Contents

CloseThreadpoolCleanupGroup

CloseThreadpoolCleanupGroupMembers

CloseThreadpoolIo

CloseThreadpoolTimer

CloseThreadpoolWait

CloseThreadpoolWork

ConvertFiberToThread

ConvertThreadToFiber

ConvertThreadToFiberEx

CreateDispatcherQueueController

CreateFiber

CreateFiberEx

CreateJobObject

CreateProcess

CreateProcessAsUser

CreateProcessWithLogonW

CreateProcessWithTokenW

CreateRemoteThread

CreateRemoteThreadEx

CreateThread

CreateThreadpool

CreateThreadpoolCleanupGroup

CreateThreadpoolIo

CreateThreadpoolTimer

CreateThreadpoolWait

CreateThreadpoolWork

CreateUmsCompletionList

CreateUmsThreadContext

DeleteFiber

DeleteProcThreadAttributeList

DeleteUmsCompletionList

DeleteUmsThreadContext

DequeueUmsCompletionListItems

DestroyThreadpoolEnvironment

DisassociateCurrentThreadFromCallback

EnterUmsSchedulingMode

ExecuteUmsThread

ExitProcess

ExitThread

FiberProc

FlsAlloc

PFLS_CALLBACK_FUNCTION

FlsFree

FlsGetValue

FlsSetValue

FlushProcessWriteBuffers

FreeEnvironmentStrings

FreeLibraryWhenCallbackReturns

FreeMemoryJobObject

GetActiveProcessorCount

GetActiveProcessorGroupCount

GetAutoRotationState

GetDisplayAutoRotationPreferencesByProcessId

GetDisplayAutoRotationPreferences

GetCommandLine

GetCurrentProcess

GetCurrentProcessId

GetCurrentProcessorNumber

GetCurrentProcessorNumberEx

GetCurrentThread

GetCurrentThreadId

GetCurrentThreadStackLimits

GetCurrentUmsThread

GetEnvironmentStrings

GetEnvironmentVariable

GetExitCodeProcess

GetExitCodeThread

GetGuiResources

GetLogicalProcessorInformation

GetLogicalProcessorInformationEx

GetMaximumProcessorCount

GetMaximumProcessorGroupCount

GetNextUmsListItem

GetNumaAvailableMemoryNode

GetNumaAvailableMemoryNodeEx

GetNumaHighestNodeNumber

GetNumaNodeNumberFromHandle

GetNumaNodeProcessorMask

GetNumaNodeProcessorMaskEx

GetNumaProcessorNode

GetNumaProcessorNodeEx

GetNumaProximityNode

GetNumaProximityNodeEx

GetPriorityClass

GetProcessAffinityMask

GetProcessDefaultCpuSets

GetProcessGroupAffinity

GetProcessHandleCount

GetProcessId

GetProcessIdOfThread

GetProcessInformation

GetProcessIoCounters

GetProcessMitigationPolicy

GetProcessPriorityBoost

GetProcessShutdownParameters

GetProcessTimes

GetProcessVersion

GetProcessWorkingSetSize

GetProcessWorkingSetSizeEx

GetProcessorSystemCycleTime

GetStartupInfo

GetSystemCpuSetInformation

GetThreadDescription

GetThreadGroupAffinity

GetThreadId

GetThreadIdealProcessorEx

GetThreadInformation

GetThreadIOPendingFlag

GetThreadPriority

GetThreadPriorityBoost

GetThreadSelectedCpuSets

GetThreadTimes

GetUmsCompletionListEvent

GetUmsSystemThreadInformation

InitializeProcThreadAttributeList

InitializeThreadpoolEnvironment

IoCompletionCallback

IsImmersiveProcess

IsProcessInJob

IsProcessCritical

IsThreadAFiber

IsThreadpoolTimerSet

IsWow64Message

IsWow64Process

IsWow64Process2

LeaveCriticalSectionWhenCallbackReturns

NeedCurrentDirectoryForExePath

NtGetCurrentProcessorNumber

NtQueryInformationProcess

NtQueryInformationThread

OpenJobObject

OpenProcess

OpenThread

QueryFullProcessImageName

QueryIdleProcessorCycleTime

QueryIdleProcessorCycleTimeEx

QueryInformationJobObject

QueryIoRateControlInformationJobObject

QueryProcessAffinityUpdateMode

QueryProcessCycleTime

QueryProtectedPolicy

QueryThreadCycleTime

QueryThreadpoolStackInformation

QueryUmsThreadInformation

QueueUserWorkItem

ReleaseMutexWhenCallbackReturns

ReleaseSemaphoreWhenCallbackReturns

ResumeThread

SetDisplayAutoRotationPreferences

SetEnvironmentVariable

SetEventWhenCallbackReturns

SetInformationJobObject

SetIoRateControlInformationJobObject

SetPriorityClass

SetProcessAffinityMask

SetProcessAffinityUpdateMode

SetProcessDefaultCpuSets

SetProcessInformation

SetProcessMitigationPolicy

SetProcessPriorityBoost

SetProcessRestrictionExemption

SetProcessShutdownParameters

SetProcessWorkingSetSize

SetProcessWorkingSetSizeEx

SetProtectedPolicy

SetThreadAffinityMask

SetThreadDescription

SetThreadGroupAffinity

SetThreadIdealProcessor

SetThreadIdealProcessorEx

SetThreadInformation

SetThreadpoolCallbackCleanupGroup

SetThreadpoolCallbackLibrary

SetThreadpoolCallbackPersistent

SetThreadpoolCallbackPool

SetThreadpoolCallbackPriority

SetThreadpoolCallbackRunsLong

SetThreadpoolStackInformation

SetThreadpoolThreadMaximum

SetThreadpoolThreadMinimum

SetThreadpoolTimer

SetThreadpoolTimerEx

SetThreadpoolWait

SetThreadpoolWaitEx

SetThreadPriority

SetThreadPriorityBoost

SetThreadSelectedCpuSets

SetThreadStackGuarantee

SetUmsThreadInformation

SimpleCallback

Sleep

SleepEx

StartThreadpoolIo

SubmitThreadpoolWork

SuspendThread

SwitchToFiber

SwitchToThread

TerminateJobObject

TerminateProcess

TerminateThread

ThreadProc

TimerCallback

TlsAlloc

TlsFree

TlsGetValue

TlsSetValue

