Portable Systems Group

Windows NT Process Structure

Author: *Mark Lucovsky*

Revision 1.27, January 14, 1992

1. Overview

This specification describes the **Windows NT** process structure.

The **Windows NT** system is designed to support both an **OS/2** and a **POSIX** operating system environment. Rather than packaging all of the capabilities of these operating system environments into the **Windows NT** kernel and executive, the system has been designed so that robust, protected subsystems can be built to provide the necessary API emulation.

The **Windows NT** approach is very similar to the approach taken in Carnegie Mellon's MACH operating system. The MACH system design is based on a simple process structure, IPC mechanism, and virtual memory system. Using these primitives, MACH is able to implement both **POSIX** and **Unix 4.3BSD** operating system environments as protected subsystems.

Like MACH, the **Windows NT** process structure provides a very basic set of services. The system does not provide a hierarchical process tree structure, global process names (PIDs), process grouping, job control, complex process or thread termination semantics, or other more traditional process structures. It does provide a complete set of services that subsystems can use to provide the set of semantics that are required by a particular operating system environment.

Using this set of services, vendors and users can develop applications based on either the **OS/2** or **POSIX** APIs (implemented as protected subsystems by Microsoft). An alternative to this is to develop applications using the native **Windows NT** system services or to develop custom subsystems and have the applications use these subsystems.

2. Process Structure Objects

The process structure is based on two types of objects. A *process object* represents an address space, a set of objects (resources) visible to the process, and a set of threads that executes in the context of the process. A *thread object* represents the basic schedulable entity in the system. It contains its own set of machine registers, its own kernel stack, a thread environment block (TEB), and user stack in the address space of its process.

The **Windows NT** process structure works with the overall **Windows NT** security architecture. Each process is assigned an access token, called the *primary token* of the process. The primary token is used by default by the process's threads when referencing a **Windows NT** object.

In addition to the primary token, each thread may have an *impersonation token* associated with it. When this is done, the impersonation token, rather than the process's primary token, is used for access validation purposes. This is done to allow efficient impersonation of clients in a client-server model.

3. Process Object APIs

The following programming interfaces support the process object:

NtCreateProcess - Creates a process object.

NtTerminateProcess - Terminates a process object.
NtCurrentProcess - Identifies the currently executing process.
NtCurrentPeb - Returns the address of the current processes Process Environment Block (PEB).
NtOpenProcess - Creates a handle to a process object.
NtQueryInformationProcess - Returns information about the process.
NtSetInformationProcess - Sets information about the process.

3.1 Access Type And Privilege Information

Object type-specific access types:

The object type-specific access types are defined below.

PROCESS_TERMINATE - Required to terminate a process.

PROCESS_CREATE_THREAD - Required to create a thread in a process.

PROCESS_VM_OPERATION - Required to manipulate the address space of a process. This does not include reading and writing the memory of a process.

PROCESS_VM_READ - Required to read the virtual memory of a process (through **Windows NT APIs**).

PROCESS_VM_WRITE - Required to write the virtual memory of a process (through **Windows NT APIs**).

PROCESS_DUP_HANDLE - Required to duplicate an object handle visible to a process.

PROCESS_CREATE_PROCESS - Required to create a process.

PROCESS_SET_QUOTA - Required to modify the quota limits of a process.

PROCESS_SET_INFORMATION - Required to modify certain attributes of a process.

PROCESS_QUERY_INFORMATION - Required to read certain attributes of a process. This access type is also needed to open the primary token of a process (using **NtOpenProcessToken()**).

PROCESS_SET_PORT - Required to set the debug or exception port of a process.

Generic Access Masks:

The object type-specific mapping of generic access types to non-generic access types for this object type are:

GENERIC_READ	**STANDARD_READ |** **PROCESS_VM_READ |** **PROCESS_QUERY_INFORMATION**
GENERIC_WRITE	**STANDARD_WRITE |** **PROCESS_TERMINATE |** **PROCESS_CREATE_THREAD |** **PROCESS_VM_OPERATION |** **PROCESS_VM_WRITE |** **PROCESS_DUP_HANDLE |** **PROCESS_CREATE_PROCESS |** **PROCESS_SET_QUOTA |** **PROCESS_SET_INFORMATION |** **PROCESS_SET_PORT**
GENERIC_EXECUTE	**STANDARD_EXECUTE |** **SYNCHRONIZE**

Standard Access Types:

This object type supports the optional SYNCHRONIZE standard access type. All required access types are supported by the object manager.

The mask of all supported access types for this object is:

PROCESS_ALL_ACCESS	**STANDARD_RIGHTS_REQUIRED |** **SYNCHRONIZE |** **PROCESS_TERMINATE |** **PROCESS_CREATE_THREAD |** **PROCESS_VM_OPERATION |** **PROCESS_VM_READ |** **PROCESS_VM_WRITE |** **PROCESS_DUP_HANDLE |** **PROCESS_CREATE_PROCESS |** **PROCESS_SET_QUOTA |** **PROCESS_SET_INFORMATION |** **PROCESS_QUERY_INFORMATION |** **PROCESS_SET_PORT**

Privileges Defined Or Used:

This object type defines or uses the following privileges:

SeAssignPrimaryTokenPrivilege - This privilege is needed to assign a new primary token for a process.

3.2 NtCreateProcess

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

NTSTATUS
NtCreateProcess(
OUT PHANDLE *ProcessHandle,*
IN ACCESS_MASK *DesiredAccess,*
IN POBJECT_ATTRIBUTES *ObjectAttributes* **OPTIONAL***,*
IN HANDLE *ParentProcess,*
IN BOOLEAN *InheritObjectTable,*
IN HANDLE *SectionHandle* **OPTIONAL***,*
IN HANDLE *DebugPort* **OPTIONAL***,*
IN HANDLE *ExceptionPort* **OPTIONAL**
);

Parameters:

ProcessHandle - A pointer to a variable that will receive the process object handle value.

DesiredAccess - The desired types of access to the created process.

ObjectAttributes - An optional pointer to a structure that specifies the object's attributes. Refer to the *Object Management Specification* for details. Note that *OBJ_PERMANENT*, *OBJ_EXCLUSIVE*, *OBJ_OPEN_IF*, and *OBJ_OPEN_LINK* are not valid attributes for a process object.

ParentProcess - An open handle to a process object. The new process is created using some of the attributes of the specified parent process. *PROCESS_CREATE_PROCESS* access to this process is required.

InheritObjectTable - A flag which determines whether or not the new process will be created with an object table whose initial contents come from the specified parent process. A value of false causes the new process to be created with an empty object table. A value of true causes the new process to be created by cloning the parent process's object table. All objects in the parent process's object table marked with the *OBJ_INHERIT* attribute appear in the new process's object table with exactly the same handle values, attributes, and granted access.

SectionHandle - An optional open handle to a section object. If the value of the argument is not null, then it specifies a handle to a section object backed by an image file the process is being created to run. *SECTION_MAP_EXECUTE* access to the section object is required.

DebugPort - An optional open handle to a port object. If specified, the port is assigned as the process's debugger port; otherwise, the process is created without a debugger port. *PORT_WRITE* and *PORT_READ* access to the port object are required.

ExceptionPort - An optional open handle to a port object. If specified, the port is assigned as the process's exception port; otherwise, the process is created without an exception port. *PORT_WRITE* and *PORT_READ* access to the port object are required.

Creating a process object causes a new process to be created. The new process shares some of its initial attributes with the specified parent process.

- The new process is created with an object table. The table is either an empty table, or a clone of the parent process's object table. This is a function of the *InheritObjectTable* parameter.
- The access token of the new process is identical to the access token of the parent process.
- The quota limits of the new process are identical to the quota limits of the parent process.
- The base priority of the new process is identical to the base priority of the parent process.

The address space of the new process is defined by the specified section handle or the address space of the specified parent process. If the section handle is not null, the section object must be backed by an image file. The address space of the new process is created by mapping a view of the entire section object. Otherwise, the address space of the process is created by copying or sharing those pieces of the parent process's address space marked as **PAG_COPY/PAG_SHARE** into the address space of the new process.

The new process is created without any threads.

Each process is created with a Process Environment Block (PEB). The PEB is readable and writeable by the application, but can only be deleted by the system. The PEB is partially initialized by the system and is placed in the address space of the. If the process is created without a section handle, then the new processes PEB is shared "copy on write" with the parent process PEB.

The PEB contains process global context such as startup parameters, image base address, a Mutant object handle for process wide synchronization, and loader data structures.

The function **NtCurrentPeb** returns the address of the current processes PEB. Access to PEB locations must be made through this API.

The process object is a waitable object. A wait performed on a process object is satisfied when the process becomes signaled. A process becomes signaled when its last thread terminates, or if a process without a thread is terminated with **NtTerminateProcess**.

Both the debugger and exception ports are used by the exception handling system within **Windows NT**. The role that these ports play in exception handling is described in another document.

3.3 NtTerminateProcess

A process can be terminated with the **NtTerminateProcess** function:

NTSTATUS
NtTerminateProcess(
IN HANDLE *ProcessHandle OPTIONAL,*
IN NTSTATUS *ExitStatus*
);

Parameters:

ProcessHandle - An optional parameter, that if specified, supplies an open handle with **PROCESS_TERMINATE** access to the process to terminate. If this parameter is not supplied, then **PROCESS_TERMINATE** access is required to the current process and the API terminates all threads in the process except for the calling thread.

ExitStatus - A value that specifies the exit status of the process to be terminated.

