages.

In a multiprocessor environment, this function maintains hardware translation buffer coherence. On return from this function, all threads on all processors are guaranteed to see the correct mapping.

To compile an application that uses this function, define the `_WIN32_WINNT` macro as `0x0500` or later. For more information, see [Using the Windows Headers](#).

Examples

For an example, see [AWE Example](#).

Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	memoryapi.h (include Windows.h, Memoryapi.h)
Library	oncore.lib
DLL	Kernel32.dll

See also

[Address Windowing Extensions](#)

[AllocateUserPhysicalPages](#)

[FreeUserPhysicalPages](#)

[MapUserPhysicalPagesScatter](#)

[Memory Management Functions](#)

MapViewOfFile function (memoryapi.h)

Article10/30/2024

Maps a view of a file mapping into the address space of a calling process.

To specify a suggested base address for the view, use the [MapViewOfFileEx](#) function. However, this practice is not recommended.

Syntax

C++

```
LPVOID MapViewOfFile(  
    [in] HANDLE hFileMappingObject,  
    [in] DWORD dwDesiredAccess,  
    [in] DWORD dwFileOffsetHigh,  
    [in] DWORD dwFileOffsetLow,  
    [in] SIZE_T dwNumberOfBytesToMap  
);
```

Parameters

[in] hFileMappingObject

A handle to a file mapping object. The [CreateFileMapping](#) and [OpenFileMapping](#) functions return this handle.

[in] dwDesiredAccess

The type of access to a file mapping object, which determines the page protection of the pages. This parameter can be one of the following values, or a bitwise OR combination of multiple values where appropriate.

 Expand table

Value	Meaning
FILE_MAP_ALL_ACCESS	<p>A read/write view of the file is mapped. The file mapping object must have been created with PAGE_READWRITE or PAGE_EXECUTE_READWRITE protection.</p> <p>When used with the MapViewOfFile function, FILE_MAP_ALL_ACCESS is equivalent to FILE_MAP_WRITE.</p>

FILE_MAP_READ	<p>A read-only view of the file is mapped. An attempt to write to the file view results in an access violation.</p> <p>The file mapping object must have been created with PAGE_READONLY, PAGE_READWRITE, PAGE_EXECUTE_READ, or PAGE_EXECUTE_READWRITE protection.</p>
FILE_MAP_WRITE	<p>A read/write view of the file is mapped. The file mapping object must have been created with PAGE_READWRITE or PAGE_EXECUTE_READWRITE protection.</p> <p>When used with MapViewOfFile, (FILE_MAP_WRITE FILE_MAP_READ) and FILE_MAP_ALL_ACCESS are equivalent to FILE_MAP_WRITE.</p>

Using bitwise OR, you can combine the values above with these values.

[Expand table](#)

Value	Meaning
FILE_MAP_COPY	<p>A copy-on-write view of the file is mapped. The file mapping object must have been created with PAGE_READONLY, PAGE_EXECUTE_READ, PAGE_WRITECOPY, PAGE_EXECUTE_WRITECOPY, PAGE_READWRITE, or PAGE_EXECUTE_READWRITE protection.</p> <p>When a process writes to a copy-on-write page, the system copies the original page to a new page that is private to the process. The new page is backed by the paging file. The protection of the new page changes from copy-on-write to read/write.</p> <p>When copy-on-write access is specified, the system and process commit charge taken is for the entire view because the calling process can potentially write to every page in the view, making all pages private. The contents of the new page are never written back to the original file and are lost when the view is unmapped.</p>
FILE_MAP_EXECUTE	<p>An executable view of the file is mapped (mapped memory can be run as code). The file mapping object must have been created with PAGE_EXECUTE_READ, PAGE_EXECUTE_WRITECOPY, or PAGE_EXECUTE_READWRITE protection.</p> <p>Windows Server 2003 and Windows XP: This value is available starting with Windows XP with SP2 and Windows Server 2003 with SP1.</p>

FILE_MAP_LARGE_PAGES	<p>Starting with Windows 10, version 1703, this flag specifies that the view should be mapped using large page support. The size of the view must be a multiple of the size of a large page reported by the GetLargePageMinimum function, and the file-mapping object must have been created using the SEC_LARGE_PAGES option. If you provide a non-null value for <i>lpBaseAddress</i>, then the value must be a multiple of GetLargePageMinimum.</p> <p>Note: On OS versions before Windows 10, version 1703, the FILE_MAP_LARGE_PAGES flag has no effect. On these releases, the view is automatically mapped using large pages if the section was created with the SEC_LARGE_PAGES flag set.</p>
FILE_MAP_TARGETS_INVALID	<p>Sets all the locations in the mapped file as invalid targets for Control Flow Guard (CFG). This flag is similar to PAGE_TARGETS_INVALID. Use this flag in combination with the execute access right FILE_MAP_EXECUTE. 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.</p>

For file mapping objects created with the **SEC_IMAGE** attribute, the *dwDesiredAccess* parameter has no effect, and should be set to any valid value such as **FILE_MAP_READ**.

For more information about access to file mapping objects, see [File Mapping Security and Access Rights](#).

[in] dwFileOffsetHigh

A high-order **DWORD** of the file offset where the view begins.

[in] dwFileOffsetLow

A low-order **DWORD** of the file offset where the view is to begin. The combination of the high and low offsets must specify an offset within the file mapping. They must also match the virtual memory allocation granularity of the system. That is, the offset must be a multiple of the VirtualAlloc allocation granularity. To obtain the VirtualAlloc memory allocation granularity of the system, use the [GetSystemInfo](#) function, which fills in the members of a **SYSTEM_INFO** structure.

[in] dwNumberOfBytesToMap

The number of bytes of a file mapping to map to the view. All bytes must be within the maximum size specified by [CreateFileMapping](#). If this parameter is 0 (zero), the mapping

extends from the specified offset to the end of the file mapping.

Return value

If the function succeeds, the return value is the starting address of the mapped view.

If the function fails, the return value is **NULL**. To get extended error information, call [GetLastError](#).

Remarks

Mapping a file makes the specified portion of a file visible in the address space of the calling process.

For files that are larger than the address space, you can only map a small portion of the file data at one time. When the first view is complete, you can unmap it and map a new view.

To obtain the size of a view, use the [VirtualQuery](#) function.

Multiple views of a file (or a file mapping object and its mapped file) are *coherent* if they contain identical data at a specified time. This occurs if the file views are derived from any file mapping object that is backed by the same file. A process can duplicate a file mapping object handle into another process by using the [DuplicateHandle](#) function, or another process can open a file mapping object by name by using the [OpenFileMapping](#) function.

With one important exception, file views derived from any file mapping object that is backed by the same file are coherent or identical at a specific time. Coherency is guaranteed for views within a process and for views that are mapped by different processes.

The exception is related to remote files. Although **MapViewOfFile** works with remote files, it does not keep them coherent. For example, if two computers both map a file as writable, and both change the same page, each computer only sees its own writes to the page. When the data gets updated on the disk, it is not merged.

A mapped view of a file is not guaranteed to be coherent with a file that is being accessed by the [ReadFile](#) or [WriteFile](#) function.

Do not store pointers in the memory mapped file; store offsets from the base of the file mapping so that the mapping can be used at any address.

To guard against **EXCEPTION_IN_PAGE_ERROR** exceptions, use structured exception handling to protect any code that writes to or reads from a memory mapped view of a file other than the page file. For more information, see [Reading and Writing From a File View](#).

When modifying a file through a mapped view, the last modification timestamp may not be updated automatically. If required, the caller should use [SetFileTime](#) to set the timestamp.

If a file mapping object is backed by the paging file ([CreateFileMapping](#) is called with the *hFile* parameter set to **INVALID_HANDLE_VALUE**), the paging file must be large enough to hold the entire mapping. If it is not, **MapViewOfFile** fails. The initial contents of the pages in a file mapping object backed by the paging file are 0 (zero).

When a file mapping object that is backed by the paging file is created, the caller can specify whether **MapViewOfFile** should reserve and commit pages at the same time (**SEC_COMMIT**) or simply reserve pages (**SEC_RESERVE**). Mapping the file makes the entire mapped virtual address range unavailable to other allocations in the process. After a page from the reserved range is committed, it cannot be freed or decommitted by calling [VirtualFree](#). Reserved and committed pages are released when the view is unmapped and the file mapping object is closed. For details, see the [UnmapViewOfFile](#) and [CloseHandle](#) functions.

To have a file with executable permissions, an application must call [CreateFileMapping](#) with either **PAGE_EXECUTE_READWRITE** or **PAGE_EXECUTE_READ**, and then call **MapViewOfFile** with **FILE_MAP_EXECUTE | FILE_MAP_WRITE** or **FILE_MAP_EXECUTE | FILE_MAP_READ**.

In Windows Server 2012, this function is supported by the following technologies.

[Expand table](#)

Technology	Supported
Server Message Block (SMB) 3.0 protocol	Yes
SMB 3.0 Transparent Failover (TFO)	Yes
SMB 3.0 with Scale-out File Shares (SO)	Yes
Cluster Shared Volume File System (CsvFS)	Yes
Resilient File System (ReFS)	Yes

When CsvFs is paused this call might fail with an error indicating that there is a lock conflict.

Examples

For an example, see [Creating Named Shared Memory](#).

Requirements

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	memoryapi.h (include Windows.h, Memoryapi.h)
Library	onecore.lib
DLL	Kernel32.dll

See also

[CreateFileMapping](#)

[Creating a File View](#)

[DuplicateHandle](#)

[GetSystemInfo](#)

[MapViewOfFileEx](#)

[Memory Management Functions](#)

[OpenFileMapping](#)

[SYSTEM_INFO](#)

[UnmapViewOfFile](#)

MapViewOfFile2 function (memoryapi.h)

Article 02/22/2024

Maps a view of a file or a pagefile-backed section into the address space of the specified process.

Syntax

C++

```
PVOID MapViewOfFile2(  
    [in]          HANDLE    FileMappingHandle,  
    [in]          HANDLE    ProcessHandle,  
    [in]          ULONG64   Offset,  
    [in, optional] PVOID    BaseAddress,  
    [in]          SIZE_T    ViewSize,  
    [in]          ULONG     AllocationType,  
    [in]          ULONG     PageProtection  
);
```

Parameters

[in] FileMappingHandle

A **HANDLE** to a section that is to be mapped into the address space of the specified process.

[in] ProcessHandle

A **HANDLE** to a process into which the section will be mapped. The handle must have the **PROCESS_VM_OPERATION** access mask.

[in] Offset

The offset from the beginning of the section. This must be 64k aligned.

[in, optional] BaseAddress

The desired base address of the view. The address is rounded down to the nearest 64k boundary. If this parameter is **NULL**, the system picks the base address.

[in] ViewSize

The number of bytes to map. A value of zero (0) specifies that the entire section is to be mapped.

[in] AllocationType

The type of allocation. This parameter can be zero (0) or one of the following constant values:

- **MEM_RESERVE** - Maps a reserved view.
- **MEM_LARGE_PAGES** - Maps a large page view. This flag specifies that the view should be mapped using [large page support](#). The size of the view must be a multiple of the size of a large page reported by the [GetLargePageMinimum](#) function, and the file-mapping object must have been created using the **SEC_LARGE_PAGES** option. If you provide a non-null value for the *BaseAddress* parameter, then the value must be a multiple of **GetLargePageMinimum**.

[in] PageProtection

The desired page protection.

For file-mapping objects created with the **SEC_IMAGE** attribute, the *PageProtection* parameter has no effect, and should be set to any valid value such as **PAGE_READONLY**.

Return value

Returns the base address of the mapped view, if successful. Otherwise, returns **NULL** and extended error status is available using [GetLastError](#).

Remarks

This function is implemented as an inline function in the header and can't be found in any export library or DLL. It's the same as calling [MapViewOfFileNuma2](#) with the last parameter set to `NUMA_NO_PREFERRED_NODE`.

Requirements

[Expand table](#)

Requirement	Value
Minimum supported client	Windows 10, version 1703 [desktop apps only]
Minimum supported server	Windows Server 2016 [desktop apps only]
Target Platform	Windows
Header	memoryapi.h (include Windows.h, Memoryapi.h)

Requirement	Value
Library	onecore.lib
DLL	Kernel32.dll

See also

[MapViewOfFile](#)

[MapViewOfFileNuma2](#)

MapViewOfFile3 function (memoryapi.h)

Article 11/20/2024

Maps a view of a file or a pagefile-backed section into the address space of the specified process.

Syntax

C++

```
PVOID MapViewOfFile3(  
    [in] HANDLE FileMapping,  
    [in] HANDLE Process,  
    [in, optional] PVOID BaseAddress,  
    [in] ULONG64 Offset,  
    [in] SIZE_T ViewSize,  
    [in] ULONG AllocationType,  
    [in] ULONG PageProtection,  
    [in, out, optional] MEM_EXTENDED_PARAMETER *ExtendedParameters,  
    [in] ULONG ParameterCount  
);
```

Parameters

[in] FileMapping

A **HANDLE** to a section that is to be mapped into the address space of the specified process.

[in] Process

A **HANDLE** to a process into which the section will be mapped.

[in, optional] BaseAddress

The desired base address of the view (the address is rounded down to the nearest 64k boundary).

If this parameter is **NULL**, the system picks the base address.

If *BaseAddress* is not **NULL**, then any provided [MEM_ADDRESS_REQUIREMENTS](#) must consist of all zeroes.

[in] Offset

The offset from the beginning of the section.

The offset must be 64k aligned or aligned to `GetLargePageMinimum` when `MEM_LARGE_PAGES` is used in `AllocationType`. Furthermore, the offset must be page-aligned to the underlying page size granted by `VirtualAlloc2` when `MEM_REPLACE_PLACEHOLDER` is used in `AllocationType`.


[in] `ViewSize`

The number of bytes to map. A value of zero (0) specifies that the entire section is to be mapped.

The size must always be a multiple of the page size.

[in] `AllocationType`

The type of memory allocation. This parameter can be zero (0) or one of the following values.

 Expand table

Value	Meaning
MEM_RESERVE 0x00002000	Maps a reserved view.
MEM_REPLACE_PLACEHOLDER 0x00004000	<p>Replaces a placeholder with a mapped view. Only data/pf-backed section views are supported (no images, physical memory, etc.). When you replace a placeholder, <i>BaseAddress</i> and <i>ViewSize</i> must exactly match those of the placeholder, and any provided MEM_ADDRESS_REQUIREMENTS structure must consist of all zeroes.</p> <p>After you replace a placeholder with a mapped view, to free that mapped view back to a placeholder, see the <i>UnmapFlags</i> parameter of UnmapViewOfFileEx and UnmapViewOfFile2.</p> <p>A placeholder is a type of reserved memory region.</p> <p>The 64k alignment requirements on <i>Offset</i> and <i>BaseAddress</i> do not apply when this flag is specified.</p>
MEM_LARGE_PAGES 0x20000000	Maps a large page view. This flag specifies that the view should be mapped using large page support . The size of the view must be a multiple of the size of a large page reported by the GetLargePageMinimum function, and the file-mapping object must have been created using the SEC_LARGE_PAGES option. If you provide a non-null value for the <i>BaseAddress</i> parameter, then the value must be a multiple of GetLargePageMinimum .

The 64k alignment requirements on *Offset* do not apply when this flag is specified.

[in] `PageProtection`

The desired page protection.

For file-mapping objects created with the **SEC_IMAGE** attribute, the *PageProtection* parameter has no effect, and should be set to any valid value such as **PAGE_READONLY**.

[in, out, optional] `ExtendedParameters`

An optional pointer to one or more extended parameters of type [MEM_EXTENDED_PARAMETER](#). Each of those extended parameter values can itself have a *Type* field of either **MemExtendedParameterAddressRequirements** or **MemExtendedParameterNumaNode**. If no **MemExtendedParameterNumaNode** extended parameter is provided, then the behavior is the same as for the [VirtualAlloc/MapViewOfFile](#) functions (that is, the preferred NUMA node for the physical pages is determined based on the ideal processor of the thread that first accesses the memory).

[in] `ParameterCount`

The number of extended parameters pointed to by *ExtendedParameters*.

Return value

Returns the base address of the mapped view, if successful. Otherwise, returns **NULL** and extended error status is available using [GetLastError](#).

Remarks

This API helps support high-performance games, and server applications, which have particular requirements around managing their virtual address space. For example, mapping memory on top of a previously reserved region; this is useful for implementing an automatically wrapping ring buffer. And allocating memory with specific alignment; for example, to enable your application to commit large/huge page-mapped regions on demand.

Using this function for new allocations, you can:


- specify a range of virtual address space and a power-of-2 alignment restriction
- specify an arbitrary number of extended parameters
- specify a preferred NUMA node for the physical memory as an extended parameter
- specify a placeholder operation (specifically, replacement).

To specify the NUMA node, see the *ExtendedParameters* parameter.

Examples

For a code example, see Scenario 1 in [VirtualAlloc2](#).

Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10, version 1803 [desktop apps only]
Minimum supported server	Windows Server 2016 [desktop apps only]
Target Platform	Windows
Header	memoryapi.h (include Windows.h)
Library	oncore.lib
DLL	Kernel32.dll

See also

[VirtualAlloc2](#)

[MapViewOfFile](#)

[MapViewOfFile2](#)

[MapViewOfFileNuma2](#)

MapViewOfFile3FromApp function (memoryapi.h)

Article 07/21/2023

Maps a view of a file mapping into the address space of a calling Windows Store app.

Using this function, you can: for new allocations, specify a range of virtual address space and a power-of-2 alignment restriction; specify an arbitrary number of extended parameters; specify a preferred NUMA node for the physical memory as an extended parameter; and specify a placeholder operation (specifically, replacement).

To specify the NUMA node, see the *ExtendedParameters* parameter.

Syntax

C++

```
PVOID MapViewOfFile3FromApp(  
    [in]          HANDLE          FileMapping,  
    [in]          HANDLE          Process,  
    [in, optional] PVOID          BaseAddress,  
    [in]          ULONG64         Offset,  
    [in]          SIZE_T          ViewSize,  
    [in]          ULONG           AllocationType,  
    [in]          ULONG           PageProtection,  
    [in, out, optional] MEM_EXTENDED_PARAMETER *ExtendedParameters,  
    [in]          ULONG           ParameterCount  
);
```

Parameters

[in] FileMapping

A **HANDLE** to a section that is to be mapped into the address space of the specified process.

[in] Process

A **HANDLE** to a process into which the section will be mapped.

[in, optional] BaseAddress

The desired base address of the view. The address is rounded down to the nearest 64k boundary.

If this parameter is `NULL`, the system picks the base address.

If *BaseAddress* is not `NULL`, then any provided [MEM_ADDRESS_REQUIREMENTS](#) structure must consist of all zeroes.

[in] Offset

The offset from the beginning of the section. This must be 64k aligned.

[in] ViewSize

The number of bytes to map. A value of zero (0) specifies that the entire section is to be mapped.

The size must always be a multiple of the page size.

[in] AllocationType

The type of memory allocation. This parameter can be zero (0) or one of the following values.

 Expand table

Value	Meaning
MEM_RESERVE 0x00002000	Maps a reserved view.
MEM_REPLACE_PLACEHOLDER 0x00004000	<p>Replaces a placeholder with a mapped view. Only data/pf-backed section views are supported (no images, physical memory, etc.). When you replace a placeholder, <i>BaseAddress</i> and <i>ViewSize</i> must exactly match those of the placeholder, and any provided MEM_ADDRESS_REQUIREMENTS structure must consist of all zeroes.</p> <p>After you replace a placeholder with a mapped view, to free that mapped view back to a placeholder, see the <i>UnmapFlags</i> parameter of UnmapViewOfFileEx and UnmapViewOfFile2.</p> <p>A placeholder is a type of reserved memory region.</p>
MEM_LARGE_PAGES 0x20000000	Maps a large page view. See large page support .

[in] PageProtection

The desired page protection.

For file-mapping objects created with the **SEC_IMAGE** attribute, the *PageProtection* parameter has no effect, and should be set to any valid value such as **PAGE_READONLY**.

[in, out, optional] **ExtendedParameters**

An optional pointer to one or more extended parameters of type **MEM_EXTENDED_PARAMETER**. Each of those extended parameter values can itself have a *Type* field of either **MemExtendedParameterAddressRequirements** or **MemExtendedParameterNumaNode**. If no **MemExtendedParameterNumaNode** extended parameter is provided, then the behavior is the same as for the **VirtualAlloc/MapViewOfFile** functions (that is, the preferred NUMA node for the physical pages is determined based on the ideal processor of the thread that first accesses the memory).

[in] **ParameterCount**

The number of extended parameters pointed to by *ExtendedParameters*.

Return value

Returns the base address of the mapped view, if successful. Otherwise, returns **NULL** and extended error status is available using **GetLastError**.

Remarks

This API helps support high-performance games, and server applications, which have particular requirements around managing their virtual address space. For example, mapping memory on top of a previously reserved region; this is useful for implementing an automatically wrapping ring buffer. And allocating memory with specific alignment; for example, to enable your application to commit large/huge page-mapped regions on demand.

With one important exception, file views derived from any file mapping object that is backed by the same file are coherent or identical at a specific time. Coherency is guaranteed for views within a process and for views that are mapped by different processes.

The exception is related to remote files. Although **MapViewOfFile3FromApp** works with remote files, it does not keep them coherent. For example, if two computers both map a file as writable, and both change the same page, each computer only sees its own writes to the page. When the data gets updated on the disk, it is not merged.

You can only successfully request executable protection if your app has the **codeGeneration** capability.

Examples

For a code example, see Scenario 1 in [Virtual2Alloc](#).

Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10 [desktop apps only]
Minimum supported server	Windows Server 2016 [desktop apps only]
Target Platform	Windows
Header	memoryapi.h (include Windows.h)
Library	WindowsApp.lib
DLL	Kernel32.dll

See also

[CreateFileMapping](#)

[Creating a File View](#)

[DuplicateHandle](#)

[GetSystemInfo](#)

[MapViewOfFile3](#)

[MapViewOfFileEx](#)

[Memory Management Functions](#)

[OpenFileMapping](#)

[SYSTEM_INFO](#)

[UnmapViewOfFile](#)

MapViewOfFileEx function (memoryapi.h)

Article10/30/2024

Maps a view of a file mapping into the address space of a calling process. A caller can optionally specify a suggested base memory address for the view.