TpInitializeCallbackEnviron

TpDestroyCallbackEnviron

TpSetCallbackActivationContext

TpSetCallbackCleanupGroup

TpSetCallbackFinalizationCallback

TpSetCallbackLongFunction

TpSetCallbackNoActivationContext

TpSetCallbackPersistent

TpSetCallbackPriority

TpSetCallbackRaceWithDll

TpSetCallbackThreadpool

TrySubmitThreadpoolCallback

UmsSchedulerProc

UmsThreadYield

UpdateProcThreadAttribute

UserHandleGrantAccess

WaitCallback

WaitForInputIdle

JOBOBJECT_IO_RATE_CONTROL_INFORMATION

JOBOBJECT_LIMIT_VIOLATION_INFORMATION

JOBOBJECT_LIMIT_VIOLATION_INFORMATION_2

JOBOBJECT_NET_RATE_CONTROL_INFORMATION

JOBOBJECT_NOTIFICATION_LIMIT_INFORMATION

JOBOBJECT_NOTIFICATION_LIMIT_INFORMATION_2

JOBOBJECT_SECURITY_LIMIT_INFORMATION

MEMORY_PRIORITY_INFORMATION

NUMA_NODE_RELATIONSHIP

ORIENTATION_PREFERENCE

PEB

PEB_LDR_DATA

PROCESS_INFORMATION

PROCESS_MEMORY_EXHAUSTION_INFO

PROCESS_MITIGATION_ASLR_POLICY

PROCESS_MITIGATION_BINARY_SIGNATURE_POLICY

PROCESS_MITIGATION_CONTROL_FLOW_GUARD_POLICY

PROCESS_MITIGATION_DEP_POLICY

PROCESS_MITIGATION_DYNAMIC_CODE_POLICY

PROCESS_MITIGATION_EXTENSION_POINT_DISABLE_POLICY

PROCESS_MITIGATION_FONT_DISABLE_POLICY

PROCESS_MITIGATION_IMAGE_LOAD_POLICY

PROCESS_MITIGATION_STRICT_HANDLE_CHECK_POLICY

PROCESS_MITIGATION_SYSTEM_CALL_DISABLE_POLICY

PROCESSOR_GROUP_INFO

PROCESSOR_NUMBER

PROCESSOR_RELATIONSHIP

RTL_USER_PROCESS_PARAMETERS

STARTUPINFO

STARTUPINFOEX

SYSTEM_CPU_SET_INFORMATION

SYSTEM_LOGICAL_PROCESSOR_INFORMATION

Processes and Threads

9/16/2022 • 2 minutes to read • Edit Online

An application consists of one or more processes. A *process*, in the simplest terms, is an executing program. One or more threads run in the context of the process. A *thread* is the basic unit to which the operating system allocates processor time. A thread can execute any part of the process code, including parts currently being executed by another thread.

A *job object* allows groups of processes to be managed as a unit. Job objects are namable, securable, sharable objects that control attributes of the processes associated with them. Operations performed on the job object affect all processes associated with the job object.

A *thread pool* is a collection of worker threads that efficiently execute asynchronous callbacks on behalf of the application. The thread pool is primarily used to reduce the number of application threads and provide management of the worker threads.

A *fiber* is a unit of execution that must be manually scheduled by the application. Fibers run in the context of the threads that schedule them.

User-mode scheduling (UMS) is a lightweight mechanism that applications can use to schedule their own threads. UMS threads differ from fibers in that each UMS thread has its own thread context instead of sharing the thread context of a single thread.

- What's New in Processes and Threads
- About Processes and Threads
- Using Processes and Threads
- Process and Thread Reference

What's New in Processes and Threads

9/16/2022 • 3 minutes to read • Edit Online

Windows 7 and Windows Server 2008 R2 include the following new programming elements for processes and threads.

New Capabilities

The 64-bit versions of Windows 7 and Windows Server 2008 R2 support more than 64 logical processors on a single computer. For more information, see Processor Groups.

User-mode scheduling (UMS) is a lightweight mechanism that applications can use to schedule their own threads. For more information, see User-Mode Scheduling.

New Functions

The following new functions are used with processors and processor groups.

FUNCTION	DESCRIPTION
CreateRemoteThreadEx	Creates a thread that runs in the virtual address space of another process and optionally specifies extended attributes such as processor group affinity.
GetActiveProcessorCount	Returns the number of active processors in a processor group or in the system.
GetActiveProcessorGroupCount	Returns the number of active processor groups in the system.
GetCurrentProcessorNumberEx	Retrieves the processor group and number of the logical processor in which the calling thread is running.
GetLogicalProcessorInformationEx	Retrieves information about the relationships of logical processors and related hardware.
GetMaximumProcessorCount	Returns the maximum number of logical processors that a processor group or the system can have.
GetMaximumProcessorGroupCount	Returns the maximum number of processor groups that the system can have.
GetNumaAvailableMemoryNodeEx	Retrieves the amount of memory that is available in the specified node as a USHORT value.
GetNumaNodeNumberFromHandle	Retrieves the NUMA node associated with the underlying device for a file handle.
GetNumaNodeProcessorMaskEx	Retrieves the processor mask for the specified NUMA node as a USHORT value.