Terminating a process causes the specified process and all of its threads to terminate. Any threads in the process that are suspended are resumed by this service so that they can begin termination. The handles of the process's threads are not explicitly closed by this service. The handle to the process being terminated is also not closed by this service. If any thread in the process was suspended and resumed by this API and informational status code of *STATUS_THREAD_WAS_SUSPENDED* is returned.

In order to terminate a process, the calling thread must have *PROCESS_TERMINATE* access to the specified process.

After all of the process's threads are terminated (and set to the signaled state), the process's object table is processed by closing all open handles.

The process object is signaled upon termination, and its exit status is updated to reflect the value of the exit status argument. Once a process object becomes signaled, no more threads can be created in the process.

The process's address space remains valid until the process object itself is deleted (the last handle to the process object is closed).

3.4 NtCurrentProcess

An object handle to the current process can be fabricated with the **NtCurrentProcess** function:

HANDLE
NtCurrentProcess();

The **NtCurrentProcess** function returns a pseudo handle to the currently executing process. The handle can be used whenever a handle to a process object is required (e.g. **NtTerminateProcess**).

When the system is asked to translate an object handle into an object pointer, the object type is a process object, and the object handle is the pseudo handle returned by **NtCurrentProcess**, the following occurs.

- The **SECURITY_DESCRIPTOR** of the current process is checked against the desired access specified in the object translation call. If access is denied a failure status is returned to the caller.

- If access is allowed, the appropriate reference count in the current process object is adjusted and a pointer to the current process object is returned.

This function is designed mainly for the use of native applications so that they can refer to their own process in process termination calls, thread creation calls, and address space modification calls without having to explicitly open their process by name or otherwise obtain a handle to their own process. A similar function exists to reference the currently executing thread.

3.5 NtCurrentPeb

The address of the current processes **PEB** can be located with the **NtCurrentPeb** function:

PPEB
NtCurrentPeb()

The **NtCurrentPeb** function returns the address of the current processes **PEB**. The **PEB** consists of a single page in the address space of the process. The page is allocated and deallocated by the system at process creation/process termination. Only the system may delete a processes **PEB**. The **PEB** contains the following:

Peb Structure

BOOLEAN *InheritedAddressSpace* - A flag set by the system to indicate that the processes initial address space was from inheritance rather than from a mapping a section.

HANDLE *Mutant* - Contains a handle to a mutant object. Various portions of the system use this mutant to synchronize within the process. The functions **RtlAcquirePebLock** and **RtlReleasePebLock** may be used to access this field.

PCOFF_HEADERS *ImageBaseAddress* - Contains the address of the image header of the processes initial image.

PPEB_LDR_DATA *Ldr* - Contains the address of the loaders per-process data. The value of this pointer is null until the first thread of a process initialzes the loader.

PEB_SM_DATA *Sm* - Contains Session Manager specific information.

PRTL_USER_PROCESS_PARAMETERS *ProcessParameters* - Contains the address of the processes startup parameters.

PVOID *SubsystemData* - Contains the address of subsystem specific data.

PPEB_FREE_BLOCK *FreeList* - Contains the address of a dynamic area in the PEB. Calls to **RtlAllocateFromPeb** and **RtlFreeToPeb** are satisfied from this area.

3.6 NtOpenProcess

A handle to a process object can be created with the **NtOpenProcess** function:

NTSTATUS
NtOpenProcess(
OUT PHANDLE *ProcessHandle,*
IN ACCESS_MASK *DesiredAccess,*
IN POBJECT_ATTRIBUTES *ObjectAttributes,*
IN PCLIENT_ID *ClientId* **OPTIONAL**
);

Parameters:

ProcessHandle - A pointer to a variable that will receive the process object handle value.

DesiredAccess - The desired types of access to the opened process. For a complete description of desired access flags, refer to the **NtCreateProcess API** description.

ObjectAttributes - An pointer to a structure that specifies the object's attributes. Refer to the *Object Management Specification* for details. Note that *OBJ_PERMANENT, OBJ_EXCLUSIVE, OBJ_OPEN_IF*, and *OBJ_OPEN_LINK* are not valid attributes for a process object.

ClientId - An optional parameter that if specified, supplies the client ID of a thread whose process is to be opened. It is an error to specify this parameter along with the an *ObjectAttributes* variable that contains a process name.

Opening a process object causes a new handle to be created. The access that the new handle has to the process object is a function of the desired access and any *SECURITY_DESCRIPTOR* on the process object

3.7 NtQueryInformationProcess

Selected information about a process can be retrieved using the **NtQueryInformationProcess** function.

NTSTATUS
NtQueryInformationProcess(
IN HANDLE *ProcessHandle,*
IN PROCESSINFOCLASS *ProcessInformationClass*
OUT PVOID *ProcessInformation,*
IN ULONG *ProcessInformationLength,*
OUT PULONG *ReturnLength* **OPTIONAL**
);

Parameters:

ProcessHandle - A variable that specifies the handle to a process from which to retrieve information.

ProcessInformationClass - A variable that specifies the type of information to retrieve from the specified process object.

ProcessInformationClass Values

ProcessBasicInformation - Returns the basic information about the specified process. This information class value requires *PROCESS_QUERY_INFORMATION* access to the process.

ProcessQuotaLimits - Returns the quota limits of the specified process. This information class requires *PROCESS_QUERY_INFORMATION* access to the process.

ProcessIoCounters - Returns the input/output counters of the specified process. This information class requires *PROCESS_QUERY_INFORMATION* access to the process.

ProcessVmCounters - Returns the virtual memory counters of the specified process. This information class requires *PROCESS_QUERY_INFORMATION* access to the process.

ProcessTimes - Returns the cpu time usage of the specified process. This information class requires *PROCESS_QUERY_INFORMATION* access to the process.

ProcessLdtInformation - Returns the contents of the Ldt for the process. Requires *PROCESS_VM_READ* access to the process. Returns *STATUS_NOT_SUPPORTED* on non i386 (and compatible) processors.

ProcessInformation - A pointer to a buffer that will receive information about the specified process. The format and contents of the buffer depend on the specified information class being queried.

ProcessInformation Format by Information Class

ProcessBasicInformation - Data type is *PPROCESS_BASIC_INFORMATION*.

PROCESS_BASIC_INFORMATION Structure

NTSTATUS *ExitStatus* - Specifies the exit status of the process. This field only contains meaningful information if the process is in the signaled state; otherwise, it contains a value of "exit status pending".

PPEB *PebBaseAddress* - Specifies the base address of the processes PEB.

KPRIORITY *BasePriority* - Specifies the base priority of the process.

KAFFINITY *AffinityMask* - Specifies the default affinity mask assigned to each thread in the process during thread creation.

ProcessQuotaLimits - Data type is *PQUOTA_LIMITS*.

QUOTA_LIMITS Structure

ULONG *PagedPoolLimit* - Specifies the maximum amount of paged pool (in bytes) that can be used by the process.

ULONG *NonPagedPoolLimit* - Specifies the maximum amount of nonpaged pool (in bytes) that can be used by the process.

ULONG *MinimumWorkingSetSize* - Specifies the minimum working set size (in bytes) for the process.

ULONG *MaximumWorkingSetSize* - Specifies the maximum working set size (in bytes) for the process.

ULONG *PagefileLimit* - Specifies the maximum amount of pagefile space (in bytes) that can be used by the process.

TIME *TimeLimit* - Specifies the maximum number of 100ns units that the process can execute for.

ProcessIoCounters - Data type is *PIO_COUNTERS.*

IO_COUNTERS Structure

ULONG *ReadOperationCount* - Specifies the number of read I/O operations performed by the process.

ULONG *WriteOperationCount* - Specifies the number of write I/O operations performed by the process.

ULONG *OtherOperationCount* - Specifies the number of other I/O operations (not read or write) performed by the process.

LARGE_INTEGER *ReadTransferCount* - Specifies the number of bytes transferred through read I/O operations.

LARGE_INTEGER *WriteTransferCount* - Specifies the number of bytes transferred through write I/O operations.

LARGE_INTEGER *OtherTransferCount* - Specifies the number of bytes transferred through other I/O operations.

ProcessVmCounters - Data type is *PVM_COUNTERS.*

VM_COUNTERS Structure

ULONG *PeakVirtualSize* - Specifies the largest virtual address space size (in bytes) that the process has reached.

ULONG *VirtualSize* - Specifies the current virtual address space size (in bytes) of the process.

ULONG *PageFaultCount* - Specifies the number of pagefaults incurred by the process.

ULONG *PeakWorkingSetSize* - Specifies the largest working set size (in bytes) that the process has reached.

ULONG *WorkingSetSize* - Specifies the current working set size (in bytes) of the process.

ULONG *QuotaPeakPagedPoolSize* - Specifies the largest amount of paged pool (in bytes) that the process has used and has been charged quota for.

ULONG *QuotaPagedPoolSize* - Specifies the current amount of paged pool (in bytes) in use by the process and being charged to the process.

ULONG *QuotaNonPeakPagedPoolSize* - Specifies the largest amount of nonpaged pool (in bytes) that the process has used and has been charged quota for.

ULONG *QuotaNonPagedPoolSize* - Specifies the current amount of nonpaged pool (in bytes) in use by the process and being charged to the process.

ULONG *PagefileUsage* - Specifies the current amount of pagefile space (in bytes) in use by the process.

ProcessTimes - Data type is *PKERNEL_USER_TIMES*.

KERNEL_USER_TIMES Structure

TIME *UserTime* - Specifies the number of 100ns units that the process has spent executing in user mode.

TIME *KernelTime* - Specifies the number of 100ns units that the process has spent executing in kernel mode.

TIME *CreateTime* - Specifies the time that the process was created.

TIME *ExitTime* - Specifies the time that the process terminated.