To specify the NUMA node for the physical memory, see [MapViewOfFileExNuma](#).

Syntax

```
C++

LPVOID MapViewOfFileEx(
    [in]          HANDLE  hFileMappingObject,
    [in]          DWORD   dwDesiredAccess,
    [in]          DWORD   dwFileOffsetHigh,
    [in]          DWORD   dwFileOffsetLow,
    [in]          SIZE_T  dwNumberOfBytesToMap,
    [in, optional] LPVOID lpBaseAddress
);
```


Parameters

[in] hFileMappingObject

A handle to a file mapping object. The [CreateFileMapping](#) and [OpenFileMapping](#) functions return this handle.

[in] dwDesiredAccess

The type of access to a file mapping object, which determines the page protection of the pages. This parameter can be one of the following values, or a bitwise OR combination of multiple values where appropriate.

 Expand table

Value	Meaning
FILE_MAP_ALL_ACCESS	<p>A read/write view of the file is mapped. The file mapping object must have been created with PAGE_READWRITE or PAGE_EXECUTE_READWRITE protection.</p> <p>When used with the MapViewOfFileEx function, FILE_MAP_ALL_ACCESS is equivalent to FILE_MAP_WRITE.</p>

FILE_MAP_READ	<p>A read-only view of the file is mapped. An attempt to write to the file view results in an access violation.</p> <p>The file mapping object must have been created with PAGE_READONLY, PAGE_READWRITE, PAGE_EXECUTE_READ, or PAGE_EXECUTE_READWRITE protection.</p>
FILE_MAP_WRITE	<p>A read/write view of the file is mapped. The file mapping object must have been created with PAGE_READWRITE or PAGE_EXECUTE_READWRITE protection.</p> <p>When used with MapViewOfFileEx, (FILE_MAP_WRITE FILE_MAP_READ) and FILE_MAP_ALL_ACCESS are equivalent to FILE_MAP_WRITE.</p>

Using bitwise OR, you can combine the values above with these values.

 Expand table

Value	Meaning
FILE_MAP_COPY	<p>A copy-on-write view of the file is mapped. The file mapping object must have been created with PAGE_READONLY, PAGE_EXECUTE_READ, PAGE_WRITECOPY, PAGE_EXECUTE_WRITECOPY, PAGE_READWRITE, or PAGE_EXECUTE_READWRITE protection.</p> <p>When a process writes to a copy-on-write page, the system copies the original page to a new page that is private to the process. The new page is backed by the paging file. The protection of the new page changes from copy-on-write to read/write.</p> <p>When copy-on-write access is specified, the system and process commit charge taken is for the entire view because the calling process can potentially write to every page in the view, making all pages private. The contents of the new page are never written back to the original file and are lost when the view is unmapped.</p>
FILE_MAP_LARGE_PAGES	<p>Starting with Windows 10, version 1703, this flag specifies that the view should be mapped using large page support. The size of the view must be a multiple of the size of a large page reported by the GetLargePageMinimum function, and the file-mapping object must have been created using the SEC_LARGE_PAGES option. If you provide a non-null value for <i>lpBaseAddress</i>, then the value must be a multiple of GetLargePageMinimum.</p>

FILE_MAP_EXECUTE	<p>An executable view of the file is mapped (mapped memory can be run as code). The file mapping object must have been created with PAGE_EXECUTE_READ, PAGE_EXECUTE_WRITECOPY, or PAGE_EXECUTE_READWRITE protection.</p> <p>Windows Server 2003 and Windows XP: This value is available starting with Windows XP with SP2 and Windows Server 2003 with SP1.</p>
FILE_MAP_TARGETS_INVALID	<p>Sets all the locations in the mapped file as invalid targets for Control Flow Guard (CFG). This flag is similar to PAGE_TARGETS_INVALID. Use this flag in combination with the execute access right FILE_MAP_EXECUTE. 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.</p>

For file-mapping objects created with the **SEC_IMAGE** attribute, the *dwDesiredAccess* parameter has no effect, and should be set to any valid value such as **FILE_MAP_READ**.

For more information about access to file mapping objects, see [File Mapping Security and Access Rights](#).

[in] dwFileOffsetHigh

The high-order **DWORD** of the file offset where the view is to begin.

[in] dwFileOffsetLow

The low-order **DWORD** of the file offset where the view is to begin. The combination of the high and low offsets must specify an offset within the file mapping. They must also match the memory allocation granularity of the system. That is, the offset must be a multiple of the allocation granularity. To obtain the memory allocation granularity of the system, use the [GetSystemInfo](#) function, which fills in the members of a **SYSTEM_INFO** structure.

[in] dwNumberOfBytesToMap

The number of bytes of a file mapping to map to a view. All bytes must be within the maximum size specified by [CreateFileMapping](#). If this parameter is 0 (zero), the mapping extends from the specified offset to the end of the file mapping.

[in, optional] lpBaseAddress

A pointer to the memory address in the calling process address space where mapping begins. This must be a multiple of the system's memory allocation granularity, or the function fails. To determine the memory allocation granularity of the system, use the [GetSystemInfo](#) function. If there is not enough address space at the specified address, the function fails.

If *lpBaseAddress* is **NULL**, the operating system chooses the mapping address. In this scenario, the function is equivalent to the [MapViewOfFile](#) function.

While it is possible to specify an address that is safe now (not used by the operating system), there is no guarantee that the address will remain safe over time. Therefore, it is better to let the operating system choose the address. In this case, you would not store pointers in the memory mapped file, you would store offsets from the base of the file mapping so that the mapping can be used at any address.

Return value

If the function succeeds, the return value is the starting address of the mapped view.

If the function fails, the return value is **NULL**. To get extended error information, call [GetLastError](#).

Remarks

Mapping a file makes the specified portion of the file visible in the address space of the calling process.

For files that are larger than the address space, you can only map a small portion of the file data at one time. When the first view is complete, then you unmap it and map a new view.

To obtain the size of a view, use the [VirtualQueryEx](#) function.

The initial contents of the pages in a file mapping object backed by the page file are 0 (zero).

Typically, the suggested address is used to specify that a file should be mapped at the same address in multiple processes. This requires the region of address space to be available in all involved processes. No other memory allocation can take place in the region that is used for mapping, including the use of the [VirtualAlloc](#) or [VirtualAllocEx](#) function to reserve memory.

If the *lpBaseAddress* parameter specifies a base offset, the function succeeds if the specified memory region is not already in use by the calling process. The system does not ensure that the same memory region is available for the memory mapped file in other 32-bit processes.

Multiple views of a file (or a file mapping object and its mapped file) are *coherent* if they contain identical data at a specified time. This occurs if the file views are derived from the same file mapping object. A process can duplicate a file mapping object handle into another process by using the [DuplicateHandle](#) function, or another process can open a file mapping object by name by using the [OpenFileMapping](#) function.

With one important exception, file views derived from any file mapping object that is backed by the same file are coherent or identical at a specific time. Coherency is guaranteed for views within a process and for views that are mapped by different processes.

The exception is related to remote files. Although **MapViewOfFileEx** works with remote files, it does not keep them coherent. For example, if two computers both map a file as writable, and both change the same page, each computer only sees its own writes to the page. When the data gets updated on the disk, it is not merged.


A mapped view of a file is not guaranteed to be coherent with a file being accessed by the [ReadFile](#) or [WriteFile](#) function.

To guard against **EXCEPTION_IN_PAGE_ERROR** exceptions, use structured exception handling to protect any code that writes to or reads from a memory mapped view of a file other than the page file. For more information, see [Reading and Writing From a File View](#).

When modifying a file through a mapped view, the last modification timestamp may not be updated automatically. If required, the caller should use [SetFileTime](#) to set the timestamp.


To have a file with executable permissions, an application must call [CreateFileMapping](#) with either **PAGE_EXECUTE_READWRITE** or **PAGE_EXECUTE_READ**, and then call **MapViewOfFileEx** with **FILE_MAP_EXECUTE | FILE_MAP_WRITE** or **FILE_MAP_EXECUTE | FILE_MAP_READ**.

In Windows Server 2012, this function is supported by the following technologies.

 [Expand table](#)

Technology	Supported
Server Message Block (SMB) 3.0 protocol	Yes
SMB 3.0 Transparent Failover (TFO)	Yes
SMB 3.0 with Scale-out File Shares (SO)	Yes
Cluster Shared Volume File System (CsvFS)	Yes
Resilient File System (ReFS)	Yes

Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	memoryapi.h (include Windows.h, Memoryapi.h)
Library	oncore.lib
DLL	Kernel32.dll

See also

[CreateFileMapping](#)

[Creating a File View](#)

[DuplicateHandle](#)

[File Mapping Functions](#)

[GetSystemInfo](#)

[MapViewOfFileExNuma](#)

[OpenFileMapping](#)

[ReadFile](#)

[SYSTEM_INFO](#)

[UnmapViewOfFile](#)

[VirtualAlloc](#)

[WriteFile](#)

MapViewOfFileFromApp function (memoryapi.h)

Article10/30/2024

Maps a view of a file mapping into the address space of a calling Windows Store app.

Syntax

```
C++

PVOID MapViewOfFileFromApp(
    [in] HANDLE    hFileMappingObject,
    [in] ULONG     DesiredAccess,
    [in] ULONG64   FileOffset,
    [in] SIZE_T    NumberOfBytesToMap
);
```


Parameters

[in] hFileMappingObject

A handle to a file mapping object. The [CreateFileMappingFromApp](#) function returns this handle.

[in] DesiredAccess


The type of access to a file mapping object, which determines the page protection of the pages. This parameter can be one of the following values, or a bitwise OR combination of multiple values where appropriate.

 Expand table

Value	Meaning
FILE_MAP_ALL_ACCESS	<p>A read/write view of the file is mapped. The file mapping object must have been created with PAGE_READWRITE protection.</p> <p>When used with MapViewOfFileFromApp, FILE_MAP_ALL_ACCESS is equivalent to FILE_MAP_WRITE.</p>
FILE_MAP_READ	<p>A read-only view of the file is mapped. An attempt to write to the file view results in an access violation.</p>

	The file mapping object must have been created with PAGE_READONLY , PAGE_READWRITE , PAGE_EXECUTE_READ , or PAGE_EXECUTE_READWRITE protection.
FILE_MAP_WRITE	<p>A read/write view of the file is mapped. The file mapping object must have been created with PAGE_READWRITE protection.</p> <p>When used with MapViewOfFileFromApp, (FILE_MAP_WRITE FILE_MAP_READ) is equivalent to FILE_MAP_WRITE.</p>

Using bitwise OR, you can combine the values above with these values.

 Expand table

Value	Meaning
FILE_MAP_COPY	<p>A copy-on-write view of the file is mapped. The file mapping object must have been created with PAGE_READONLY, PAGE_EXECUTE_READ, PAGE_WRITECOPY, or PAGE_READWRITE protection.</p> <p>When a process writes to a copy-on-write page, the system copies the original page to a new page that is private to the process. The new page is backed by the paging file. The protection of the new page changes from copy-on-write to read/write.</p> <p>When copy-on-write access is specified, the system and process commit charge taken is for the entire view because the calling process can potentially write to every page in the view, making all pages private. The contents of the new page are never written back to the original file and are lost when the view is unmapped.</p>
FILE_MAP_LARGE_PAGES	Starting with Windows 10, version 1703, this flag specifies that the view should be mapped using large page support . The size of the view must be a multiple of the size of a large page reported by the GetLargePageMinimum function, and the file-mapping object must have been created using the SEC_LARGE_PAGES option. If you provide a non-null value for <i>lpBaseAddress</i> , then the value must be a multiple of GetLargePageMinimum .
FILE_MAP_TARGETS_INVALID	Sets all the locations in the mapped file as invalid targets for Control Flow Guard (CFG). This flag is similar to PAGE_TARGETS_INVALID . Use this flag in combination with the execute access right FILE_MAP_EXECUTE . 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.

For file-mapping objects created with the **SEC_IMAGE** attribute, the *dwDesiredAccess* parameter has no effect, and should be set to any valid value such as **FILE_MAP_READ**.

For more information about access to file mapping objects, see [File Mapping Security and Access Rights](#).

[in] FileOffset

The file offset where the view is to begin. The offset must specify an offset within the file mapping. They must also match the memory allocation granularity of the system. That is, the offset must be a multiple of the allocation granularity. To obtain the memory allocation granularity of the system, use the [GetSystemInfo](#) function, which fills in the members of a **SYSTEM_INFO** structure.

[in] NumberOfBytesToMap

The number of bytes of a file mapping to map to the view. All bytes must be within the maximum size specified by [CreateFileMappingFromApp](#). If this parameter is 0 (zero), the mapping extends from the specified offset to the end of the file mapping.

Return value

If the function succeeds, the return value is the starting address of the mapped view.

If the function fails, the return value is **NULL**. To get extended error information, call [GetLastError](#).


Remarks

With one important exception, file views derived from any file mapping object that is backed by the same file are coherent or identical at a specific time. Coherency is guaranteed for views within a process and for views that are mapped by different processes.

The exception is related to remote files. Although **MapViewOfFileFromApp** works with remote files, it does not keep them coherent. For example, if two computers both map a file as writable, and both change the same page, each computer only sees its own writes to the page. When the data gets updated on the disk, it is not merged.

You can only successfully request executable protection if your app has the **codeGeneration** capability.

Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 8 [desktop apps UWP apps]
Minimum supported server	Windows Server 2012 [desktop apps UWP apps]
Target Platform	Windows
Header	memoryapi.h (include Windows.h)
Library	onecore.lib
DLL	Kernel32.dll

See also

[CreateFileMapping](#)

[Creating a File View](#)

[DuplicateHandle](#)

[GetSystemInfo](#)

[MapViewOfFile](#)

[MapViewOfFileEx](#)

[Memory Management Functions](#)

[OpenFileMapping](#)

[SYSTEM_INFO](#)

[UnmapViewOfFile](#)

MapViewOfFileNuma2 function (memoryapi.h)

Article02/22/2024

Maps a view of a file or a pagefile-backed section into the address space of the specified process.

Syntax

C++

```
PVOID MapViewOfFileNuma2(  
    [in]          HANDLE   FileMappingHandle,  
    [in]          HANDLE   ProcessHandle,  
    [in]          ULONG64  Offset,  
    [in, optional] PVOID   BaseAddress,  
    [in]          SIZE_T   ViewSize,  
    [in]          ULONG    AllocationType,  
    [in]          ULONG    PageProtection,  
    [in]          ULONG    PreferredNode  
);
```

Parameters

[in] FileMappingHandle

A **HANDLE** to a section that is to be mapped into the address space of the specified process.

[in] ProcessHandle

A **HANDLE** to a process into which the section will be mapped.

[in] Offset

The offset from the beginning of the section. This must be 64k aligned.

[in, optional] BaseAddress

The desired base address of the view. The address is rounded down to the nearest 64k boundary. If this parameter is **NULL**, the system picks the base address.

[in] ViewSize

The number of bytes to map. A value of zero (0) specifies that the entire section is to be mapped.

[in] `AllocationType`

The type of allocation. This parameter can be zero (0) or one of the following constant values:

- **MEM_RESERVE** - Maps a reserved view
- **MEM_LARGE_PAGES** - Maps a large page view

[in] `PageProtection`

The desired page protection.

For file-mapping objects created with the **SEC_IMAGE** attribute, the *PageProtection* parameter has no effect, and should be set to any valid value such as **PAGE_READONLY**.

[in] `PreferredNode`

The preferred NUMA node for this memory.

Return value

Returns the base address of the mapped view, if successful. Otherwise, returns **NULL** and extended error status is available using [GetLastError](#).

Requirements

 [Expand table](#)

Requirement	Value
Minimum supported client	Windows 10, version 1703 [desktop apps only]
Minimum supported server	Windows Server 2016 [desktop apps only]
Target Platform	Windows
Header	memoryapi.h (include Windows.h, Memoryapi.h)
Library	Onecore.lib; Onecoreuap.lib
DLL	Api-ms-win-core-memory-l1-1-5.dll

See also

MapViewOfFile

MapViewOfFileNuma

OfferVirtualMemory function (memoryapi.h)

Article 05/14/2022

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.

The specified pages will be marked as inaccessible, removed from the process working set, and will not be written to the paging file.

To later reclaim offered pages, call [ReclaimVirtualMemory](#).

Syntax

C++

```
DWORD OfferVirtualMemory(  
    [in] PVOID          VirtualAddress,  
    [in] SIZE_T         Size,  
    [in] OFFER_PRIORITY Priority  
);
```

Parameters

[in] VirtualAddress

Page-aligned starting address of the memory to offer.

[in] Size

Size, in bytes, of the memory region to offer. *Size* must be an integer multiple of the system page size.

[in] Priority

Priority indicates how important the offered memory is to the application. A higher priority increases the probability that the offered memory can be reclaimed intact when calling [ReclaimVirtualMemory](#). The system typically discards lower priority memory before discarding higher priority memory. *Priority* must be one of the following values.

 Expand table

Value	Meaning
VMOfferPriorityVeryLow 0x00000001	The offered memory is very low priority, and should be the first discarded.
VMOfferPriorityLow 0x00000002	The offered memory is low priority.
VMOfferPriorityBelowNormal 0x00000003	The offered memory is below normal priority.
VMOfferPriorityNormal 0x00000004	The offered memory is of normal priority to the application, and should be the last discarded.

Return value

ERROR_SUCCESS if successful; a [System Error Code](#) otherwise.

Remarks

To reclaim offered pages, call [ReclaimVirtualMemory](#). The data in reclaimed pages may have been discarded, in which case the contents of the memory region is undefined and must be rewritten by the application.

Do not call **OfferVirtualMemory** to offer virtual memory that is locked. Doing so will unlock the specified range of pages.

Note that offering and reclaiming virtual memory is similar to using the MEM_RESET and MEM_RESET_UNDO memory allocation flags, except that **OfferVirtualMemory** removes the memory from the process working set and restricts access to the offered pages until they are reclaimed.

Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 8.1 Update [desktop apps UWP apps]
Minimum supported server	Windows Server 2012 R2 Update [desktop apps UWP apps]
Target Platform	Windows

Requirement	Value
Header	memoryapi.h (include Windows.h, Memoryapi.h)
Library	onecore.lib
DLL	Kernel32.dll

See also

- [DiscardVirtualMemory](#)
- [Memory Management Functions](#)
- [ReclaimVirtualMemory](#)
- [Virtual Memory Functions](#)
- [VirtualAlloc](#)
- [VirtualFree](#)
- [VirtualLock](#)
- [VirtualQuery](#)

OpenFileMappingFromApp function (memoryapi.h)

Article 07/27/2022

Opens a named file mapping object.

Syntax

C++

```
HANDLE OpenFileMappingFromApp(  
    [in] ULONG    DesiredAccess,  
    [in] BOOL     InheritHandle,  
    [in] PCWSTR   Name  
);
```

Parameters

[in] DesiredAccess

The access to the file mapping object. This access is checked against any security descriptor on the target file mapping object. For a list of values, see [File Mapping Security and Access Rights](#). You can only open the file mapping object for **FILE_MAP_EXECUTE** access if your app has the **codeGeneration** capability.

[in] InheritHandle

If this parameter is **TRUE**, a process created by the [CreateProcess](#) function can inherit the handle; otherwise, the handle cannot be inherited.

[in] Name

The name of the file mapping object to be opened. If there is an open handle to a file mapping object by this name and the security descriptor on the mapping object does not conflict with the *DesiredAccess* parameter, the open operation succeeds. The name can have a "Global" or "Local" prefix to explicitly open an object in the global or session namespace. The remainder of the name can contain any character except the backslash character (\). For more information, see [Kernel Object Namespaces](#). Fast user switching is implemented using Terminal Services sessions. The first user to log on uses session 0, the next user to log on uses session 1, and so on. Kernel object names must follow the guidelines outlined for Terminal Services so that applications can support multiple users.

Return value

If the function succeeds, the return value is an open handle to the specified file mapping object.

If the function fails, the return value is **NULL**. To get extended error information, call [GetLastError](#).

Remarks

You can call **OpenFileMappingFromApp** from Windows Store apps with just-in-time (JIT) capabilities to use JIT functionality. The app must include the **codeGeneration** capability in the app manifest file to use JIT capabilities. **OpenFileMappingFromApp** lets Windows Store apps use the [MemoryMappedFile](#) class in the .NET Framework.

The handle that **OpenFileMappingFromApp** returns can be used with any function that requires a handle to a file mapping object.

When modifying a file through a mapped view, the last modification timestamp may not be updated automatically. If required, the caller should use [SetFileTime](#) to set the timestamp.

When it is no longer needed, the caller should call release the handle returned by **OpenFileMappingFromApp** with a call to [CloseHandle](#).

Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10 [desktop apps UWP apps]
Minimum supported server	Windows Server 2016 [desktop apps UWP apps]
Target Platform	Windows
Header	memoryapi.h (include Windows.h)
Library	WindowsApp.lib
DLL	Kernel32.dll

See also

CreateFileMapping

File Mapping Functions

Memory Management Functions

OpenFileMapping

Sharing Files and Memory

OpenFileMappingW function (memoryapi.h)

Article 07/27/2022

Opens a named file mapping object.

Syntax

C++

```
HANDLE OpenFileMappingW(  
    [in] DWORD    dwDesiredAccess,  
    [in] BOOL     bInheritHandle,  
    [in] LPCWSTR  lpName  
);
```

Parameters

[in] dwDesiredAccess

The access to the file mapping object. This access is checked against any security descriptor on the target file mapping object. For a list of values, see [File Mapping Security and Access Rights](#).

[in] bInheritHandle

If this parameter is **TRUE**, a process created by the [CreateProcess](#) function can inherit the handle; otherwise, the handle cannot be inherited.

[in] lpName

The name of the file mapping object to be opened. If there is an open handle to a file mapping object by this name and the security descriptor on the mapping object does not conflict with the *dwDesiredAccess* parameter, the open operation succeeds. The name can have a "Global" or "Local" prefix to explicitly open an object in the global or session namespace. The remainder of the name can contain any character except the backslash character (\). For more information, see [Kernel Object Namespaces](#). Fast user switching is implemented using Terminal Services sessions. The first user to log on uses session 0, the next user to log on uses session 1, and so on. Kernel object names must

follow the guidelines outlined for Terminal Services so that applications can support multiple users.

Return value

If the function succeeds, the return value is an open handle to the specified file mapping object.

If the function fails, the return value is **NULL**. To get extended error information, call [GetLastError](#).

