Portable Systems Group

Windows NT Virtual Memory Specification

Author: *Lou Perazzoli*

Original Draft 1.0, December 15, 1988
Revision 4.0 April 28, 1993

Windows NT Virtual Memory Specification

Windows NT Virtual Memory Specification

1. Overview

This specification describes the virtual memory component for the portable New Technology (**Windows NT**) system. **Windows NT** virtual memory includes the following:

- Virtual memory support for the **POSIX** *fork* and *exec* operations, which enable compliance with the **POSIX** standard.
- Mapping of files into virtual memory and paging directly to/from those files. Files larger than 2 Gb are mapped via partial views of the file.
- Protection of shared memory and mapped files via Access Control Lists (**ACL**s), which is required to achieve a **DOD** security rating of **C2** or higher.
- Application control of virtual address space allocation and the mapping of shared memory.
- Copy-on-write pages with the ability to establish guard pages and set page protection.
- Creation of committed and/or reserved private memory without creating any kind of memory object.

1.1 Object Orientation

The basic architecture of the **Windows NT** system is object based. This means that all operating system abstractions presented at the API level are in the form of objects and a set of operations on those objects. This allows a stylized set of operations for each object, uniform naming across objects, uniform protection of objects, and uniform sharing of objects.

Typically there is an operation to create a new instance of an object (**NtCreate**_*object)* and to establish access (create a handle) for an existing object (**NtOpen**_*object)*. These basic operations are generally augmented by a set of object-specific operations. A handle is closed with a generic close operation (**NtClose**).

The treatment of objects here is minimal. A separate specification, *Windows NT Object Management,* more fully covers the object orientation of **Windows NT**.

1.2 Virtual Memory

Virtual memory is supported in the **Windows NT** system by section objects, a set of operations that may be performed on section objects, and various other services that directly manipulate the process virtual address space. In addition to section objects and their corresponding services, a set of operations are also provided to reserve and commit virtual memory private to a process.

A section object is a shareable entity that can be mapped into the virtual address space of a process. It can be backed by a paging file (e.g., demand zero pages) or by a file (mapped file).

Mapping a section into the virtual address space creates a process-private *view* of the section. The view can be partial or complete. Several different views of a section can be concurrently mapped within the same or different processes. When a view of a section is created, the corresponding process virtual address space is reserved and optionally committed.

Views are allocated on a hardware dependent **Allocation Granularity** (64kb on x86 and MIPS) virtual address boundary. The allocation granularity is determined by cache coherency issues and the desire to support larger page sizes in a compatible manner.

In general, it is not desirable for programs to directly control the allocation of the process virtual address space. However, the system has a need for this capability when a program is activated (i.e., the program file is mapped into the address space when the image is started), and some applications that use tag bits in pointers also want to control placement so that certain address bits are guaranteed to be zero. Thus the proposed interface provides for placement control, but it is optional.

Each section can have an optional name and Access Control List (**ACL**). This provides the basis whereby a section can be shared in a controlled manner. An unnamed section is a private section and can only be shared with another process via inheritance (i.e., the fork mechanism required to support **POSIX** compliance). Named sections can be shared by any other process that has access to the section.

The operating system does not use views as protection domains, and therefore never checks to ensure that an argument data structure resides within a single view. Thus an argument to a **Windows NT** service may span one or more views or private pages.

The default base address of all program images is the **Allocation Granularity** (zero-based program images do not allow uninitialized pointers to manifest themselves as access violations and are more difficult to debug). However, the base may be explicitly set to any desired value.

1.3 Page Protections

The virtual memory services allow the specification of *execute* access for page protections. On hardware which does not support execute access, the page protections for execute access will be treated as read. Therefore execute-only access would be treated as read-only, execute-read-write would be treated as read-write, etc. However, in the query operations, the actual set page protection would be returned.

1.4 Page File Quota and Commitment

The memory management system keeps track of page file usage on a global basis, termed *commitment*, and on a per process basis as *page file quota*. Commitment and page file quota are charged whenever virtual memory is created which requires backing store from the paging file.

The following explains the actions for each service which potentially creates pages destined for the paging file:

NtCreateSection (mapping file) __No commitment or page file quota is charged.

NtCreateSection (paging file) __Charge commitment for any committed pages within the section. The commitment is returned when the section is deleted.

NtAllocateVirtualMemory (reserve) __Charge commitment and page file quota for the page table pages required to map the potentially committed pages.

NtAllocateVirtualMemory (reserve & commit) __Charge commitment and page file quota for both the page table pages required to map the virtual memory and committed pages.

NtAllocateVirtualMemory (commit private pages) __Charge commitment and page file quota for each page of memory committed.

NtAllocateVirtualMemory (commit shared pages) __Charge page file quota and commitment if page protection is write-copy. Charge commitment if the page is within a view of a paging file backed section.

NtMapViewOfSection (mapping file) __Charge commitment and page file quota for the page table pages required to map the virtual memory. If the protection of the section is write-copy, charge page file quota and commitment for all pages in the view.

NtMapViewOfSection (paging file) __Charge page file quota as though all pages in the section are committed. Charge commitment and page file quota for the page table pages required to map the virtual memory. If the protection of the section is write-copy, charge page file quota and commitment for all pages in the view.

NtProtectVirtualMemory (within a view) __If the page protection is write-copy, charge commitment and page file quota for each newly protected page which is not already write-copy or private. If the page protection is not write-copy, and the previous page protection was write-copy, return the commitment and page file quota for that page.

NtFreeVirtualMemory __Returned the charged commitment and page file quota.

2. Virtual Memory Operations

The following subsections describe the virtual memory operations that can be performed in the **Windows NT** system. A definition and an explanation of each operation is given.

The **API**s described include:

NtCreateSection - Create section and open handle
NtOpenSection - Open handle to existing section
NtMapViewOfSection - Map view of section

NtExtendSection - Extend the size of section
NtUnmapViewOfSection - Unmap view of section
NtAllocateVirtualMemory - Commit/reserve region
NtFreeVirtualMemory - Decommit/release region
NtReadVirtualMemory - Read memory from specified process
NtWriteVirtualMemory - Write memory to specified process
NtFlushVirtualMemory - Flush modified pages to file
NtLockVirtualMemory - Lock region process/system
NtUnlockVirtualMemory - Unlock region process/system
NtProtectVirtualMemory - Protect region
NtQueryVirtualMemory - Get information about region
NtQuerySection - Get information about section
NtCreatePagingFile - Create a paging file
NtFlushInstructionCache - Flushes the instruction cache.
NtFlushWriteBuffer - Flushes the write buffer on the current processor.
NtClose - Close handle

Each **API** returns a status value (error code) that signifies the success or failure of the operation.

2.1 Create Section

A section object can be created and a handle opened for access to the section with the **NtCreateSection** function:

NTSTATUS
NtCreateSection (
OUT PHANDLE *SectionHandle,*
IN ACCESS_MASK *DesiredAccess,*
IN POBJECT_ATTRIBUTES *ObjectAttributes* **OPTIONAL**,
IN PLARGE_INTEGER *MaximumSize* **OPTIONAL**,
IN ULONG *SectionPageProtection,*
IN ULONG *AllocationAttributes,*
IN HANDLE *FileHandle* **OPTIONAL,**
);

Parameters:

SectionHandle - A pointer to a variable that will receive the section object handle value.

DesiredAccess - The desired types of access for the section. The following object type specific access flags can be specified in addition to the *STANDARD_ACCESS_REQUIRED* flags described in the Object Management Specification.

DesiredAccess Flags

SECTION_MAP_EXECUTE - Execute access to the section is desired.

SECTION_MAP_READ - Read access to the section is desired.

SECTION_MAP_WRITE - Write and read access to the section is desired.

SECTION_QUERY - Query access to the section is desired.

SECTION_EXTEND_SIZE - The ability to extend the size of the section is desired.

ObjectAttributes - An optional pointer to a structure that specifies the object's attributes. Refer to the *Object Management Specification* for details.

MaximumSize - A pointer to the maximum size of the section in bytes. For page file backed sections, this value is rounded up to the host page size. If this argument is unspecified or the value is specified as zero, and a file handle is specified, the section size is set to the size of the file.

SectionPageProtection - Specifies the underlying page protection for the section. For files mapped as images, this parameter is ignored and the underlying page protection is taken from the mapped file's image header.

Section Page Protection Values

PAGE_READONLY - Read access to the committed region of pages is allowed. An attempt to write or execute the committed region results in an access violation.

PAGE_READWRITE - Read, and write access to the region of committed pages are allowed. If write access to the underlying section is allowed, then a single copy of the pages is shared. Otherwise the pages are shared read-only/copy-on-write.