ProcessLdtInformation - Data type is *PPROCESS_LDT_INFORMATION*.

PROCESS_LDT_INFORMATION Structure

ULONG *Start* - Specifies the starting offset in the LDT to return descriptors from. It must be 0 mod 8. If this value is larger than the current size of the LDT, no information will be put into the LdtEntries field.

ULONG *Length* - Supplies the length of the section of the LDT to return. Must be 0 mod 8. Returns the length of the Ldt. Will always be set.

LDT_ENTRY *LdtEntries[1]* - Variable size array of LDT_ENTRYs, is the actual Ldt data in hardware format.

ProcessInformationLength - Specifies the length in bytes of the process information buffer (i.e. size of the information structure).

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

3.8 NtSetInformationProcess

Selected information can be set in a process using the **NtSetInformationProcess** function.

NTSTATUS
NtSetInformationProcess(
IN HANDLE *ProcessHandle,*
IN PROCESSINFOCLASS *ProcessInformationClass,*
IN PVOID *ProcessInformation,*
IN ULONG *ProcessInformationLength*
);

Parameters:

ProcessHandle - A variable that specifies the handle to a process to set information into.

ProcessInformationClass - A variable that specifies the type of information to set into the specified process object.

ProcessInformationClass Values

ProcessBasePriority - Sets the base priority of the specified process. This information class value requires *PROCESS_SET_INFORMATION* access to the process.

ProcessQuotaLimits - Sets the quota limits associated with the process. This information class value requires *PROCESS_SET_QUOTA* access to the process. If an attempt is made to increase quota, a privilege check is done to ensure that the calling process has ***TBD*** privilege.

ProcessAccessToken - Sets the primary access token of the specified process. This information class requires *PROCESS_SET_INFORMATION* access to the process. Furthermore, the caller must have **SeAssignPrimaryTokenPrivilege** privilege.

Since the process access token is inherited during process creation, this operation only needs to be performed when a process is being created for a new user or for a privileged application.

ProcessDebugPort - Sets the debug port of the specified process. If the process already has a debug port either through process creation, or a previous call to **NtSetInformationProcess** then an error is returned. This information class requires *PROCESS_SET_PORT* access to the process.

ProcessExceptionPort - Sets the exception port of the specified process. If the process already has an exception port either through process creation, or a previous call to **NtSetInformationProcess** then an error is returned. This information class requires *PROCESS_SET_PORT* access to the process.

ProcessLdtInformation - Returns the contents of the Ldt for the process. Requires *PROCESS_VM_WRITE* access to the process. Returns *STATUS_NOT_SUPPORTED* on non i386 (and compatible) processors.

ProcessLdtSize - Returns the size of the Ldt for the process. *PROCESS_VM_WRITE* access required. Returns *STATUS_NOT_SUPPORTED* on non i386 (and compatible) processors.

ProcessInformation - A pointer to a buffer that contains the information to set in the specified process. The format and contents of the buffer depend on the specified information class being queried.

ProcessInformation Format by Information Class

ProcessBasePriority - Data type is *KPRIORITY*.

KPRIORITY *BasePriority* - Specifies the base priority of the process.

ProcessQuotaLimits - Data type is *PQUOTA_LIMITS*.

QUOTA_LIMITS Structure

ULONG *PagedPoolLimit* - Specifies the maximum amount of paged pool (in bytes) that can be used by the process.

ULONG *NonPagedPoolLimit* - Specifies the maximum amount of nonpaged pool (in bytes) that can be used by the process.

ULONG *MinimumWorkingSetSize* - Specifies the minimum working set size (in bytes) for the process.

ULONG *MaximumWorkingSetSize* - Specifies the maximum working set size (in bytes) for the process.

ULONG *PagefileLimit* - Specifies the maximum amount of pagefile space (in bytes) that can be used by the process.

TIME *TimeLimit* - Specifies the maximum number of 100ns units that the process can execute for.

ProcessAccessToken - Data type is *PHANDLE*. The handle is expected to be to a Token object. The handle must have been opened to provide **TOKEN_ASSIGN_PRIMARY** access.

ProcessDebugPort - Data type is *PHANDLE*.

ProcessExceptionPort - Data type is *PHANDLE*.

ProcessLdtInformation - Data type is *PPROCESS_LDT_INFORMATION*.

PROCESS_LDT_INFORMATION Structure

ULONG *Start* - Offset in Ldt of first entry to set. Must be 0 mod 8.

ULONG *Length* - Length of section of Ldt to set. Must be 0 mod 8.

LDT_ENTRY *LdtEntries[1]* - Variable size array of LDT_ENTRYs, is the actual Ldt data in hardware format.

ProcessLdtSize - Data type is *PPROCESS_LDT_SIZE*.

PROCESS_LDT_SIZE Structure

ULONG *Length* - Size to set Ldt to. Setting 0 sets a null Ldt. Can be used to truncate the Ldt. Must be 0 mod 8.

ProcessInformationLength - Specifies the length in bytes of the process information buffer.

4. Thread Object APIs

The following programming interfaces support the thread object:

NtCreateThread - Creates a thread object.
NtTerminateThread - Terminates a thread object.
NtCurrentThread - Identifies the currently executing thread.
NtCurrentTeb - Returns the address of the current thread's Thread Environment Block (TEB).
NtSuspendThread - Suspends user-mode execution of a thread.
NtResumeThread - Resumes user-mode execution of a thread.
NtGetContextThread - Returns the user-mode context of a thread.
NtSetContextThread - Sets the user-mode context of a thread.
NtOpenThread - Returns a handle to a thread object.
NtQueryInformationThread - Returns information about the thread.
NtSetInformationThread - Sets information about the thread.
NtImpersonateThread - Set one thread to be impersonating another thread.
NtAlertThread - Alerts the specified thread.
NtTestAlert - Tests for an alert condition.
NtAlertResumeThread - Alerts and resumes the specified thread.
NtRegisterThreadTerminationPort - Adds a port notification descriptor to the specified thread.

4.1 Access Type And Privilege Information

Object type-specific access types:

The object type-specific access types are defined below.

THREAD_TERMINATE - Required to terminate a thread.

THREAD_SUSPEND_RESUME - Required to suspend or resume a thread.

THREAD_ALERT - Required to alert a thread using either **NtAlertThread** or **NtAlertResumeThread**.

THREAD_GET_CONTEXT - Required to read a thread's context (using **NtGetContextThread**).

THREAD_SET_CONTEXT - Required to modify a thread's context (using **NtSetContextThread**).

THREAD_SET_INFORMATION - Required to modify certain attributes of a thread.

THREAD_QUERY_INFORMATION - Required to read certain attributes of a thread. This access type is also needed to open the impersonation token of a thread (using **NtOpenThreadToken()**).

THREAD_SET_THREAD_TOKEN - Required to explicitly assign an impersonation token to the thread. In some cases, impersonation will happen automatically (e.g., as a result of a call from a client via LPC). However, to explicitly assign an impersonation token (via a handle) to a thread (also via a handle), requires this access to the thread.

THREAD_IMPERSONATE - Required to directly impersonate a thread. In some instances this access is not required to impersonate a thread. In particular, when a thread calls a server using an communication session layer that supports security quality of service[1], then the server does not need to directly access the thread to impersonate. However, in some cases it is desireable to allow a server to impersonate a thread without using a communication session layer to impersonate a client. In that case, the target client thread may be opened for this access, and then a call made to **NtImpersonateThread()**.

Generic Access Masks:

The object type-specific mapping of generic access types to non-generic access types for this object type are:

---

[1] See the Windows NT Local Security Specification for more on security quality of service.

GENERIC_READ	**STANDARD_READ |** **THREAD_GET_CONTEXT |** **THREAD_QUERY_INFORMATION**
GENERIC_WRITE	**STANDARD_WRITE |** **THREAD_TERMINATE |** **THREAD_SUSPEND_RESUME |** **THREAD_THREAD_ALERT |** **THREAD_SET_CONTEXT |** **THREAD_SET_INFORMATION**
GENERIC_EXECUTE	**STANDARD_EXECUTE |** **THREAD_SET_THREAD_TOKEN |** **SYNCHRONIZE**

Standard Access Types:

This object type supports the optional SYNCHRONIZE standard access type. All required access types are supported by the object manager.

The mask of all supported access types for this object is:

THREAD_ALL_ACCESS	**STANDARD_RIGHTS_REQUIRED |** **SYNCHRONIZE |** **THREAD_GET_CONTEXT |** **THREAD_QUERY_INFORMATION |** **THREAD_TERMINATE |** **THREAD_SUSPEND_RESUME |** **THREAD_THREAD_ALERT |** **THREAD_SET_CONTEXT |** **THREAD_SET_INFORMATION |** **THREAD_SET_THREAD_TOKEN |** **THREAD_IMPERSONATE |** **THREAD_DIRECT_IMPERSONATION**

4.2 NtCreateThread

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

NTSTATUS
NtCreateThread(
OUT PHANDLE *ThreadHandle,*
IN ACCESS_MASK *DesiredAccess,*
IN POBJECT_ATTRIBUTES *ObjectAttributes* **OPTIONAL**,
IN HANDLE *ProcessHandle,*
OUT PCLIENT_ID *ClientId,*
IN PCONTEXT *ThreadContext,*
IN PINITIAL_TEB *InitialTeb,*
IN BOOLEAN *CreateSuspended*
);

Parameters:

ThreadHandle - A pointer to a variable that will receive the thread object handle value.

DesiredAccess - The desired types of access to the created thread.