Remarks

The handle that **OpenFileMapping** returns can be used with any function that requires a handle to a file mapping object.

When modifying a file through a mapped view, the last modification timestamp may not be updated automatically. If required, the caller should use [SetFileTime](#) to set the timestamp.

When it is no longer needed, the caller should call release the handle returned by **OpenFileMapping** with a call to [CloseHandle](#).

In Windows Server 2012, this function is supported by the following technologies.

Technology	Supported
Server Message Block (SMB) 3.0 protocol	Yes
SMB 3.0 Transparent Failover (TFO)	Yes
SMB 3.0 with Scale-out File Shares (SO)	Yes
Cluster Shared Volume File System (CsvFS)	Yes
Resilient File System (ReFS)	Yes

Examples

For an example, see [Creating Named Shared Memory](#).

Requirements

Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	memoryapi.h (include Windows.h, Memoryapi.h)
Library	oncore.lib
DLL	Kernel32.dll

See also

[CreateFileMapping](#)

[File Mapping Functions](#)

[Memory Management Functions](#)

[Sharing Files and Memory](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

PrefetchVirtualMemory function (memoryapi.h)

Article 05/14/2022

Provides an efficient mechanism to bring into memory potentially discontinuous virtual address ranges in a process address space.

Syntax

C++

```
BOOL PrefetchVirtualMemory(  
    [in] HANDLE                hProcess,  
    [in] ULONG_PTR            NumberOfEntries,  
    [in] PWIN32_MEMORY_RANGE_ENTRY VirtualAddresses,  
    [in] ULONG                 Flags  
);
```

Parameters

[in] `hProcess`

Handle to the process whose virtual address ranges are to be prefetched. Use the [GetCurrentProcess](#) function to use the current process.

[in] `NumberOfEntries`

Number of entries in the array pointed to by the *VirtualAddresses* parameter.

[in] `VirtualAddresses`

Pointer to an array of [WIN32_MEMORY_RANGE_ENTRY](#) structures which each specify a virtual address range to be prefetched. The virtual address ranges may cover any part of the process address space accessible by the target process.

[in] `Flags`

Reserved. Must be 0.

Return value

If the function succeeds, the return value is a nonzero value.

If the function fails, the return value is 0 (zero). To get extended error information, call [GetLastError](#).

Remarks

The **PrefetchVirtualMemory** function is targeted at applications that know with reasonable confidence the set of addresses they will be accessing. If it's likely that these addresses are no longer resident in memory (i.e. they have been paged out to disk), calling the **PrefetchVirtualMemory** function on those address ranges before access will reduce the overall latency because the API will efficiently bring in those address ranges from disk using large, concurrent I/O requests where possible.


The **PrefetchVirtualMemory** function allows applications to make efficient use of disk hardware by issuing large, concurrent I/Os where possible when the application provides a list of process address ranges that are going to be accessed. Even for a single address range (e.g. a file mapping), the **PrefetchVirtualMemory** function can provide performance improvements by issuing a single large I/O rather than the many smaller I/Os that would be issued via page faulting.

The **PrefetchVirtualMemory** function is purely a performance optimization: prefetching is not necessary for accessing the target address ranges. The prefetched memory is not added to the target process' working set; it is cached in physical memory. When the prefetched address ranges are accessed by the target process, they will be added to the working set.

Since the **PrefetchVirtualMemory** function can never be necessary for correct operation of applications, it is treated as a strong hint by the system and is subject to usual physical memory constraints where it can completely or partially fail under low-memory conditions. It can also create memory pressure if called with large address ranges, so applications should only prefetch address ranges they will actually use.

To compile an application that calls this function, define `_WIN32_WINNT` as `_WIN32_WINNT_WIN8` or higher. For more information, see [Using the Windows Headers](#).

Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]

Requirement	Value
Target Platform	Windows
Header	memoryapi.h (include Windows.h, Memoryapi.h)
Library	onecore.lib
DLL	Kernel32.dll

See also

[Memory Management Functions](#)

[WIN32_MEMORY_RANGE_ENTRY](#)

QueryMemoryResourceNotification function (memoryapi.h)

Article 02/22/2024

Retrieves the state of the specified memory resource object.

Syntax

C++

```
BOOL QueryMemoryResourceNotification(  
    [in] HANDLE ResourceNotificationHandle,  
    [out] PBOOL ResourceState  
);
```

Parameters

[in] ResourceNotificationHandle

A handle to a memory resource notification object. The [CreateMemoryResourceNotification](#) function returns this handle.

[out] ResourceState

The memory pointed to by this parameter receives the state of the memory resource notification object. The value of this parameter is set to **TRUE** if the specified memory condition exists, and **FALSE** if the specified memory condition does not exist.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. For more error information, call [GetLastError](#).

Remarks

Unlike the [wait functions](#), **QueryMemoryResourceNotification** does not block the calling thread. Therefore, it is an efficient way to check the state of physical memory before proceeding with an operation.

To compile an application that uses this function, define the `_WIN32_WINNT` macro as 0x0501 or later. For more information, see [Using the Windows Headers](#).

Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	memoryapi.h (include Windows.h, Memoryapi.h)
Library	oncore.lib
DLL	Kernel32.dll

See also

[CreateMemoryResourceNotification](#)

[Memory Management Functions](#)

QueryVirtualMemoryInformation function (memoryapi.h)

Article 10/13/2021

The **QueryVirtualMemoryInformation** function returns information about a page or a set of pages within the virtual address space of the specified process.

Syntax

C++

```
BOOL QueryVirtualMemoryInformation(  
    [in] HANDLE Process,  
    [in] const VOID *VirtualAddress,  
    [in] WIN32_MEMORY_INFORMATION_CLASS MemoryInformationClass,  
    [out] PVOID MemoryInformation,  
    [in] SIZE_T MemoryInformationSize,  
    [out, optional] PSIZE_T ReturnSize  
);
```

Parameters

[in] Process

A handle for the process in whose context the pages to be queried reside.

[in] VirtualAddress

The address of the region of pages to be queried. This value is rounded down to the next host-page-address boundary.

[in] MemoryInformationClass

The memory information class about which to retrieve information. The only supported value is **MemoryRegionInfo**.

[out] MemoryInformation

A pointer to a buffer that receives the specified information.

If the *MemoryInformationClass* parameter has a value of **MemoryRegionInfo**, this parameter must point to a [WIN32_MEMORY_REGION_INFORMATION](#) structure.

[in] MemoryInformationSize

Specifies the length in bytes of the memory information buffer.

[out, optional] ReturnSize

An optional pointer which, if specified, receives the number of bytes placed in the memory information buffer.

Return value

Returns **TRUE** on success. Returns **FALSE** for failure. To get extended error information, call [GetLastError](#).

Remarks

If the *MemoryInformationClass* parameter has a value of **MemoryRegionInfo**, the *MemoryInformation* parameter must point to a [WIN32_MEMORY_REGION_INFORMATION](#) structure. The *VirtualAddress* parameter must point to an address within a valid memory allocation. If the *VirtualAddress* parameter points to an unallocated memory region, the function fails.

Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10, version 1607 [desktop apps only]
Minimum supported server	Windows Server 2016 [desktop apps only]
Target Platform	Windows
Header	memoryapi.h (include Windows.h)
Library	Onecore.lib
DLL	Api-ms-win-core-memory-l1-1-4.dll

See also

[GetLastError](#)

[MEMORY_BASIC_INFORMATION](#)

ReadProcessMemory function (memoryapi.h)

Article 05/14/2022

Syntax

C++

```
BOOL ReadProcessMemory(  
    [in] HANDLE hProcess,  
    [in] LPCVOID lpBaseAddress,  
    [out] LPVOID lpBuffer,  
    [in] SIZE_T nSize,  
    [out] SIZE_T *lpNumberOfBytesRead  
);
```

Parameters

[in] hProcess

A handle to the process with memory that is being read. The handle must have PROCESS_VM_READ access to the process.

[in] lpBaseAddress

A pointer to the base address in the specified process from which to read. Before any data transfer occurs, the system verifies that all data in the base address and memory of the specified size is accessible for read access, and if it is not accessible the function fails.

[out] lpBuffer

A pointer to a buffer that receives the contents from the address space of the specified process.

[in] nSize

The number of bytes to be read from the specified process.

[out] lpNumberOfBytesRead

A pointer to a variable that receives the number of bytes transferred into the specified buffer. If *lpNumberOfBytesRead* is **NULL**, the parameter is ignored.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is 0 (zero). To get extended error information, call [GetLastError](#).


The function fails if the requested read operation crosses into an area of the process that is inaccessible.

Remarks

ReadProcessMemory copies the data in the specified address range from the address space of the specified process into the specified buffer of the current process. Any process that has a handle with PROCESS_VM_READ access can call the function.

The entire area to be read must be accessible, and if it is not accessible, the function fails.

Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	memoryapi.h (include Windows.h)
Library	oncore.lib
DLL	Kernel32.dll

See also

[Debugging Functions](#), [OpenProcess](#), [Process Functions for Debugging](#), [VirtualAllocEx](#), [WriteProcessMemory](#)

ReclaimVirtualMemory function (memoryapi.h)

Article 02/22/2024

Reclaims a range of memory pages that were offered to the system with [OfferVirtualMemory](#).

If the offered memory has been discarded, the contents of the memory region is undefined and must be rewritten by the application. If the offered memory has not been discarded, it is reclaimed intact.

Syntax

C++

```
DWORD ReclaimVirtualMemory(  
    [in] void const *VirtualAddress,  
    [in] SIZE_T      Size  
);
```

Parameters

[in] VirtualAddress

Page-aligned starting address of the memory to reclaim.

[in] Size

Size, in bytes, of the memory region to reclaim. Size must be an integer multiple of the system page size.

Return value

Returns ERROR_SUCCESS if successful and the memory was reclaimed intact.

Returns ERROR_BUSY if successful but the memory was discarded and must be rewritten by the application. In this case, the contents of the memory region is undefined.

Returns a [System Error Code](#) otherwise.

Remarks

Reclaimed memory pages can be used by the application, and will be written to the system paging file if paging occurs.

If the function returns `ERROR_SUCCESS`, the data in the reclaimed pages is valid. If the function returns `ERROR_BUSY`, the data in the reclaimed pages was discarded by the system and is no longer valid. For this reason, memory should only be offered to the system if the application does not need or can regenerate the data.

Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 8.1 Update [desktop apps UWP apps]
Minimum supported server	Windows Server 2012 R2 Update [desktop apps UWP apps]
Target Platform	Windows
Header	memoryapi.h (include Windows.h, Memoryapi.h)
Library	oncore.lib
DLL	Kernel32.dll

See also

[Memory Management Functions](#)

[OfferVirtualMemory](#)

[Virtual Memory Functions](#)

[VirtualAlloc](#)

[VirtualFree](#)

[VirtualLock](#)

[VirtualQuery](#)

RegisterBadMemoryNotification function (memoryapi.h)

Article02/22/2024

Registers a bad memory notification that is called when one or more bad memory pages are detected and the system cannot remove at least one of them (for example if the pages contains modified data that has not yet been written to the pagefile.)

Syntax

```
C++

PVOID RegisterBadMemoryNotification(
    [in] PBAD_MEMORY_CALLBACK_ROUTINE Callback
);
```

Parameters

[in] Callback

A pointer to the application-defined [BadMemoryCallbackRoutine](#) function to register.


Return value

Registration handle that represents the callback notification. Can be passed to the [UnregisterBadMemoryNotification](#) function when no longer needed.

Remarks

To compile an application that calls this function, define `_WIN32_WINNT` as `_WIN32_WINNT_WIN8` or higher. For more information, see [Using the Windows Headers](#).

Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 8 [desktop apps only]

Requirement	Value
Minimum supported server	Windows Server 2012 [desktop apps only]
Target Platform	Windows
Header	memoryapi.h (include Windows.h, Memoryapi.h)
Library	oncore.lib
DLL	Kernel32.dll

ResetWriteWatch function (memoryapi.h)

Article 02/22/2024

Resets the write-tracking state for a region of virtual memory. Subsequent calls to the [GetWriteWatch](#) function only report pages that are written to since the reset operation.

64-bit Windows on Itanium-based systems: Due to the difference in page sizes, **ResetWriteWatch** is not supported for 32-bit applications.

Syntax

C++

```
UINT ResetWriteWatch(  
    [in] LPVOID lpBaseAddress,  
    [in] SIZE_T dwRegionSize  
);
```

Parameters

[in] lpBaseAddress

A pointer to the base address of the memory region for which to reset the write-tracking state.

This address must be in a memory region that is allocated by the [VirtualAlloc](#) function with **MEM_WRITE_WATCH**.

[in] dwRegionSize

The size of the memory region for which to reset the write-tracking information, in bytes.

Return value

If the function succeeds, the return value is 0 (zero).

If the function fails, the return value is a nonzero value.

Remarks


The **ResetWriteWatch** function can be useful to an application such as a garbage collector. The application calls the [GetWriteWatch](#) function to retrieve the list of written pages, and then

writes to those pages as part of its cleanup operation. Then the garbage collector calls **ResetWriteWatch** to remove the write-tracking records caused by the cleanup.

You can also reset the write-tracking state of a memory region by specifying **WRITE_WATCH_FLAG_RESET** when you call [GetWriteWatch](#).

If you use **ResetWriteWatch**, you must ensure that no threads write to the region during the interval between the [GetWriteWatch](#) and **ResetWriteWatch** calls. Otherwise, there may be written pages that you not detect.

Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	memoryapi.h (include Windows.h, Memoryapi.h)
Library	onecore.lib
DLL	Kernel32.dll

See also

[GetWriteWatch](#)

[Memory Management Functions](#)

[VirtualAlloc](#)

SetProcessValidCallTargets function (memoryapi.h)

Article 10/13/2021

Provides Control Flow Guard (CFG) with a list of valid indirect call targets and specifies whether they should be marked valid or not. The valid call target information is provided as a list of offsets relative to a virtual memory range (start and size of the range). The call targets specified should be 16-byte aligned and in ascending order.

Syntax

C++

```
BOOL SetProcessValidCallTargets(  
    [in]     HANDLE          hProcess,  
    [in]     PVOID          VirtualAddress,  
    [in]     SIZE_T          RegionSize,  
    [in]     ULONG           NumberOfOffsets,  
    [in, out] PCFG_CALL_TARGET_INFO OffsetInformation  
);
```

Parameters

[in] hProcess

The handle to the target process.

[in] VirtualAddress

The start of the virtual memory region whose call targets are being marked valid. The memory region must be allocated using one of the executable [memory protection constants](#).

[in] RegionSize

The size of the virtual memory region.

[in] NumberOfOffsets

The number of offsets relative to the virtual memory ranges.

[in, out] OffsetInformation

A list of offsets and flags relative to the virtual memory ranges.

Return value

TRUE if the operation was successful; otherwise, FALSE. To retrieve error values for this function, call [GetLastError](#).

Remarks

This function does not succeed if Control Flow Guard is not enabled for the target process. This can be checked using [GetProcessMitigationPolicy](#).

Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10 [desktop apps UWP apps]
Minimum supported server	Windows Server 2016 [desktop apps UWP apps]
Target Platform	Windows
Header	memoryapi.h (include Windows.h, Memoryapi.h)
Library	WindowsApp.lib
DLL	Kernelbase.dll

SetProcessWorkingSetSize function (memoryapi.h)

Article 12/09/2022

Sets the minimum and maximum working set sizes for the specified process.

Syntax

C++

```
BOOL SetProcessWorkingSetSize(  
    [in] HANDLE hProcess,  
    [in] SIZE_T dwMinimumWorkingSetSize,  
    [in] SIZE_T dwMaximumWorkingSetSize  
);
```

Parameters

[in] hProcess

A handle to the process whose working set sizes is to be set.

The handle must have the **PROCESS_SET_QUOTA** access right. For more information, see [Process Security and Access Rights](#).

[in] dwMinimumWorkingSetSize

The minimum working set size for the process, in bytes. The virtual memory manager attempts to keep at least this much memory resident in the process whenever the process is active.

This parameter must be greater than zero but less than or equal to the maximum working set size. The default size is 50 pages (for example, this is 204,800 bytes on systems with a 4K page size). If the value is greater than zero but less than 20 pages, the minimum value is set to 20 pages.

If both *dwMinimumWorkingSetSize* and *dwMaximumWorkingSetSize* have the value (**SIZE_T**)–1, the function removes as many pages as possible from the working set of the specified process.

[in] dwMaximumWorkingSetSize

The maximum working set size for the process, in bytes. The virtual memory manager attempts to keep no more than this much memory resident in the process whenever the process is active and available memory is low.

This parameter must be greater than or equal to 13 pages (for example, 53,248 on systems with a 4K page size), and less than the system-wide maximum (number of available pages minus 512 pages). The default size is 345 pages (for example, this is 1,413,120 bytes on systems with a 4K page size).

If both *dwMinimumWorkingSetSize* and *dwMaximumWorkingSetSize* have the value (**SIZE_T**)–1, the function removes as many pages as possible from the working set of the specified process.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. Call [GetLastError](#) to obtain extended error information.

Remarks

The working set of a process is the set of memory pages in the virtual address space of the process that are currently resident in physical memory. These pages are available for an application to use without triggering a page fault. For more information about page faults, see [Working Set](#). The minimum and maximum working set sizes affect the virtual memory paging behavior of a process.

The working set of the specified process can be emptied by specifying the value (**SIZE_T**)–1 for both the minimum and maximum working set sizes. This removes as many pages as possible from the working set. The [EmptyWorkingSet](#) function can also be used for this purpose.

If the values of either *dwMinimumWorkingSetSize* or *dwMaximumWorkingSetSize* are greater than the process' current working set sizes, the specified process must have the **SE_INC_WORKING_SET_NAME** privilege. All users generally have this privilege. For more information about security privileges, see [Privileges](#).


Windows Server 2003 and Windows XP: The specified process must have the **SE_INC_BASE_PRIORITY_NAME** privilege. Users in the Administrators and Power Users groups generally have this privilege.

The operating system allocates working set sizes on a first-come, first-served basis. For example, if an application successfully sets 40 megabytes as its minimum working set size on a 64-megabyte system, and a second application requests a 40-megabyte working set size, the operating system denies the second application's request.

Using the **SetProcessWorkingSetSize** function to set an application's minimum and maximum working set sizes does not guarantee that the requested memory will be reserved, or that it will remain resident at all times. When the application is idle, or a low-memory situation causes a demand for memory, the operating system can reduce the application's working set. An application can use the [VirtualLock](#) function to lock ranges of the application's virtual address space in memory; however, that can potentially degrade the performance of the system.

When you increase the working set size of an application, you are taking away physical memory from the rest of the system. This can degrade the performance of other applications and the system as a whole. It can also lead to failures of operations that require physical memory to be present (for example, creating processes, threads, and kernel pool). Thus, you must use the **SetProcessWorkingSetSize** function carefully. You must always consider the performance of the whole system when you are designing an application.

Requirements

 [Expand table](#)

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	memoryapi.h
Library	oncore.lib
DLL	Kernel32.dll

See also

[GetProcessWorkingSetSize](#)

[Process Working Set](#)

[Process and Thread Functions](#)

[Processes](#)

[SetProcessWorkingSetSizeEx](#)

[VirtualLock](#)

SetProcessWorkingSetSizeEx function (memoryapi.h)

Article 09/23/2022

Sets the minimum and maximum working set sizes for the specified process.

Syntax

C++

```
BOOL SetProcessWorkingSetSizeEx(  
    [in] HANDLE hProcess,  
    [in] SIZE_T dwMinimumWorkingSetSize,  
    [in] SIZE_T dwMaximumWorkingSetSize,  
    [in] DWORD Flags  
);
```

Parameters

[in] hProcess

A handle to the process whose working set sizes is to be set.

The handle must have **PROCESS_SET_QUOTA** access rights. For more information, see [Process Security and Access Rights](#).

[in] dwMinimumWorkingSetSize

The minimum working set size for the process, in bytes. The virtual memory manager attempts to keep at least this much memory resident in the process whenever the process is active.

This parameter must be greater than zero but less than or equal to the maximum working set size. The default size is 50 pages (for example, this is 204,800 bytes on systems with a 4K page size). If the value is greater than zero but less than 20 pages, the minimum value is set to 20 pages.

If both *dwMinimumWorkingSetSize* and *dwMaximumWorkingSetSize* have the value (**SIZE_T**)–1, the function removes as many pages as possible from the working set of the specified process.

[in] dwMaximumWorkingSetSize

The maximum working set size for the process, in bytes. The virtual memory manager attempts to keep no more than this much memory resident in the process whenever the process is active.

and available memory is low.

This parameter must be greater than or equal to 13 pages (for example, 53,248 on systems with a 4K page size), and less than the system-wide maximum (number of available pages minus 512 pages). The default size is 345 pages (for example, this is 1,413,120 bytes on systems with a 4K page size).

If both *dwMinimumWorkingSetSize* and *dwMaximumWorkingSetSize* have the value (SIZE_T)–1, the function removes as many pages as possible from the working set of the specified process. For details, see Remarks.

[in] Flags

The flags that control the enforcement of the minimum and maximum working set sizes.

 Expand table

Value	Meaning
QUOTA_LIMITS_HARDWS_MIN_DISABLE 0x00000002	The working set may fall below the minimum working set limit if memory demands are high. This flag cannot be used with QUOTA_LIMITS_HARDWS_MIN_ENABLE .
QUOTA_LIMITS_HARDWS_MIN_ENABLE 0x00000001	The working set will not fall below the minimum working set limit. This flag cannot be used with QUOTA_LIMITS_HARDWS_MIN_DISABLE .
QUOTA_LIMITS_HARDWS_MAX_DISABLE 0x00000008	<i>The working set may exceed the maximum working set limit if there is abundant memory.</i> This flag cannot be used with QUOTA_LIMITS_HARDWS_MAX_ENABLE .
QUOTA_LIMITS_HARDWS_MAX_ENABLE 0x00000004	The working set will not exceed the maximum working set limit. This flag cannot be used with QUOTA_LIMITS_HARDWS_MAX_DISABLE .

Return value

If the function is succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#). If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The working set of a process is the set of memory pages in the virtual address space of the process that are currently resident in physical memory. These pages are available for an application to use without triggering a page fault. For more information about page faults, see [Working Set](#). The minimum and maximum working set sizes affect the virtual memory paging behavior of a process.