PAGE_WRITECOPY - Read and write access to the region of committed pages is allowed. The pages are shared read-only/copy-on-write.

PAGE_EXECUTE - Execute access to the committed region of pages is allowed. An attempt to read or write the committed region results in an access violation.

PAGE_EXECUTE_READ - Execute and read access to the region of committed pages is allowed. An attempt to write the committed region results in an access violation.

PAGE_EXECUTE_READWRITE - Execute, read and write access to the region of committed pages is allowed.

PAGE_EXECUTE_WRITECOPY - Read, execute, and write access to the region of committed pages is allowed. The pages are shared read-only/copy-on-write.

AllocationAttributes - A set of flags that describes the allocation attributes of the section. One of *SEC_RESERVE, SEC_COMMIT* or *SEC_IMAGE* must be

supplied. If *SEC_IMAGE* is specified the only other valid option is *SEC_BASED*.

AllocationAttributes Flags

SEC_BASED - The section is a based section. Attempt to find a location that allows mapping in the current process which does not conflict with other SEC_BASED sections. If a view to a SEC_BASED section cannot be mapped at the specified address in the process and error is returned. **NOTE: SEC_BASED does not prevent other mappings or allocations from colliding with the based section, it merely guarantees that either the section is mapped at the based address or an error is returned.**

SEC_RESERVE - All pages of the section are set to the reserved state.

SEC_COMMIT - All pages of the section are set to the commit state.

SEC_IMAGE - The file specified by the file handle is an executable image file.

SEC_NOCACHE - All pages of the section are to be set as non-cacheable.

FileHandle - An optional handle of an open file object. If the value of this handle is NULL, then the section is backed by a paging file. Otherwise, the section is backed by the specified data file.

Creating a section creates an object that describes a region of potentially shareable memory and opens a handle for access to the section object. The section can be backed by a paging file or a specified data file. An open section handle can be used to map a view of the section into the virtual address space of the subject process.

If the section is given a name, then it can be shared at any virtual address with other processes that can open the section (see **NtOpenSection** below). The section can also be specified as "based" in which case it can also be shared at a fixed address in all processes that map a view of the section.

If the section is shareable (i.e., it is given a name), then the Access Control List (**ACL**) specifies which users can access the section. If the section is not given a name, then only the creating process and its descendants can access the section.

Various object attributes can be chosen for the section such that access to the section can be inherited by the child process when a new process is created. This capability is required to support the **POSIX** standard.

The *OBJ_OPENIF* object attribute allows the section to be created if a section object with the specified name doesn't already exist. This is useful when two or more processes dynamically create a temporary section to hold shared data while one or more processes that operate on the shared data are active. If this option is specified and a section object with the same name already exists, then the desired access to

the section object by the subject process is verified and an open handle is returned for the existing object.

A section can be specified as temporary or permanent. A temporary object is deleted when the last open handle to the object is closed. This can result from closing the handle (see **NtClose** below) or by terminating a process. A permanent object is deleted by first opening a handle to the object, marking it temporary, and then closing the handle. The object then behaves much like a temporary object and is deleted when the last open handle is closed.

If the section is mapped by a file, then the **ACL** on the file is used to control access to the section unless the user ID of the subject process is the owner of the file, in which case the specified **ACL** is used. The desired access types must be allowed by the section **ACL** and must be compatible with the open mode of the file (i.e., write access is not allowed to a file that is opened for read-only access).

If the file is open for read-write access, then the file acts as backing store for both reads and writes of pages in the section. Otherwise, the file is used for inpaging and no outpaging to the file occurs (i.e., any modified pages are written to a paging file).

In addition to quota errors and object management errors associated with creating objects, the following status values may be returned by the function:

- *STATUS_NORMAL* - Normal, successful completion.
- *STATUS_INVALID_PARAMETER* - Error, an invalid parameter was specified.
- *STATUS_INVALID_PAGE_PROTECTION* - Error, an invalid page protection was specified.
- *STATUS_INVALID_FILE* - Error, an invalid file handle was specified.
- *STATUS_NOT_IMAGE* - Error, an attempt to map file as an image which is not an image file.
- *STATUS_SECTION_TOO_BIG* - Error, an attempt to map create a section which is bigger than the file which it backs.

2.2 Open Section

A handle can be opened for access to an existing section object with the **NtOpenSection** function:

NTSTATUS
NtOpenSection(
OUT PHANDLE *SectionHandle,*
IN ACCESS_MASK *DesiredAccess*,
IN POBJECT_ATTRIBUTES *ObjectAttributes*
);

Parameters:

Sectionhandle - A pointer to a variable that will receive the section object handle value.

DesiredAccess - The desired types of access for the section. The following object type specific access flags can be specified in addition to the *STANDARD_ACCESS_REQUIRED* flags described in the Object Management Specification.

DesiredAccess Flags

SECTION_MAP_EXECUTE - Execute access to the section is desired.

SECTION_MAP_READ - Read access to the section is desired.

SECTION_MAP_WRITE - Write and read access to the section is desired.

SECTION_QUERY - Query access to the section is desired.

SECTION_EXTEND_SIZE - The ability to extend the size of the section is desired.

ObjectAttributes - A pointer to a structure that specifies the object's attributes. Refer to the *Object Management Specification* for details.

Opening a section causes a handle for the object to be opened so that a view of the section can be mapped into the virtual address space of the subject process.

A process cannot open a section object unless the desired access types are allowed by the section object **ACL**, and, if the section is backed by a data file, are also compatible with the open mode of the associated data file.

In addition to quota errors and object management errors associated with opening objects, the following status values may be returned by the function:

- *STATUS_NORMAL* - Normal, successful completion.
- *STATUS_INVALID_PARAMETER* - Error, an invalid parameter was specified.

2.3 Map View Of Section

A view of a section can be mapped into the virtual address space of a subject process with the **NtMapViewOfSection** function:

NTSTATUS
NtMapViewOfSection(
IN HANDLE *SectionHandle,*
IN HANDLE *ProcessHandle,*
IN OUT PVOID **BaseAddress,*
IN ULONG *ZeroBits,*
IN ULONG *CommitSize,*
IN OUT PLARGE_INTEGER *SectionOffset* **OPTIONAL**,
IN OUT PULONG *ViewSize,*
IN SECTION_INHERIT *InheritDisposition,*
IN ULONG *AllocationType,*
IN ULONG *Protect*
);

Parameters:

SectionHandle - An open handle to a section object.

ProcessHandle - An open handle to a process object.

BaseAddress - A pointer to a variable that will receive the base address of the view. If the initial value of this argument is not NULL, then the view is allocated starting at the specified virtual address must be a multiple of the allocation granularity. If the initial value of this argument is NULL, then the operating system determines where to allocate the view using the information specified by the *ZeroBits* argument value and the section allocation attributes (i.e., SEC_BASED).

ZeroBits - The number of high-order address bits that must be zero in the base address of the section view. The value of this argument must be less than 21 and is only used when the operating system determines where to allocate the view (i.e., when *BaseAddress* is NULL).

CommitSize - The size of the initially committed region of the view in bytes. *CommitSize* is only meaningful for page-file backed sections, mapped sections, both data and image are always committed at section creation time and is ignored for mapped files. This value is rounded up to the next host-page-size boundary.

SectionOffset - Optionally supplies a pointer to the offset from the beginning of the section to the view in bytes. This value must be a multiple of allocation granularity. If the section was created with the *SEC_IMAGE*, this argument must be NULL.

ViewSize - A pointer to a variable that will receive the actual size in bytes of the view. If the value of this argument is zero, then a view of the section will be mapped starting at the specified section offset and continuing to the end of the section. Otherwise the initial value of this argument specifies the size of the view in bytes and is rounded up to the next host page size boundary.

InheritDisposition - A value that specifies how the view is to be shared by a child process created with a create process operation.

InheritDisposition Values

ViewShare - Inherit view and share a single copy of the committed pages with a child process using the current protection value.

ViewUnmap - Do not map the view into a child process.

AllocationType - A set of flags that describes the type of allocation that is to be performed for the specified region of pages.

AllocationType Flags

MEM_TOP_DOWN - The specified region is to be allocated from the highest portion of the address space possible based on the *ZeroBits* argument.

MEM_LARGE_PAGES - Only valid with physical memory mappings. The specified view should be mapped with the largest page size possible.

MEM_DOS_LIM - Only valid on x86, provided for DOS/VDM compatibility. Allows views to be mapped on 4kb boundaries rather than allocation granularity.

Protect - The protection desired for the region of initially committed pages.

Protect Values

PAGE_NOACCESS - No access to the committed region of pages is allowed. An attempt to read, write, or execute the committed region results in an access violation (i.e., a GP fault).