ObjectAttributes - An optional pointer to a structure that specifies the object's attributes. Refer to the *Object Management Specification* for details. Note that *OBJ_PERMANENT, OBJ_EXCLUSIVE, OBJ_OPEN_IF*, and *OBJ_OPEN_LINK* are not valid attributes for a thread object.

ProcessHandle - An open handle to the process object that the thread is to run in. The subject thread must have *PROCESS_CREATE_THREAD* access to this process. The value of this argument may be the value returned by **NtCurrentProcess** to specify that the new thread is to be created in the context of the current process.

ClientId - A pointer to a structure that will receive the client identifier of the new thread. Each thread in the system is assigned a client identifier value. A client identifier remains valid from the time the thread is created until it is terminated. The value of the client identifier is unique for each thread in the system. The client identifier contains two fields. One field is unique for each process in the system, and one field is unique for each thread in the system.

ClientId Structure

ULONG *UniqueProcess* - Unique value for each process in the system.

ULONG *UniqueThread* - Unique value for each thread in the system.

ThreadContext - A pointer to the structure that contains the new thread's initial user mode context.

InitialTeb - A pointer to a structure that specifies initial values for portions of the thread's TEB.

InitialTeb Structure

PVOID *StackBase* - Contains the base address of the thread's stack.

PVOID *StackLimit* - Contains the stack limit for the thread.

PVOID *EnvironmentPointer* - Unspecified.

CreateSuspended - A parameter that specifies whether or not the thread is to be created suspended. If the value of this parameter is **TRUE**, then the thread is created in a suspended state. The thread will not begin executing until it is explicitly resumed using **NtResumeThread**. If the value of this parameter is **FALSE**, then the thread begins execution in user-mode using the specified context.

Creating a thread object causes a new thread to be created. The new thread is assigned some of its initial attributes from the process object it is being created to run in.

- The new thread's priority is the same as its process's base priority.
- The new thread's processor affinity mask is the same as its process's default processor affinity mask.
- The new thread's access token is the same as its process's.

All threads begin execution with a user-mode APC to system code that is part of each processes address space. This code optionally initializes the loaders data structures and resolves dynamic link library references. When the APC routine returns, the thread's context is restored. Normally, this context is the same as that specified during thread creation.

The thread object is a waitable object. A wait performed on a thread object is satisfied when the thread becomes signaled. A thread becomes signaled when it terminates.

Each thread is created with a Thread Environment Block (TEB). The TEB is readable and writeable by the application, but can only be deleted by the system. The TEB is partially initialized by the system and is placed in the address space of the specified process.

The TEB contains thread local context such as stack base and bounds, environment pointer (used by subsystems/dll's), thread local storage descriptors, and the thread's client id. The thread's creator is responsible for initializing the TEB's stack base and bounds since it is also responsible for creating the thread's stack.

The function **NtCurrentTeb** returns the address of the current thread's TEB. Access to TEB locations must be made through this API. The TEB of each thread is located at a different address. The system will guarantee that TEB access of the form:

```
foo = NtCurrentTeb()->StackBase;
NtCurrentTeb()->EnvironmentPointer = &PsxEnvironment;
```

will cause locations in the current thread's TEB to be referenced.

4.3 NtTerminateThread

A thread can be terminated with the **NtTerminateThread** function:

NTSTATUS
NtTerminateThread(
    IN HANDLE *ThreadHandle* **OPTIONAL**,
    IN NTSTATUS *ExitStatus*
    );

Parameters:

ThreadHandle - An optional parameter, that if specified, supplies an open handle with **THREAD_TERMINATE** access to the thread to terminate. If this parameter is not supplied, then **THREAD_TERMINATE** access is required to the current thread and the API terminates the current thread in the process except for the case where the current thread is the last thread in the current process. In this case, a status code of *STATUS_CANT_TERMINATE_SELF* is returned.

ExitStatus - A value that specifies the exit status of the thread to be terminated.

Terminating a thread causes the specified thread to terminate its execution. If the target thread is currently suspended, it will be resumed so that it can begin termination. Once termination begins, the thread will no longer execute in either user mode or kernel mode. The handle to the thread being terminated is not closed by this service. If the thread was suspended and resumed by this API an informational status code of *STATUS_THREAD_WAS_SUSPENDED* is returned.

In order to terminate a thread, the calling thread must have *THREAD_TERMINATE* access to the specified thread.

Once a thread has become the target thread in a valid call to **NtTerminateThread** (i.e. the calling thread has *THREAD_TERMINATE* access to the target thread), the target thread will terminate without executing another instruction in user-mode. This is accomplished by queueing a special kernel-mode APC to the thread which queues a user-mode APC to the target thread and user-mode alerts the thread. The kernel routine associated with the user-mode APC will cause the thread to terminate itself. To guarantee the delivery of the user-mode APC (i.e. to bypass the alert mechanism), the user APC pending bit in the target thread is set during the execution of the special kernel-mode APC.

During thread termination, the terminating thread's port notification list is processed. For each entry in the list, a thread termination datagram is sent to the port. The system blindly ignores any errors sending this datagram (e.g. port disconnect...).

After the thread is terminated (and set to the signaled state), the thread's TEB is deallocated from the address space of the thread's process and its exit status is updated to reflect the value of the exit status argument. The system does not delete the thread's user-mode stack.

Once terminated, the thread's client identifier is available for re-use.

If the terminating thread is the last thread in its process, its process is terminated (via an internal call to **NtTerminateProcess(NtCurrentProcess(), ExitStatus);**). There is no mechanism that a subsystem can use to prevent this from happening.

4.4 NtCurrentThread

An object handle to the current thread can be fabricated with the **NtCurrentThread** function:

HANDLE
NtCurrentThread();

The **NtCurrentThread** function returns a pseudo handle to the currently executing thread. The handle can be used whenever a handle to a thread object is required (e.g. **NtTerminateThread**).

When the system is asked to translate an object handle into an object pointer, the object type is a thread object, and the object handle is the pseudo handle returned by **NtCurrentThread**, the following occurs.

- The **SECURITY_DESCRIPTOR** of the current thread is checked against the desired access specified in the object translation call. If access is denied, a failure status is returned to the caller.

- If access is allowed, the appropriate reference count in the current thread object is adjusted and a pointer to the current thread object is returned.

This function is designed mainly for the use of native applications so that they can refer to their own thread in thread termination calls, thread creation calls, and thread control calls without having to explicitly open their thread by name or otherwise obtain a handle to their own thread.

4.5 NtCurrentTeb

The address of the current thread's **TEB** can be located with the **NtCurrentTeb** function:

PTEB
NtCurrentTeb()

The **NtCurrentTeb** function returns the address of the current thread's **TEB**. The **TEB** consists of a single page in the address space of the thread's process. The page is allocated and deallocated by the system at thread creation/thread termination. Only the system may delete a thread's **TEB**. The **TEB** contains the following:

Teb Structure

PEXCEPTION_REGISTRATION_RECORD *ExceptionRegistrationRecord* - Contains the base address of the thread's exception handler chain. This field is only used on implementations that require this sort of exception handler registration.

PVOID *StackBase* - Contains the base address of the thread's stack.

PVOID *StackLimit* - Contains the stack limit for the thread.

PVOID *EnvironmentPointer* - Unspecified.

ULONG *Version* - Unspecified.

PVOID *ArbitraryUserPointer* - Unspecified.

CLIENT_ID *ClientId* - Contains the client identifier of the thread.

PVOID *ActiveRpcHandle* - Reserved for use by the *Microsoft Remote Procedure Call Runtime Package*.

PVOID *ThreadLocalStoragePointer* - Reserved for runtime support.

PPEB *ProcessEnvironmentBlock* - Contains the base address of the thread's PEB.

PVOID *UserReserved[USER_RESERVED_TEB]* - TEB locations reserved for applications.

PVOID *SystemReserved[SYSTEM_RESERVED_TEB]* - TEB locations reserved for Microsoft system software.

4.6 NtSuspendThread

A thread can be suspended with the **NtSuspendThread** function:

```
NTSTATUS
NtSuspendThread(
    IN HANDLE ThreadHandle,
    OUT PULONG PreviousSuspendCount OPTIONAL
    );
```

Parameters:

ThreadHandle - A handle to the thread to be suspended.

PreviousSuspendCount - A pointer to the variable that receives the thread's previous suspend count.

Suspending a thread causes the thread to stop executing in user-mode. If the thread is resumed without altering its context and its previous suspend count is one, then the thread resumes execution at the point that it was suspended. If the specified thread is either terminated or is currently terminating, an error status of *STATUS_THREAD_IS_TERMINATING* is returned.

The suspension of a thread is controlled by a suspend count. This count has a maximum value. If an attempt is made to suspend a thread whose suspend count is at its maximum, an error is returned When an attempt is made to suspend a thread, the thread's suspend count is incremented. If the previous value of the suspend count was zero, then a kernel mode APC is queued to the thread. When the APC executes, it causes the thread to wait on its built-in suspend semaphore (the wait is not alertable). The previous value of the thread's suspend count is returned to the caller. A non-zero value indicates that the thread was previously suspended. The value plus 1 specifies the number of calls to **NtResumeThread** that must be made in order to bring the thread out of the suspend state.

This service requires *THREAD_SUSPEND_RESUME* access to the specified thread.

4.7 NtResumeThread

A thread can be resumed with the **NtResumeThread** function:

NTSTATUS
NtResumeThread(
IN HANDLE *ThreadHandle,*
OUT PULONG *PreviousSuspendCount* **OPTIONAL**
);

Parameters:

ThreadHandle - A handle to the thread to be resumed.

PreviousSuspendCount - A pointer to the variable that receives the thread's previous suspend count.

Resuming a thread reverses the effects of a previous call to **NtSuspendThread**.