The working set of the specified process can be emptied by specifying the value (**SIZE_T**)–1 for both the minimum and maximum working set sizes. This removes as many pages as possible from the working set. The [EmptyWorkingSet](#) function can also be used for this purpose.

If the values of either *dwMinimumWorkingSetSize* or *dwMaximumWorkingSetSize* are greater than the process' current working set sizes, the specified process must have the **SE_INC_WORKING_SET_NAME** privilege. All users generally have this privilege. For more information about security privileges, see [Privileges](#).

Windows Server 2003: The specified process must have the **SE_INC_BASE_PRIORITY_NAME** privilege. Users in the Administrators and Power Users groups generally have this privilege.

The operating system allocates working set sizes on a first-come, first-served basis. For example, if an application successfully sets 40 megabytes as its minimum working set size on a 64-megabyte system, and a second application requests a 40-megabyte working set size, the operating system denies the second application's request.

By default, using the [SetProcessWorkingSetSize](#) function to set an application's minimum and maximum working set sizes does not guarantee that the requested memory will be reserved, or that it will remain resident at all times. When an application is idle, or a low-memory situation causes a demand for memory, the operating system can reduce the application's working set below its minimum working set limit. If memory is abundant, the system might allow an application to exceed its maximum working set limit. The **QUOTA_LIMITS_HARDWS_MIN_ENABLE** and **QUOTA_LIMITS_HARDWS_MAX_ENABLE** flags enable you to ensure that limits are enforced.

When you increase the working set size of an application, you are taking away physical memory from the rest of the system. This can degrade the performance of other applications and the system as a whole. It can also lead to failures of operations that require physical memory to be present (for example, creating processes, threads, and kernel pool). Thus, you must use the [SetProcessWorkingSetSize](#) function carefully. You must always consider the performance of the whole system when you are designing an application.

An application can use the [VirtualLock](#) function to lock ranges of the application's virtual address space in memory; however, that can potentially degrade the performance of the

system.

Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows Vista [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	memoryapi.h (include Windows.h on Windows Vista, Windows 7, Windows Server 2008 Windows Server 2008 R2)
Library	oncore.lib
DLL	Kernel32.dll

See also

[GetProcessWorkingSetSizeEx](#)

[Process Working Set](#)

[Process and Thread Functions](#)

[Processes](#)

[VirtualLock](#)

[Working Set](#)

SetSystemFileCacheSize function (memoryapi.h)

Article05/14/2022

Limits the size of the working set for the file system cache.

Syntax

C++

```
BOOL SetSystemFileCacheSize(  
    [in] SIZE_T MinimumFileCacheSize,  
    [in] SIZE_T MaximumFileCacheSize,  
    [in] DWORD  Flags  
);
```

Parameters

[in] MinimumFileCacheSize

The minimum size of the file cache, in bytes. The virtual memory manager attempts to keep at least this much memory resident in the system file cache.

To flush the cache, specify (SIZE_T) -1.

[in] MaximumFileCacheSize

The maximum size of the file cache, in bytes. The virtual memory manager enforces this limit only if this call or a previous call to **SetSystemFileCacheSize** specifies **FILE_CACHE_MAX_HARD_ENABLE**.

To flush the cache, specify (SIZE_T) -1.

[in] Flags

The flags that enable or disable the file cache limits. If this parameter is 0 (zero), the size limits retain the current setting, which is either disabled or enabled.

 Expand table

Value	Meaning
FILE_CACHE_MAX_HARD_DISABLE	Disable the maximum size limit.

0x2	The FILE_CACHE_MAX_HARD_DISABLE and FILE_CACHE_MAX_HARD_ENABLE flags are mutually exclusive.
FILE_CACHE_MAX_HARD_ENABLE 0x1	Enable the maximum size limit. The FILE_CACHE_MAX_HARD_DISABLE and FILE_CACHE_MAX_HARD_ENABLE flags are mutually exclusive.
FILE_CACHE_MIN_HARD_DISABLE 0x8	Disable the minimum size limit. The FILE_CACHE_MIN_HARD_DISABLE and FILE_CACHE_MIN_HARD_ENABLE flags are mutually exclusive.
FILE_CACHE_MIN_HARD_ENABLE 0x4	Enable the minimum size limit. The FILE_CACHE_MIN_HARD_DISABLE and FILE_CACHE_MIN_HARD_ENABLE flags are mutually exclusive.

Return value

If the function succeeds, the return value is a nonzero value.

If the function fails, the return value is 0 (zero). To get extended error information, call [GetLastError](#).

Remarks

The calling process must enable the **SE_INCREASE_QUOTA_NAME** privilege.

Setting the *MaximumFileCacheSize* parameter to a very low value can adversely affect system performance.

To compile an application that uses this function, define **_WIN32_WINNT** as 0x0502 or later. For more information, see [Using the Windows Headers](#).

The **FILE_CACHE_*** constants will be defined in the Windows header files starting with the Windows SDK for Windows Server 2008. If you are using header files from an earlier version of the SDK, add the following definitions to your code.


C++

```
#ifndef FILE_CACHE_FLAGS_DEFINED

#define FILE_CACHE_MAX_HARD_ENABLE    0x00000001
#define FILE_CACHE_MAX_HARD_DISABLE  0x00000002
#define FILE_CACHE_MIN_HARD_ENABLE    0x00000004
#define FILE_CACHE_MIN_HARD_DISABLE  0x00000008
```

```
#endif // FILE_CACHE_FLAGS_DEFINED
```

Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows Vista, Windows XP Professional x64 Edition [desktop apps only]
Minimum supported server	Windows Server 2008, Windows Server 2003 with SP1 [desktop apps only]
Target Platform	Windows
Header	memoryapi.h (include Windows.h, Memoryapi.h)
Library	onecore.lib
DLL	Kernel32.dll

See also

[GetSystemFileCacheSize](#)

[Memory Management Functions](#)

UnmapViewOfFile function (memoryapi.h)

Article 08/02/2023

Unmaps a mapped view of a file from the calling process's address space.

Syntax

C++

```
BOOL UnmapViewOfFile(  
    [in] LPCVOID lpBaseAddress  
);
```

Parameters

[in] lpBaseAddress

A pointer to the base address of the mapped view of a file that is to be unmapped. This value must be identical to the value returned by a previous call to one of the functions in the [MapViewOfFile](#) family.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks


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.

Although an application may close the file handle used to create a file mapping object, the system holds the corresponding file open until the last view of the file is unmapped. Files for which the last view has not yet been unmapped are held open with no sharing restrictions.

In Windows Server 2012, this function is supported by the following technologies.

 [Expand table](#)

Technology	Supported
Server Message Block (SMB) 3.0 protocol	Yes
SMB 3.0 Transparent Failover (TFO)	Yes
SMB 3.0 with Scale-out File Shares (SO)	Yes
Cluster Shared Volume File System (CsvFS)	Yes
Resilient File System (ReFS)	Yes

Examples

For an example, see [Creating a View Within a File](#).

Requirements

 [Expand table](#)

Requirement	Value
Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	memoryapi.h (include Windows.h, Memoryapi.h)
Library	oncore.lib
DLL	Kernel32.dll

See also

[Closing a File Mapping Object](#)

[File Mapping Functions](#)

[MapViewOfFile](#)

[MapViewOfFileEx](#)

[Memory Management Functions](#)

UnmapViewOfFile2 function (memoryapi.h)

Article02/22/2024

Unmaps a previously mapped view of a file or a pagefile-backed section.

Syntax

```
C++

BOOL UnmapViewOfFile2(
    [in] HANDLE Process,
    [in] PVOID BaseAddress,
    [in] ULONG UnmapFlags
);
```

Parameters

[in] Process

A **HANDLE** to the process from which the section will be unmapped.

[in] BaseAddress

The base address of a previously mapped view that is to be unmapped. This value must be identical to the value returned by a previous call to one of the functions in the [MapViewOfFile](#) family.

[in] UnmapFlags

This parameter can be zero (0) or one of the following values.

 Expand table

Value	Meaning
MEM_UNMAP_WITH_TRANSIENT_BOOST 0x00000001	Specifies that the priority of the pages being unmapped should be temporarily boosted (with automatic short term decay) because the caller expects that these pages will be accessed again shortly from another thread. For more information about memory priorities, see the SetThreadInformation(ThreadMemoryPriority) function.
MEM_PRESERVE_PLACEHOLDER 0x00000002	Unmaps a mapped view back to a placeholder (after you've replaced a placeholder with a mapped view using

Return value

Returns **TRUE** if successful. Otherwise, returns **FALSE** and extended error status is available using [GetLastError](#).

Requirements

[Expand table](#)

Requirement	Value
Minimum supported client	Windows 10, version 1703 [desktop apps only]
Minimum supported server	Windows Server 2016 [desktop apps only]
Target Platform	Windows
Header	memoryapi.h (include Windows.h, Memoryapi.h)
Library	WindowsApp.lib
DLL	Kernelbase.dll

See also

[MapViewOfFile2](#)[UnmapViewOfFile](#)[UnmapViewOfFileEx](#)

UnmapViewOfFileEx function (memoryapi.h)

Article02/22/2024

This is an extended version of [UnmapViewOfFile](#) that takes an additional flags parameter.

Syntax

```
C++

BOOL UnmapViewOfFileEx(
    [in] PVOID BaseAddress,
    [in] ULONG UnmapFlags
);
```

Parameters

[in] BaseAddress

A pointer to the base address of the mapped view of a file that is to be unmapped. This value must be identical to the value returned by a previous call to one of the functions in the [MapViewOfFile](#) family.

[in] UnmapFlags

This parameter can be one of the following values.

 Expand table

Value	Meaning
MEM_UNMAP_WITH_TRANSIENT_BOOST 0x00000001	Specifies that the priority of the pages being unmapped should be temporarily boosted (with automatic short term decay) because the caller expects that these pages will be accessed again shortly from another thread. For more information about memory priorities, see the SetThreadInformation(ThreadMemoryPriority) function.
MEM_PRESERVE_PLACEHOLDER 0x00000002	Unmaps a mapped view back to a placeholder (after you've replaced a placeholder with a mapped view using MapViewOfFile3 or MapViewOfFile3FromApp).

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

For more information about the behavior of this function, see the [UnmapViewOfFile](#) function.

Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 8 [desktop apps UWP apps]
Minimum supported server	Windows Server 2012 [desktop apps UWP apps]
Target Platform	Windows
Header	memoryapi.h (include Windows.h, Memoryapi.h)
Library	oncore.lib
DLL	Kernel32.dll

UnregisterBadMemoryNotification function (memoryapi.h)

Article02/22/2024

Closes the specified bad memory notification handle.

Syntax

```
C++

BOOL UnregisterBadMemoryNotification(
    [in] PVOID RegistrationHandle
);
```

Parameters

[in] RegistrationHandle

Registration handle returned from the [RegisterBadMemoryNotification](#) function.

Return value


If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

To compile an application that calls this function, define `_WIN32_WINNT` as `_WIN32_WINNT_WIN8` or higher. For more information, see [Using the Windows Headers](#).

Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 8 [desktop apps only]

Requirement	Value
Minimum supported server	Windows Server 2012 [desktop apps only]
Target Platform	Windows
Header	memoryapi.h (include Windows.h, Memoryapi.h)
Library	oncore.lib
DLL	Kernel32.dll

VirtualAlloc function (memoryapi.h)

Article 02/05/2024

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.

To allocate memory in the address space of another process, use the [VirtualAllocEx](#) function.

Syntax

C++

```
LPVOID VirtualAlloc(  
    [in, optional] LPVOID lpAddress,  
    [in]           SIZE_T dwSize,  
    [in]           DWORD  flAllocationType,  
    [in]           DWORD  flProtect  
);
```

Parameters

[in, optional] lpAddress

The starting address of the region to allocate. If the memory is being reserved, the specified address is rounded down to the nearest multiple of the allocation granularity. If the memory is already reserved and is being committed, the address is rounded down to the next page boundary. To determine the size of a page and the allocation granularity on the host computer, use the [GetSystemInfo](#) function. If this parameter is **NULL**, the system determines where to allocate the region.

If this address is within an enclave that you have not initialized by calling [InitializeEnclave](#), **VirtualAlloc** allocates a page of zeros for the enclave at that address. The page must be previously uncommitted, and will not be measured with the EEXTEND instruction of the Intel Software Guard Extensions programming model.


If the address is within an enclave that you initialized, then the allocation operation fails with the **ERROR_INVALID_ADDRESS** error. That is true for enclaves that do not support dynamic memory management (i.e. SGX1). SGX2 enclaves will permit allocation, and the page must be accepted by the enclave after it has been allocated.

[in] dwSize

The size of the region, in bytes. If the *lpAddress* parameter is **NULL**, this value is rounded up to the next page boundary. Otherwise, the allocated pages include all pages containing one or more bytes in the range from *lpAddress* to *lpAddress+dwSize*. This means that a 2-byte range straddling a page boundary causes both pages to be included in the allocated region.

[in] `flAllocationType`

The type of memory allocation. This parameter must contain one of the following values.

 [Expand table](#)

Value	Meaning
MEM_COMMIT 0x00001000	<p>Allocates memory charges (from the overall size of memory and the paging files on disk) for the specified reserved memory pages. The function also guarantees that when the caller later initially accesses the memory, the contents will be zero. Actual physical pages are not allocated unless/until the virtual addresses are actually accessed.</p> <p>To reserve and commit pages in one step, call VirtualAlloc with <code>MEM_COMMIT MEM_RESERVE</code>.</p> <p>Attempting to commit a specific address range by specifying MEM_COMMIT without MEM_RESERVE and a non-NULL <i>lpAddress</i> fails unless the entire range has already been reserved. The resulting error code is ERROR_INVALID_ADDRESS.</p> <p>An attempt to commit a page that is already committed does not cause the function to fail. This means that you can commit pages without first determining the current commitment state of each page.</p> <p>If <i>lpAddress</i> specifies an address within an enclave, <i>flAllocationType</i> must be MEM_COMMIT.</p>
MEM_RESERVE 0x00002000	<p>Reserves a range of the process's virtual address space without allocating any actual physical storage in memory or in the paging file on disk.</p> <p>You can commit reserved pages in subsequent calls to the VirtualAlloc function. To reserve and commit pages in one step, call VirtualAlloc with <code>MEM_COMMIT MEM_RESERVE</code>.</p> <p>Other memory allocation functions, such as malloc and LocalAlloc, cannot use a reserved range of memory until it is released.</p>
MEM_RESET 0x00080000	<p>Indicates that data in the memory range specified by <i>lpAddress</i> and <i>dwSize</i> is no longer of interest. The pages should not be read from or written to the paging file.</p>

However, the memory block will be used again later, so it should not be decommitted. This value cannot be used with any other value.

Using this value does not guarantee that the range operated on with **MEM_RESET** will contain zeros. If you want the range to contain zeros, decommit the memory and then recommit it.

When you specify **MEM_RESET**, the **VirtualAlloc** function ignores the value of *flProtect*. However, you must still set *flProtect* to a valid protection value, such as **PAGE_NOACCESS**.

VirtualAlloc returns an error if you use **MEM_RESET** and the range of memory is mapped to a file. A shared view is only acceptable if it is mapped to a paging file.

MEM_RESET_UNDO
0x1000000

MEM_RESET_UNDO should only be called on an address range to which **MEM_RESET** was successfully applied earlier. It indicates that the data in the specified memory range specified by *lpAddress* and *dwSize* is of interest to the caller and attempts to reverse the effects of **MEM_RESET**. If the function succeeds, that means all data in the specified address range is intact. If the function fails, at least some of the data in the address range has been replaced with zeroes.

This value cannot be used with any other value. If **MEM_RESET_UNDO** is called on an address range which was not **MEM_RESET** earlier, the behavior is undefined. When you specify **MEM_RESET**, the **VirtualAlloc** function ignores the value of *flProtect*. However, you must still set *flProtect* to a valid protection value, such as **PAGE_NOACCESS**.

Windows Server 2008 R2, Windows 7, Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP: The **MEM_RESET_UNDO** flag is not supported until Windows 8 and Windows Server 2012.

This parameter can also specify the following values as indicated.

 Expand table

Value	Meaning
MEM_LARGE_PAGES 0x20000000	<p>Allocates memory using large page support.</p> <p>The size and alignment must be a multiple of the large-page minimum. To obtain this value, use the GetLargePageMinimum function.</p>

	If you specify this value, you must also specify MEM_RESERVE and MEM_COMMIT .
MEM_PHYSICAL 0x00400000	Reserves an address range that can be used to map Address Windowing Extensions (AWE) pages. This value must be used with MEM_RESERVE and no other values.
MEM_TOP_DOWN 0x00100000	Allocates memory at the highest possible address. This can be slower than regular allocations, especially when there are many allocations.
MEM_WRITE_WATCH 0x00200000	Causes the system to track pages that are written to in the allocated region. If you specify this value, you must also specify MEM_RESERVE . To retrieve the addresses of the pages that have been written to since the region was allocated or the write-tracking state was reset, call the GetWriteWatch function. To reset the write-tracking state, call GetWriteWatch or ResetWriteWatch . The write-tracking feature remains enabled for the memory region until the region is freed.

[in] `flProtect`

The memory protection for the region of pages to be allocated. If the pages are being committed, you can specify any one of the [memory protection constants](#).

If *lpAddress* specifies an address within an enclave, *flProtect* cannot be any of the following values:

- **PAGE_NOACCESS**
- **PAGE_GUARD**
- **PAGE_NOCACHE**
- **PAGE_WRITECOMBINE**

When allocating dynamic memory for an enclave, the *flProtect* parameter must be **PAGE_READWRITE** or **PAGE_EXECUTE_READWRITE**.

Return value

If the function succeeds, the return value is the base address of the allocated region of pages.

If the function fails, the return value is **NULL**. To get extended error information, call [GetLastError](#).

Remarks

Each page has an associated [page state](#). The **VirtualAlloc** function can perform the following operations:

- Commit a region of reserved pages
- Reserve a region of free pages
- Simultaneously reserve and commit a region of free pages

VirtualAlloc cannot reserve a reserved page. It can commit a page that is already committed. This means you can commit a range of pages, regardless of whether they have already been committed, and the function will not fail.

You can use **VirtualAlloc** to reserve a block of pages and then make additional calls to **VirtualAlloc** to commit individual pages from the reserved block. This enables a process to reserve a range of its virtual address space without consuming physical storage until it is needed.

If the *lpAddress* parameter is not **NULL**, the function uses the *lpAddress* and *dwSize* parameters to compute the region of pages to be allocated. The current state of the entire range of pages must be compatible with the type of allocation specified by the *flAllocationType* parameter. Otherwise, the function fails and none of the pages are allocated. This compatibility requirement does not preclude committing an already committed page, as mentioned previously.

To execute dynamically generated code, use **VirtualAlloc** to allocate memory and the [VirtualProtect](#) function to grant **PAGE_EXECUTE** access.

The **VirtualAlloc** function can be used to reserve an [Address Windowing Extensions](#) (AWE) region of memory within the virtual address space of a specified process. This region of memory can then be used to map physical pages into and out of virtual memory as required by the application. The **MEM_PHYSICAL** and **MEM_RESERVE** values must be set in the *AllocationType* parameter. The **MEM_COMMIT** value must not be set. The page protection must be set to **PAGE_READWRITE**.

The [VirtualFree](#) function can decommit a committed page, releasing the page's storage, or it can simultaneously decommit and release a committed page. It can also release a reserved page, making it a free page.

When creating a region that will be executable, the calling program bears responsibility for ensuring cache coherency via an appropriate call to [FlushInstructionCache](#) once the code has been set in place. Otherwise attempts to execute code out of the newly executable region may produce unpredictable results.

Examples

For an example, see [Reserving and Committing Memory](#).

Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	memoryapi.h (include Windows.h, Memoryapi.h)
Library	oncore.lib
DLL	Kernel32.dll

See also

[Memory Management Functions](#)

[Virtual Memory Functions](#)

[VirtualAllocEx](#)

[VirtualFree](#)

[VirtualLock](#)

[VirtualProtect](#)

[VirtualQuery](#)

[Vertdll APIs available in VBS enclaves](#)

VirtualAlloc2 function (memoryapi.h)

Article 06/26/2024

Reserves, commits, or changes the state of a region of memory within the virtual address space of a specified process (allocated memory is initialized to zero).

Syntax

C++

```
PVOID VirtualAlloc2(  
    [in, optional] HANDLE Process,  
    [in, optional] PVOID BaseAddress,  
    [in] SIZE_T Size,  
    [in] ULONG AllocationType,  
    [in] ULONG PageProtection,  
    [in, out, optional] MEM_EXTENDED_PARAMETER *ExtendedParameters,  
    [in] ULONG ParameterCount  
);
```

Parameters

[in, optional] Process

The handle to a process. The function allocates memory within the virtual address space of this process.

The handle must have the **PROCESS_VM_OPERATION** access right. For more information, see [Process Security and Access Rights](#).

If *Process* is **NULL**, the function allocates memory for the calling process.

[in, optional] BaseAddress

The pointer that specifies a desired starting address for the region of pages that you want to allocate.

If *BaseAddress* is **NULL**, the function determines where to allocate the region.

If *BaseAddress* is not **NULL**, then any provided [MEM_ADDRESS_REQUIREMENTS](#) structure must consist of all zeroes, and the base address must be a multiple of the system allocation granularity. To determine the allocation granularity, use the [GetSystemInfo](#) function.

If this address is within an enclave that you have not initialized by calling [InitializeEnclave](#), **VirtualAlloc2** allocates a page of zeros for the enclave at that address. The page must be previously uncommitted, and will not be measured with the EEXTEND instruction of the Intel Software Guard Extensions programming model.

If the address is within an enclave that you initialized, then the allocation operation fails with the **ERROR_INVALID_ADDRESS** error. That is true for enclaves that do not support dynamic memory management (i.e. SGX1). SGX2 enclaves will permit allocation, and the page must be accepted by the enclave after it has been allocated.

[in] Size

The size of the region of memory to allocate, in bytes.

The size must always be a multiple of the page size.

If *BaseAddress* is not **NULL**, the function allocates all pages that contain one or more bytes in the range from *BaseAddress* to *BaseAddress*+*Size*. This means, for example, that a 2-byte range that straddles a page boundary causes the function to allocate both pages.

[in] AllocationType

The type of memory allocation. This parameter must contain one of the following values.