PAGE_READONLY - Read access to the committed region of pages is allowed. An attempt to write or execute the committed region results in an access violation.

PAGE_READWRITE - Read, and write access to the region of committed pages are allowed. If write access to the underlying section is allowed, then a single copy of the pages is shared. Otherwise the pages are shared read-only/copy-on-write.

PAGE_WRITECOPY - Read and write access to the region of committed pages is allowed. The pages are shared read-only/copy-on-write.

PAGE_EXECUTE - Execute access to the committed region of pages is allowed. An attempt to read or write the committed region results in an access violation.

PAGE_EXECUTE_READ - Execute and read access to the region of committed pages is allowed. An attempt to write the committed region results in an access violation.

PAGE_EXECUTE_READWRITE - Execute, read and write access to the region of committed pages is allowed.

PAGE_EXECUTE_WRITECOPY - Read, execute and write access to the region of committed pages is allowed. The pages are shared read-only/copy-on-write.

PAGE_GUARD - Protect the page with the underlying page protection, however, access to the region causes a "*guard page entered*" condition to be raised in the subject process. This value is only valid with one of the page protections except PAGE_NOACCESS.

Mapping a view of a section into the virtual address space of a subject process causes a region of the virtual address space to be reserved and, optionally, committed. The issuing process must have *PROCESS_VM_OPERATION* access to the subject process, and the following access to the section:

Protect Value	**Section Access Required**
PAGE_NOACCESS	*SECTION_MAP_READ*
PAGE_READONLY	*SECTION_MAP_READ*
PAGE_READWRITE	*SECTION_MAP_WRITE, SECTION_MAP_READ*
PAGE_WRITECOPY	*SECTION_MAP_READ*
PAGE_EXECUTE	*SECTION_MAP_EXECUTE*
PAGE_EXECUTE_READ	*SECTION_MAP_READ, SECTION_MAP_EXECUTE*
PAGE_EXECUTE_READWRITE	*SECTION_MAP_EXECUTE, SECTION_MAP_READ, SECTION_MAP_WRITE*
PAGE_EXECUTE_WRITECOPY	*SECTION_MAP_EXECUTE, SECTION_MAP_READ*

In addition to the section access, the specified page protection must be compatible with the *SectionPageProtection* specified when the section was created.

Desired View Protection	**Section Protection Required**
PAGE_NOACCESS	Any
PAGE_READONLY	Any except *PAGE_NOACCESS* and *PAGE_EXECUTE*
PAGE_READWRITE	*PAGE_READWRITE*, *PAGE_EXECUTE_READWRITE*
PAGE_WRITECOPY	Any except *PAGE_NOACCESS* and *PAGE_EXECUTE*
PAGE_EXECUTE	*PAGE_EXECUTE, PAGE_EXECUTE_READ, PAGE_EXECUTE_READWRITE, PAGE_EXECUTE_WRITECOPY*
PAGE_EXECUTE_READ	*PAGE_EXECUTE_READ*, *PAGE_EXECUTE_READWRITE, PAGE_EXECUTE_WRITECOPY*
PAGE_EXECUTE_READWRITE	*PAGE_EXECUTE_READWRITE*
PAGE_EXECUTE_WRITECOPY	*PAGE_EXECUTE_READ*, *PAGE_EXECUTE_READWRITE, PAGE_EXECUTE_WRITECOPY*

The view size and section offset determine the region of the section that is mapped into the virtual address space of the subject process. The commit size determines how much of the view is initially committed. The committed pages, if any, start at the beginning of the view and extend upward.

Several different views of a section can be concurrently mapped into the virtual address space of a process. Likewise, several different views of a section can also be concurrently mapped into the virtual address space of several processes.

If the operating system determines the virtual address allocation for the view (i.e. *BaseAddress* is NULL) and the section is based, then the region chosen is the one that was reserved when the section was created.

If the operating system determines the virtual address allocation for the view and the section is not based, then the allocation is such that the specified number of high-order address bits are zero in the base address of the view. This capability is provided so that applications that use address bits for tag bits need not explicitly manage the virtual-address-space allocation themselves.

If the operating system determines the virtual address allocation for the view, the *ViewSize* is zero, and the complete section has been mapped before, the returned based address will be the base address where the complete section is already mapped. This allows library routines to map complete views of sections without having to determine if the section has been previously mapped.

If the operating system does not determine the virtual address allocation (i.e., *BaseAddress* is not null), then an attempt is made to map the view starting at the specified base address and extending upward. If any page within this region is already reserved or committed, then the view cannot be mapped.

Committed pages are initialized with the specified protection value which must be compatible with the granted access to the section. Reserved pages are given a protection value of no access. Any attempt to access these pages results in an access violation unless another sharer has previously committed the pages.

The following status values may be returned by the function:

- *STATUS_NORMAL* - Normal, successful completion.
- *STATUS_INVALID_PARAMETER* - Error, an invalid parameter was specified.
- *STATUS_NO_ACCESS* - Error, access denied to specified object.
- *STATUS_NO_QUOTA* - Error, insufficient quota to create the specified section.
- *STATUS_NO_MEMORY* - Error, insufficient virtual memory to map specified view.
- *STATUS_SECTION_PROTECTION* - Error, the specified protection is not compatible with the underlying section protection.
- *STATUS_INVALID_PAGE_PROTECTION* - Error, an invalid page protection was specified.
- *STATUS_CONFLICTING_ADDRESSES* - Error, the specified address range conflicts with an existing address range.

2.4 Extend Size Of Section

An existing section which maps a data file can be extended with the **NtExtendSection** function:

NTSTATUS
NtExtendSection (
    IN HANDLE *SectionHandle*,
    IN OUT PLARGE_INTEGER *NewSectionSize*
    );

Parameters:

SectionHandle - An open handle to a section object that maps a data file. *SECTION_EXTEND_SIZE* access to this handle is required.

NewSectionSize - A pointer to a variable that supplies the new size for the section. This variable receives the new size of the section. If the specified size is less than the current size, this variable receives the current size.

The extend section service allows a user to extend the size of a section that maps a data file. If the current size of the section is greater than the specified size, the section size is not changed and the current section size is written to the *NewSectionSize*.

If the current section size is less than the new section size, the current file allocation size is checked and if the file allocation size is greater than the specified new section size, the section is extended.

If, however, the file allocation size is less than the specified section size, an attempt is made to set the file allocation size to the specified new section size. If this succeeds, the section is extended. If this fails the section size is unchanged and the returned status indicates why the file allocation could not be increased.

2.5 Unmap View Of Section

A view of a section can be unmapped from the virtual address space of a subject process with the **NtUnmapViewOfSection** function:

NTSTATUS
NtUnmapViewOfSection(
    IN HANDLE *ProcessHandle*,
    IN PVOID *BaseAddress*
    );

Parameters:

ProcessHandle - An open handle to a process object.

BaseAddress - A virtual address within the view which is to be unmapped.

The entire view of the section specified by the base address parameter is unmapped from the virtual address space of the specified process. The base address argument may be any virtual address within the view. The issuing process must have *PROCESS_VM_OPERATION* access to the subject process.

The virtual address region occupied by the view is no longer reserved and is available to map other views or private pages. If the view was also the last reference to the underlying section (i.e., no open handles exist to the section object), then all committed pages in the section are decommitted and the section is deleted.

The following status values may be returned by the function:

- *STATUS_NORMAL* - Normal, successful completion.
- *STATUS_INVALID_PARAMETER* - Error, an invalid parameter was specified.
- *STATUS_NO_ACCESS* - Error, access denied to specified object.

2.6 Allocate Virtual Memory

A region of pages within the virtual address space of a subject process can be reserved and/or committed with the **NtAllocateVirtualMemory** function:

NTSTATUS
NtAllocateVirtualMemory(
    IN HANDLE *ProcessHandle,*
    IN OUT PVOID **BaseAddress,*
    IN ULONG *ZeroBits,*
    IN OUT PULONG *RegionSize,*
    IN ULONG *AllocationType,*
    IN ULONG *Protect*
    );

Parameters:

ProcessHandle - An open handle to a process object.

BaseAddress - A pointer to a variable that will receive the base address of the allocated region of pages. If the initial value of this argument is not NULL and the memory is being reserved, then the region is allocated starting at the specified virtual address rounded down to the next 64K byte boundary. If the memory is already reserved and is being committed, this value is rounded down to a host-page-size boundary. If the initial value of this argument is NULL, then the operating system determines where to allocate the region.

ZeroBits - The number of high-order address bits that must be zero in the base address of the section view. The value of this argument must be less than 21 and is only used when the operating system determines where to allocate the view (i.e., when *BaseAddress* is NULL).