When an attempt is made to resume a thread, the thread's suspend count is examined. If the count is zero, then the service returns the suspend count. Otherwise, the count is decremented and if the count reaches zero, the thread resumes. In either case, the previous value of the thread's suspend count is returned. A non-zero value indicates that the thread was previously suspended. The value minus 1 specifies the number of calls to **NtResumeThread** that must be made in order to bring the thread out of the suspend state.

This service requires *THREAD_SUSPEND_RESUME* access to the specified thread.

4.8 NtGetContextThread

A thread's user-mode machine context can be read using the **NtGetContextThread** function:

NTSTATUS
NtGetContextThread(
    IN HANDLE *ThreadHandle,*
    IN OUT PCONTEXT *ThreadContext*
    );

Parameters:

ThreadHandle - An open handle to the thread object from which to retrieve context information.

ThreadContext - A pointer to the structure that will receive the user mode context of the specified thread. The initial value of the context flags field indicates the type and amount of context returned by this function.

The **NtGetContextThread** function is designed to facilitate the implementation of debuggers, and to allow subsystems to control the execution flow of their threads (e.g.; emulate signal delivery or APC delivery).

The **NtGetContextThread** function is absolutely NOT PORTABLE! The layout, contents, and length of the *PCONTEXT* structure depend on the processor and system architecture of the system servicing the **NtGetContextThread** function.

This service requires *THREAD_GET_CONTEXT* access to the specified thread.

The **NtGetContextThread** function is implemented by:

- Validating its arguments and translating the thread handle.
- Assuming everything is valid, it allocates a buffer for the thread's user-mode context. It then queues a special kernel-mode APC to the thread, and waits on an event located in the allocated buffer.
- When the APC executes, the thread dumps its user-mode context into the buffer and sets an event (located in the allocated buffer) indicating that the context dump is complete.

    *\ The APC is actually a special kernel mode APC, so that it can work even on a thread that is stuck in a suspend. *

- The target thread returns to whatever it was doing, and the thread calling **NtGetContextThread** copies the user-mode context from the allocated buffer into the thread context buffer passed in the system service. The allocated buffer is freed and the **NtGetContextThread** service completes.

The specified thread does not need to be in a suspend state in order to call **NtGetContextThread** (subsystems and debuggers must explicitly do this if that is what is required). There is nothing to prevent a thread from calling **NtGetContextThread** on itself.

4.9 NtSetContextThread

A thread's user-mode machine context can be altered using the **NtSetContextThread** function:

NTSTATUS
NtSetContextThread(
    IN HANDLE *ThreadHandle,*
    IN OUT PCONTEXT *ThreadContext*
    );

Parameters:

ThreadHandle - An open handle to the thread whose context is to be set.

ThreadContext - A pointer to the structure that contains the user-mode context to be restored into the specified thread. The initial value of the context flags field indicates the type and amount of context that will be restored by this function.

The **NtSetContextThread** function is designed to facilitate the implementation of debuggers, and to allow subsystems to control the execution flow of their threads (e.g.; emulate signal delivery).

The **NtSetContextThread** function is absolutely NOT PORTABLE! The layout, contents, and length of the *PCONTEXT* structure depend on the processor and system architecture of the system servicing the **NtSetContextThread** function. Some fields of the *PCONTEXT* structure contain registers that contain both user-mode and kernel-mode context. Setting kernel-mode portions of these registers is not an error, but is ignored.

This service requires *THREAD_SET_CONTEXT* access to the specified thread.

The **NtSetContextThread** function is implemented by:

- Validating its arguments and translating the thread handle.
- Any kernel-mode only portions of fields in the *PCONTEXT* structure are set to a benign value.
- Assuming everything is valid, it allocates a buffer for the thread's user-mode context and copies the contents of the *ThreadContext* parameter into this buffer. It then queues a kernel-mode APC to the thread, and waits on an event located in the allocated buffer.
- When the APC executes, it writes the thread's user-mode context using the contents of the buffer and sets an event (located in the allocated buffer) indicating it is done with the buffer.

*\ The APC is actually a special kernel mode APC, so that it can work even on a thread that is stuck in a suspend. *

- The target thread returns to whatever it was doing. When the target thread transitions into user-mode, its user-mode context will be restored using the context passed in during the call.

- The thread calling **NtSetContextThread** frees the allocated buffer and completes the service.

The specified thread does not need to be in a suspend state in order to call **NtSetContextThread** (subsystems and debuggers must explicitly do this if that is what is required). There is also nothing that prevents the thread making the call to **NtSetContextThread** from being the target thread in the call.

4.10 NtOpenThread

A handle to a thread object can be created with the **NtOpenThread** function:

NTSTATUS
NtOpenThread(
OUT PHANDLE *ThreadHandle,*
IN ACCESS_MASK *DesiredAccess,*
IN POBJECT_ATTRIBUTES *ObjectAttributes,*
IN PCLIENT_ID *ClientId* **OPTIONAL**
);

Parameters:

ThreadHandle - A pointer to a variable that will receive the thread object handle value.

DesiredAccess - The desired types of access to the opened thread. For a complete description of desired access flags, refer to the **NtCreateThread API** description.

ObjectAttributes - An pointer to a structure that specifies the object's attributes. Refer to the *Object Management Specification* for details. Note that *OBJ_PERMANENT, OBJ_EXCLUSIVE, OBJ_OPEN_IF*, and *OBJ_OPEN_LINK* are not valid attributes for a thread object.

ClientId - An optional parameter that if specified, supplies the client identifier of the thread to be opened. It is an error to specify this parameter along with an *ObjectAttributes* variable that contains a thread name.

Opening a thread object causes a new handle to be created. The access that the new handle has to the thread object is a function of the desired access and any *SECURITY_DESCRIPTOR* on the thread object.

4.11 NtQueryInformationThread

Selected information about a thread can be retrieved using the **NtQueryInformationThread** function.

NTSTATUS
NtQueryInformationThread(
    IN HANDLE *ThreadHandle,*
    IN THREADINFOCLASS *ThreadInformationClass*
    OUT PVOID *ThreadInformation,*
    IN ULONG *ThreadInformationLength,*
    OUT PULONG *ReturnLength* **OPTIONAL**
    );

Parameters:

ThreadHandle - An open handle to the thread object from which to retrieve information.

ThreadInformationClass - A variable that specifies the type of information to retrieve from the specified thread object.

ThreadInformationClass Values

ThreadBasicInformation - Returns the basic information about the specified thread. This information class value requires *THREAD_QUERY_INFORMATION* access to the thread.

ThreadTimes - Returns the cpu time usage of the specified thread. This information class requires *THREAD_QUERY_INFORMATION* access to the thread.

ThreadDescriptorTableEntry - Returns a descriptor from appropriate descriptor table for the thread. This information class will return a descriptor from either the Ldt, or the Gdt for the thread. This information class is only available on x86 processors, and returns STATUS_NOT_IMPLEMENTED on other processors. This information class requires *THREAD_QUERY_INFORMATION* access to the thread.

ThreadInformation - A pointer to a buffer that will receive information about the specified thread. The format and contents of the buffer depend on the specified information class being queried.

ThreadInformation Format by Information Class

ThreadBasicInformation - Data type is *PTHREAD_BASIC_INFORMATION*.

THREAD_BASIC_INFORMATION Structure

ULONG *ExitStatus* - Specifies the exit status of the thread. This field only contains meaningful information if the thread is in the signaled state; otherwise, it contains a value of "exit status pending".

PTEB *TebBaseAddress* - Specifies the virtual address of the thread's **TEB**.

CLIENT_ID *ClientId* - Specifies the thread's client identifier.

KPRIORITY *Priority* - Specifies the current priority of the thread.

KAFFINITY *AffinityMask* - Specifies the current processor affinity mask of the thread.

ThreadTimes - Data type is *PKERNEL_USER_TIMES*.

KERNEL_USER_TIMES Structure

TIME *UserTime* - Specifies the number of 100ns units that the thread has spent executing in user mode.

TIME *KernelTime* - Specifies the number of 100ns units that the thread has spent executing in kernel mode.

TIME *CreateTime* - Specifies the time that the thread was created.

TIME *ExitTime* - Specifies the time that the thread terminated.

ThreadDescriptorTableEntry - Data type is *PDESCRIPTOR_TABLE_ENTRY*

DESCRIPTOR_TABLE_ENTRY Structure

ULONG *Selector* - Specifies the number of the descriptor to return.

LDT_ENTRY *Descriptor* - Returns the descriptor contents.

ThreadInformationLength - Specifies the length in bytes of the thread information buffer (i.e.: the size of the information structure).

ReturnLength - An optional parameter that if specified, receives the number of bytes placed in thread information buffer.

4.12 NtSetInformationThread

Selected information can be set in a thread using the **NtSetInformationThread** function.

NTSTATUS
NtSetInformationThread(
IN HANDLE *ThreadHandle,*
IN THREADINFOCLASS *ThreadInformationClass,*
IN PVOID *ThreadInformation,*
IN ULONG *ThreadInformationLength*
);

Parameters:

ThreadHandle - A variable that specifies the handle to the thread to set information into.

ThreadInformationClass - A variable that specifies the type of information to set into the specified thread object.

ThreadInformationClass Values

ThreadPriority - Sets the priority of the specified thread. This information class value requires *THREAD_SET_INFORMATION* access to the thread.

ThreadAffinityMask - Sets the processor affinity mask of the specified thread. This information class requires *THREAD_SET_INFORMATION* access to the thread.