FUNCTION	DESCRIPTION
GetNumaProcessorNodeEx	Retrieves the node number of the specified logical processor as a USHORT value.
GetNumaProximityNodeEx	Retrieves the node number as a USHORT value for the specified proximity identifier.
GetProcessGroupAffinity	Retrieves the processor group affinity of the specified process.
GetProcessorSystemCycleTime	Retrieves the cycle time each processor in the specified group spent executing deferred procedure calls (DPCs) and interrupt service routines (ISRs).
GetThreadGroupAffinity	Retrieves the processor group affinity of the specified thread.
GetThreadIdealProcessorEx	Retrieves the processor number of the ideal processor for the specified thread.
QueryIdleProcessorCycleTimeEx	Retrieves the accumulated cycle time for the idle thread on each logical processor in the specified processor group.
SetThreadGroupAffinity	Sets the processor group affinity for the specified thread.
SetThreadIdealProcessorEx	Sets the ideal processor for the specified thread and optionally retrieves the previous ideal processor.

The following new functions are used with thread pools.

FUNCTION	DESCRIPTION
QueryThreadpoolStackInformation	Retrieves the stack reserve and commit sizes for threads in the specified thread pool.
SetThreadpoolCallbackPersistent	Specifies that the callback should run on a persistent thread.
SetThreadpoolCallbackPriority	Specifies the priority of a callback function relative to other work items in the same thread pool.
SetThreadpoolStackInformation	Sets the stack reserve and commit sizes for new threads in the specified thread pool.

The following new functions are used with UMS.

FUNCTION	DESCRIPTION
CreateUmsCompletionList	Creates a UMS completion list.
CreateUmsThreadContext	Creates a UMS thread context to represent a UMS worker thread.

FUNCTION	DESCRIPTION
DeleteUmsCompletionList	Deletes the specified UMS completion list. The list must be empty.
DeleteUmsThreadContext	Deletes the specified UMS thread context. The thread must be terminated.
DequeueUmsCompletionListItems	Retrieves UMS worker threads from the specified UMS completion list.
EnterUmsSchedulingMode	Converts the calling thread into a UMS scheduler thread.
ExecuteUmsThread	Runs the specified UMS worker thread.
GetCurrentUmsThread	Returns the UMS thread context of the calling UMS thread.
GetNextUmsListItem	Returns the next UMS thread context in a list of UMS thread contexts.
GetUmsCompletionListEvent	Retrieves a handle to the event associated with the specified UMS completion list.
QueryUmsThreadInformation	Retrieves information about the specified UMS worker thread.
SetUmsThreadInformation	Sets application-specific context information for the specified UMS worker thread.
UmsSchedulerProc	The application-defined UMS scheduler entry point function associated with a UMS completion list.
UmsThreadYield	Yields control to the UMS scheduler thread on which the calling UMS worker thread is running.

New Structures

STRUCTURE	DESCRIPTION
CACHE_RELATIONSHIP	Describes cache attributes.
GROUP_AFFINITY	Contains a processor group-specific affinity, such as the affinity of a thread.
GROUP_RELATIONSHIP	Contains information about processor groups.
NUMA_NODE_RELATIONSHIP	Contains information about a NUMA node in a processor group.
PROCESSOR_GROUP_INFO	Contains the number and affinity of processors in a processor group.

STRUCTURE	DESCRIPTION
PROCESSOR_NUMBER	Represents a logical processor in a processor group.
PROCESSOR_RELATIONSHIP	Contains information about affinity within a processor group.
SYSTEM_LOGICAL_PROCESSOR_INFORMATION_EX	Contains information about the relationships of logical processors and related hardware.
UMS_CREATE_THREAD_ATTRIBUTES	Specifies attributes for a UMS worker thread.
UMS_SCHEDULER_STARTUP_INFO	Specifies attributes for a UMS scheduler thread

About Processes and Threads

9/16/2022 • 2 minutes to read • Edit Online

Each *process* provides the resources needed to execute a program. A process has a virtual address space, executable code, open handles to system objects, a security context, a unique process identifier, environment variables, a priority class, minimum and maximum working set sizes, and at least one thread of execution. Each process is started with a single thread, often called the *primary thread*, but can create additional threads from any of its threads.

A *thread* is the entity within a process that can be scheduled for execution. All threads of a process share its virtual address space and system resources. In addition, each thread maintains exception handlers, a scheduling priority, thread local storage, a unique thread identifier, and a set of structures the system will use to save the thread context until it is scheduled. The *thread context* includes the thread's set of machine registers, the kernel stack, a thread environment block, and a user stack in the address space of the thread's process. Threads can also have their own security context, which can be used for impersonating clients.

Microsoft Windows supports *preemptive multitasking*, which creates the effect of simultaneous execution of multiple threads from multiple processes. On a multiprocessor computer, the system can simultaneously execute as many threads as there are processors on the computer.

A *job object* allows groups of processes to be managed as a unit. Job objects are namable, securable, sharable objects that control attributes of the processes associated with them. Operations performed on the job object affect all processes associated with the job object.