RegionSize - A pointer to a variable that will receive the actual size in bytes of the allocated region of pages. The initial value of this argument specifies the size in bytes of the region and is rounded up to the next host-page-size boundary.

AllocationType - A set of flags that describes the type of allocation that is to be performed for the specified region of pages. One of *MEM_COMMIT* or *MEM_RESERVED* is required, both are acceptable (i.e., MEM_COMMIT | MEM_RESERVE).

AllocationType Flags

MEM_COMMIT - The specified region of pages is to be committed.

MEM_RESERVE - The specified region of pages is to be reserved.

MEM_TOP_DOWN - The specified region is to be allocated from the highest portion of the address space possible based on the *ZeroBits* argument.

Protect - The protection desired for the committed region of pages.

Protect Values

PAGE_NOACCESS - No access to the committed region of pages is allowed. An attempt to read, write, or execute the committed region results in an access violation (i.e., a GP fault).

PAGE_READONLY - Read access to the committed region of pages is allowed. An attempt to write or execute the committed region results in an access violation.

PAGE_READWRITE - Read and write access to the committed region of pages is allowed. If write access to the underlying section is allowed, then a single copy of the pages are shared. Otherwise, the pages are shared read-only/copy-on-write.

PAGE_WRITECOPY - Read and write access to the region of committed pages is allowed. The pages are shared read-only/copy-on-write.

PAGE_EXECUTE - Execute access to the committed region of pages is allowed. An attempt to read or write the committed region results in an access violation.

PAGE_EXECUTE_READ - Execute and read access to the region of committed pages is allowed. An attempt to write the committed region results in an access violation.

PAGE_EXECUTE_READWRITE - Execute, read and write access to the region of committed pages is allowed.

PAGE_EXECUTE_WRITECOPY - Read, execute, and write access to the region of committed pages is allowed. The pages are shared read-only/copy-on-write.

PAGE_GUARD - Protect the page with the underlying page protection, however, access to the region causes a "*guard page entered*" condition to be raised in the subject process. This value is only valid with one of the page protections except PAGE_NOACCESS.

PAGE_NOCACHE - Disable the placement of committed pages into the data cache. This is only valid for pages which are not contained within a view of a section. For pages which are contained in a view of a section, the nocache attribute may be specified when the section is created, in which case it cannot be changed. This value is only valid with one of the page protections except PAGE_NOACCESS.

This function can be used to commit a region of previously reserved pages (i.e., from a mapped view or a previous call to this function), to reserve a region of private pages, or to reserve and commit a region of private pages. This function also can be used to create a sparse population of committed private or mapped pages. The issuing process must have *PROCESS_VM_OPERATION* access to the subject process.

If the initial value of the base address parameter is NULL, then the operating system allocates a region of private pages large enough to fulfill the specified allocation request from the virtual address space of the subject process. The base address of this region is returned in the base address parameter. Private pages are given an inherit disposition of equivalent to *ViewShare*.

Process address map entries are scanned from the base address upward until the entire range of pages can be allocated or a failure occurs. If the entire range cannot be allocated, an appropriate status value is returned and no pages are mapped.

Each page in the process virtual address space is either private or mapped into a view of a section. Private pages can be in one of three states:

1. Free - Not committed or reserved, and inaccessible

2. Committed - Allocated backing storage with access controlled by a protection code

3. Reserved - Reserved, not committed, and inaccessible

Pages that are mapped into a view of a section can be in one of two states:

1. Committed - Allocated backing storage with access controlled by a protection code

2. Reserved - Reserved, not committed, and inaccessible, but can be auto-committed if an access to the page is attempted and the page has already been committed in the section mapped by the view (i.e., the page has been committed by another sharer of the section)

As each page is considered for allocation, its state and whether it is a private or mapped page is determined. Private pages are handled as follows:

1. Free - A private page that is free can be reserved and/or committed.

2. Committed - A private page that is already committed is left unchanged (i.e., it is still committed and its protection is not changed).

3. Reserved - A private page that is reserved can be committed. An attempt to reserve a page already in the reserved state has no effect.

Pages that are mapped into a view of a section are handled as follows:

1. Committed - A mapped page that is already committed cannot be changed to reserved. A shared page that is already committed is unchanged, however, in certain cases it's protection may be changed. This is due to the fact that shared pages, even though committed, may not be active in the process and hence have the original protection of the mapping. In committing the page the mapping state is not checked on a page by page basis.

2. Reserved - A mapped page that is reserved can be committed.

The protection value applied to committed pages that are contained within a mapped view of a section must be compatible with the access granted to the underlying section. Note that the underlying protection of the section does not change, only the specified pages contained in the process's view. Any protection value can be applied to committed private pages. Reserved pages are given a protection value of no access.

Pages that are backed by a paging file are committed as demand-zero pages (i.e., the first attempt to read or write the page causes a page of zeros to be created). Pages that are backed by a data file are committed such that they map pages of the data file.

The following status values may be returned by the function:

- *STATUS_NORMAL* - Normal, successful completion.

- *STATUS_INVALID_PARAMETER* - Error, an invalid parameter was specified.

- *STATUS_INVALID_PAGE_PROTECTION* - Error, an invalid page protection was specified.

- *STATUS_CONFLICTING_ADDRESSES* - Error, the specified address range conflicts with an existing address range.

- *STATUS_NO_ACCESS* - Error, access denied to specified object.

2.7 Free Virtual Memory

A region of pages within the virtual address space of a subject process can be decommitted and/or released with the **NtFreeVirtualMemory** function:

NTSTATUS
NtFreeVirtualMemory(
    IN HANDLE *ProcessHandle,*
    IN OUT PVOID **BaseAddress,*
    IN OUT PULONG *RegionSize,*
    IN ULONG *FreeType*
    );

Parameters:

ProcessHandle - An open handle to a process object.

BaseAddress - A pointer to a variable that will receive the base address of the region of pages to be freed. The initial value of this argument is the base address of the region of pages to be freed. This value is rounded down to the next host-page-address boundary.

RegionSize - A pointer to a variable that will receive the actual size in bytes of the freed region of pages. The initial value of this argument is rounded up to the next host-page-size boundary. If this value is zero and the *BaseAddress* is the starting address of the allocated region, the complete range of pages allocated together is freed or decommitted.

FreeType - A set of flags that describes the type of free that is to be performed for the specified region of pages. One of the following:

FreeType Flags

MEM_DECOMMIT - The specified region of pages is to be decommitted.

MEM_RELEASE - The specified region of pages is to be released.

This function can be used to decommit a region of previously committed pages (i.e. from a mapped view or from an allocation of virtual memory), to release a region of previously reserved private pages, and to decommit and release a region of previously committed private pages. The issuing process must have *PROCESS_VM_OPERATION* access to the subject process.

Process address map entries are scanned from the base address upward until the entire range of pages can be freed or until a failure occurs. If the entire range cannot be freed, an appropriate status value is returned and no pages are freed.

As each page is considered for deallocation, its state and whether it is a private or mapped page is determined. Private pages are handled as follows:

1. Free - A private page that is free cannot be released or decommitted.

2. Committed - A private page that is committed can be released and/or decommitted.

3. Reserved - A private page that is reserved can be released or decommitted. Decommitting a reserved page leaves the page in the reserved state.

Pages that are mapped into a view of a section are handled as follows:

1. Committed - A mapped page that is committed cannot be decommitted or released.

2. Reserved - A mapped page that is reserved cannot be decommitted or released.

If the desired type of free is allowed for the specified *RegionSize*, then page attributes are established as necessary in the process address map, and the current length of the freed region is updated.

If the desired type of free cannot be performed on the entire range, then an appropriate status value is returned and none of the specified region is freed.

Decommitting a private page causes the backing storage for the page to be released to the appropriate paging file and the address map entry for the corresponding page to be returned to the reserved state.

Decommitted and released pages are given a protection value of no access.

The following status values may be returned by the function:

- *STATUS_NORMAL* - Normal, successful completion.

- *STATUS_INVALID_PARAMETER* - Error, an invalid parameter was specified.

- *STATUS_NO_ACCESS* - Error, access denied to specified object.

- *STATUS_UNABLE_TO_FREE_VM* - Error, the specified virtual memory could not be released. This could be caused by the virtual memory being a system structure (TEB or PEB) or being part of a mapped view, or the specified size larger than the original allocation.

- *STATUS_VM_NOT_AT_BASE* - Error, the region size was specified as zero, but the starting address was not the beginning of the allocation.

- *STATUS_MEMORY_NOT_ALLOCATED* - Error, no memory as been allocated at the specified base address.