[Expand table](#)

Value	Meaning
MEM_COMMIT 0x00001000	<p>Allocates memory charges (from the overall size of memory and the paging files on disk) for the specified reserved memory pages. The function also guarantees that when the caller later initially accesses the memory, the contents will be zero. Actual physical pages are not allocated unless/until the virtual addresses are actually accessed.</p> <p>To reserve and commit pages in one step, call VirtualAlloc2 with MEM_COMMIT MEM_RESERVE.</p> <p>Attempting to commit a specific address range by specifying MEM_COMMIT without MEM_RESERVE and a non-NULL <i>BaseAddress</i> fails unless the entire range has already been reserved. The resulting error code is ERROR_INVALID_ADDRESS.</p> <p>An attempt to commit a page that is already committed does not cause the function to fail. This means that you can commit pages without first determining the current commitment state of each page.</p>

	<p>If <i>BaseAddress</i> specifies an address within an enclave, <i>AllocationType</i> must be MEM_COMMIT.</p>
MEM_RESERVE 0x00002000	<p>Reserves a range of the process's virtual address space without allocating any actual physical storage in memory or in the paging file on disk.</p> <p>You commit reserved pages by calling VirtualAlloc2 again with MEM_COMMIT. To reserve and commit pages in one step, call VirtualAlloc2 with MEM_COMMIT MEM_RESERVE.</p> <p>Other memory allocation functions, such as malloc and LocalAlloc, cannot use reserved memory until it has been released.</p>
MEM_REPLACE_PLACEHOLDER 0x00004000	<p>Replaces a placeholder with a normal private allocation. Only data/pf-backed section views are supported (no images, physical memory, etc.). When you replace a placeholder, <i>BaseAddress</i> and <i>Size</i> must exactly match those of the placeholder, and any provided MEM_ADDRESS_REQUIREMENTS structure must consist of all zeroes.</p> <p>After you replace a placeholder with a private allocation, to free that allocation back to a placeholder, see the <i>dwFreeType</i> parameter of VirtualFree and VirtualFreeEx.</p> <p>A placeholder is a type of reserved memory region.</p>
MEM_RESERVE_PLACEHOLDER 0x00040000	<p>To create a placeholder, call VirtualAlloc2 with MEM_RESERVE MEM_RESERVE_PLACEHOLDER and <i>PageProtection</i> set to PAGE_NOACCESS. To free/split/coalesce a placeholder, see the <i>dwFreeType</i> parameter of VirtualFree and VirtualFreeEx.</p> <p>A placeholder is a type of reserved memory region.</p>
MEM_RESET 0x00080000	<p>Indicates that data in the memory range specified by <i>BaseAddress</i> and <i>Size</i> is no longer of interest. The pages should not be read from or written to the paging file. However, the memory block will be used again later, so it should not be decommitted. This value cannot be used with any other value.</p> <p>Using this value does not guarantee that the range operated on with MEM_RESET will contain zeros. If you want the range to contain zeros, decommit the memory and then recommit it.</p> <p>When you use MEM_RESET, the VirtualAlloc2 function ignores the value of <i>fProtect</i>. However, you must still set <i>fProtect</i> to a valid protection value, such as PAGE_NOACCESS.</p> <p>VirtualAlloc2 returns an error if you use MEM_RESET and the range of memory is mapped to a file. A shared view is only</p>

acceptable if it is mapped to a paging file.


MEM_RESET_UNDO
0x1000000

MEM_RESET_UNDO should only be called on an address range to which **MEM_RESET** was successfully applied earlier. It indicates that the data in the specified memory range specified by *BaseAddress* and *Size* is of interest to the caller and attempts to reverse the effects of **MEM_RESET**. If the function succeeds, that means all data in the specified address range is intact. If the function fails, at least some of the data in the address range has been replaced with zeroes.

This value cannot be used with any other value. If **MEM_RESET_UNDO** is called on an address range which was not **MEM_RESET** earlier, the behavior is undefined. When you specify **MEM_RESET**, the **VirtualAlloc2** function ignores the value of *PageProtection*. However, you must still set *PageProtection* to a valid protection value, such as **PAGE_NOACCESS**.

Windows Server 2008 R2, Windows 7, Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP: The **MEM_RESET_UNDO** flag is not supported until Windows 8 and Windows Server 2012.

This parameter can also specify the following values as indicated.

 Expand table

Value	Meaning
MEM_LARGE_PAGES 0x20000000	<p>Allocates memory using large page support.</p> <p>The size and alignment must be a multiple of the large-page minimum. To obtain this value, use the GetLargePageMinimum function.</p> <p>If you specify this value, you must also specify MEM_RESERVE and MEM_COMMIT.</p>
MEM_64K_PAGES 0x20400000	<p>A hint to the operating system to map the memory using 64K pages, if possible.</p> <p>A 64K page is a region of memory that is 64K in size, virtually and physically contiguous, and virtually and physically aligned on a 64K boundary.</p> <p>By default, memory allocated using MEM_64K_PAGES is pageable, and physical pages backing the memory are allocated on demand (at the time of access). If physical memory is too fragmented to assemble a physically</p>

	<p>contiguous 64K page, all or part of a MEM_64K_PAGES allocation may be mapped using non-contiguous small pages instead.</p> <p>If MEM_64K_PAGES is combined with the MEM_EXTENDED_PARAMETER_NONPAGED attribute, the allocation will be mapped using non-paged 64K pages. In that case, if contiguous 64K pages cannot be obtained, the allocation will fail.</p> <p>If MEM_64K_PAGES is specified, the Size and BaseAddress parameters must both be multiples of 64K (BaseAddress may be NULL).</p>
MEM_PHYSICAL 0x00400000	<p>Reserves an address range that can be used to map Address Windowing Extensions (AWE) pages.</p> <p>This value must be used with MEM_RESERVE and no other values.</p>
MEM_TOP_DOWN 0x00100000	<p>Allocates memory at the highest possible address. This can be slower than regular allocations, especially when there are many allocations.</p>

[in] PageProtection

The memory protection for the region of pages to be allocated. If the pages are being committed, you can specify any one of the [memory protection constants](#).

If *BaseAddress* specifies an address within an enclave, *PageProtection* cannot be any of the following values:

- PAGE_NOACCESS
- PAGE_GUARD
- PAGE_NOCACHE
- PAGE_WRITECOMBINE

When allocating dynamic memory for an enclave, the *PageProtection* parameter must be **PAGE_READWRITE** or **PAGE_EXECUTE_READWRITE**.

[in, out, optional] ExtendedParameters

An optional pointer to one or more extended parameters of type [MEM_EXTENDED_PARAMETER](#). Each of those extended parameter values can itself have a *Type* field of either **MemExtendedParameterAddressRequirements** or **MemExtendedParameterNumaNode**. If no **MemExtendedParameterNumaNode** extended parameter is provided, then the behavior is the same as for the [VirtualAlloc/MapViewOfFile](#)

functions (that is, the preferred NUMA node for the physical pages is determined based on the ideal processor of the thread that first accesses the memory).

[in] ParameterCount

The number of extended parameters pointed to by *ExtendedParameters*.

Return value

If the function succeeds, the return value is the base address of the allocated region of pages.

If the function fails, the return value is **NULL**. To get extended error information, call [GetLastError](#).

Remarks

This function lets you specify:

- a range of virtual address space and a power-of-2 alignment restriction for new allocations
- an arbitrary number of extended parameters
- a preferred NUMA node for the physical memory as an extended parameter (see the *ExtendedParameters* parameter)
- a placeholder operation (specifically, replacement).

This API provides specialized techniques for managing virtual memory in support of high-performance games and server applications. For example, placeholders allow a reserved memory range to be explicitly partitioned, overlaid, and re-mapped; this can be used to implement arbitrarily extendable regions or virtual memory ring buffers. **VirtualAlloc2** also allows for allocating memory with a specific memory-alignment.

Each page has an associated [page state](#). The **VirtualAlloc2** function can perform the following operations:

- Commit a region of reserved pages
- Reserve a region of free pages
- Simultaneously reserve and commit a region of free pages

VirtualAlloc2 can commit pages that are already committed, but cannot reserve pages that are already reserved. This means you can commit a range of pages, regardless of whether they have already been committed, and the function will not fail. In general however, only a minimal range of mostly uncommitted pages should be specified, because committing a large number of pages that are already committed can cause the **VirtualAlloc2** call to take much longer.

You can use **VirtualAlloc2** to reserve a block of pages and then make additional calls to **VirtualAlloc2** to commit individual pages from the reserved block. This enables a process to reserve a range of its virtual address space without consuming physical storage until it is needed.

If the *lpAddress* parameter is not **NULL**, the function uses the *lpAddress* and *dwSize* parameters to compute the region of pages to be allocated. The current state of the entire range of pages must be compatible with the type of allocation specified by the *flAllocationType* parameter. Otherwise, the function fails and none of the pages is allocated. This compatibility requirement does not preclude committing an already committed page; see the preceding list.

To execute dynamically generated code, use **VirtualAlloc2** to allocate memory, and the **VirtualProtectEx** function to grant **PAGE_EXECUTE** access.

The **VirtualAlloc2** function can be used to reserve an [Address Windowing Extensions](#) (AWE) region of memory within the virtual address space of a specified process. This region of memory can then be used to map physical pages into and out of virtual memory as required by the application. The **MEM_PHYSICAL** and **MEM_RESERVE** values must be set in the *AllocationType* parameter. The **MEM_COMMIT** value must not be set. The page protection must be set to **PAGE_READWRITE**.

The **VirtualFreeEx** function can decommit a committed page, releasing the page's storage, or it can simultaneously decommit and release a committed page. It can also release a reserved page, making it a free page.

When creating a region that will be executable, the calling program bears responsibility for ensuring cache coherency via an appropriate call to [FlushInstructionCache](#) once the code has been set in place. Otherwise attempts to execute code out of the newly executable region may produce unpredictable results.

Examples

Scenario 1. Create a circular buffer by mapping two adjacent views of the same shared memory section.

C++

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

//
// This function creates a ring buffer by allocating a pagefile-backed section
// and mapping two views of that section next to each other. This way if the
// last record in the buffer wraps it can still be accessed in a linear fashion
```



```

// using its base VA.
//

void*
CreateRingBuffer (
    unsigned int bufferSize,
    _Outptr_ void** secondaryView
)
{
    BOOL result;
    HANDLE section = nullptr;
    SYSTEM_INFO sysInfo;
    void* ringBuffer = nullptr;
    void* placeholder1 = nullptr;
    void* placeholder2 = nullptr;
    void* view1 = nullptr;
    void* view2 = nullptr;

    GetSystemInfo (&sysInfo);

    if ((bufferSize % sysInfo.dwAllocationGranularity) != 0) {
        return nullptr;
    }

    //
    // Reserve a placeholder region where the buffer will be mapped.
    //

    placeholder1 = (PCHAR) VirtualAlloc2 (
        nullptr,
        nullptr,
        2 * bufferSize,
        MEM_RESERVE | MEM_RESERVE_PLACEHOLDER,
        PAGE_NOACCESS,
        nullptr, 0
    );

    if (placeholder1 == nullptr) {
        printf ("VirtualAlloc2 failed, error %#x\n", GetLastError());
        goto Exit;
    }

    //
    // Split the placeholder region into two regions of equal size.
    //

    result = VirtualFree (
        placeholder1,
        bufferSize,
        MEM_RELEASE | MEM_PRESERVE_PLACEHOLDER
    );

    if (result == FALSE) {
        printf ("VirtualFreeEx failed, error %#x\n", GetLastError());
        goto Exit;
    }
}

```

```

}

placeholder2 = (void*) ((ULONG_PTR) placeholder1 + bufferSize);

//
// Create a pagefile-backed section for the buffer.
//

section = CreateFileMapping (
    INVALID_HANDLE_VALUE,
    nullptr,
    PAGE_READWRITE,
    0,
    bufferSize, nullptr
);

if (section == nullptr) {
    printf ("CreateFileMapping failed, error %#x\n", GetLastError());
    goto Exit;
}

//
// Map the section into the first placeholder region.
//

view1 = MapViewOfFile3 (
    section,
    nullptr,
    placeholder1,
    0,
    bufferSize,
    MEM_REPLACE_PLACEHOLDER,
    PAGE_READWRITE,
    nullptr, 0
);

if (view1 == nullptr) {
    printf ("MapViewOfFile3 failed, error %#x\n", GetLastError());
    goto Exit;
}

//
// Ownership transferred, don't free this now.
//

placeholder1 = nullptr;

//
// Map the section into the second placeholder region.
//

view2 = MapViewOfFile3 (
    section,
    nullptr,
    placeholder2,

```

```

    0,
    bufferSize,
    MEM_REPLACE_PLACEHOLDER,
    PAGE_READWRITE,
    nullptr, 0
);

if (view2 == nullptr) {
    printf ("MapViewOfFile3 failed, error %#x\n", GetLastError());
    goto Exit;
}

//
// Success, return both mapped views to the caller.
//

ringBuffer = view1;
*secondaryView = view2;

placeholder2 = nullptr;
view1 = nullptr;
view2 = nullptr;

Exit:

if (section != nullptr) {
    CloseHandle (section);
}

if (placeholder1 != nullptr) {
    VirtualFree (placeholder1, 0, MEM_RELEASE);
}

if (placeholder2 != nullptr) {
    VirtualFree (placeholder2, 0, MEM_RELEASE);
}

if (view1 != nullptr) {
    UnmapViewOfFileEx (view1, 0);
}

if (view2 != nullptr) {
    UnmapViewOfFileEx (view2, 0);
}

return ringBuffer;
}

int __cdecl wmain()
{
    char* ringBuffer;
    void* secondaryView;
    unsigned int bufferSize = 0x10000;

    ringBuffer = (char*) CreateRingBuffer (bufferSize, &secondaryView);

```

```

if (ringBuffer == nullptr) {
    printf ("CreateRingBuffer failed\n");
    return 0;
}

//
// Make sure the buffer wraps properly.
//

ringBuffer[0] = 'a';

if (ringBuffer[bufferSize] == 'a') {
    printf ("The buffer wraps as expected\n");
}

UnmapViewOfFile (ringBuffer);
UnmapViewOfFile (secondaryView);
}

```

Scenario 2. Specify a preferred NUMA node when allocating memory.

C++

```

void*
AllocateWithPreferredNode (size_t size, unsigned int numaNode)
{
    MEM_EXTENDED_PARAMETER param = {0};

    param.Type = MemExtendedParameterNumaNode;
    param.ULong = numaNode;

    return VirtualAlloc2 (
        nullptr, nullptr,
        size,
        MEM_RESERVE | MEM_COMMIT,
        PAGE_READWRITE,
        &param, 1);
}

```

Scenario 3. Allocate memory in a specific virtual address range (below 4GB, in this example) and with specific alignment.

C++

```

void*
AllocateAlignedBelow2GB (size_t size, size_t alignment)

```

```

{
    MEM_ADDRESS_REQUIREMENTS addressReqs = {0};
    MEM_EXTENDED_PARAMETER param = {0};


    addressReqs.Alignment = alignment;
    addressReqs.HighestEndingAddress = (PVOID)(ULONG_PTR) 0x7fffffff;

    param.Type = MemExtendedParameterAddressRequirements;
    param.Pointer = &addressReqs;

    return VirtualAlloc2 (
        nullptr, nullptr,
        size,
        MEM_RESERVE | MEM_COMMIT,
        PAGE_READWRITE,
        &param, 1);
}

```

Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10 [desktop apps only]
Minimum supported server	Windows Server 2016 [desktop apps only]
Target Platform	Windows
Header	memoryapi.h (include Windows.h)
Library	oncore.lib
DLL	Kernel32.dll

See also

[Memory Management Functions](#)

[ReadProcessMemory](#)

[Virtual Memory Functions](#)

[VirtualAllocExNuma](#)

[VirtualFreeEx](#)

VirtualLock

VirtualProtect

VirtualQuery

WriteProcessMemory

VirtualAlloc2FromApp function (memoryapi.h)

Article 07/27/2022

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.

Using this function, you can: for new allocations, specify a range of virtual address space and a power-of-2 alignment restriction; specify an arbitrary number of extended parameters; specify a preferred NUMA node for the physical memory as an extended parameter; and specify a placeholder operation (specifically, replacement).

To specify the NUMA node, see the *ExtendedParameters* parameter.

Syntax

C++

```
PVOID VirtualAlloc2FromApp(  
    [in, optional] HANDLE Process,  
    [in, optional] PVOID BaseAddress,  
    [in] SIZE_T Size,  
    [in] ULONG AllocationType,  
    [in] ULONG PageProtection,  
    [in, out, optional] MEM_EXTENDED_PARAMETER *ExtendedParameters,  
    [in] ULONG ParameterCount  
);
```

Parameters

[in, optional] Process

The handle to a process. The function allocates memory within the virtual address space of this process.

The handle must have the **PROCESS_VM_OPERATION** access right. For more information, see [Process Security and Access Rights](#).

[in, optional] BaseAddress

The pointer that specifies a desired starting address for the region of pages that you want to allocate.

If an explicit base address is specified, then it must be a multiple of the system allocation granularity. To determine the size of a page and the allocation granularity on the host computer, use the [GetSystemInfo](#) function.

If *BaseAddress* is **NULL**, the function determines where to allocate the region.

[in] **Size**

The size of the region of memory to allocate, in bytes.

The size must always be a multiple of the page size.

If *BaseAddress* is not **NULL**, the function allocates all pages that contain one or more bytes in the range from *BaseAddress* to *BaseAddress*+*Size*. This means, for example, that a 2-byte range that straddles a page boundary causes the function to allocate both pages.

[in] **AllocationType**

The type of memory allocation. This parameter must contain one of the following values.

Value	Meaning
MEM_COMMIT 0x00001000	<p>Allocates memory charges (from the overall size of memory and the paging files on disk) for the specified reserved memory pages. The function also guarantees that when the caller later initially accesses the memory, the contents will be zero. Actual physical pages are not allocated unless/until the virtual addresses are actually accessed.</p> <p>To reserve and commit pages in one step, call Virtual2AllocFromApp with MEM_COMMIT MEM_RESERVE.</p> <p>Attempting to commit a specific address range by specifying MEM_COMMIT without MEM_RESERVE and a non-NULL <i>BaseAddress</i> fails unless the entire range has already been reserved. The resulting error code is ERROR_INVALID_ADDRESS.</p> <p>An attempt to commit a page that is already committed does not cause the function to fail. This means that you can commit pages without first determining the current commitment state of each page.</p>
MEM_RESERVE 0x00002000	<p>Reserves a range of the process's virtual address space without allocating any actual physical storage in memory</p>

or in the paging file on disk.

You can commit reserved pages in subsequent calls to the **Virtual2AllocFromApp** function. To reserve and commit pages in one step, call **Virtual2AllocFromApp** with **MEM_COMMIT | MEM_RESERVE**.

Other memory allocation functions, such as **malloc** and **LocalAlloc**, cannot use a reserved range of memory until it is released.

MEM_REPLACE_PLACEHOLDER
0x00004000

Replaces a placeholder with a normal private allocation. Only data/pf-backed section views are supported (no images, physical memory, etc.). When you replace a placeholder, *BaseAddress* and *Size* must exactly match those of the placeholder.

After you replace a placeholder with a private allocation, to free that allocation back to a placeholder, see the *dwFreeType* parameter of **VirtualFree** and **VirtualFreeEx**.

A placeholder is a type of reserved memory region.

MEM_RESERVE_PLACEHOLDER
0x00040000

To create a placeholder, call **VirtualAlloc2** with **MEM_RESERVE | MEM_RESERVE_PLACEHOLDER** and *PageProtection* set to **PAGE_NOACCESS**. To free/split/coalesce a placeholder, see the *dwFreeType* parameter of **VirtualFree** and **VirtualFreeEx**.

A placeholder is a type of reserved memory region.

MEM_RESET
0x00080000

Indicates that data in the memory range specified by *BaseAddress* and *Size* is no longer of interest. The pages should not be read from or written to the paging file. However, the memory block will be used again later, so it should not be decommitted. This value cannot be used with any other value.

Using this value does not guarantee that the range operated on with **MEM_RESET** will contain zeros. If you want the range to contain zeros, decommit the memory and then recommit it.

When you specify **MEM_RESET**, the **Virtual2AllocFromApp** function ignores the value of *Protection*. However, you must still set *Protection* to a valid protection value, such as **PAGE_NOACCESS**.

Virtual2AllocFromApp returns an error if you use **MEM_RESET** and the range of memory is mapped to a file. A shared view is only acceptable if it is mapped to a paging file.

MEM_RESET_UNDO

0x1000000

MEM_RESET_UNDO should only be called on an address range to which **MEM_RESET** was successfully applied earlier. It indicates that the data in the specified memory range specified by *BaseAddress* and *Size* is of interest to the caller and attempts to reverse the effects of **MEM_RESET**. If the function succeeds, that means all data in the specified address range is intact. If the function fails, at least some of the data in the address range has been replaced with zeroes.

This value cannot be used with any other value. If **MEM_RESET_UNDO** is called on an address range which was not **MEM_RESET** earlier, the behavior is undefined. When you specify **MEM_RESET**, the **Virtual2AllocFromApp** function ignores the value of *Protection*. However, you must still set *Protection* to a valid protection value, such as **PAGE_NOACCESS**.

This parameter can also specify the following values as indicated.

Value	Meaning
MEM_LARGE_PAGES 0x20000000	<p>Allocates memory using large page support.</p> <p>The size and alignment must be a multiple of the large-page minimum. To obtain this value, use the GetLargePageMinimum function.</p> <p>If you specify this value, you must also specify MEM_RESERVE and MEM_COMMIT.</p>
MEM_PHYSICAL 0x00400000	<p>Reserves an address range that can be used to map Address Windowing Extensions (AWE) pages.</p> <p>This value must be used with MEM_RESERVE and no other values.</p>
MEM_TOP_DOWN 0x00100000	<p>Allocates memory at the highest possible address. This can be slower than regular allocations, especially when there are many allocations.</p>
MEM_WRITE_WATCH 0x00200000	<p>Causes the system to track pages that are written to in the allocated region. If you specify this value, you must also specify MEM_RESERVE.</p> <p>To retrieve the addresses of the pages that have been written to since the region was allocated or the write-tracking state was reset, call the GetWriteWatch function. To reset the write-tracking state, call GetWriteWatch or</p>

[ResetWriteWatch](#). The write-tracking feature remains enabled for the memory region until the region is freed.

[in] PageProtection

The memory protection for the region of pages to be allocated. If the pages are being committed, you can specify one of the [memory protection constants](#). The following constants generate an error:

- **PAGE_EXECUTE**
- **PAGE_EXECUTE_READ**
- **PAGE_EXECUTE_READWRITE**
- **PAGE_EXECUTE_WRITECOPY**

[in, out, optional] ExtendedParameters

An optional pointer to one or more extended parameters of type [MEM_EXTENDED_PARAMETER](#). Each of those extended parameter values can itself have a *Type* field of either **MemExtendedParameterAddressRequirements** or **MemExtendedParameterNumaNode**. If no **MemExtendedParameterNumaNode** extended parameter is provided, then the behavior is the same as for the [VirtualAlloc/MapViewOfFile](#) functions (that is, the preferred NUMA node for the physical pages is determined based on the ideal processor of the thread that first accesses the memory).

[in] ParameterCount

The number of extended parameters pointed to by *ExtendedParameters*.

Return value

If the function succeeds, the return value is the base address of the allocated region of pages.

If the function fails, the return value is **NULL**. To get extended error information, call [GetLastError](#).

Remarks

This API helps support high-performance games, and server applications, which have particular requirements around managing their virtual address space. For example, mapping memory on top of a previously reserved region; this is useful for implementing an automatically wrapping ring buffer. And allocating memory with specific alignment;

for example, to enable your application to commit large/huge page-mapped regions on demand.

You can call **Virtual2AllocFromApp** from Windows Store apps with just-in-time (JIT) capabilities to use JIT functionality. The app must include the **codeGeneration** capability in the app manifest file to use JIT capabilities.

Each page has an associated [page state](#). The **Virtual2AllocFromApp** function can perform the following operations:

- Commit a region of reserved pages
- Reserve a region of free pages
- Simultaneously reserve and commit a region of free pages

Virtual2AllocFromApp cannot reserve a reserved page. It can commit a page that is already committed. This means you can commit a range of pages, regardless of whether they have already been committed, and the function will not fail.

You can use **Virtual2AllocFromApp** to reserve a block of pages and then make additional calls to **Virtual2AllocFromApp** to commit individual pages from the reserved block. This enables a process to reserve a range of its virtual address space without consuming physical storage until it is needed.

If the *BaseAddress* parameter is not **NULL**, the function uses the *BaseAddress* and *Size* parameters to compute the region of pages to be allocated. The current state of the entire range of pages must be compatible with the type of allocation specified by the *AllocationType* parameter. Otherwise, the function fails and none of the pages are allocated. This compatibility requirement does not preclude committing an already committed page, as mentioned previously.

Virtual2AllocFromApp does not allow the creation of executable pages.

The **Virtual2AllocFromApp** function can be used to reserve an [Address Windowing Extensions](#) (AWE) region of memory within the virtual address space of a specified process. This region of memory can then be used to map physical pages into and out of virtual memory as required by the application. The **MEM_PHYSICAL** and **MEM_RESERVE** values must be set in the *AllocationType* parameter. The **MEM_COMMIT** value must not be set. The page protection must be set to **PAGE_READWRITE**.

The [VirtualFree](#) function can decommit a committed page, releasing the page's storage, or it can simultaneously decommit and release a committed page. It can also release a reserved page, making it a free page.

When creating a region that will be executable, the calling program bears responsibility for ensuring cache coherency via an appropriate call to [FlushInstructionCache](#) once the code has been set in place. Otherwise attempts to execute code out of the newly executable region may produce unpredictable results.

Examples

For code examples, see [Virtual2Alloc](#).

Requirements

Minimum supported client	Windows 10 [desktop apps only]
Minimum supported server	Windows Server 2016 [desktop apps only]
Target Platform	Windows
Header	memoryapi.h (include Windows.h)
Library	WindowsApp.lib
DLL	Kernel32.dll

See also

[Memory Management Functions](#)

[Virtual Memory Functions](#)

[VirtualAlloc](#)

[VirtualAllocEx](#)

[VirtualFree](#)

[VirtualLock](#)

[VirtualProtectFromApp](#)

[VirtualQuery](#)

Feedback

Was this page helpful?

 Yes

 No

[Get help at Microsoft Q&A](#)

VirtualAllocEx function (memoryapi.h)

Article 07/27/2022

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.

To specify the NUMA node for the physical memory, see [VirtualAllocExNuma](#).

Syntax

C++