An application can use the *thread pool* to reduce the number of application threads and provide management of the worker threads. Applications can queue work items, associate work with waitable handles, automatically queue based on a timer, and bind with I/O.

User-mode scheduling (UMS) is a lightweight mechanism that applications can use to schedule their own threads. An application can switch between UMS threads in user mode without involving the system scheduler and regain control of the processor if a UMS thread blocks in the kernel. Each UMS thread has its own thread context instead of sharing the thread context of a single thread. The ability to switch between threads in user mode makes UMS more efficient than thread pools for short-duration work items that require few system calls.

A *fiber* is a unit of execution that must be manually scheduled by the application. Fibers run in the context of the threads that schedule them. Each thread can schedule multiple fibers. In general, fibers do not provide advantages over a well-designed multithreaded application. However, using fibers can make it easier to port applications that were designed to schedule their own threads.

For more information, see the following topics:

- Multitasking
- Scheduling
- Multiple Threads
- Child Processes
- Thread Pools
- Job Objects
- User-Mode Scheduling
- Fibers

Multitasking

9/16/2022 • 2 minutes to read • Edit Online

A multitasking operating system divides the available processor time among the processes or threads that need it. The system is designed for preemptive multitasking; it allocates a processor *time slice* to each thread it executes. The currently executing thread is suspended when its time slice elapses, allowing another thread to run. When the system switches from one thread to another, it saves the context of the preempted thread and restores the saved context of the next thread in the queue.

The length of the time slice depends on the operating system and the processor. Because each time slice is small (approximately 20 milliseconds), multiple threads appear to be executing at the same time. This is actually the case on multiprocessor systems, where the executable threads are distributed among the available processors. However, you must use caution when using multiple threads in an application, because system performance can decrease if there are too many threads.

For more information, see the following topics:

- Advantages of Multitasking
- When to Use Multitasking
- Multitasking Considerations

Advantages of Multitasking

9/16/2022 • 2 minutes to read • Edit Online

To the user, the advantage of multitasking is the ability to have several applications open and working at the same time. For example, a user can edit a file with one application while another application is recalculating a spreadsheet.

To the application developer, the advantage of multitasking is the ability to create applications that use more than one process and to create processes that use more than one thread of execution. For example, a process can have a user interface thread that manages interactions with the user (keyboard and mouse input), and worker threads that perform other tasks while the user interface thread waits for user input. If you give the user interface thread a higher priority, the application will be more responsive to the user, while the worker threads use the processor efficiently during the times when there is no user input.

When to Use Multitasking

9/16/2022 • 2 minutes to read • Edit Online

There are two ways to implement multitasking: as a single process with multiple threads or as multiple processes, each with one or more threads. An application can put each thread that requires a private address space and private resources into its own process, to protect it from the activities of other process threads.

A multithreaded process can manage mutually exclusive tasks with threads, such as providing a user interface and performing background calculations. Creating a multithreaded process can also be a convenient way to structure a program that performs several similar or identical tasks concurrently. For example, a named pipe server can create a thread for each client process that attaches to the pipe. This thread manages the communication between the server and the client. Your process could use multiple threads to accomplish the following tasks:

- Manage input for multiple windows.
- Manage input from several communications devices.
- Distinguish tasks of varying priority. For example, a high-priority thread manages time-critical tasks, and a low-priority thread performs other tasks.
- Allow the user interface to remain responsive, while allocating time to background tasks.

It is typically more efficient for an application to implement multitasking by creating a single, multithreaded process, rather than creating multiple processes, for the following reasons:

- The system can perform a context switch more quickly for threads than processes, because a process has more overhead than a thread does (the process context is larger than the thread context).
- All threads of a process share the same address space and can access the process's global variables, which can simplify communication between threads.
- All threads of a process can share open handles to resources, such as files and pipes.

There are other techniques you can use in the place of multithreading. The most significant of these are as follows: asynchronous input and output (I/O), I/O completion ports, asynchronous procedure calls (APC), and the ability to wait for multiple events.

A single thread can initiate multiple time-consuming I/O requests that can run concurrently using asynchronous I/O. Asynchronous I/O can be performed on files, pipes, and serial communication devices. For more information, see Synchronization and Overlapped Input and Output.

A single thread can block its own execution while waiting for any one or all of several events to occur. This is more efficient than using multiple threads, each waiting for a single event, and more efficient than using a single thread that consumes processor time by continually checking for events to occur. For more information, see Wait Functions.

Multitasking Considerations

9/16/2022 • 2 minutes to read • Edit Online

The recommended guideline is to use as few threads as possible, thereby minimizing the use of system resources. This improves performance. Multitasking has resource requirements and potential conflicts to be considered when designing your application. The resource requirements are as follows:

- The system consumes memory for the context information required by both processes and threads. Therefore, the number of processes and threads that can be created is limited by available memory.
- Keeping track of a large number of threads consumes significant processor time. If there are too many threads, most of them will not be able to make significant progress. If most of the current threads are in one process, threads in other processes are scheduled less frequently.

Providing shared access to resources can create conflicts. To avoid them, you must synchronize access to shared resources. This is true for system resources (such as communications ports), resources shared by multiple processes (such as file handles), or the resources of a single process (such as global variables) accessed by multiple threads. Failure to synchronize access properly (in the same or in different processes) can lead to problems such as deadlock and race conditions. The synchronization objects and functions you can use to coordinate resource sharing among multiple threads. For more information about synchronization, see Synchronizing Execution of Multiple Threads. Reducing the number of threads makes it easier and more effective to synchronize resources.