ThreadImpersonationToken - A handle to an impersonation token to be assigned as the impersonation token of the thread. This requires **THREAD_SET_THREAD_TOKEN** to the thread object and **TOKEN_IMPERSONATE** access to the token object. If the handle value is null, then any impersonation already in progress is discontinued.

ThreadInformation - A pointer to a buffer that contains the information to set in the specified thread. The format and contents of the buffer depend on the specified information class being queried.

ThreadInformation Format by Information Class

ThreadPriority - Data type is *PKPRIORITY*.

KPRIORITY *Priority* - Specifies the priority of the thread.

ThreadAffinityMask - Data type is *PKAFFINITY*.

KAFFINITY *AffinityMask* - Specifies the affinity mask assigned to the specified thread. The specified mask is anded with the process' default affinity mask and with the system wide affinity mask (which specifies the entire set of active processors in the system). The net effect is to limit a threads allowable affinity mask such that it is a subset of the maximum affinity mask in the current

configuration, and is also a subset of the affinity allowed to the process. Attempting to set an affinity that specifies no processors is an error condition.

ThreadImpersonationToken - Data type is *PHANDLE*. The handle value is that of an impersonation token, or may be null to indicate impersonation is to be discontinued.

ThreadInformationLength - Specifies the length in bytes of the thread information buffer.

4.13 NtImpersonateThread

Sets a server thread to be impersonating a client thread.

NTSTATUS
NtImpersonateThread(
IN HANDLE *ServerThread*,
IN HANDLE *ClientThread*
);

Parameters:

ServerThread - A handle to the thread which is to be set to impersonate the client thread. This handle must be open for **THREAD_SET_THREAD_TOKEN** access.

ClientThread - A handle to the thread to be impersonated. This handle must be open for **THREAD_IMPERSONATE** access.

This service causes the thread specified by the *ServerThread* argument to impersonate the thread specified by the *ClientThread* argument. The impersonation will have the following security quality of service parameters:

- Delegation Level.
- Dynamic Tracking.
- Not EffectiveOnly.

4.14 NtAlertThread

A thread can be alerted with the **NtAlertThread** function:

NTSTATUS
NtAlertThread(
IN HANDLE *ThreadHandle*
);

Parameters:

ThreadHandle - A handle to the thread to be alerted.

This function provides a mechanism that can be used to interrupt thread execution in the caller's previous mode (if this service is called from user mode the alert mode is user; otherwise, the alert mode is kernel) at well defined points.

Each thread has an alerted flag for each of the processor modes user and kernel. These flags are set by calling the **NtAlertThread** function.

If **NtAlertThread** is called and the target thread is in a wait state, then several additional tests are performed to determine the correct action to take.

If the mode of the wait is user (e.g. **NtWait** was called from user mode), and the alert mode is user, then a thread specific user mode **APC** is queued to the thread and the thread's wait will complete with a status of "alerted". When the **APC** executes it will raise the "alerted" condition.

If the mode of the wait is user or kernel, and the wait is alertable, then the thread's wait will complete with a status of "alerted".

If the target thread is not in a wait state, then the appropriate alerted bit in the target thread is set. Executing an **NtTestAlert**, or an alertable **NtWait** will clear the bit, return a status, and possibly queue a user mode **APC**.

This service requires *THREAD_ALERT* access to the specified thread.

4.15 NtTestAlert

A thread can test its alerted flag using the **NtTestAlert** function.

NTSTATUS
NtTestAlert();

The **NtTestAlert** function tests the calling thread's alerted flag for the thread's previous processor mode (i.e. if this function is called from user mode, the user mode alerted flag is tested; otherwise, the kernel mode alerted flag is tested). If the appropriate alerted flag is set, then the status value "alerted" is returned and the alerted flag is cleared; otherwise, a "normal" status value is returned. If the alerted flag was set and the previous mode is user, then a user **APC** is queued to the thread. When the **APC** executes, it will raise the "alerted" condition.

In addition, **NtTestAlert** tests whether a user **APC** should be delivered. If the previous mode is user and the user **APC** queue contains an entry, then **APC** pending is set in the thread (this will cause an **APC** to be delivered to the thread on a transition from kernel mode into user mode).

4.16 NtAlertResumeThread

A thread can be alerted and resumed with the **NtAlertResumeThread** function:

NTSTATUS
NtAlertResumeThread(
    IN HANDLE *ThreadHandle,*
    OUT PULONG *PreviousSuspendCount* **OPTIONAL**
    );

Parameters:

ThreadHandle - A handle to the thread to be alerted and resumed.

PreviousSuspendCount - A pointer to the variable that receives the thread's previous suspend count.

Resuming and alerting a thread reverses the effects of a previous call to **NtSuspendThread** and causes the thread to be interrupted out of an alertable kernel mode wait with a status of "alerted". This function is provided to allow a subsystem to resume a thread and interrupt it out of an interruptible system service.

When an attempt is made to resume and alert a thread, the thread is alerted with a kernel mode alert, and its suspend count is examined. If the count is zero, then the service returns the suspend count. Otherwise, the count is decremented and if the count reaches zero, the thread resumes. In either case, the previous value of the thread's suspend count is returned. A non-zero value indicates that the thread was previously suspended. The value minus 1 specifies the number of calls to **NtResumeThread** that must be made in order to bring the thread out of the suspend state.

If the thread was waiting in a kernel mode alertable wait, its wait completes with a status of alerted.

This service requires *THREAD_SUSPEND_RESUME* and *THREAD_ALERT* access to the specified thread.

4.17 NtRegisterThreadTerminationPort

A thread can arrange for a port to be notified when it terminates using **NtRegisterThreadTerminationPort**.

NTSTATUS
NtRegisterThreadTerminationPort(
    IN HANDLE *PortHandle*
    );

Parameters:

ULONG *PortHandle* - A handle to the port object that is to be notified when the subject thread terminates.

The **NtRegisterThreadTerminationPort** function is designed to allow a thread to specify a port object that is to be send a thread termination datagram when the subject thread terminates. Multiple calls to this service cause multiple ports to be notified when the thread terminates.

Each thread has a list of ports that are to be notified via a thread termination datagram when the thread terminates. When a thread terminates, the list is scanned and for each entry in the list, a thread termination datagram specifying the thread's client identifier and exit status is sent to the port. If during the send operation any errors occur (e.g. the port's connection was broken...) the system skips to the next entry in the list.

> *\ There is no need to provide this type of service at the process level since all of the process's port objects are closed during process termination. When a port object is closed (for the last time) its connections are broken, and the port that it was connected to is notified. *

The service is useful for subsystems that maintain per thread state (e.g. The Presentation Manager (**PM**) Subsystem). During the subsystem initialization that occurs in the client thread (e.g. calling **WinInitialize**), a call can be made to **NtRegisterThreadTerminationPort** specifying the port to the subsystem. When the thread terminates, the subsystem will receive a thread termination datagram. This datagram can be used as a signal to the subsystem that allows it to free up any thread specific resources.

Another use of this service is to allow a process to be notified when one of its own threads terminates. In order to do this, a multithreaded process creates a port to itself. A monitor thread monitors this port for thread termination datagrams. Each thread (in its startup routine) calls **NtRegisterThreadTerminationPort** specifying the port. Whenever a thread in the process terminates, the monitor thread is notified via the termination datagram. The monitor thread can use this event to perform appropriate actions.

4.18 NtImpersonateThread

NTSTATUS
NtImpersonateThread(
IN HANDLE *ServerThreadHandle*,
IN HANDLE *ClientThreadHandle*,
IN PSECURITY_QUALITY_OF_SERVICE *SecurityQos*
)

Arguments:

ServerThreadHandle - Is a handle to the server thread (the impersonator, or doing the impersonation). This handle must be open for THREAD_IMPERSONATE access.

ClientThreadHandle - Is a handle to the Client thread (the impersonatee, or one being impersonated). This handle must be open for THREAD_DIRECT_IMPERSONATION access.

SecurityQos - A pointer to security quality of service information indicating what form of impersonation is to be performed.

Return Value:

STATUS_SUCCESS - Indicates the call completed successfully.

Routine Description:

This routine is used to cause the server thread to impersonate the client thread. The impersonation is done according to the specified quality of service parameters.

5. System Information API

The following programming interface provide support for querying information about the system:

NtQuerySystemInformation - Returns information about the system.

5.1 NtQuerySystemInformation

Information about the system can be retreived using the NtQuerySystemInformation system service.

NTSTATUS
NtQuerySystemInformation(
IN SYSTEM_INFORMATION_CLASS *SystemInformationClass,*
OUT PVOID *SystemInformation,*
IN ULONG *SystemInformationLength,*
OUT PULONG *ReturnLength* **OPTIONAL**
)

Parameters:

SystemInformationClass - The system information class about which to retrieve information.

SystemInformation - A pointer to a buffer which receives the specified information. The format and content of the buffer depend on the specified system information class.

SystemInformation Format by Information Class:

SystemBasicInformation - Data type is PSYSTEM_BASIC_INFORMATION

SYSTEM_BASIC_INFORMATION Structure

ULONG *OemMachineId* - An OEM specific bit pattern that identifies the machine configuration.

ULONG *TimerResolutionInMicroSeconds* - The resolution of the hardware time. All time values in Windows NT are specified as 64-bit LARGE_INTEGER values in units of 100 nanoseconds. This field allows an application to understand how many of the low order bits of a system time value are insignificant.

ULONG *PageSize* - The physical page size for virtual memory objects. Physical memory is committed in PageSize chunks.

ULONG *AllocationGranularity* - The logical page size for virtual memory objects. Allocating 1 byte of virtual memory will actually allocate AllocationGranularity bytes of virtual memory. Storing into that byte will commit the first physical page of the virtual memory.