2.8 Read Virtual Memory

Data can be read from the address space of another process with the **NtReadVirtualMemory** function:

NTSTATUS
NtReadVirtualMemory(
IN HANDLE *ProcessHandle,*
IN PVOID *BaseAddress,*
OUT PVOID *Buffer,*
IN ULONG *BufferSize,*
OUT PULONG *NumberOfBytesRead* **OPTIONAL**
);

Parameters:

ProcessHandle - An open handle to a process object.

BaseAddress - The base address in the specified process of the region of pages to be read.

Buffer - The address of a buffer that receives the contents from the specified process address space.

BufferSize - The requested number of bytes to read from the specified process.

NumberOfBytesRead - Receives the actual number of bytes transferred into the specified buffer.

This function reads data from the base address in the specified process and places the data in the specified buffer. The *NtReadVirtualMemory* function probes both the input and the output buffers before any bytes are copied. If either the *Buffer* fails a probe for write, or the *BaseAddress* fails a probe for read, the function returns an error and the *NumberOfBytesRead* parameter is returned as zero.

If the probe operations are successful, an attempt is made to copy the number of bytes specified in *BufferSize*. The *NumberOfBytesRead* parameter returns the actual number of bytes copied from the specified process into the buffer. The issuing process must have *PROCESS_VM_READ* access to the subject process.

The following status values may be returned by the function:

- *STATUS_NORMAL* - Normal, successful completion.
- *STATUS_PARTIAL_COPY* - Warning, due to protection conflicts not all the requested bytes could be copied.
- *STATUS_INVALID_PARAMETER* - Error, an invalid parameter was specified.
- *STATUS_NO_ACCESS* - Error, access denied to specified object.
- *STATUS_ACCESS_VIOLATION* - Error, one of the memory regions was not completely accessible.

2.9 Write Virtual Memory

Data can be written to the address space of another process with the **NtWriteVirtualMemory** function:

NTSTATUS
NtWriteVirtualMemory(
    IN HANDLE *ProcessHandle,*
    OUT PVOID *BaseAddress,*
    IN PVOID *Buffer,*
    IN ULONG *BufferSize,*
    OUT PULONG *NumberOfBytesWritten* **OPTIONAL**
    );

Parameters:

ProcessHandle - An open handle to a process object.

BaseAddress - The base address in the specified process of the region of pages to be written.

Buffer - The address of a buffer that contains the contents to be written into the specified process address space.

BufferSize - The requested number of bytes to write into the specified process.

NumberOfBytesWritten - Receives the actual number of bytes transferred into the specified address space.

This function writes data from the specified buffer in the current process to the specified base address in the specified process. The *NtWriteVirtualMemory* function probes both the input and the output buffers before any bytes are copied. If either the *Buffer* fails a probe for read, or the *BaseAddress* fails a probe for write, the function returns an error and the *NumberOfBytesRead* parameter is returned as zero.

If the probe operations are successful, An attempt is made to copy the number of bytes specified in *BufferSize*. The *NumberOfBytesWritten* parameter returns the actual number of bytes copied from the buffer to the specified process. The issuing process must have *PROCESS_VM_WRITE* access to the subject process.

The following status values may be returned by the function:

- *STATUS_NORMAL* - Normal, successful completion.
- *STATUS_PARTIAL_COPY* - Warning, due to access violations not all the requested bytes could be copied.
- *STATUS_INVALID_PARAMETER* - Error, an invalid parameter was specified.
- *STATUS_NO_ACCESS* - Error, access denied to specified object.

- *STATUS_ACCESS_VIOLATION* - Error, one of the memory regions was not completely accessible.

2.10 Flush Virtual Memory

A region of pages within the virtual address space of a subject process can be forced to be written back into the corresponding data file (if they have been modified since they were last written) with the **NtFlushVirtualMemory** function:

NTSTATUS
NtFlushVirtualMemory(
    IN HANDLE *ProcessHandle,*
    IN OUT PVOID **BaseAddress,*
    IN OUT PULONG *RegionSize,*
    OUT PIO_STATUS_BLOCK *IoStatus*
    );

Parameters:

ProcessHandle - An open handle to a process object.

BaseAddress - A pointer to a variable that will receive the base address of the region of pages to flush. The initial value of this argument is the base address of the region of pages to flush. This value is rounded down to the next host-page-address boundary.

RegionSize - A pointer to a variable that will receive the actual size in bytes of the flushed region of pages. The initial value of this argument is rounded up to the next host-page-size boundary. If the *RegionSize* is specified as 0, the range from the base address until the last address mapped in this view is flushed.

IoStatus - A pointer to an I/O status block that receives the I/O status from the last page written.

Process address map entries are scanned from the base address upward until the entire specified range of pages has been flushed. The actual size of the flushed region and an appropriate status value are returned. The issuing process must have *PROCESS_VM_OPERATE* access to the subject process.

As each page is considered for flushing, its state is determined. If the page is committed, mapped into a view of a section that is backed by a data file, and has been modified in memory but not yet written back into the file, then a write of the modified page is initiated. Otherwise, no operation is performed on the page.

This function can be used to ensure that a consistent state of the data within a file is maintained in the presence of various sequences of updates (e.g., forced writes of log pages, etc.).

If an I/O error occurs while writing pages, the *RegionSize* contains the starting virtual address of the write which failed, and the *IoStatus* contains the failure status.

The following status values may be returned by the function:

- *STATUS_NORMAL* - Normal, successful completion.
- *STATUS_INVALID_PARAMETER* - Error, an invalid parameter was specified.
- *STATUS_NO_ACCESS* - Error, access denied to specified object.
- Any errors possible from an NtWriteFile service.

2.11 Lock Virtual Memory

A region of pages within the virtual address space of a subject process can be locked for process residency and/or system residency with the **NtLockVirtualMemory** function:

```
NTSTATUS
NtLockVirtualMemory(
    IN HANDLE ProcessHandle,
    IN OUT PVOID *BaseAddress,
    IN OUT PULONG RegionSize,
    IN ULONG MapType
    );
```

Parameters:

ProcessHandle - An open handle to a process object.

BaseAddress - A pointer to a variable that will receive the base address of the region of pages to lock. The initial value of this argument is the base address of the region of pages to lock. This value is rounded down to the next host-page-address boundary.

RegionSize - A pointer to a variable that receives the actual size in bytes of the locked region of pages. The initial value of this argument is rounded up to the next host-page-size boundary.

MapType - The map type flags.

MapType Flags

MAP_PROCESS - Process residency

MAP_SYSTEM - System residency

Locking a region of pages in an address map causes the residency attributes of the corresponding pages to be set such that they are not eligible for paging.

Locking a page for system residency causes the page to remain memory resident until it is explicitly unlocked. A special privilege is required in a server system to lock a page for system residency.

Locking a page for process residency causes the page to remain memory resident while the subject process is a member of the balance set (i.e., the set of processes that are actively being considered for execution).

Note that changing the protection of a locked page to *PAGE_NOACCESS* or *PAGE_GUARD* causes the page to become unlocked. In addition, locked pages are not inherited as locked, they are unlocked in the new process.

If the entire *RegionSize* cannot be locked, an appropriate status code is returned and none of the pages is locked. The issuing process must have *PROCESS_VM_OPERATE* access to the subject process.

The following status values may be returned by the function:

- *STATUS_NORMAL* - Normal, successful completion.
- *STATUS_WAS_LOCKED* - Warning, at least one of the pages in the specified region was already locked.
- *STATUS_INVALID_PARAMETER* - Error, an invalid parameter was specified.
- *STATUS_NO_ACCESS* - Error, access denied to specified object.
- *STATUS_NO_QUOTA* - Error, insufficient quota to lock the specified region.

2.12 Unlock Virtual Memory

A region of pages within the virtual address space of a subject process can be unlocked from process residency and/or system residency with the **NtUnlockVirtualMemory** function:

```
NTSTATUS
NtUnlockVirtualMemory(
    IN HANDLE ProcessHandle,
    IN OUT PVOID *BaseAddress,
    IN OUT PULONG RegionSize,
    IN ULONG MapType
    );
```

Parameters:

ProcessHandle - An open handle to a process object.

BaseAddress - A pointer to a variable that will receive the base address of the region of pages to unlock. The initial value of this argument is the base address of the region of pages to unlock. This value is rounded down to the next host-page-address boundary.

RegionSize - A pointer to a variable that receives the actual size in bytes of the unlocked region of pages. The initial value of this argument is rounded up to the next host-page-size boundary.

MapType - The map type flags.

MapType Flags

MAP_PROCESS - Process residency

MAP_SYSTEM - System residency

Unlocking a region of pages causes the residency attributes of the corresponding pages to be set such that they are eligible for paging.

Unlocking a process-resident page causes the page to become pageable until it is explicitly locked.

Unlocking a system-resident page causes the page to become pageable until it is explicitly locked. A special privilege is required in a server system to unlock a system-resident page.