A good design for a multithreaded application is the pipeline server. In this design, you create one thread per processor and build queues of requests for which the application maintains the context information. A thread would process all requests in a queue before processing requests in the next queue.

Scheduling

9/16/2022 • 2 minutes to read • Edit Online

The system scheduler controls multitasking by determining which of the competing threads receives the next processor time slice. The scheduler determines which thread runs next using scheduling priorities.

For more information, see the following topics:

- Scheduling Priorities
- Context Switches
- Priority Boosts
- Priority Inversion
- Multiple Processors
- NUMA Support
- Thread Ordering Service
- Multimedia Class Scheduler Service
- Processor Groups

Scheduling Priorities

9/16/2022 • 4 minutes to read • Edit Online

Threads are scheduled to run based on their *scheduling priority*. Each thread is assigned a scheduling priority. The priority levels range from zero (lowest priority) to 31 (highest priority). Only the zero-page thread can have a priority of zero. (The zero-page thread is a system thread responsible for zeroing any free pages when there are no other threads that need to run.)

The system treats all threads with the same priority as equal. The system assigns time slices in a round-robin fashion to all threads with the highest priority. If none of these threads are ready to run, the system assigns time slices in a round-robin fashion to all threads with the next highest priority. If a higher-priority thread becomes available to run, the system ceases to execute the lower-priority thread (without allowing it to finish using its time slice) and assigns a full time slice to the higher-priority thread. For more information, see Context Switches.

The priority of each thread is determined by the following criteria:

- The priority class of its process
- The priority level of the thread within the priority class of its process

The priority class and priority level are combined to form the *base priority* of a thread. For information on the dynamic priority of a thread, see Priority Boosts.

Priority Class

Each process belongs to one of the following priority classes:

IDLE_PRIORITY_CLASS
BELOW_NORMAL_PRIORITY_CLASS
NORMAL_PRIORITY_CLASS
ABOVE_NORMAL_PRIORITY_CLASS
HIGH_PRIORITY_CLASS
REALTIME_PRIORITY_CLASS

By default, the priority class of a process is NORMAL_PRIORITY_CLASS. Use the **CreateProcess** function to specify the priority class of a child process when you create it. If the calling process is IDLE_PRIORITY_CLASS or BELOW_NORMAL_PRIORITY_CLASS, the new process will inherit this class. Use the **GetPriorityClass** function to determine the current priority class of a process and the **SetPriorityClass** function to change the priority class of a process.

Processes that monitor the system, such as screen savers or applications that periodically update a display, should use IDLE_PRIORITY_CLASS. This prevents the threads of this process, which do not have high priority, from interfering with higher priority threads.

Use HIGH_PRIORITY_CLASS with care. If a thread runs at the highest priority level for extended periods, other threads in the system will not get processor time. If several threads are set at high priority at the same time, the threads lose their effectiveness. The high-priority class should be reserved for threads that must respond to time-critical events. If your application performs one task that requires the high-priority class while the rest of its tasks are normal priority, use **SetPriorityClass** to raise the priority class of the application temporarily; then reduce it after the time-critical task has been completed. Another strategy is to create a high-priority process that has all of its threads blocked most of the time, awakening threads only when critical tasks are needed. The important point is that a high-priority thread should execute for a brief time, and only when it has time-critical work to perform.

You should almost never use REALTIME_PRIORITY_CLASS, because this interrupts system threads that manage mouse input, keyboard input, and background disk flushing. This class can be appropriate for applications that "talk" directly to hardware or that perform brief tasks that should have limited interruptions.

Priority Level

The following are priority levels within each priority class:

THREAD_PRIORITY_IDLE
THREAD_PRIORITY_LOWEST
THREAD_PRIORITY_BELOW_NORMAL
THREAD_PRIORITY_NORMAL
THREAD_PRIORITY_ABOVE_NORMAL
THREAD_PRIORITY_HIGHEST
THREAD_PRIORITY_TIME_CRITICAL

All threads are created using THREAD_PRIORITY_NORMAL. This means that the thread priority is the same as the process priority class. After you create a thread, use the **SetThreadPriority** function to adjust its priority relative to other threads in the process.

A typical strategy is to use THREAD_PRIORITY_ABOVE_NORMAL or THREAD_PRIORITY_HIGHEST for the process's input thread, to ensure that the application is responsive to the user. Background threads, particularly those that are processor intensive, can be set to THREAD_PRIORITY_BELOW_NORMAL or THREAD_PRIORITY_LOWEST, to ensure that they can be preempted when necessary. However, if you have a thread waiting for another thread with a lower priority to complete some task, be sure to block the execution of the waiting high-priority thread. To do this, use a wait function, critical section, or the **Sleep** function, **SleepEx**, or **SwitchToThread** function. This is preferable to having the thread execute a loop. Otherwise, the process may become deadlocked, because the thread with lower priority is never scheduled.

To determine the current priority level of a thread, use the **GetThreadPriority** function.

Base Priority

The process priority class and thread priority level are combined to form the base priority of each thread.

The following table shows the base priority for combinations of process priority class and thread priority value.