```
LPVOID VirtualAllocEx(  
    [in]          HANDLE hProcess,  
    [in, optional] LPVOID lpAddress,  
    [in]          SIZE_T dwSize,  
    [in]          DWORD  flAllocationType,  
    [in]          DWORD  flProtect  
);
```

Parameters

[in] hProcess

The handle to a process. The function allocates memory within the virtual address space of this process.

The handle must have the **PROCESS_VM_OPERATION** access right. For more information, see [Process Security and Access Rights](#).

[in, optional] lpAddress

The pointer that specifies a desired starting address for the region of pages that you want to allocate.

If you are reserving memory, the function rounds this address down to the nearest multiple of the allocation granularity.

If you are committing memory that is already reserved, the function rounds this address down to the nearest page boundary. To determine the size of a page and the allocation granularity on the host computer, use the [GetSystemInfo](#) function.

If *lpAddress* is **NULL**, the function determines where to allocate the region.

If this address is within an enclave that you have not initialized by calling [InitializeEnclave](#), **VirtualAllocEx** allocates a page of zeros for the enclave at that address. The page must be previously uncommitted, and will not be measured with the EEXTEND instruction of the Intel Software Guard Extensions programming model.

If the address is within an enclave that you initialized, then the allocation operation fails with the **ERROR_INVALID_ADDRESS** error. That is true for enclaves that do not support dynamic memory management (i.e. SGX1). SGX2 enclaves will permit allocation, and the page must be accepted by the enclave after it has been allocated.

[in] `dwSize`

The size of the region of memory to allocate, in bytes.

If *lpAddress* is **NULL**, the function rounds *dwSize* up to the next page boundary.

If *lpAddress* is not **NULL**, the function allocates all pages that contain one or more bytes in the range from *lpAddress* to *lpAddress*+*dwSize*. This means, for example, that a 2-byte range that straddles a page boundary causes the function to allocate both pages.

[in] `flAllocationType`

The type of memory allocation. This parameter must contain one of the following values.

[Expand table](#)

Value	Meaning
MEM_COMMIT 0x00001000	<p>Allocates memory charges (from the overall size of memory and the paging files on disk) for the specified reserved memory pages. The function also guarantees that when the caller later initially accesses the memory, the contents will be zero. Actual physical pages are not allocated unless/until the virtual addresses are actually accessed.</p> <p>To reserve and commit pages in one step, call VirtualAllocEx with <code>MEM_COMMIT MEM_RESERVE</code>.</p> <p>Attempting to commit a specific address range by specifying MEM_COMMIT without MEM_RESERVE and a non-NULL <i>lpAddress</i> fails unless the entire range has already been reserved. The resulting error code is ERROR_INVALID_ADDRESS.</p> <p>An attempt to commit a page that is already committed does not cause the function to fail. This means that you can commit pages without first determining the current commitment state of each page.</p>

If *lpAddress* specifies an address within an enclave, *flAllocationType* must be **MEM_COMMIT**.

MEM_RESERVE
0x00002000

Reserves a range of the process's virtual address space without allocating any actual physical storage in memory or in the paging file on disk.

You commit reserved pages by calling **VirtualAllocEx** again with **MEM_COMMIT**. To reserve and commit pages in one step, call **VirtualAllocEx** with **MEM_COMMIT | MEM_RESERVE**.

Other memory allocation functions, such as **malloc** and **LocalAlloc**, cannot use reserved memory until it has been released.

MEM_RESET
0x00080000

Indicates that data in the memory range specified by *lpAddress* and *dwSize* is no longer of interest. The pages should not be read from or written to the paging file. However, the memory block will be used again later, so it should not be decommitted. This value cannot be used with any other value.

Using this value does not guarantee that the range operated on with **MEM_RESET** will contain zeros. If you want the range to contain zeros, decommit the memory and then recommit it.

When you use **MEM_RESET**, the **VirtualAllocEx** function ignores the value of *flProtect*. However, you must still set *flProtect* to a valid protection value, such as **PAGE_NOACCESS**.

VirtualAllocEx returns an error if you use **MEM_RESET** and the range of memory is mapped to a file. A shared view is only acceptable if it is mapped to a paging file.

MEM_RESET_UNDO
0x1000000


MEM_RESET_UNDO should only be called on an address range to which **MEM_RESET** was successfully applied earlier. It indicates that the data in the specified memory range specified by *lpAddress* and *dwSize* is of interest to the caller and attempts to reverse the effects of **MEM_RESET**. If the function succeeds, that means all data in the specified address range is intact. If the function fails, at least some of the data in the address range has been replaced with zeroes.

This value cannot be used with any other value. If **MEM_RESET_UNDO** is called on an address range which was not **MEM_RESET** earlier, the behavior is undefined. When you specify **MEM_RESET**, the **VirtualAllocEx** function ignores the value of *flProtect*. However, you must still set *flProtect* to a valid protection value, such as **PAGE_NOACCESS**.

Windows Server 2008 R2, Windows 7, Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP: The

MEM_RESET_UNDO flag is not supported until Windows 8 and Windows Server 2012.

This parameter can also specify the following values as indicated.

 Expand table

Value	Meaning
MEM_LARGE_PAGES 0x20000000	<p>Allocates memory using large page support.</p> <p>The size and alignment must be a multiple of the large-page minimum. To obtain this value, use the GetLargePageMinimum function.</p> <p>If you specify this value, you must also specify MEM_RESERVE and MEM_COMMIT.</p>
MEM_PHYSICAL 0x00400000	<p>Reserves an address range that can be used to map Address Windowing Extensions (AWE) pages.</p> <p>This value must be used with MEM_RESERVE and no other values.</p>
MEM_TOP_DOWN 0x00100000	<p>Allocates memory at the highest possible address. This can be slower than regular allocations, especially when there are many allocations.</p>

[in] `flProtect`

The memory protection for the region of pages to be allocated. If the pages are being committed, you can specify any one of the [memory protection constants](#).

If *lpAddress* specifies an address within an enclave, *flProtect* cannot be any of the following values:

- **PAGE_NOACCESS**
- **PAGE_GUARD**
- **PAGE_NOCACHE**
- **PAGE_WRITECOMBINE**

When allocating dynamic memory for an enclave, the *flProtect* parameter must be **PAGE_READWRITE** or **PAGE_EXECUTE_READWRITE**.

Return value

If the function succeeds, the return value is the base address of the allocated region of pages.

If the function fails, the return value is **NULL**. To get extended error information, call [GetLastError](#).

Remarks

Each page has an associated [page state](#). The **VirtualAllocEx** function can perform the following operations:

- Commit a region of reserved pages
- Reserve a region of free pages
- Simultaneously reserve and commit a region of free pages

VirtualAllocEx cannot reserve a reserved page. It can commit a page that is already committed. This means you can commit a range of pages, regardless of whether they have already been committed, and the function will not fail.

You can use **VirtualAllocEx** to reserve a block of pages and then make additional calls to **VirtualAllocEx** to commit individual pages from the reserved block. This enables a process to reserve a range of its virtual address space without consuming physical storage until it is needed.

If the *lpAddress* parameter is not **NULL**, the function uses the *lpAddress* and *dwSize* parameters to compute the region of pages to be allocated. The current state of the entire range of pages must be compatible with the type of allocation specified by the *flAllocationType* parameter. Otherwise, the function fails and none of the pages is allocated. This compatibility requirement does not preclude committing an already committed page; see the preceding list.

To execute dynamically generated code, use **VirtualAllocEx** to allocate memory and the [VirtualProtectEx](#) function to grant **PAGE_EXECUTE** access.

The **VirtualAllocEx** function can be used to reserve an [Address Windowing Extensions](#) (AWE) region of memory within the virtual address space of a specified process. This region of memory can then be used to map physical pages into and out of virtual memory as required by the application. The **MEM_PHYSICAL** and **MEM_RESERVE** values must be set in the *AllocationType* parameter. The **MEM_COMMIT** value must not be set. The page protection must be set to **PAGE_READWRITE**.

The [VirtualFreeEx](#) function can decommit a committed page, releasing the page's storage, or it can simultaneously decommit and release a committed page. It can also release a reserved page, making it a free page.

When creating a region that will be executable, the calling program bears responsibility for ensuring cache coherency via an appropriate call to [FlushInstructionCache](#) once the code has been set in place. Otherwise attempts to execute code out of the newly executable region may produce unpredictable results.

Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	memoryapi.h (include Windows.h, Memoryapi.h)
Library	oncore.lib
DLL	Kernel32.dll

See also

[Memory Management Functions](#)

[ReadProcessMemory](#)

[Virtual Memory Functions](#)

[VirtualAllocExNuma](#)

[VirtualFreeEx](#)

[VirtualLock](#)

[VirtualProtect](#)

[VirtualQuery](#)

[WriteProcessMemory](#)

VirtualAllocExNuma function (memoryapi.h)

Article 05/14/2022

Reserves, commits, or changes the state of a region of memory within the virtual address space of the specified process, and specifies the NUMA node for the physical memory.

Syntax

C++

```
LPVOID VirtualAllocExNuma(  
    [in]          HANDLE hProcess,  
    [in, optional] LPVOID lpAddress,  
    [in]          SIZE_T dwSize,  
    [in]          DWORD  flAllocationType,  
    [in]          DWORD  flProtect,  
    [in]          DWORD  nndPreferred  
);
```

Parameters

[in] hProcess

The handle to a process. The function allocates memory within the virtual address space of this process.

The handle must have the **PROCESS_VM_OPERATION** access right. For more information, see [Process Security and Access Rights](#).

[in, optional] lpAddress

The pointer that specifies a desired starting address for the region of pages that you want to allocate.

If you are reserving memory, the function rounds this address down to the nearest multiple of the allocation granularity.

If you are committing memory that is already reserved, the function rounds this address down to the nearest page boundary. To determine the size of a page and the allocation granularity on the host computer, use the [GetSystemInfo](#) function.

If *lpAddress* is **NULL**, the function determines where to allocate the region.

[in] `dwSize`

The size of the region of memory to be allocated, in bytes.

If `lpAddress` is **NULL**, the function rounds `dwSize` up to the next page boundary.

If `lpAddress` is not **NULL**, the function allocates all pages that contain one or more bytes in the range from `lpAddress` to `(lpAddress+dwSize)`. This means, for example, that a 2-byte range that straddles a page boundary causes the function to allocate both pages.

[in] `flAllocationType`

The type of memory allocation. This parameter must contain one of the following values.

[Expand table](#)

Value	Meaning
MEM_COMMIT 0x00001000	<p>Allocates memory charges (from the overall size of memory and the paging files on disk) for the specified reserved memory pages. The function also guarantees that when the caller later initially accesses the memory, the contents will be zero. Actual physical pages are not allocated unless/until the virtual addresses are actually accessed.</p> <p>To reserve and commit pages in one step, call the function with <code>MEM_COMMIT MEM_RESERVE</code>.</p> <p>Attempting to commit a specific address range by specifying MEM_COMMIT without MEM_RESERVE and a non-NULL <code>lpAddress</code> fails unless the entire range has already been reserved. The resulting error code is ERROR_INVALID_ADDRESS.</p> <p>An attempt to commit a page that is already committed does not cause the function to fail. This means that you can commit pages without first determining the current commitment state of each page.</p>
MEM_RESERVE 0x00002000	<p>Reserves a range of the process's virtual address space without allocating any actual physical storage in memory or in the paging file on disk.</p> <p>You commit reserved pages by calling the function again with MEM_COMMIT. To reserve and commit pages in one step, call the function with <code>MEM_COMMIT MEM_RESERVE</code>.</p> <p>Other memory allocation functions, such as malloc and LocalAlloc, cannot use reserved memory until it has been released.</p>

MEM_RESET
0x00080000

Indicates that data in the memory range specified by *lpAddress* and *dwSize* is no longer of interest. The pages should not be read from or written to the paging file. However, the memory block will be used again later, so it should not be decommitted. This value cannot be used with any other value.

Using this value does not guarantee that the range operated on with **MEM_RESET** will contain zeros. If you want the range to contain zeros, decommit the memory and then recommit it.

When you use **MEM_RESET**, the function ignores the value of *flProtect*. However, you must still set *flProtect* to a valid protection value, such as **PAGE_NOACCESS**.

The function returns an error if you use **MEM_RESET** and the range of memory is mapped to a file. A shared view is only acceptable if it is mapped to a paging file.

MEM_RESET_UNDO
0x1000000

MEM_RESET_UNDO should only be called on an address range to which **MEM_RESET** was successfully applied earlier. It indicates that the data in the specified memory range specified by *lpAddress* and *dwSize* is of interest to the caller and attempts to reverse the effects of **MEM_RESET**. If the function succeeds, that means all data in the specified address range is intact. If the function fails, at least some of the data in the address range has been replaced with zeroes.

This value cannot be used with any other value. If **MEM_RESET_UNDO** is called on an address range which was not **MEM_RESET** earlier, the behavior is undefined. When you specify **MEM_RESET**, the **VirtualAllocExNuma** function ignores the value of *flProtect*. However, you must still set *flProtect* to a valid protection value, such as **PAGE_NOACCESS**.

Windows Server 2008 R2, Windows 7, Windows Server 2008 and Windows Vista: The **MEM_RESET_UNDO** flag is not supported until Windows 8 and Windows Server 2012.

This parameter can also specify the following values as indicated.

 Expand table

Value	Meaning
MEM_LARGE_PAGES 0x20000000	Allocates memory using large page support . The size and alignment must be a multiple of the large-page minimum. To obtain this value, use the

[GetLargePageMinimum](#) function.

If you specify this value, you must also specify **MEM_RESERVE** and **MEM_COMMIT**.

MEM_PHYSICAL
0x00400000

Reserves an address range that can be used to map [Address Windowing Extensions](#) (AWE) pages.

This value must be used with **MEM_RESERVE** and no other values.

MEM_TOP_DOWN
0x00100000

Allocates memory at the highest possible address.

[in] `flProtect`

The memory protection for the region of pages to be allocated. If the pages are being committed, you can specify any one of the [memory protection constants](#).

Protection attributes specified when protecting a page cannot conflict with those specified when allocating a page.

[in] `nndPreferred`

The NUMA node where the physical memory should reside.

Used only when allocating a new VA region (either committed or reserved). Otherwise this parameter is ignored when the API is used to commit pages in a region that already exists

Return value

If the function succeeds, the return value is the base address of the allocated region of pages.

If the function fails, the return value is **NULL**. To get extended error information, call [GetLastError](#).

Remarks

Each page has an associated [page state](#). The **VirtualAllocExNuma** function can perform the following operations:

- Commit a region of reserved pages
- Reserve a region of free pages
- Simultaneously reserve and commit a region of free pages

VirtualAllocExNuma cannot reserve a reserved page. It can commit a page that is already committed. This means you can commit a range of pages, regardless of whether they have already been committed, and the function will not fail.

You can use **VirtualAllocExNuma** to reserve a block of pages and then make additional calls to **VirtualAllocExNuma** to commit individual pages from the reserved block. This enables a process to reserve a range of its virtual address space without consuming physical storage until it is needed.

If the *lpAddress* parameter is not **NULL**, the function uses the *lpAddress* and *dwSize* parameters to compute the region of pages to be allocated. The current state of the entire range of pages must be compatible with the type of allocation specified by the *flAllocationType* parameter. Otherwise, the function fails and none of the pages is allocated. This compatibility requirement does not preclude committing an already committed page; see the preceding list.

Because **VirtualAllocExNuma** does not allocate any physical pages, it will succeed whether or not the pages are available on that node or elsewhere in the system. The physical pages are allocated on demand. If the preferred node runs out of pages, the memory manager will use pages from other nodes. If the memory is paged out, the same process is used when it is brought back in.

To execute dynamically generated code, use **VirtualAllocExNuma** to allocate memory and the [VirtualProtectEx](#) function to grant **PAGE_EXECUTE** access.

The **VirtualAllocExNuma** function can be used to reserve an [Address Windowing Extensions](#) (AWE) region of memory within the virtual address space of a specified process. This region of memory can then be used to map physical pages into and out of virtual memory as required by the application. The **MEM_PHYSICAL** and **MEM_RESERVE** values must be set in the *AllocationType* parameter. The **MEM_COMMIT** value must not be set. The page protection must be set to **PAGE_READWRITE**.

The [VirtualFreeEx](#) function can decommit a committed page, releasing the page's storage, or it can simultaneously decommit and release a committed page. It can also release a reserved page, making it a free page.

To compile an application that uses this function, define **_WIN32_WINNT** as 0x0600 or later.

Examples

For an example, see [Allocating Memory from a NUMA Node](#).

Requirements

Requirement	Value
Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	memoryapi.h (include Windows.h, Memoryapi.h)
Library	oncore.lib
DLL	Kernel32.dll

See also

[Memory Management Functions](#)

[NUMA Support](#)

[Virtual Memory Functions](#)

[VirtualAllocEx](#)

[VirtualFreeEx](#)

[VirtualLock](#)

[VirtualProtect](#)

[VirtualQuery](#)

VirtualAllocFromApp function (memoryapi.h)

Article 07/27/2022

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.

Syntax

C++