ULONG *MinimumUserModeAddress* - The smallest valid user mode address. The first AllocationGranullarity bytes of the virtual address space are reserved. This forces access violations for code the dereferences a zero pointer.

ULONG *MaximumUserModeAddress* - The largest valid used mode address. The next AllocationGranullarity bytes of the virtual address space are reserved. This allows system service routines to validate user mode pointer parameters quickly.

KAFFINITY *ActiveProcessorsAffinityMask* - The system wide affinity mask that specifies the set of processors configured into the system. This set represents the maximum allowable affinity of any thread within the system.

CCHAR *NumberOfProcessors* - The number of processors in the current hardware configuration.

SystemProcessorInformation - Data type is SYSTEM_PROCESSOR_INFORMATION

SYSTEM_PROCESSOR_INFORMATION Structure

ULONG *ProcessorType* - The processor type.

ProcessorType Values:

PROCESSOR_INTEL_386

PROCESSOR_INTEL_486

PROCESSOR_INTEL_860

PROCESSOR_MIPS_R2000

PROCESSOR_MIPS_R3000

PROCESSOR_MIPS_R4000

ULONG *ProcessorStepping* - The processor stepping. The high order 16 bits specify the stepping letter (0==A, 1==B, etc.) and the low order 16 bits specify the stepping level (e.g. 0, 1, 2, etc.).

ULONG *ProcessorOptions* - Flags that specify processor options that may or may not be present. The flags are processor specific.

ProcessOptions flags for PROCESSOR_INTEL_386:

PROCESSOR_OPTION_387 - A 387 co-processor chip is present.

PROCESSOR_OPTION_WEITEK - A Weitek floating pointer co-processor chip is present.

SystemInformationLength - Specifies the length in bytes of the system information buffer.

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

Return Value:

NTSTATUS - *STATUS_SUCCESS* if the operation is successful and an appropriate error value otherwise.

The following status values may be returned by the function:

- *STATUS_SUCCESS* - successful completion.
- *STATUS_INVALID_INFO_CLASS* - The *SystemInformationClass* parameter did not specify a valid value.
- *STATUS_INFO_LENGTH_MISMATCH* - The value of the *SystemInformationLength* parameter did not match the length required for the information class requested by the *SystemInformationClass* parameter.
- *STATUS_ACCESS_VIOLATION* - Either the *SystemInformation* buffer pointer or the *ReturnLength* pointer value specified an invalid address.

6. Executive APIs

The following programming interfaces are available from within the **Windows NT** executive:

PsCreateSystemProcess - Creates a system process.
PsCreateSystemThread - Creates a system thread.
PsLookupProcessThreadByCid - Locates the process and thread using the specified **CID**.
PsChargePoolQuota - Charges pool quota to the specified process.
PsReturnPoolQuota - Returns pool quota to the specified process.
PsGetCurrentThread - Returns the address of the currently executing thread's thread object.

PsGetCurrentProcess - Returns the address of the process object that the currently executing thread is attached to.
ExGetPreviousMode - Returns the processor mode that the thread was executing in prior to the last trap.
PsRevertToSelf - Reverts the calling thread's access token to its original value.
PsReferencePrimaryToken - This function returns a pointer to the primary token of a process. The reference count of that primary token is incremented to protect the pointer returned.
PsDereferencePrimaryToken - This function releases a pointer to a primary token obtained using **PsReferencePrimaryToken()**.
PsReferenceImpersonationToken - This function returns a pointer to the impersonation token of a thread. The reference count of that impersonation token is incremented to protect the pointer returned.
PsDereferenceImpersonationToken - This function releases a pointer to a primary token obtained using **PsReferenceImpersonationToken()**.
PsOpenTokenOfProcess - This function does the thread specific processing of an **NtOpenThreadToken()** service.
PsOpenTokenOfThread - This function does the thread specific processing of an **NtOpenThreadToken()** service.
PsImpersonateClient -This routine sets up the specified thread so that it is impersonating the specified client.

6.1 PsCreateSystemProcess

A system process can be created using **PsCreateSystemProcess**.

NTSTATUS
PsCreateSystemProcess(
OUT HANDLE *ProcessHandle,*
IN ACCESS_MASK *DesiredAccess,*
IN POBJECT_ATTRIBUTES *ObjectAttributes* **OPTIONAL**,
);

Parameters:

ProcessHandle - A pointer to a variable that will receive the process object handle value.

DesiredAccess - The desired types of access to the created process. For a complete description of desired access flags, refer to the **NtCreateProcess API** description.

ObjectAttributes - An pointer to a structure that specifies the object's attributes. Refer to the *Object Management Specification* for details. Note that *OBJ_PERMANENT, OBJ_EXCLUSIVE, OBJ_OPEN_IF*, and *OBJ_OPEN_LINK* are not valid attributes for a process object.

Creating a system process creates a process object whose address space is initialized so that the "user" portion of the address space is empty, and the "system" portion of the address space maps the system. This option is not available from user-mode via **NtCreateProcess**. The process inherits its access token and quotas from the initial system process. It is created with an empty handle table. The process's debug and exception ports are **NULL**.

The system does not treat a process created through this API any differently than any other process. Any **Windows NT** API that requires a handle to a process object may specify a process created through this API.

6.2 PsCreateSystemThread

A system thread that executes in kernel mode can be created and a handle opened for access to the thread with the **PsCreateSystemThread** function:

NTSTATUS
PsCreateSystemThread(
    OUT PHANDLE *ThreadHandle,*
    IN ACCESS_MASK *DesiredAccess,*
    IN POBJECT_ATTRIBUTES *ObjectAttributes* **OPTIONAL**,
    IN HANDLE *ProcessHandle* **OPTIONAL**,
    OUT PCLIENT_ID *ClientId* **OPTIONAL**,
    IN PKSTART_ROUTINE *StartRoutine,*
    IN PVOID *StartContext*
    );

Parameters:

ThreadHandle - A pointer to a variable that will receive the thread object handle value.

DesiredAccess - The desired types of access to the created thread. For a complete description of desired access flags, refer to the **NtCreateThread API** description.

ObjectAttributes - An pointer to a structure that specifies the object's attributes. Refer to the *Object Management Specification* for details. Note that *OBJ_PERMANENT, OBJ_EXCLUSIVE, OBJ_OPEN_IF*, and *OBJ_OPEN_LINK* are not valid attributes for a thread object.

ProcessHandle - An open handle to the process object that the thread is to run in. The subject thread must have *PROCESS_CREATE_THREAD* access to this process. If this parameter is not supplied, then the thread will be created in the initial system process.

ClientId - A pointer to a structure that will receive the client identifier of the new thread.

StartRoutine - Supplies the address of a function in system space that the thread begins execution at. A return from this function causes the thread to terminate.

StartContext - Supplies a single argument passed to the thread when it begins execution.

Creating a system thread begins a separate thread of execution within the system. System threads may only execute in kernel-mode. A system thread has no **TEB**, or user-mode context. It is not possible to terminate a system thread using **NtTerminateThread** unless the thread is terminating itself.

6.3 PsLookupProcessThreadByCid

A process and thread can be located by client id using the **PsLookupProcessThreadByCid** function:

NTSTATUS
PsLookupProcessThreadByCid(
IN PCID *Cid,*
OUT PEPROCESS *Process* **OPTIONAL,**
OUT PETHREAD *Thread*
);

Parameters:

Cid - A pointer to the client id whose thread and process are to be located.

Process - An optional parameter that if specified receives a referenced pointer to the process object associated with the specified client id.

Thread - A parameter that receives a referenced pointer to the thread object associated with the specified client id.

6.4 PsChargePoolQuota

Pool quota can be charged to the specified process using the **PsChargePoolQuota** function:

VOID
PsChargePoolQuota(
IN PEPROCESS *Process,*
IN POOL_TYPE *PoolType***,**
IN ULONG *Amount*
);

Parameters:

Process - Supplies the address of a process to charge pool quota to.

PoolType - Supplies the pool type to charge the quota for.

Amount - Supplies the amount of quota to charge to the process.

The **PsChargePoolQuota** function is designed to charge pool quota to a process subject to the quota limits of that process. If the quota charge would cause the process to exceed its quota limit for the specified pool type, then an *STATUS_QUOTA_EXCEEDED* exception is raised and the charge is not made. Otherwise, the quota pool usage of the specified process is adjusted (incremented) to account for the quota being charged to the process.

6.5 PsReturnPoolQuota

Pool quota can be returned to the specified process using the **PsReturnPoolQuota** function:

VOID
PsReturnPoolQuota(
    IN PEPROCESS *Process,*
    IN POOL_TYPE *PoolType***,**
    IN ULONG *Amount*
    );

Parameters:

Process - Supplies the address of a process to return pool quota to.

PoolType - Supplies the pool type to return the quota for.

Amount - Supplies the amount of quota to return to the process.

The **PsReturnPoolQuota** function is designed to return pool quota to a process to reverse the effects of a previous call to **PsChargePoolQuota**. The system will catch attemps to return more quota to the process than the process has been charged for and bug check. Otherwise, the quota pool usage of the specified process is adjusted (decremented) to account for the quota being returned to the process.

6.6 PsGetCurrentThread

The address of the thread object of the currently executing thread is returned using the **GetCurrentThread** function:

PETHREAD
PsGetCurrentThread();

6.7 PsGetCurrentProcess

The address of the process object that the currently executing thread is attached to is returned using the **PsGetCurrentProcess** function:

PEPROCESS
PsGetCurrentProcess();

6.8 KeGetPreviousMode

The processor mode that the current thread was running in prior to the last trap or interrupt can be determined using the **KeGetPreviousMode** function:

KPROCESSOR_MODE
KeGetPreviousMode();

The **KeGetPreviousMode** function is used mainly inside **Windows NT** system services to determine the processor mode that the thread was executing in prior to the system service.