PROCESS PRIORITY CLASS	THREAD PRIORITY LEVEL	BASE PRIORITY
IDLE_PRIORITY_CLASS	THREAD_PRIORITY_IDLE	1
	THREAD_PRIORITY_LOWEST	2
	THREAD_PRIORITY_BELOW_NORMAL	3
	THREAD_PRIORITY_NORMAL	4
	THREAD_PRIORITY_ABOVE_NORMAL	5
	THREAD_PRIORITY_HIGHEST	6

	THREAD_PRIORITY_TIME_CRITICAL	15
BELOW_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_IDLE	1
	THREAD_PRIORITY_LOWEST	4
	THREAD_PRIORITY_BELOW_NORMAL	5
	THREAD_PRIORITY_NORMAL	6
	THREAD_PRIORITY_ABOVE_NORMAL	7
	THREAD_PRIORITY_HIGHEST	8
	THREAD_PRIORITY_TIME_CRITICAL	15
NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_IDLE	1
	THREAD_PRIORITY_LOWEST	6
	THREAD_PRIORITY_BELOW_NORMAL	7
	THREAD_PRIORITY_NORMAL	8
	THREAD_PRIORITY_ABOVE_NORMAL	9
	THREAD_PRIORITY_HIGHEST	10
	THREAD_PRIORITY_TIME_CRITICAL	15
ABOVE_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_IDLE	1
	THREAD_PRIORITY_LOWEST	8
	THREAD_PRIORITY_BELOW_NORMAL	9
	THREAD_PRIORITY_NORMAL	10
	THREAD_PRIORITY_ABOVE_NORMAL	11

	THREAD_PRIORITY_HIGHEST	12
	THREAD_PRIORITY_TIME_CRITICAL	15
HIGH_PRIORITY_CLASS	THREAD_PRIORITY_IDLE	1
	THREAD_PRIORITY_LOWEST	11
	THREAD_PRIORITY_BELOW_NORMAL	12
	THREAD_PRIORITY_NORMAL	13
	THREAD_PRIORITY_ABOVE_NORMAL	14
	THREAD_PRIORITY_HIGHEST	15
	THREAD_PRIORITY_TIME_CRITICAL	15
REALTIME_PRIORITY_CLASS	THREAD_PRIORITY_IDLE	16
	THREAD_PRIORITY_LOWEST	22
	THREAD_PRIORITY_BELOW_NORMAL	23
	THREAD_PRIORITY_NORMAL	24
	THREAD_PRIORITY_ABOVE_NORMAL	25
	THREAD_PRIORITY_HIGHEST	26
	THREAD_PRIORITY_TIME_CRITICAL	31

Context Switches

9/16/2022 • 2 minutes to read • Edit Online

The scheduler maintains a queue of executable threads for each priority level. These are known as *ready threads*. When a processor becomes available, the system performs a *context switch*. The steps in a context switch are:

1. Save the context of the thread that just finished executing.
2. Place the thread that just finished executing at the end of the queue for its priority.
3. Find the highest priority queue that contains ready threads.
4. Remove the thread at the head of the queue, load its context, and execute it.

The following classes of threads are not ready threads.

- Threads created with the CREATE_SUSPENDED flag
- Threads halted during execution with the **SuspendThread** or **SwitchToThread** function
- Threads waiting for a synchronization object or input.

Until threads that are suspended or blocked become ready to run, the scheduler does not allocate any processor time to them, regardless of their priority.

The most common reasons for a context switch are:

- The time slice has elapsed.
- A thread with a higher priority has become ready to run.
- A running thread needs to wait.

When a running thread needs to wait, it relinquishes the remainder of its time slice.

Priority Boosts

9/16/2022 • 2 minutes to read • Edit Online

Each thread has a *dynamic priority*. This is the priority the scheduler uses to determine which thread to execute. Initially, a thread's dynamic priority is the same as its base priority. The system can boost and lower the dynamic priority, to ensure that it is responsive and that no threads are starved for processor time. The system does not boost the priority of threads with a base priority level between 16 and 31. Only threads with a base priority between 0 and 15 receive dynamic priority boosts.

The system boosts the dynamic priority of a thread to enhance its responsiveness as follows.

- When a process that uses NORMAL_PRIORITY_CLASS is brought to the foreground, the scheduler boosts the priority class of the process associated with the foreground window, so that it is greater than or equal to the priority class of any background processes. The priority class returns to its original setting when the process is no longer in the foreground.
- When a window receives input, such as timer messages, mouse messages, or keyboard input, the scheduler boosts the priority of the thread that owns the window.
- When the wait conditions for a blocked thread are satisfied, the scheduler boosts the priority of the thread. For example, when a wait operation associated with disk or keyboard I/O finishes, the thread receives a priority boost.

  You can disable the priority-boosting feature by calling the **SetProcessPriorityBoost** or **SetThreadPriorityBoost** function. To determine whether this feature has been disabled, call the **GetProcessPriorityBoost** or **GetThreadPriorityBoost** function.

After raising a thread's dynamic priority, the scheduler reduces that priority by one level each time the thread completes a time slice, until the thread drops back to its base priority. A thread's dynamic priority is never less than its base priority.

Priority Inversion

9/16/2022 • 2 minutes to read • Edit Online

Priority inversion occurs when two or more threads with different priorities are in contention to be scheduled. Consider a simple case with three threads: thread 1, thread 2, and thread 3. Thread 1 is high priority and becomes ready to be scheduled. Thread 2, a low-priority thread, is executing code in a critical section. Thread 1, the high-priority thread, begins waiting for a shared resource from thread 2. Thread 3 has medium priority. Thread 3 receives all the processor time, because the high-priority thread (thread 1) is waiting for shared resources from the low-priority thread (thread 2). Thread 2 will not leave the critical section, because it does not have the highest priority and will not be scheduled.

The scheduler solves this problem by randomly boosting the priority of the ready threads (in this case, the low priority lock-holders). The low priority threads run long enough to exit the critical section, and the high-priority thread can enter the critical section. If the low-priority thread does not get enough CPU time to exit the critical section the first time, it will get another chance during the next round of scheduling.