```
PVOID VirtualAllocFromApp(  
    [in, optional] PVOID BaseAddress,  
    [in]           SIZE_T Size,  
    [in]           ULONG AllocationType,  
    [in]           ULONG Protection  
);
```

Parameters

[in, optional] BaseAddress

The starting address of the region to allocate. If the memory is being reserved, the specified address is rounded down to the nearest multiple of the allocation granularity. If the memory is already reserved and is being committed, the address is rounded down to the next page boundary. To determine the size of a page and the allocation granularity on the host computer, use the [GetSystemInfo](#) function. If this parameter is **NULL**, the system determines where to allocate the region.

[in] Size

The size of the region, in bytes. If the *BaseAddress* parameter is **NULL**, this value is rounded up to the next page boundary. Otherwise, the allocated pages include all pages containing one or more bytes in the range from *BaseAddress* to *BaseAddress*+*Size*. This means that a 2-byte range straddling a page boundary causes both pages to be included in the allocated region.

[in] AllocationType

The type of memory allocation. This parameter must contain one of the following values.

 Expand table

Value	Meaning
MEM_COMMIT 0x00001000	<p>Allocates memory charges (from the overall size of memory and the paging files on disk) for the specified reserved memory pages. The function also guarantees that when the caller later initially accesses the memory, the contents will be zero. Actual physical pages are not allocated unless/until the virtual addresses are actually accessed.</p> <p>To reserve and commit pages in one step, call VirtualAllocFromApp with MEM_COMMIT MEM_RESERVE.</p> <p>Attempting to commit a specific address range by specifying MEM_COMMIT without MEM_RESERVE and a non-NULL <i>BaseAddress</i> fails unless the entire range has already been reserved. The resulting error code is ERROR_INVALID_ADDRESS.</p> <p>An attempt to commit a page that is already committed does not cause the function to fail. This means that you can commit pages without first determining the current commitment state of each page.</p>
MEM_RESERVE 0x00002000	<p>Reserves a range of the process's virtual address space without allocating any actual physical storage in memory or in the paging file on disk.</p> <p>You can commit reserved pages in subsequent calls to the VirtualAllocFromApp function. To reserve and commit pages in one step, call VirtualAllocFromApp with MEM_COMMIT MEM_RESERVE.</p> <p>Other memory allocation functions, such as malloc and LocalAlloc, cannot use a reserved range of memory until it is released.</p>
MEM_RESET 0x00080000	<p>Indicates that data in the memory range specified by <i>BaseAddress</i> and <i>Size</i> is no longer of interest. The pages should not be read from or written to the paging file. However, the memory block will be used again later, so it should not be decommitted. This value cannot be used with any other value.</p> <p>Using this value does not guarantee that the range operated on with MEM_RESET will contain zeros. If you want the range to contain zeros, decommit the memory and then recommit it.</p> <p>When you specify MEM_RESET, the VirtualAllocFromApp function ignores the value of <i>Protection</i>. However, you must still set <i>Protection</i> to a valid protection value, such as PAGE_NOACCESS.</p>

VirtualAllocFromApp returns an error if you use **MEM_RESET** and the range of memory is mapped to a file. A shared view is only acceptable if it is mapped to a paging file.

MEM_RESET_UNDO

0x1000000

MEM_RESET_UNDO should only be called on an address range to which **MEM_RESET** was successfully applied earlier. It indicates that the data in the specified memory range specified by *BaseAddress* and *Size* is of interest to the caller and attempts to reverse the effects of **MEM_RESET**. If the function succeeds, that means all data in the specified address range is intact. If the function fails, at least some of the data in the address range has been replaced with zeroes.

This value cannot be used with any other value. If **MEM_RESET_UNDO** is called on an address range which was not **MEM_RESET** earlier, the behavior is undefined. When you specify **MEM_RESET**, the **VirtualAllocFromApp** function ignores the value of *Protection*. However, you must still set *Protection* to a valid protection value, such as **PAGE_NOACCESS**.

This parameter can also specify the following values as indicated.

 Expand table

Value	Meaning
MEM_LARGE_PAGES 0x20000000	<p>Allocates memory using large page support.</p> <p>The size and alignment must be a multiple of the large-page minimum. To obtain this value, use the GetLargePageMinimum function.</p> <p>If you specify this value, you must also specify MEM_RESERVE and MEM_COMMIT.</p>
MEM_PHYSICAL 0x00400000	<p>Reserves an address range that can be used to map Address Windowing Extensions (AWE) pages.</p> <p>This value must be used with MEM_RESERVE and no other values.</p>
MEM_TOP_DOWN 0x00100000	<p>Allocates memory at the highest possible address. This can be slower than regular allocations, especially when there are many allocations.</p>
MEM_WRITE_WATCH 0x00200000	<p>Causes the system to track pages that are written to in the allocated region. If you specify this value, you must also specify MEM_RESERVE.</p>

To retrieve the addresses of the pages that have been written to since the region was allocated or the write-tracking state was reset, call the [GetWriteWatch](#) function. To reset the write-tracking state, call **GetWriteWatch** or [ResetWriteWatch](#). The write-tracking feature remains enabled for the memory region until the region is freed.

[in] Protection

The memory protection for the region of pages to be allocated. If the pages are being committed, you can specify one of the [memory protection constants](#). The following constants generate an error:

- **PAGE_EXECUTE**
- **PAGE_EXECUTE_READ**
- **PAGE_EXECUTE_READWRITE**
- **PAGE_EXECUTE_WRITECOPY**

Return value

If the function succeeds, the return value is the base address of the allocated region of pages.

If the function fails, the return value is **NULL**. To get extended error information, call [GetLastError](#).

Remarks

You can call **VirtualAllocFromApp** from Windows Store apps with just-in-time (JIT) capabilities to use JIT functionality. The app must include the **codeGeneration** capability in the app manifest file to use JIT capabilities.

Each page has an associated [page state](#). The **VirtualAllocFromApp** function can perform the following operations:

- Commit a region of reserved pages
- Reserve a region of free pages
- Simultaneously reserve and commit a region of free pages

VirtualAllocFromApp cannot reserve a reserved page. It can commit a page that is already committed. This means you can commit a range of pages, regardless of whether they have already been committed, and the function will not fail.

You can use **VirtualAllocFromApp** to reserve a block of pages and then make additional calls to **VirtualAllocFromApp** to commit individual pages from the reserved block. This enables a process to reserve a range of its virtual address space without consuming physical storage until it is needed.

If the *BaseAddress* parameter is not **NULL**, the function uses the *BaseAddress* and *Size* parameters to compute the region of pages to be allocated. The current state of the entire range of pages must be compatible with the type of allocation specified by the *AllocationType* parameter. Otherwise, the function fails and none of the pages are allocated. This compatibility requirement does not preclude committing an already committed page, as mentioned previously.


VirtualAllocFromApp does not allow the creation of executable pages.

The **VirtualAllocFromApp** function can be used to reserve an [Address Windowing Extensions](#) (AWE) region of memory within the virtual address space of a specified process. This region of memory can then be used to map physical pages into and out of virtual memory as required by the application. The **MEM_PHYSICAL** and **MEM_RESERVE** values must be set in the *AllocationType* parameter. The **MEM_COMMIT** value must not be set. The page protection must be set to **PAGE_READWRITE**.

The [VirtualFree](#) function can decommit a committed page, releasing the page's storage, or it can simultaneously decommit and release a committed page. It can also release a reserved page, making it a free page.

When creating a region that will be executable, the calling program bears responsibility for ensuring cache coherency via an appropriate call to [FlushInstructionCache](#) once the code has been set in place. Otherwise attempts to execute code out of the newly executable region may produce unpredictable results.

Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10 [desktop apps UWP apps]
Minimum supported server	Windows Server 2016 [desktop apps UWP apps]
Target Platform	Windows
Header	memoryapi.h (include Windows.h)

Requirement	Value
Library	WindowsApp.lib
DLL	Kernel32.dll

See also

[Memory Management Functions](#)

[Virtual Memory Functions](#)

[VirtualAlloc](#)

[VirtualAllocEx](#)

[VirtualFree](#)

[VirtualLock](#)

[VirtualProtectFromApp](#)

[VirtualQuery](#)

VirtualFree function (memoryapi.h)

Article 02/05/2024

Releases, decommits, or releases and decommits a region of pages within the virtual address space of the calling process.

To free memory allocated in another process by the [VirtualAllocEx](#) function, use the [VirtualFreeEx](#) function.

Syntax

C++

```
BOOL VirtualFree(  
    [in] LPVOID lpAddress,  
    [in] SIZE_T dwSize,  
    [in] DWORD  dwFreeType  
);
```

Parameters

[in] lpAddress

A pointer to the base address of the region of pages to be freed.

If the *dwFreeType* parameter is **MEM_RELEASE**, this parameter must be the base address returned by the [VirtualAlloc](#) function when the region of pages is reserved.

[in] dwSize


The size of the region of memory to be freed, in bytes.

If the *dwFreeType* parameter is **MEM_RELEASE**, this parameter must be 0 (zero). The function frees the entire region that is reserved in the initial allocation call to [VirtualAlloc](#).

If the *dwFreeType* parameter is **MEM_DECOMMIT**, the function decommits all memory pages that contain one or more bytes in the range from the *lpAddress* parameter to `(lpAddress+dwSize)`. This means, for example, that a 2-byte region of memory that straddles a page boundary causes both pages to be decommitted. If *lpAddress* is the base address returned by [VirtualAlloc](#) and *dwSize* is 0 (zero), the function decommits the entire region that is allocated by **VirtualAlloc**. After that, the entire region is in the reserved state.


[in] dwFreeType

The type of free operation. This parameter must be one of the following values.

 Expand table

Value	Meaning
MEM_DECOMMIT 0x00004000	<p>Decommits the specified region of committed pages. After the operation, the pages are in the reserved state.</p> <p>The function does not fail if you attempt to decommit an uncommitted page. This means that you can decommit a range of pages without first determining the current commitment state.</p> <p>The MEM_DECOMMIT value is not supported when the <i>lpAddress</i> parameter provides the base address for an enclave. This is true for enclaves that do not support dynamic memory management (i.e. SGX1). SGX2 enclaves permit MEM_DECOMMIT anywhere in the enclave.</p>
MEM_RELEASE 0x00008000	<p>Releases the specified region of pages, or placeholder (for a placeholder, the address space is released and available for other allocations). After this operation, the pages are in the free state.</p> <p>If you specify this value, <i>dwSize</i> must be 0 (zero), and <i>lpAddress</i> must point to the base address returned by the VirtualAlloc function when the region is reserved. The function fails if either of these conditions is not met.</p> <p>If any pages in the region are committed currently, the function first decommits, and then releases them.</p> <p>The function does not fail if you attempt to release pages that are in different states, some reserved and some committed. This means that you can release a range of pages without first determining the current commitment state.</p>

When using **MEM_RELEASE**, this parameter can additionally specify one of the following values.

 Expand table

Value	Meaning
MEM_COALESCE_PLACEHOLDERS 0x00000001	<p>To coalesce two adjacent placeholders, specify MEM_RELEASE MEM_COALESCE_PLACEHOLDERS. When you coalesce placeholders, <i>lpAddress</i> and <i>dwSize</i> must exactly match the overall range of the placeholders to be merged.</p>

MEM_PRESERVE_PLACEHOLDER
0x00000002

Frees an allocation back to a placeholder (after you've replaced a placeholder with a private allocation using [VirtualAlloc2](#) or [Virtual2AllocFromApp](#) [↗]).

To split a placeholder into two placeholders, specify `MEM_RELEASE | MEM_PRESERVE_PLACEHOLDER`.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is 0 (zero). To get extended error information, call [GetLastError](#).

Remarks

Each page of memory in a process virtual address space has a [Page State](#). The **VirtualFree** function can decommit a range of pages that are in different states, some committed and some uncommitted. This means that you can decommit a range of pages without first determining the current commitment state of each page. Decommitting a page releases its physical storage, either in memory or in the paging file on disk.

If a page is decommitted but not released, its state changes to reserved. Subsequently, you can call [VirtualAlloc](#) to commit it, or **VirtualFree** to release it. Attempts to read from or write to a reserved page results in an access violation exception.

The **VirtualFree** function can release a range of pages that are in different states, some reserved and some committed. This means that you can release a range of pages without first determining the current commitment state of each page. The entire range of pages originally reserved by the [VirtualAlloc](#) function must be released at the same time.

If a page is released, its state changes to free, and it is available for subsequent allocation operations. After memory is released or decommitted, you can never refer to the memory 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 need to keep information, do not decommit or free memory that contains the information.

The **VirtualFree** function can be used on an AWE region of memory, and it invalidates any physical page mappings in the region when freeing the address space. However, the physical page is not deleted, and the application can use them. The application must explicitly call [FreeUserPhysicalPages](#) to free the physical pages. When the process is terminated, all resources are cleaned up automatically.

Windows 10, version 1709 and later and Windows 11: To delete the enclave when you finish using it, call [DeleteEnclave](#). You cannot delete a VBS enclave by calling the **VirtualFree** or [VirtualFreeEx](#) function. You can still delete an SGX enclave by calling **VirtualFree** or **VirtualFreeEx**.

Windows 10, version 1507, Windows 10, version 1511, Windows 10, version 1607 and Windows 10, version 1703: To delete the enclave when you finish using it, call the **VirtualFree** or [VirtualFreeEx](#) function and specify the following values:

- The base address of the enclave for the *lpAddress* parameter.
- 0 for the *dwSize* parameter.
- **MEM_RELEASE** for the *dwFreeType* parameter.

Examples

For an example, see [Reserving and Committing Memory](#).

Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	memoryapi.h (include Windows.h, Memoryapi.h)
Library	oncore.lib
DLL	Kernel32.dll

See also

[Memory Management Functions](#)

[Virtual Memory Functions](#)

[VirtualFreeEx](#)

[Vertdll APIs available in VBS enclaves](#)

VirtualFreeEx function (memoryapi.h)

Article 05/24/2022

Releases, decommits, or releases and decommits a region of memory within the virtual address space of a specified process.

Syntax

C++

```
BOOL VirtualFreeEx(  
    [in] HANDLE hProcess,  
    [in] LPVOID lpAddress,  
    [in] SIZE_T dwSize,  
    [in] DWORD dwFreeType  
);
```

Parameters

[in] hProcess

A handle to a process. The function frees memory within the virtual address space of the process.

The handle must have the **PROCESS_VM_OPERATION** access right. For more information, see [Process Security and Access Rights](#).

[in] lpAddress

A pointer to the starting address of the region of memory to be freed.

If the *dwFreeType* parameter is **MEM_RELEASE**, *lpAddress* must be the base address returned by the [VirtualAllocEx](#) function when the region is reserved.

[in] dwSize

The size of the region of memory to free, in bytes.

If the *dwFreeType* parameter is **MEM_RELEASE**, *dwSize* must be 0 (zero). The function frees the entire region that is reserved in the initial allocation call to [VirtualAllocEx](#).

If *dwFreeType* is **MEM_DECOMMIT**, the function decommits all memory pages that contain one or more bytes in the range from the *lpAddress* parameter to `(lpAddress+dwSize)`. This means, for example, that a 2-byte region of memory that straddles a page boundary causes both

pages to be decommitted. If *lpAddress* is the base address returned by [VirtualAllocEx](#) and *dwSize* is 0 (zero), the function decommits the entire region that is allocated by **VirtualAllocEx**. After that, the entire region is in the reserved state.

[in] dwFreeType

The type of free operation. This parameter must be one of the following values.

[Expand table](#)

Value	Meaning
MEM_DECOMMIT 0x00004000	<p>Decommits the specified region of committed pages. After the operation, the pages are in the reserved state.</p> <p>The function does not fail if you attempt to decommit an uncommitted page. This means that you can decommit a range of pages without first determining the current commitment state.</p> <p>The MEM_DECOMMIT value is not supported when the <i>lpAddress</i> parameter provides the base address for an enclave. This is true for enclaves that do not support dynamic memory management (i.e. SGX1). SGX2 enclaves permit MEM_DECOMMIT anywhere in the enclave.</p>
MEM_RELEASE 0x00008000	<p>Releases the specified region of pages, or placeholder (for a placeholder, the address space is released and available for other allocations). After this operation, the pages are in the free state.</p> <p>If you specify this value, <i>dwSize</i> must be 0 (zero), and <i>lpAddress</i> must point to the base address returned by the VirtualAlloc function when the region is reserved. The function fails if either of these conditions is not met.</p> <p>If any pages in the region are committed currently, the function first decommits, and then releases them.</p> <p>The function does not fail if you attempt to release pages that are in different states, some reserved and some committed. This means that you can release a range of pages without first determining the current commitment state.</p>

When using **MEM_RELEASE**, this parameter can additionally specify one of the following values.

[Expand table](#)

Value	Meaning
-------	---------

MEM_COALESCE_PLACEHOLDERS 0x00000001	To coalesce two adjacent placeholders, specify <code>MEM_RELEASE MEM_COALESCE_PLACEHOLDERS</code> . When you coalesce placeholders, <i>lpAddress</i> and <i>dwSize</i> must exactly match the overall range of the placeholders to be merged.
MEM_PRESERVE_PLACEHOLDER 0x00000002	<p>Frees an allocation back to a placeholder (after you've replaced a placeholder with a private allocation using VirtualAlloc2 or Virtual2AllocFromApp).</p> <p>To split a placeholder into two placeholders, specify <code>MEM_RELEASE MEM_PRESERVE_PLACEHOLDER</code>.</p>

Return value

If the function succeeds, the return value is a nonzero value.

If the function fails, the return value is 0 (zero). To get extended error information, call [GetLastError](#).

Remarks

Each page of memory in a process virtual address space has a [Page State](#). The **VirtualFreeEx** function can decommit a range of pages that are in different states, some committed and some uncommitted. This means that you can decommit a range of pages without first determining the current commitment state of each page. Decommitting a page releases its physical storage, either in memory or in the paging file on disk.

If a page is decommitted but not released, its state changes to reserved. Subsequently, you can call [VirtualAllocEx](#) to commit it, or **VirtualFreeEx** to release it. Attempting to read from or write to a reserved page results in an access violation exception.

The **VirtualFreeEx** function can release a range of pages that are in different states, some reserved and some committed. This means that you can release a range of pages without first determining the current commitment state of each page. The entire range of pages originally reserved by [VirtualAllocEx](#) must be released at the same time.

If a page is released, its state changes to free, and it is available for subsequent allocation operations. After memory is released or decommitted, you can never refer to the memory again. Any information that may have been in that memory is gone forever. Attempts to read from or write to a free page results in an access violation exception. If you need to keep information, do not decommit or free memory that contains the information.

The **VirtualFreeEx** function can be used on an AWE region of memory and it invalidates any physical page mappings in the region when freeing the address space. However, the physical pages are not deleted, and the application can use them. The application must explicitly call [FreeUserPhysicalPages](#) to free the physical pages. When the process is terminated, all resources are automatically cleaned up.

Windows 10, version 1709 and later and Windows 11: To delete the enclave when you finish using it, call [DeleteEnclave](#). You cannot delete a VBS enclave by calling the [VirtualFree](#) or **VirtualFreeEx** function. You can still delete an SGX enclave by calling **VirtualFree** or **VirtualFreeEx**.

Windows 10, version 1507, Windows 10, version 1511, Windows 10, version 1607 and Windows 10, version 1703: To delete the enclave when you finish using it, call the [VirtualFree](#) or **VirtualFreeEx** function and specify the following values:

- The base address of the enclave for the *lpAddress* parameter.
- 0 for the *dwSize* parameter.
- **MEM_RELEASE** for the *dwFreeType* parameter.

Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	memoryapi.h (include Windows.h, Memoryapi.h)
Library	oncore.lib
DLL	Kernel32.dll

See also

[Memory Management Functions](#)

[Virtual Memory Functions](#)

[VirtualAllocEx](#)

VirtualLock function (memoryapi.h)

Article 08/23/2022

Locks the specified region of the process's virtual address space into physical memory, ensuring that subsequent access to the region will not incur a page fault.

Syntax

C++

```
BOOL VirtualLock(  
    [in] LPVOID lpAddress,  
    [in] SIZE_T dwSize  
);
```

Parameters

[in] lpAddress

A pointer to the base address of the region of pages to be locked.

[in] dwSize

The size of the region to be locked, in bytes. The region of affected pages includes all pages that contain one or more bytes in the range from the *lpAddress* parameter to `(lpAddress+dwSize)`. This means that a 2-byte range straddling a page boundary causes both pages to be locked.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

All pages in the specified region must be committed. Memory protected with **PAGE_NOACCESS** cannot be locked.

Locking pages into memory may degrade the performance of the system by reducing the available RAM and forcing the system to swap out other critical pages to the paging file. Each version of Windows has a limit on the maximum number of pages a process can lock. This limit is intentionally small to avoid severe performance degradation. Applications that need to lock larger numbers of pages must first call the [SetProcessWorkingSetSize](#) function to increase their minimum and maximum working set sizes. The maximum number of pages that a process can lock is equal to the number of pages in its minimum working set minus a small overhead.

Pages that a process has locked remain in physical memory until the process unlocks them or terminates. These pages are guaranteed not to be written to the pagefile while they are locked.


To unlock a region of locked pages, use the [VirtualUnlock](#) function. Locked pages are automatically unlocked when the process terminates.

This function is not like the [GlobalLock](#) or [LocalLock](#) function in that it does not increment a lock count and translate a handle into a pointer. There is no lock count for virtual pages, so multiple calls to the [VirtualUnlock](#) function are never required to unlock a region of pages.

Examples

For an example, see [Creating Guard Pages](#).

Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	memoryapi.h (include Windows.h, Memoryapi.h)
Library	oncore.lib
DLL	Kernel32.dll

See also

[Memory Management Functions](#)

[SetProcessWorkingSetSize](#)

[Virtual Memory Functions](#)

[VirtualUnlock](#)

VirtualProtect function (memoryapi.h)

Article 02/05/2024

Changes the protection on a region of committed pages in the virtual address space of the calling process.

To change the access protection of any process, use the [VirtualProtectEx](#) function.

Syntax

C++

```
BOOL VirtualProtect(  
    [in] LPVOID lpAddress,  
    [in] SIZE_T dwSize,  
    [in] DWORD  flNewProtect,  
    [out] PDWORD lpflOldProtect  
);
```

Parameters

[in] lpAddress

The address of the starting page of the region of pages whose access protection attributes are to be changed.

All pages in the specified region must be within the same reserved region allocated when calling the [VirtualAlloc](#) or [VirtualAllocEx](#) function using **MEM_RESERVE**. The pages cannot span adjacent reserved regions that were allocated by separate calls to **VirtualAlloc** or **VirtualAllocEx** using **MEM_RESERVE**.

[in] dwSize

The size of the region whose access protection attributes are to be changed, in bytes. The region of affected pages includes all pages containing one or more bytes in the range from the **lpAddress** parameter to **(lpAddress+dwSize)**. This means that a 2-byte range straddling a page boundary causes the protection attributes of both pages to be changed.

[in] flNewProtect

The memory protection option. This parameter can be one of the [memory protection constants](#).

For mapped views, this value must be compatible with the access protection specified when the view was mapped (see [MapViewOfFile](#), [MapViewOfFileEx](#), and [MapViewOfFileExNuma](#)).

[out] lpfl0ldProtect

A pointer to a variable that receives the previous access protection value of the first page in the specified region of pages. If this parameter is **NULL** or does not point to a valid variable, the function fails.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

You can set the access protection value on committed pages only. If the state of any page in the specified region is not committed, the function fails and returns without modifying the access protection of any pages in the specified region.

The **PAGE_GUARD** protection modifier establishes guard pages. Guard pages act as one-shot access alarms. For more information, see [Creating Guard Pages](#).

It is best to avoid using **VirtualProtect** to change page protections on memory blocks allocated by [GlobalAlloc](#), [HeapAlloc](#), or [LocalAlloc](#), because multiple memory blocks can exist on a single page. The heap manager assumes that all pages in the heap grant at least read and write access.

When protecting a region that will be executable, the calling program bears responsibility for ensuring cache coherency via an appropriate call to [FlushInstructionCache](#) once the code has been set in place. Otherwise attempts to execute code out of the newly executable region may produce unpredictable results.

Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps UWP apps]

Requirement	Value
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	memoryapi.h (include Windows.h, Memoryapi.h)
Library	oncore.lib
DLL	Kernel32.dll

See also

[Memory Management Functions](#)

[Memory Protection Constants](#)

[Virtual Memory Functions](#)

[VirtualAlloc](#)

[VirtualProtectEx](#)

[Vertdll APIs available in VBS enclaves](#)

VirtualProtectEx function (memoryapi.h)

Article 05/24/2022

Changes the protection on a region of committed pages in the virtual address space of a specified process.

Syntax

C++