6.9 PsRevertToSelf

The current can switch to its original access token using the **PsRevertToSelf** function:

VOID
PsRevertToSelf();

The **PsRevertToSelf** function switches the access token used by the calling thread back to its original value. This is the same token that would have been in effect if the thread had never called **PsImpersonateThread**.

6.10 PsReferencePrimaryToken

PACCESS_TOKEN
PsReferencePrimaryToken(
  IN PEPROCESS *Process*
)

Arguments:

Process - Supplies the address of the process whose primary token is to be referenced.

Return Value:

A pointer to the specified process's primary token.

Routine Description:

This function returns a pointer to the primary token of a process. The reference count of that primary token is incremented to protect the pointer returned.

When the pointer is no longer needed, it should be freed using **PsDereferencePrimaryToken()**.

6.11 PsDereferencePrimaryToken

VOID
PsDereferencePrimaryToken(
  IN PACCESS_TOKEN *PrimaryToken*
)

Arguments:

PrimaryToken - Pointer to a token obtained using **PsReferencePrimaryToken()**.

Return Value:

None.

Routine Description:

This function causes the referenced primary token to be dereferenced. This token is expected to have been referenced using **PsReferencePrimaryToken()**.

6.12 PsReferenceImpersonationToken

PACCESS_TOKEN
PsReferenceImpersonationToken(
  IN PETHREAD *Thread*,
  OUT PBOOLEAN *CopyOnOpen*,
  OUT PBOOLEAN *EffectiveOnly*,
  OUT PSECURITY_IMPERSONATION_LEVEL *ImpersonationLevel*,
)

Arguments:

Thread - Supplies the address of the thread whose impersonation token is to be referenced.

CopyOnOpen - The current value of the Thread->CopyOnOpen field.

EffectiveOnly - The current value of the Thread->EffectiveOnly field.

ImpersonationLevel - The current value of the Thread->ImpersonationLevel field.

Return Value:

A pointer to the specified thread's impersonation token.

If the thread is not currently impersonating a client, then NULL is returned.

Routine Description:

This function returns a pointer to the impersonation token of a thread. The reference count of that impersonation token is incremented to protect the pointer returned.

If the thread is not currently impersonating a client, then a null pointer is returned.

If the thread is impersonating a client, then information about the means of impersonation are also returned (ImpersonationLevel).

If a non-null value is returned, then **PsDereferenceImpersonationToken()** must be called to decrement the token's reference count when the pointer is no longer needed.

6.13 PsDereferenceImpersonationToken

VOID
PsDereferenceImpersonationToken(
  IN PACCESS_TOKEN *ImpersonationToken*
)

Arguments:

ImpersonationToken - Pointer to a token obtained using **PsReferenceImpersonationToken()**.

Return Value:

None.

Routine Description:

This function causes the referenced impersonation token to be dereferenced. This token is expected to have been referenced using **PsReferenceImpersonationToken()**.

6.14 PsOpenTokenOfProcess

NTSTATUS
PsOpenTokenOfProcess(
  IN HANDLE *ProcessHandle*,
  OUT PACCESS_TOKEN **Token*
)

Arguments:

ProcessHandle - Supplies a handle to a process object whose primary token is to be opened.

Token - If successful, receives a pointer to the process's token object.

Return Value:

STATUS_SUCCESS - Indicates the call completed successfully.

Status may also be any value returned by an attemp the reference the process object for PROCESS_QUERY_INFORMATION access.

Routine Description:

This function does the process specific processing of an NtOpenProcessToken() service.

The service validates that the handle has appropriate access to referenced process. If so, it goes on to reference the primary token object to prevent it from going away while the rest of the NtOpenProcessToken() request is processed.

NOTE: If this call completes successfully, the caller is responsible for decrementing the reference count of the target token. This must be done using the PsDereferencePrimaryToken() API.

6.15 PsOpenTokenOfThread

NTSTATUS
PsOpenTokenOfThread(
IN HANDLE *ThreadHandle*,
OUT PACCESS_TOKEN **Token*,
OUT PBOOLEAN CopyOnOpen,
OUT PBOOLEAN *EffectiveOnly*,
OUT PSECURITY_IMPERSONATION_LEVEL *ImpersonationLevel*
)

Arguments:

ThreadHandle - Supplies a handle to a thread object.

Token - If successful, receives a pointer to the thread's token object.

CopyOnOpen - The current value of the Thread->CopyOnOpen field.

EffectiveOnly - The current value of the Thread->EffectiveOnly field.

ImpersonationLevel - The current value of the Thread->ImpersonationLevel field.

Return Value:

STATUS_SUCCESS - Indicates the call completed successfully.

STATUS_NO_TOKEN - Indicates the referenced thread is not currently impersonating a client.

STATUS_CANT_OPEN_ANONYMOUS - Indicates the client requested anonymous impersonation level. An anonymous token can not be openned.

status may also be any value returned by an attemp the reference the thread object for THREAD_QUERY_INFORMATION access.

Routine Description:

This function does the thread specific processing of an **NtOpenThreadToken()** service.

The service validates that the handle has appropriate access to reference the thread. If so, it goes on to increment the reference count of the token object to prevent it from going away while the rest of the NtOpenThreadToken() request is processed.

NOTE: If this call completes successfully, the caller is responsible for decrementing the reference count of the target token. This must be done using PsDereferenceImpersonationToken().

6.16 PsImpersonateClient

VOID
PsImpersonateClient(
IN PETHREAD *Thread*,
IN BOOLEAN *CopyOnOpen*,
IN BOOLEAN *EffectiveOnly*,
IN SECURITY_IMPERSONATION_LEVEL *ImpersonationLevel*
)

Arguments:

Thread - points to the thread which is going to impersonate a client.

CopyOnOpen - If TRUE, indicates the token is considered to be private by the assigner and should be copied if opened. For example, a session layer may be using a token to represent a client's context. If the session is trying to synchronize the context of the client, then user mode code should not be given direct access to the session layer's token.

This field is ANDed with the DirectAccess field of the ClientContext to establish the CopyOnOpen value actually assigned to the impersonation.

CopyOnOpen - If TRUE, indicates the token is considered to be private by the assigner and should be copied if opened. For example, a session layer may be using a token to represent a client's context. If the session is trying to synchronize the context of the client, then user mode code should not be given direct access to the session layer's token.

Basically, session layers should always specify TRUE for this, while tokens assigned by the server itself (handle based) should specify FALSE.

EffectiveOnly - Is a boolean value to be assigned as the Thread->EffectiveOnly field value for the impersonation. A value of FALSE indicates the server is allowed to enable currently disabled groups and privileges.

ImpersonationLevel - Is the impersonation level that the server is allowed to access the token with.

Return Value:

STATUS_SUCCESS - Indicates the call completed successfully.

Routine Description:

This routine sets up the specified thread so that it is impersonating the specified client. This will result in the reference count of the token representing the client being incremented to reflect the new reference.

If the thread is currently impersonating a client, that token will be dereferenced.

Revision History

Revision 1.2

1. Simplify Create Thread.
2. Remove create if, permanent, and other object options that are only there for orthoganality.
3. Add port notification handlers.
4. Add 32 bit exit status for process and thread termination.
5. Add NtAlertThread/NtAlertResumeThread.
6. Add get thread info.
7. Add debugger port and subsystem port to process creation.
8. Add process get/set info place holders.

Revision 1.3

1. Complicate create thread
2. Reorganize considerations

Revision 1.15, August 20, 1990, Jim Kelly

1. Eliminated previous token query information levels. This is done using NtOpenProcessToken() and NtOpenThreadToken().
2. Added information level allowing the setting of a primary token.
3. Added PsReferenceImpersonationToken() and PsDereferenceImpersonationToken().
4. Added PsReferencePrimaryToken() and PsDereferencePrimaryToken().
5. Added PsImpersonateClient().
6. Added PsOpenTokenOfProcess() and PsOpenTokenOfThread().
7. Eliminated PsLockToken(), PsUnlockToken(), and PsImpersonateThread().
8. Minor grammatical and spelling corrections.
9. Removed *TokenLength* field from THREAD_BASIC_INFORMATION structure.

Revision 1.22, February 7, 1991, Jim Kelly.

1. Changed THREAD_IMPERSONATE_CLIENT access type to be THREAD_SET_THREAD_TOKEN.

2. Added the ability to directly impersonate a thread. This resulted in a new API (**NtImpersonateThread()**) and a new access type (**THREAD_IMPERSONATE**).

3. Corrected minor typos.

Revision 1.24, February 28, 1991, Mark Lucovsky.

1) ???

Revision 1.24, April 21, 1991, Jim Kelly (JimK).

1. Added NtImpersonateThread() service.

Revision 1.25, May 2, 1991, Bryan Willman (bryanwi).

1. Added ProcessLdtInformation and ProcessLdtSize to set of data types for NtQueryInformationProcess and NtSetInformationProcess.

Revision 1.26, May 24, 1991, Dave Hastings (daveh).

1. Added ThreadDescriptorTableEntry to NtQueryInformationThread.

2. Allowed querying of specific regions of the LDT for ProcessLdtInformation.

Revision 1.27, January 14, 1992, Jim Kelly (JimK).

1. Eliminated PROCESS_SET_ACCESS_TOKEN as an access type. Changing the primary token of a process will be protected by PROCESS_SET_INFORMATION followed by a privilege test at the time the change is requested (rather than at open time).