Multiple Processors

9/16/2022 • 2 minutes to read • Edit Online

Computers with multiple processors are typically designed for one of two architectures: non-uniform memory access (NUMA) or symmetric multiprocessing (SMP).

In a NUMA computer, each processor is closer to some parts of memory than others, making memory access faster for some parts of memory than other parts. Under the NUMA model, the system attempts to schedule threads on processors that are close to the memory being used. For more information about NUMA, see NUMA Support.

In an SMP computer, two or more identical processors or cores connect to a single shared main memory. Under the SMP model, any thread can be assigned to any processor. Therefore, scheduling threads on an SMP computer is similar to scheduling threads on a computer with a single processor. However, the scheduler has a pool of processors, so that it can schedule threads to run concurrently. Scheduling is still determined by thread priority, but it can be influenced by setting thread affinity and thread ideal processor, as discussed in this topic.

Thread Affinity

Thread affinity forces a thread to run on a specific subset of processors. Setting thread affinity should generally be avoided, because it can interfere with the scheduler's ability to schedule threads effectively across processors. This can decrease the performance gains produced by parallel processing. An appropriate use of thread affinity is testing each processor.

The system represents affinity with a bitmask called a processor affinity mask. The affinity mask is the size of the maximum number of processors in the system, with bits set to identify a subset of processors. Initially, the system determines the subset of processors in the mask.

You can obtain the current thread affinity for all threads of the process by calling the **GetProcessAffinityMask** function. Use the **SetProcessAffinityMask** function to specify thread affinity for all threads of the process. To set the thread affinity for a single thread, use the **SetThreadAffinityMask** function. The thread affinity must be a subset of the process affinity.

On systems with more than 64 processors, the affinity mask initially represents processors in a single processor group. However, thread affinity can be set to a processor in a different group, which alters the affinity mask for the process. For more information, see Processor Groups.

Thread Ideal Processor

When you specify a *thread ideal processor*, the scheduler runs the thread on the specified processor when possible. Use the **SetThreadIdealProcessor** function to specify a preferred processor for a thread. This does not guarantee that the ideal processor will be chosen but provides a useful hint to the scheduler. On systems with more than 64 processors, you can use the **SetThreadIdealProcessorEx** function to specify a preferred processor in a specific processor group.

Related topics

NUMA Support

Processor Groups

NUMA Architecture

9/16/2022 • 9 minutes to read • Edit Online

The traditional model for multiprocessor architecture is symmetric multiprocessor (SMP). In this model, each processor has equal access to memory and I/O. As more processors are added, the processor bus becomes a limitation for system performance.

System designers use non-uniform memory access (NUMA) to increase processor speed without increasing the load on the processor bus. The architecture is non-uniform because each processor is close to some parts of memory and farther from other parts of memory. The processor quickly gains access to the memory it is close to, while it can take longer to gain access to memory that is farther away.

In a NUMA system, CPUs are arranged in smaller systems called *nodes*. Each node has its own processors and memory, and is connected to the larger system through a cache-coherent interconnect bus.

The system attempts to improve performance by scheduling threads on processors that are in the same node as the memory being used. It attempts to satisfy memory-allocation requests from within the node, but will allocate memory from other nodes if necessary. It also provides an API to make the topology of the system available to applications. You can improve the performance of your applications by using the NUMA functions to optimize scheduling and memory usage.

First of all, you will need to determine the layout of nodes in the system. To retrieve the highest numbered node in the system, use the **GetNumaHighestNodeNumber** function. Note that this number is not guaranteed to equal the total number of nodes in the system. Also, nodes with sequential numbers are not guaranteed to be close together. To retrieve the list of processors on the system, use the **GetProcessAffinityMask** function. You can determine the node for each processor in the list by using the **GetNumaProcessorNode** function. Alternatively, to retrieve a list of all processors in a node, use the **GetNumaNodeProcessorMask** function.

After you have determined which processors belong to which nodes, you can optimize your application's performance. To ensure that all threads for your process run on the same node, use the **SetProcessAffinityMask** function with a process affinity mask that specifies processors in the same node. This increases the efficiency of applications whose threads need to access the same memory. Alternatively, to limit the number of threads on each node, use the **SetThreadAffinityMask** function.

Memory-intensive applications will need to optimize their memory usage. To retrieve the amount of free memory available to a node, use the **GetNumaAvailableMemoryNode** function. The **VirtualAllocExNuma** function enables the application to specify a preferred node for the memory allocation. **VirtualAllocExNuma** does not allocate any physical pages, so it will succeed whether or not the pages are available on that node or elsewhere in the system. The physical pages are allocated on demand. If the preferred node runs out of pages, the memory manager will use pages from other nodes. If the memory is paged out, the same process is used when it is brought back in.

NUMA Support on Systems With More Than 64 Logical Processors

On systems with more than 64 logical processors, nodes are assigned to processor groups according to the capacity of the nodes. The capacity of a node is the number of processors that are present when the system starts together with any additional logical processors that can be added while the system is running.

Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP: Processor groups are not supported.

Each node must be fully contained within a group. If the capacities of the nodes are relatively small, the system assigns more than one node to the same group, choosing nodes that are physically close to one another for

better performance. If a node's capacity exceeds the maximum number of processors in a group, the system splits the node into multiple smaller nodes, each small enough to fit in a group.

An ideal NUMA node for a new process can be requested using the PROC_THREAD_ATTRIBUTE_PREFERRED_NODE extended attribute when the process is created. Like a thread ideal processor, the ideal node is a hint to the scheduler, which assigns the new process to the group that contains the requested node if possible.