If the entire *RegionSize* cannot be unlocked, an appropriate status code is returned and none of the pages is unlocked. The issuing process must have *PROCESS_VM_OPERATE* access to the subject process.

The following status values may be returned by the function:

- *STATUS_NORMAL* - Normal, successful completion.
- *STATUS_INVALID_PARAMETER* - Error, an invalid parameter was specified.
- *STATUS_NO_ACCESS* - Error, access denied to specified object.

2.13 Protect Virtual Memory

The protection on a region of committed pages within the virtual address space of the subject process can be changed with the **NtProtectVirtualMemory** function:

NTSTATUS
NtProtectVirtualMemory(
    IN HANDLE *ProcessHandle,*
    IN OUT PVOID **BaseAddress,*
    IN OUT PULONG *RegionSize,*
    IN ULONG *NewProtect,*
    OUT PULONG *OldProtect*
    );

Parameters:

ProcessHandle - An open handle to a process object.

BaseAddress - A pointer to a variable that will receive the actual base address of the protected region of pages. The initial value of this argument is rounded down to the next host-page-address boundary.

RegionSize - A pointer to a variable that will receive the actual size in bytes of the protected region of pages. The initial value of this argument is rounded up to the next host-page-size boundary.

NewProtect - The new protection desired for the specified region of pages.

NewProtect Values

PAGE_NOACCESS - No access to the specified region of pages is allowed. An attempt to read, write, or execute the specified region results in an access violation (i.e., a GP fault).

PAGE_READONLY - Read-access to the specified region of pages is allowed. An attempt to execute or write the specified region results in an access violation.

PAGE_READWRITE - Read and write access to the specified region of pages is allowed. If write access to the underlying section is allowed, then a single copy of the pages are shared. Otherwise, the pages are shared read-only/copy-on-write.

PAGE_WRITECOPY - Read and write access to the region of committed pages is allowed. The pages are shared read-only/copy-on-write. This value may only be specified for an address ranges which is within a mapped view of a section.

PAGE_EXECUTE - Execute access to the specified region of pages is allowed. An attempt to read or write the specified region results in an access violation.

PAGE_EXECUTE_READ - Execute and read access to the region of committed pages is allowed. An attempt to write the committed region results in an access violation.

PAGE_EXECUTE_READWRITE - Execute, read and write access to the region of committed pages is allowed.

PAGE_EXECUTE_WRITECOPY - Read, execute, and write access to the region of committed pages is allowed. The pages are shared read-only/copy-on-write. This value may only be specified for an address ranges which is within a mapped view of a section.

PAGE_GUARD - Protect the page with the underlying page protection, however, access to the region causes a "*guard page entered*" condition to be raised in the subject process. This value is only valid with one of the page protections except PAGE_NOACCESS.

PAGE_NOCACHE - Disable the placement of committed pages into the data cache. This value is only valid when specified in combination with one of the above underlying page protections with the exception of PAGE_NOACCESS, e.g., (*PAGE_NOCACHE* | *PAGE_READWRITE*). The *PAGE_NOCACHE* attribute may not be specified on an address range which is within a mapped view of a section.

OldProtect - A pointer to a variable that will receive the old protection of the first page within the specified region of pages.

Setting the protection on a range of pages causes the old protection value to be replaced by a new protection value. The protection value can only be set on committed pages. The issuing process must have *PROCESS_VM_OPERATE* access to the subject process.

Note that setting page protections to *PAGE_NOACCESS* or *PAGE_GUARD* on a page which is locked in memory or locked in the process causes the locked page to become unlocked.

Setting the protection value to *PAGE_GUARD* causes guard pages to be established. If an access to a guard page is attempted, then the protection of the accessed page to be set to its declared access, and "guard page entered" condition is raised. This capability is intended to provide automatic stack checking, but can also be used to separate other data structures where appropriate.

As each page is considered for protecting, its state is determined. If the state of the page is not committed, the page is reserved and cannot be auto-committed, or the page is contained within a mapped view of a section and the granted access to the section is incompatible with the new protection, then an appropriate status value is returned and none of the pages in the specified region is modified.

The following status values may be returned by the function:

- *STATUS_NORMAL* - Normal, successful completion.
- *STATUS_WAS_UNLOCKED* - Warning, at least one of the pages in the specified region was unlocked due to a page protection of PAGE_NOACCESS or PAGE_GUARD.
- *STATUS_INVALID_PARAMETER* - Error, an invalid parameter was specified.
- *STATUS_INVALID_PAGE_PROTECTION* - Error, an invalid page protection was specified.
- *STATUS_NO_ACCESS* - Error, access denied to specified object.
- *STATUS_NOT_COMMITTED* - Error, some pages within the range are not committed.
- *STATUS_IS_WRITECOPY* - Warning, the protection of the region was set to *PAGE_WRITECOPY* due to the underlying nature of the section.

2.14 Query Virtual Memory

Information about a range of pages within the virtual address space of the subject process can be obtained with the **NtQueryVirtualMemory** function:

NTSTATUS
NtQueryVirtualMemory(
    IN HANDLE *ProcessHandle,*
    IN PVOID *BaseAddress,*
    IN MEMORY_INFORMATION_CLASS *MemoryInformationClass,*
    OUT PVOID *MemoryInformation,*
    IN ULONG *MemoryInformationLength,*
    OUT PULONG *ReturnLength* **OPTIONAL**
    );

Parameters:

ProcessHandle - An open handle to a process object.

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

MemoryInformationClass - The memory information class about which to retrieve information.

MemoryInformation - A pointer to a buffer that receives the specified information. The format and content of the buffer depend on the specified information class.

MemoryInformation Format by Information Class:

MemoryBasicInformation - Data type is *PMEMORY_BASIC_INFORMATION*.

MEMORY_BASIC_INFORMATION Structure

PVOID *BaseAddress* - The base address of the region.

PVOID *AllocationBase* - The allocation base of the allocation this page is contained within.

ULONG *AllocationProtect* - The protection specified when the region was initially allocated.

ULONG *RegionSize* - The size of the region in bytes beginning at the base address in which all pages have identical attributes.

ULONG *State* - The state of the pages within the region.

State Values

MEM_COMMIT - The state of the pages within the region is committed.

MEM_FREE - The state of the pages within the region is free.

MEM_RESERVE - The state of the pages within the region is reserved.

If the memory state is *MEM_FREE* other the *AllocationBase, AllocationProtect, Protect* and *Type* fields in the information are undefined.

If the memory state is *MEM_RESERVE* the information in the *Protect* field is undefined.

ULONG *Protect* - The protection of the pages within the region.

Protect Values

PAGE_NOACCESS - No access to the region of pages is allowed. An attempt to read, write, or execute within the region results in an access violation (i.e., a GP fault).

PAGE_READONLY - Read-access to the region of pages is allowed. An attempt to execute or write within the region results in an access violation.

PAGE_READWRITE - Read and write access to the region of pages is allowed. If write access to the underlying section is allowed, then a single copy of the pages are shared. Otherwise, the pages are shared read-only/copy-on-write.

PAGE_WRITECOPY - Read and write access to the region of committed pages is allowed. The pages are shared read-only/copy-on-write.

PAGE_EXECUTE - Execute access to the region of pages is allowed. An attempt to read or write within the region results in an access violation.

PAGE_EXECUTE_READ - Execute and read access to the region of committed pages is allowed. An attempt to write the committed region results in an access violation.

PAGE_EXECUTE_READWRITE - Execute, read and write access to the region of committed pages is allowed.

PAGE_EXECUTE_WRITECOPY - Read, execute and write access to the region of committed pages is allowed. The pages are shared read-only/copy-on-write.

PAGE_GUARD - Protect the page with the underlying page protection, however, access to the region causes a "*guard page entered*" condition to be raised in the subject process. This value is only valid with one of the page protections except PAGE_NOACCESS.

PAGE_NOCACHE - Disable the placement of committed pages into the data cache. This value is only valid with one of the other page protections except PAGE_NOACCESS.

ULONG *Type* - The type of pages within the region.

Type Values

MEM_PRIVATE - The pages within the region are private.

MEM_MAPPED - The pages within the region are mapped into the view of a section.

MEM_IMAGE - The pages within the region are mapped into the view of an image section.

MemoryInformationLength - Specifies the length in bytes of the memory information buffer.

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

This function provides the capability to determine the state, protection, and type of a region of pages within the virtual address space of the subject process. The issuing process must have *PROCESS_QUERY_INFORMATION* access to the subject process.

The state of the first page within the region is determined and then subsequent entries in the process address map are scanned from the base address upward until either the entire range of pages has been scanned or until a page with a nonmatching set of attributes is encountered. The region attributes, the length of the region of pages with matching attributes, and an appropriate status value are returned.