```
BOOL VirtualProtectEx(  
    [in] HANDLE hProcess,  
    [in] LPVOID lpAddress,  
    [in] SIZE_T dwSize,  
    [in] DWORD flNewProtect,  
    [out] PDWORD lpflOldProtect  
);
```

Parameters

[in] hProcess

A handle to the process whose memory protection is to be changed. The handle must have the **PROCESS_VM_OPERATION** access right. For more information, see [Process Security and Access Rights](#).

[in] lpAddress

A pointer to the base address of the region of pages whose access protection attributes are to be changed.

All pages in the specified region must be within the same reserved region allocated when calling the [VirtualAlloc](#) or [VirtualAllocEx](#) function using **MEM_RESERVE**. The pages cannot span adjacent reserved regions that were allocated by separate calls to **VirtualAlloc** or **VirtualAllocEx** using **MEM_RESERVE**.

[in] dwSize

The size of the region whose access protection attributes are changed, in bytes. The region of affected pages includes all pages containing one or more bytes in the range from the *lpAddress* parameter to `(lpAddress+dwSize)`. This means that a 2-byte range straddling a page boundary causes the protection attributes of both pages to be changed.

[in] flNewProtect

The memory protection option. This parameter can be one of the [memory protection constants](#).

For mapped views, this value must be compatible with the access protection specified when the view was mapped (see [MapViewOfFile](#), [MapViewOfFileEx](#), and [MapViewOfFileExNuma](#)).

[out] lpflOldProtect

A pointer to a variable that receives the previous access protection of the first page in the specified region of pages. If this parameter is **NULL** or does not point to a valid variable, the function fails.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

The access protection value can be set only on committed pages. If the state of any page in the specified region is not committed, the function fails and returns without modifying the access protection of any pages in the specified region.

The **PAGE_GUARD** protection modifier establishes guard pages. Guard pages act as one-shot access alarms. For more information, see [Creating Guard Pages](#).

It is best to avoid using **VirtualProtectEx** to change page protections on memory blocks allocated by [GlobalAlloc](#), [HeapAlloc](#), or [LocalAlloc](#), because multiple memory blocks can exist on a single page. The heap manager assumes that all pages in the heap grant at least read and write access.

When protecting a region that will be executable, the calling program bears responsibility for ensuring cache coherency via an appropriate call to [FlushInstructionCache](#) once the code has been set in place. Otherwise attempts to execute code out of the newly executable region may produce unpredictable results.

Requirements

Requirement	Value
Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	memoryapi.h (include Windows.h, Memoryapi.h)
Library	oncore.lib
DLL	Kernel32.dll

See also

[Memory Management Functions](#)

[Memory Protection Constants](#)

[Virtual Memory Functions](#)

[VirtualAlloc](#)

[VirtualProtect](#)

[VirtualQueryEx](#)

VirtualProtectFromApp function (memoryapi.h)

Article 07/27/2022

Changes the protection on a region of committed pages in the virtual address space of the calling process.

Syntax

C++

```
BOOL VirtualProtectFromApp(  
    [in] PVOID Address,  
    [in] SIZE_T Size,  
    [in] ULONG NewProtection,  
    [out] PULONG OldProtection  
);
```

Parameters

[in] Address

A pointer to an address that describes the starting page of the region of pages whose access protection attributes are to be changed.

All pages in the specified region must be within the same reserved region allocated when calling the [VirtualAlloc](#), [VirtualAllocFromApp](#), or [VirtualAllocEx](#) function using **MEM_RESERVE**. The pages cannot span adjacent reserved regions that were allocated by separate calls to [VirtualAlloc](#), [VirtualAllocFromApp](#), or [VirtualAllocEx](#) using **MEM_RESERVE**.

[in] Size

The size of the region whose access protection attributes are to be changed, in bytes. The region of affected pages includes all pages containing one or more bytes in the range from the *Address* parameter to `(Address+Size)`. This means that a 2-byte range straddling a page boundary causes the protection attributes of both pages to be changed.

[in] NewProtection

The memory protection option. This parameter can be one of the [memory protection constants](#).

For mapped views, this value must be compatible with the access protection specified when the view was mapped (see [MapViewOfFile](#), [MapViewOfFileEx](#), and [MapViewOfFileExNuma](#)).

The following constants generate an error:

- **PAGE_EXECUTE_READWRITE**
- **PAGE_EXECUTE_WRITECOPY**

The following constants are allowed only for apps that have the **codeGeneration** capability:

- **PAGE_EXECUTE**
- **PAGE_EXECUTE_READ**

[out] OldProtection

A pointer to a variable that receives the previous access protection value of the first page in the specified region of pages. If this parameter is **NULL** or does not point to a valid variable, the function fails.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

You can call **VirtualProtectFromApp** from Windows Store apps with just-in-time (JIT) capabilities to use JIT functionality. The app must include the **codeGeneration** capability in the app manifest file to use JIT capabilities.

You can set the access protection value on committed pages only. If the state of any page in the specified region is not committed, the function fails and returns without modifying the access protection of any pages in the specified region.

The **PAGE_GUARD** protection modifier establishes guard pages. Guard pages act as one-shot access alarms. For more information, see [Creating Guard Pages](#).

It is best to avoid using **VirtualProtectFromApp** to change page protections on memory blocks allocated by [GlobalAlloc](#), [HeapAlloc](#), or [LocalAlloc](#), because multiple memory blocks can exist on a single page. The heap manager assumes that all pages in the heap grant at least read and write access.

VirtualProtectFromApp allows you to mark pages as executable, but does not allow you to set both write and execute permissions at the same time.

When protecting a region that will be executable, the calling program bears responsibility for ensuring cache coherency via an appropriate call to [FlushInstructionCache](#) once the code has been set in place. Otherwise attempts to execute code out of the newly executable region may produce unpredictable results.

Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10 [desktop apps UWP apps]
Minimum supported server	Windows Server 2016 [desktop apps UWP apps]
Target Platform	Windows
Header	memoryapi.h (include Windows.h)
Library	WindowsApp.lib
DLL	Kernel32.dll

See also

[Memory Management Functions](#)

[Virtual Memory Functions](#)

[VirtualAllocFromApp](#)

[VirtualProtect](#)

[VirtualProtectEx](#)

VirtualQuery function (memoryapi.h)

Article 02/06/2024

Retrieves information about a range of pages in the virtual address space of the calling process.

To retrieve information about a range of pages in the address space of another process, use the [VirtualQueryEx](#) function.

Syntax

C++

```
SIZE_T VirtualQuery(  
    [in, optional] LPCVOID          lpAddress,  
    [out]           PMEMORY_BASIC_INFORMATION lpBuffer,  
    [in]           SIZE_T           dwLength  
);
```

Parameters

[in, optional] lpAddress

A pointer to the base address of the region of pages to be queried. This value is rounded down to the next page boundary. To determine the size of a page on the host computer, use the [GetSystemInfo](#) function.

If *lpAddress* specifies an address above the highest memory address accessible to the process, the function fails with **ERROR_INVALID_PARAMETER**.

[out] lpBuffer

A pointer to a [MEMORY_BASIC_INFORMATION](#) structure in which information about the specified page range is returned.

[in] dwLength

The size of the buffer pointed to by the *lpBuffer* parameter, in bytes.

Return value

The return value is the actual number of bytes returned in the information buffer.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#). Possible error values include **ERROR_INVALID_PARAMETER**.

Remarks

VirtualQuery provides information about a region of consecutive pages beginning at a specified address that share the following attributes:

- The state of all pages is the same (**MEM_COMMIT**, **MEM_RESERVE**, **MEM_FREE**, **MEM_PRIVATE**, **MEM_MAPPED**, or **MEM_IMAGE**).
- If the initial page is not free, all pages in the region are part of the same initial allocation of pages created by a single call to [VirtualAlloc](#), [MapViewOfFile](#), or one of the following extended versions of these functions: [VirtualAllocEx](#), [VirtualAllocExNuma](#), [MapViewOfFileEx](#), [MapViewOfFileExNuma](#).
- The access granted to all pages is the same (**PAGE_READONLY**, **PAGE_READWRITE**, **PAGE_NOACCESS**, **PAGE_WRITECOPY**, **PAGE_EXECUTE**, **PAGE_EXECUTE_READ**, **PAGE_EXECUTE_READWRITE**, **PAGE_EXECUTE_WRITECOPY**, **PAGE_GUARD**, or **PAGE_NOCACHE**).

The function determines the attributes of the first page in the region and then scans subsequent pages until it scans the entire range of pages or until it encounters a page with a non-matching set of attributes. The function returns the attributes and the size of the region of pages with matching attributes, in bytes. For example, if there is a 40 megabyte (MB) region of free memory, and **VirtualQuery** is called on a page that is 10 MB into the region, the function will obtain a state of **MEM_FREE** and a size of 30 MB.

If a shared copy-on-write page is modified, it becomes private to the process that modified the page. However, the **VirtualQuery** function will continue to report such pages as **MEM_MAPPED** (for data views) or **MEM_IMAGE** (for executable image views) rather than **MEM_PRIVATE**. To detect whether copy-on-write has occurred for a specific page, either access the page or lock it using the [VirtualLock](#) function to make sure the page is resident in memory, then use the [QueryWorkingSetEx](#) function to check the **Shared** bit in the extended working set information for the page. If the **Shared** bit is clear, the page is private.

This function reports on a region of pages in the memory of the calling process, and the [VirtualQueryEx](#) function reports on a region of pages in the memory of a specified process.

Requirements

Requirement	Value
Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	memoryapi.h (include Windows.h, Memoryapi.h)
Library	oncore.lib
DLL	Kernel32.dll

See also

[GetSystemInfo](#)

[MEMORY_BASIC_INFORMATION](#)

[MapViewOfFile](#)

[Memory Management Functions](#)

[Virtual Memory Functions](#)

[VirtualQueryEx](#)

[Vertdll APIs available in VBS enclaves](#)

VirtualQueryEx function (memoryapi.h)

Article 05/24/2022

Retrieves information about a range of pages within the virtual address space of a specified process.

Syntax

C++

```
SIZE_T VirtualQueryEx(  
    [in] HANDLE hProcess,  
    [in, optional] LPCVOID lpAddress,  
    [out] PMEMORY_BASIC_INFORMATION lpBuffer,  
    [in] SIZE_T dwLength  
);
```

Parameters

[in] hProcess

A handle to the process whose memory information is queried. The handle must have been opened with the **PROCESS_QUERY_INFORMATION** access right, which enables using the handle to read information from the process object. For more information, see [Process Security and Access Rights](#).

[in, optional] lpAddress

A pointer to the base address of the region of pages to be queried. This value is rounded down to the next page boundary. To determine the size of a page on the host computer, use the [GetSystemInfo](#) function.

If *lpAddress* specifies an address above the highest memory address accessible to the process, the function fails with **ERROR_INVALID_PARAMETER**.

[out] lpBuffer

A pointer to a [MEMORY_BASIC_INFORMATION](#) structure in which information about the specified page range is returned.

[in] dwLength

The size of the buffer pointed to by the *lpBuffer* parameter, in bytes.

Return value

The return value is the actual number of bytes returned in the information buffer.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#). Possible error values include **ERROR_INVALID_PARAMETER**.

Remarks

VirtualQueryEx provides information about a region of consecutive pages beginning at a specified address that share the following attributes:

- The state of all pages is the same (**MEM_COMMIT**, **MEM_RESERVE**, **MEM_FREE**, **MEM_PRIVATE**, **MEM_MAPPED**, or **MEM_IMAGE**).
- If the initial page is not free, all pages in the region are part of the same initial allocation of pages created by a single call to [VirtualAlloc](#), [MapViewOfFile](#), or one of the following extended versions of these functions: [VirtualAllocEx](#), [VirtualAllocExNuma](#), [MapViewOfFileEx](#), [MapViewOfFileExNuma](#).
- The access granted to all pages is the same (**PAGE_READONLY**, **PAGE_READWRITE**, **PAGE_NOACCESS**, **PAGE_WRITECOPY**, **PAGE_EXECUTE**, **PAGE_EXECUTE_READ**, **PAGE_EXECUTE_READWRITE**, **PAGE_EXECUTE_WRITECOPY**, **PAGE_GUARD**, or **PAGE_NOCACHE**).

The **VirtualQueryEx** function determines the attributes of the first page in the region and then scans subsequent pages until it scans the entire range of pages, or until it encounters a page with a nonmatching set of attributes. The function returns the attributes and the size of the region of pages with matching attributes, in bytes. For example, if there is a 40 megabyte (MB) region of free memory, and **VirtualQueryEx** is called on a page that is 10 MB into the region, the function will obtain a state of **MEM_FREE** and a size of 30 MB.

If a shared copy-on-write page is modified, it becomes private to the process that modified the page. However, the **VirtualQueryEx** function will continue to report such pages as **MEM_MAPPED** (for data views) or **MEM_IMAGE** (for executable image views) rather than **MEM_PRIVATE**. To detect whether copy-on-write has occurred for a specific page, either access the page or lock it using the [VirtualLock](#) function to make sure the page is resident in memory, then use the [QueryWorkingSet](#) or [QueryWorkingSetEx](#) function to check the **Shared** bit in the extended working set information for the page. If the **Shared** bit is clear, the page is private.

Requirements

Requirement	Value
Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	memoryapi.h (include Windows.h, Memoryapi.h)
Library	oncore.lib
DLL	Kernel32.dll

See also

[GetSystemInfo](#)

[MEMORY_BASIC_INFORMATION](#)

[MapViewOfFileEx](#)

[Memory Management Functions](#)

[Virtual Memory Functions](#)

[VirtualAllocEx](#)

[VirtualProtectEx](#)

VirtualUnlock function (memoryapi.h)

Article02/22/2024

Unlocks a specified range of pages in the virtual address space of a process, enabling the system to swap the pages out to the paging file if necessary.

Syntax

C++

```
BOOL VirtualUnlock(  
    [in] LPVOID lpAddress,  
    [in] SIZE_T dwSize  
);
```

Parameters

[in] lpAddress

A pointer to the base address of the region of pages to be unlocked.

[in] dwSize

The size of the region being unlocked, in bytes. The region of affected pages includes all pages containing one or more bytes in the range from the *lpAddress* parameter to `(lpAddress+dwSize)`. This means that a 2-byte range straddling a page boundary causes both pages to be unlocked.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call [GetLastError](#).

Remarks

For the function to succeed, the range specified need not match a range passed to a previous call to the [VirtualLock](#) function, but all pages in the range must be locked. If any of the pages in the specified range are not locked, **VirtualUnlock** removes such pages from the working set, sets last error to **ERROR_NOT_LOCKED**, and returns **FALSE**.

Calling **VirtualUnlock** on a range of memory that is not locked releases the pages from the process's working set.

Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps UWP apps]
Target Platform	Windows
Header	memoryapi.h (include Windows.h, Memoryapi.h)
Library	oncore.lib
DLL	Kernel32.dll

See also

[Memory Management Functions](#)

[Virtual Memory Functions](#)

[VirtualLock](#)

WIN32_MEMORY_RANGE_ENTRY structure (memoryapi.h)

Article02/22/2024

Specifies a range of memory. This structure is used by the [PrefetchVirtualMemory](#) function.

Syntax

C++

```
typedef struct _WIN32_MEMORY_RANGE_ENTRY {
    PVOID VirtualAddress;
    SIZE_T NumberOfBytes;
} WIN32_MEMORY_RANGE_ENTRY, *PWIN32_MEMORY_RANGE_ENTRY;
```

Members


VirtualAddress

NumberOfBytes

Remarks

To compile an application that calls this function, define `_WIN32_WINNT` as `_WIN32_WINNT_WIN8` or higher. For more information, see [Using the Windows Headers](#).

Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]
Header	memoryapi.h (include Windows.h, Memoryapi.h)

See also

[Memory Management Structures](#)

[PrefetchVirtualMemory](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#)  | [Get help at Microsoft Q&A](#)

WIN32_MEMORY_REGION_INFORMATION structure (memoryapi.h)

Article 04/02/2021

Contains information about a memory region. A memory region is a single allocation that is created using a memory allocation function, such as [VirtualAlloc](#) or [MapViewOfFile](#).

Syntax

C++

```
typedef struct WIN32_MEMORY_REGION_INFORMATION {
    PVOID AllocationBase;
    ULONG AllocationProtect;
    union {
        ULONG Flags;
        struct {
            ULONG Private : 1;
            ULONG MappedDataFile : 1;
            ULONG MappedImage : 1;
            ULONG MappedPageFile : 1;
            ULONG MappedPhysical : 1;
            ULONG DirectMapped : 1;
            ULONG Reserved : 26;
        } DUMMYSTRUCTNAME;
    } DUMMYUNIONNAME;
    SIZE_T RegionSize;
    SIZE_T CommitSize;
} WIN32_MEMORY_REGION_INFORMATION;
```

Members

AllocationBase

The base address of the allocation.

AllocationProtect

The page protection value that was specified when the allocation was created. Protections of individual pages within the allocation can be different from this value. To query protection values of individual pages, use the [VirtualQuery](#) function.

DUMMYUNIONNAME

DUMMYUNIONNAME.Flags

Represents all memory region flags as a single ULONG value. Applications should not use this field. Instead, test the individual bit field flags defined below.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.Private`

A value of 1 indicates that the allocation is private to the process.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.MappedDataFile`

A value of 1 indicates that the allocation is a mapped view of a data file.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.MappedImage`

A value of 1 indicates that the allocation is a mapped view of an executable image.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.MappedPageFile`

A value of 1 indicates that the allocation is a mapped view of a pagefile-backed section.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.MappedPhysical`

A value of 1 indicates that the allocation is a view of the **\Device\PhysicalMemory** section.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.DirectMapped`

A value of 1 indicates that the allocation is a mapped view of a direct-mapped file.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.Reserved`

Reserved.

`RegionSize`


The size of the allocation.

`CommitSize`

The commit charge associated with the allocation. For private allocations, this is the combined size of pages in the region that are committed, as opposed to reserved. For mapped views, this is the combined size of pages that have copy-on-write protection, or have been made private as a result of copy-on-write.


Remarks

The **WIN32_MEMORY_REGION_INFORMATION** structure contains information about a single memory allocation. In contrast, the **MEMORY_BASIC_INFORMATION** structure that is returned by the **VirtualQuery** function describes a contiguous run of pages within a single allocation that all have the same type, state, and protection. The mapping between **WIN32_MEMORY_REGION_INFORMATION** fields and memory type values returned by **VirtualQuery** is as follows:

 Expand table

WIN32_MEMORY_REGION_INFORMATION	MEMORY_BASIC_INFORMATION::Type
Private	MEM_PRIVATE
MappedDataFile	MEM_MAPPED
MappedImage	MEM_IMAGE
MappedPageFile	MEM_MAPPED
MappedPhysical	MEM_MAPPED

Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 10, version 1607 [desktop apps only]
Minimum supported server	Windows Server 2016 [desktop apps only]
Header	memoryapi.h (include Windows.h)

See also

[MEMORY_BASIC_INFORMATION](#)

[MapViewOfFile](#)

[VirtualAlloc](#)

[VirtualQuery](#)

WriteProcessMemory function (memoryapi.h)

Article 05/14/2022

Writes data to an area of memory in a specified process. The entire area to be written to must be accessible or the operation fails.

Syntax

C++

```
BOOL WriteProcessMemory(  
    [in] HANDLE hProcess,  
    [in] LPVOID lpBaseAddress,  
    [in] LPCVOID lpBuffer,  
    [in] SIZE_T nSize,  
    [out] SIZE_T *lpNumberOfBytesWritten  
);
```

Parameters

[in] hProcess

A handle to the process memory to be modified. The handle must have PROCESS_VM_WRITE and PROCESS_VM_OPERATION access to the process.

[in] lpBaseAddress

A pointer to the base address in the specified process to which data is written. Before data transfer occurs, the system verifies that all data in the base address and memory of the specified size is accessible for write access, and if it is not accessible, the function fails.

[in] lpBuffer

A pointer to the buffer that contains data to be written in the address space of the specified process.

[in] nSize

The number of bytes to be written to the specified process.

[out] lpNumberOfBytesWritten

A pointer to a variable that receives the number of bytes transferred into the specified process. This parameter is optional. If *lpNumberOfBytesWritten* is **NULL**, the parameter is ignored.

Return value

If the function succeeds, the return value is nonzero.


If the function fails, the return value is 0 (zero). To get extended error information, call [GetLastError](#). The function fails if the requested write operation crosses into an area of the process that is inaccessible.

Remarks

WriteProcessMemory copies the data from the specified buffer in the current process to the address range of the specified process. Any process that has a handle with **PROCESS_VM_WRITE** and **PROCESS_VM_OPERATION** access to the process to be written to can call the function. Typically but not always, the process with address space that is being written to is being debugged.

The entire area to be written to must be accessible, and if it is not accessible, the function fails.

Requirements

 [Expand table](#)

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	memoryapi.h (include Windows.h)
Library	oncore.lib
DLL	Kernel32.dll

See also

[Debugging Functions](#)

Process Functions for Debugging

ReadProcessMemory

VirtualAllocEx