The following status values may be returned by the function:

- *STATUS_NORMAL* - Normal, successful completion.
- *STATUS_INVALID_PARAMETER* - Error, an invalid parameter was specified.
- *STATUS_NO_ACCESS* - Error, access denied to specified object.
- *STATUS_INFO_LENGTH_MISMATCH* - Error, the specified buffer size is not large enough to hold the requested information.

- *STATUS_INVALID_INFO_CLASS* - Error, the specified information class is not valid for this service.

2.15 Query Section Information

Information about a section can be obtained with the **NtQuerySection** function:

NTSTATUS
NtQuerySection(
IN HANDLE *SectionHandle,*
IN SECTION_INFORMATION_CLASS *SectionInformationClass,*
OUT PVOID *SectionInformation,*
IN ULONG *SectionInformationLength,*
OUT PULONG *ReturnLength* **OPTIONAL**
);

Parameters:

SectionHandle - An open handle to a section object.

SectionInformationClass - The section information class about which to retrieve information.

SectionInformation - A pointer to a buffer that receives the specified information. The format and content of the buffer depend on the specified section class.

SectionInformation Format by Information Class:

SectionBasicInformation - Data type is *PSECTION_BASIC_INFORMATION.*.

SECTION_BASIC_INFORMATION Structure

PVOID *BaseAddress* - The base virtual address of the section if the section is based.

ULONG *AllocationAttributes* - The allocation attributes flags.

AllocationAttributes Flags

SEC_BASED - The section is a based section.

SEC_FILE - The section is backed by a data file.

SEC_RESERVE - All pages of the section were initially set to the reserved state.

SEC_COMMIT - All pages of the section were initially set to the committed state.

SEC_IMAGE - The section was mapped as an executable image file.

SEC_NOCACHE - All pages of the section are to be set as non-cacheable.

LARGE_INTEGER *MaximumSize* - The maximum size of the section in bytes.

SectionImageInformation - Data type is *PSECTION_IMAGE_INFORMATION*.

SECTION_IMAGE_INFORMATION Structure

PVOID *TransferAddress* - The transfer address of the image.

ULONG *ZeroBits* - The zero bits requirement for the creation of the stack.

ULONG *MaximumStackSize* - The maximum stack size required by the image.

ULONG *CommittedStackSize* - The amount of stack space to initially commit.

ULONG *SubSystemType* - Subsystem image is linked for.

ULONG *SubSystemVersion* - Subsystem version number.

ULONG *GpValue* - The value for the global pointer register.

USHORT *ImageCharacteristics* - Image characteristics.

USHORT *DllCharacteristics* - Dll characteristics.

USHORT *Machine*- Hardware platform image was built for.

USHORT *Spare1* - unused.

ULONG *LoaderFlags* - Flags specified in image for loader usage.

SectionInformationLength - Specifies the length in bytes of the section information buffer.

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

This function provides the capability to determine the base address, size, granted access, and allocation of an opened section object. The issuing process must have *SECTION_QUERY* access to the specified section.

The following status values may be returned by the function:

- *STATUS_NORMAL* - Normal, successful completion.
- *STATUS_INVALID_PARAMETER* - Error, an invalid parameter was specified.
- *STATUS_NO_ACCESS* - Error, access denied to specified object.
- *STATUS_INFO_LENGTH_MISMATCH* - Error, the specified buffer size is not large enough to hold the requested information.
- *STATUS_INVALID_INFO_CLASS* - Error, the specified information class is not valid for this service.
- *STATUS_SECTION_NOT_IMAGE* - Error, attempt to get image information on a section which does not map an image.

2.16 Create Paging File

An existing file can be declared as a paging file with the **NtCreatePagingFile** function:

NTSTATUS
NtCreatePagingFile *(*
    IN PSTRING *PageFileName,*
    IN PLARGE_INTEGER *InitialSize,*
    IN PLARGE_INTEGER *MaximumSize,*
    IN ULONG *Priority*
    );

Parameters:

PageFileName - Supplies the name of an existing file to utilize as a paging file. This file must already exist.

InitialSize - Supplies the initial size of the specified paging file in bytes. This value is rounded up to the next host size boundary and the specified paging file is extended or truncated to the initial size.

MaximumSize - Supplies the maximum number of bytes to store in the specified paging file. This value is rounded up to the next host page size. This value must be greater than or equal to the *InitialSize*.

Priority - Supplies the relative priority of the paging file with zero being the lowest priority and 0xFFFFFFFF being the highest priority. Page file space on paging files is searched for based on the priority of each paging file.

At least 8 paging files may be created. The modified page writer attempts to write pages to all specified paging file simultaneously, therefore, for maximum performance, each paging file should reside on a separate disk drive.

The following status values may be returned by the function:

- *STATUS_NORMAL* - Normal, successful completion.
- Errors resulting from attempting to open, extend, or truncate the specified file.

2.17 Flush Instruction Cache

The instruction cache for a specific process can be flushed with the **NtFlushInstructionCache** function:

NTSTATUS
NtFlushInstructionCache *(*
    IN HANDLE *ProcessHandle,*
    IN PVOID *BaseAddress* **OPTIONAL***,*
    IN ULONG *Length*
    );

Parameters:

ProcessHandle - An open handle to a process object.

BaseAddress - Optionally supplies the base address to begin the flush operation at. If not specified the whole cache is flushed.

Length - Supplies the length of the buffer to flush. Only used if BaseAddress is specified.

This routine is provided for use by system debuggers and routines which dynamically modify code segments. The issuing process must have *PROCESS_VM_OPERATION* access to the subject process

The following status values may be returned by the function:

- *STATUS_NORMAL* - Normal, successful completion.
- Errors resulting from referencing the specified process handle.

2.18 Flush Write Buffer

The write buffers are flushed with the **NtFlushWriteBuffer** function:

NTSTATUS
NtFlushWriteBuffer (
    VOID
    );

This routine flushes the write buffer on the current processor. On processors without write buffers no action is taken.

The following status values may be returned by the function:

- *STATUS_NORMAL* - Normal, successful completion.

2.19 Close Handle

An open handle to any object can be closed with the **NtClose** function:

NTSTATUS
NtClose(
    IN HANDLE *Handle*
    );

Parameters:

Handle - An open handle to an object.

This is a generic function and can be used to close an open handle to any object.

Closing an open handle to an object causes the reference count of the associated object to be decremented. If the resultant count is zero (i.e., there are no other references to the section), then the object is deleted. If the resultant count is one, the object has a name, and the object is temporary, then an attempt is made to delete the object by removing its name from the appropriate object directory. (Note that this operation may fail if another sharer manages to open the object before the name can be deleted, i.e., the removal of the name is conditional.)

Closing a handle to a section object causes all modified pages to be written to the associated file, if the section is backed by a data file.

After a close operation, the specified section handle is no longer valid.

The following status values may be returned by the function:

- *STATUS_NORMAL* - Normal, successful completion.
- *STATUS_INVALID_PARAMETER* - Error, an invalid parameter was specified.
- *STATUS_NO_ACCESS* - Error, access denied to specified object.

Windows NT Virtual Memory Specification

Revision History:

Revision 1.3, January 4, 1989

1. Add section that describes the difference between this proposal and the proposal included in the **IBM IPFS** for **Cruiser**.

2. Drop expand stack and allow set protection to establish a guard region. Accessing a guard region causes the corresponding page to be turned into a read/write page and a guard page exception to be raised.

3. Define all API functions as returning a status value that determines the success or failure of the operation.

4. Use the words "commit" and "reserve" when referring to virtual address space allocation.

5. Add flags argument to allocate and free virtual memory which signifies whether the commitment and/or reservation of the specified region is to be changed. This allows a region of private pages to be reserved without creating any kind of memory object.

6. Correct definition of giveable and gettable sections so they are temporary and mapped at a fixed address in the virtual address space of each process.

7. Correct definition of tiled to mean that the preferred mapping of the section is within the first **512mb** of the virtual address space of a process.

8. Change give and get section to work with a virtual address rather than a section handle.

9. Add function to query a region of virtual memory.

10. Clear up confusion about protection types by defining types to be no access, execute-only, read-only, read/write, and guard region.

11. Drop section offset parameter on create section operation which allows any number of section to be backed by the same data file.

12. If no **ACL** is specified for an object, then use a process default **ACL**.

13. Change close section handle to be a generic function that closes any type of handle.

14. More clearly define what permanent objects are and how they are deleted.

Revision 2.0, February 28, 1989

1. Changed format of calls to match the **Windows NT** coding guidelines.

Revision 2.1, March 16, 1989

1. Changed format of calls to match the new **Windows NT** coding guidelines.
2. Added *ProcessHandle* argument create operations which operated on the address space.
3. Change *Fork* attribute to *Inherit*.
4. Removed giveable and gettable attributes and replaced them with the based attribute.
5. Eliminated **NtGetSection** and **NtGiveSection** services.
6. Changed semantics of services that change virtual memory attributes on a range of pages to either change the total specified range or fail and change none of the range. This matches OS/2 behavior.
7. Added OBJ_EXCLUSIVE and OBJ_SYSTEM_TABLE flags to handle attributes in create section.
8. Added handle attributes to OpenSection service.
9. Enhanced map view to recognize multiple mappings of the same complete section and return the base address in subsequent mappings.
10. Changed FreeVirtualMemory to not allow previously committed shared pages to be decommitted. This matches the OS/2 behavior.
11. Changed lock and unlock virtual memory to talk about system and process residency rather than system and process address maps.
12. Add zero bits parameter to **NtAllocateVirtualMemory**.
13. Add **NtReadVirtualMemory** function.
14. Add **NtWriteVirtualMemory** function.
15. Add error return values.

Revision 2.2, May 9, 1989

1. Fix typos and minor inconsistencies.

Revision 2.3, August 7, 1989

1. Add SEC_IMAGE option to **NtCreateSection**.
2. Add PAG_NOCACHE option to the protection values.

Revision 2.4, September 7, 1989

1. Change names of PAG_READ, PAG_READWRITE, PAG_EXECUTE, PAG_NOACCESS, PAGE_NOCACHE, PAG_GUARD to PAGE_READ, etc.

2. Change names of PAG_COMMIT, PAG_RESERVE, PAG_RELEASE, PAG_DECOMMIT, PAG_PRIVATE, PAG_MAPPED, to MEM_COMMIT, MEM_RESERVE, etc.

3. Changed NtMapViewOfSection to have AllocationType parameter and changed the type of InheritDisposition from ULONG to SECITON_INHERIT.

4. Added MEM_TOP_DOWN to NtAllocateVirtualMemory and NtMapViewOfSection.

5. Changed BaseAddress from an IN to an IN OUT in NtFreeVirtualMemory, NtProtectVirtualMemory, NtLockVirtualMemoryh and NtUnlockVirtualMemory.

6. Added BaseAddress field to MEMORYBASICINFO type.

7. Added PAGE_WRITECOPY protection to NtMapViewOfSection, NtProtectVirtualMemory, and NtQueryVirtualMemory.

8. Added note to NtProtectVirtualMemory indicating that changing a locked page to PAGE_NOACCESS causes the page to be unlocked.

9. Added note to NtLockVirtualMemory to indicate that locked pages are not locked in a process which inherits the memory.

10. Changed SectionOffset in NtMapViewOfSection to IN OUT.

Revision 2.5, October 23, 1989

1. Add PAGE_EXECUTE_READ, PAGE_EXECUTE_READWRITE and PAGE_EXECUTE_WRITECOPY.

2. Change semantic of PAGE_GUARD to be similar to PAGE_NOACCESS, but instead of an "access violation" being raised, the page protection is changed to its declared protection and a "guard page entered" exception is raised. Like PAGE_NOACCESS, guard pages unlocked locked pages.

3. Added SectionPageProtection argument to NtCreateSection.

4. Added SEC_NOCACHE attribute to NtCreateSection.

5. Made SetionOffset optional for NtMapViewOfSection and changed its allocation from host page size, to system allocation granularity (64k).

6. Changed SEC_COPY, SEC_SHARE, SEC_UNMAP to ViewCopy, ViewShare, ViewUnmap in NtMapViewOfSection.

7. Removed PAGE_NOCACHE option from NtMapViewOfSection.
8. Added PAGE_GUARD option to NtMapViewOfSection.
9. Added PAGE_GUARD option to NtAllocateVirtualMemory.
10. Clarified PAGE_NOCACHE option in NtAllocateVirtualMemory.
11. Changed region size of zero to operate on complete range in NtFreeVirtualMemory.
12. Added AllocationBase and AllocationProtect to NtQueryVirtualMemory.
13. Added SECTIONIMAGEINFO to NtQuerySection.
14. For NtReadVirtualMemory the NumberOfBytesRead was changed to be OPTIONAL.
15. For NtWriteVirtualMemory the NumberOfBytesWritten was changed to be OPTIONAL.

Revision 2.6, December 1, 1989

1. Changed description of NtCreateSection and NtOpenSection to sue OBJECT_ATTRIBUTES and reference the Object Management Specification for details.
2. Changed Query services info structure names.
3. Removed all references to TILE.

Revision 2.7, January 5, 1990

1. Changed section access rights from READ, WRITE, and EXECUTE to SECTION_MAP_READ, SECTION_MAP_WRITE, and SECTION_MAP_EXECUTE.
2. Added SECTION_QUERY access right.
3. Described the type of access required on the section and process handles for various virtual memory services.

Revision 2.8, February 8, 1990

1. Changed NtReadVirtualMemory to have OUT on the buffer argument rather than IN.
2. Changed NtWriteVirtualMemory to have OUT on the base address argument rather than IN.
3. Changed both NtReadVirtualMemory and NtWriteVirtualMemory to remove the base address rounding down to the host page size.
4. Removed PAGE_NOACCESS and PAGE_GUARD as valid page protections when creating a section.
5. PAGE_NOACCESS may not be specified in combination with PAGE_GUARD or PAGE_NOCACHE.
6. Removed STATUS_BUFFER_TOO_SMALL.
7. Added status's of STATUS_INVALID_INFO_CLASS and STATUS_INFO_LENGHT_MISMATCH to query functions.
8. Disallow the combination of Commit and Release to NtFreeVirtualMemory.
9. Add STATUS_NOT_IMAGE to query vm and create section.
10. Add section on page file quota and commitment.
11. Clarify protection rules in MapViewOfSection.
12. Don't allow protection of PAGE_WRITECOPY or PAGE_EXECUTE_WRITECOPY on address ranges not mapping a view of a section.
13. Don't allow a protection of PAGE_NOCACHE on address ranges mapping a view of a section.

Revision 2.9, March 9, 1990

1. Added the following status values to various calls: STATUS_SECTION_TOO_BIG and STATUS_CONFLICTING_ADDRESS.
2. Changed DesiredAccess to type ACCESS_MASK.
3. When SEC_IMAGE is specified in NtCreateSection only accept SEC_BASED with it.
4. Limit MaximumSize in NtCreateSection to 0xFFFEFFFF.
5. Removed ViewCopy from NtMapViewOfSection.

Revision 3.0, May 31, 1990

1. Added the NtExtendSection service.
2. Added SECTION_EXTEND_SIZE access.
3. In create section SEC_COMMIT is only meaningful for page file backed sections.
4. Changed SectionSize parameter to type PLARGE_INTEGER in NtCreateSection.
5. Changed description of MaximumSize parameter in NtCreateSection.
6. Changed SectionOffset parameter to type PLARGE_INTEGER in NtMapViewOfSection.
7. Added NtCreatePagingFile routine.
8. Added NtFlushInstructionCache routine.
9. Added NtFlushWriteBuffer routine.
10. Changed NtFreeVirtualMemory to require the base address to be the start of the region if the region size is specified as zero.
11. Added more status codes to NtFreeVirtualMemory.

Revision 3.1, October 4, 1990

1. Added STATUS_NOT_COMMITTED to NtProtectVirtualMemory.
2. Added MEM_IMAGE as another type to NtQueryVirtualMemory.

Revision 3.2, January 24, 1991

1. Added SECTION_EXTEND_SIZE to NtOpenSection.
2. Clarified that SECTION_WRITE access also grants read access.
3. Clarified that private pages are inherited on a fork operation.
4. Changed parameters to NtCreatePagingFile.
5. Clarified NtReadVirtualMemory and NtWriteVirtualMemory to state that the buffers are probed before any bytes are copied.

Revision 3.3, April 25, 1991

1. For NtFlushVirtualMemory if RegionSize is zero, flush from the base address to the end of the mapped view.

Revision 4.0, April 28, 1993

1. Reflect Windows NT version 3.1.
2. Remove all references to OS/2.
3. Change references to 64k alignment to Allocation Granularity as this alignment is hardware architecture dependent.
4. Changed description of SEC_BASED.
5. Added MEM_LARGE_PAGES and MEM_DOS_LIM to NtMapViewOfSection.
6. Changed NtAllocateVirtualMemory to reflect the fact that committed pages may be committed.
7. Change NtFreeVirtualMemory to reflect the fact that reserved pages may be decommitted.
8. Updated section information structure.
9. Added parameters to NtFlushInstructionCache.

[end of vm.doc]