The extended NUMA functions GetNumaAvailableMemoryNodeEx, GetNumaNodeProcessorMaskEx, GetNumaProcessorNodeEx, and GetNumaProximityNodeEx differ from their unextended counterparts in that the node number is a **USHORT** value rather than a **UCHAR**, to accommodate the potentially greater number of nodes on a system with more than 64 logical processors. Also, the processor specified with or retrieved by the extended functions includes the processor group; the processor specified with or retrieved by the unextended functions is group-relative. For details, see the individual function reference topics.

A group-aware application can assign all of its threads to a particular node in a similar fashion to that described earlier in this topic, using the corresponding extended NUMA functions. The application uses GetLogicalProcessorInformationEx to get the list of all processors on the system. Note that the application cannot set the process affinity mask unless the process is assigned to a single group and the intended node is located in that group. Usually the application must call SetThreadGroupAffinity to limit its threads to the intended node.

Behavior starting with Windows 10 Build 20348

> **NOTE**
> Starting with Windows 10 Build 20348, the behavior of this and other NUMA functions has been modified to better support systems with nodes containing more that 64 processors.

Creating "fake" nodes to accommodate a 1:1 mapping between groups and nodes has resulted in confusing behaviors where unexpected numbers of NUMA nodes are reported and so, starting with Windows 10 Build 20348, the OS has changed to allow multiple groups to be associated with a node, and so now the true NUMA topology of the system can be reported.

As part of these changes to the OS, a number of NUMA APIs have changed to support reporting the multiple groups which can now be associated with a single NUMA node. Updated and new APIs are labeled in the table in the **NUMA API** section below.

Because the removal of node splitting can potentially impact existing applications, a registry value is available to allow opting back into the legacy node splitting behavior. Node splitting can be re-enabled by creating a **REG_DWORD** value named "SplitLargeNodes" with value 1 underneath HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\NUMA. Changes to this setting require a reboot to take effect.

```
reg add "HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\NUMA" /v SplitLargeNodes /t REG_DWORD /d 1
```

> **NOTE**
> Applications that are updated to use the new API functionality that reports the true NUMA topology will continue to work properly on systems where large node splitting has been reenabled with this registry key.

The following example first demonstrates potential issues with builds tables mapping processors to NUMA nodes using the legacy affinity APIs, which no longer provide a full cover of all processors in the system this can result in an incomplete table. The implications of such incompleteness depend on the contents of the table. If the

table simply stores the corresponding node number then this is likely only a performance issue with uncovered processors being left as part of node 0. However, if the table contains pointers to a per-node context structure this can result in NULL dereferences at runtime.

Next, the code example illustrates two workarounds for the issue. The first is to migrate to the multi-group node affinity APIs (user-mode and kernel-mode). The second is to use **KeQueryLogicalProcessorRelationship** to directly query the NUMA node associated with a given processor number.

```
//
// Problematic implementation using KeQueryNodeActiveAffinity.
//

USHORT CurrentNode;
USHORT HighestNodeNumber;
GROUP_AFFINITY NodeAffinity;
ULONG ProcessorIndex;
PROCESSOR_NUMBER ProcessorNumber;

HighestNodeNumber = KeQueryHighestNodeNumber();
for (CurrentNode = 0; CurrentNode <= HighestNodeNumber; CurrentNode += 1) {

    KeQueryNodeActiveAffinity(CurrentNode, &NodeAffinity, NULL);
    while (NodeAffinity.Mask != 0) {

        ProcessorNumber.Group = NodeAffinity.Group;
        BitScanForward(&ProcessorNumber.Number, NodeAffinity.Mask);

        ProcessorIndex = KeGetProcessorIndexFromNumber(&ProcessorNumber);

        ProcessorNodeContexts[ProcessorIndex] = NodeContexts[CurrentNode;]

        NodeAffinity.Mask &= ~((KAFFINITY)1 << ProcessorNumber.Number);
    }
}

//
// Resolution using KeQueryNodeActiveAffinity2.
//

USHORT CurrentIndex;
USHORT CurrentNode;
USHORT CurrentNodeAffinityCount;
USHORT HighestNodeNumber;
ULONG MaximumGroupCount;
PGROUP_AFFINITY NodeAffinityMasks;
ULONG ProcessorIndex;
PROCESSOR_NUMBER ProcessorNumber;
NTSTATUS Status;

MaximumGroupCount = KeQueryMaximumGroupCount();
NodeAffinityMasks = ExAllocatePool2(POOL_FLAG_PAGED,
                                    sizeof(GROUP_AFFINITY) * MaximumGroupCount,
                                    'tseT');

if (NodeAffinityMasks == NULL) {
    return STATUS_NO_MEMORY;
}

HighestNodeNumber = KeQueryHighestNodeNumber();
for (CurrentNode = 0; CurrentNode <= HighestNodeNumber; CurrentNode += 1) {

    Status = KeQueryNodeActiveAffinity2(CurrentNode,
                                        NodeAffinityMasks,
                                        MaximumGroupCount,
                                        &CurrentNodeAffinityCount);
    NT_ASSERT(NT_SUCCESS(Status));
```

```
    NT_ASSERT(NT_SUCCESS(Status));

    for (CurrentIndex = 0; CurrentIndex < CurrentNodeAffinityCount; CurrentIndex += 1) {

        CurrentAffinity = &NodeAffinityMasks[CurrentIndex];

        while (CurrentAffinity->Mask != 0) {

            ProcessorNumber.Group = CurrentAffinity.Group;
            BitScanForward(&ProcessorNumber.Number, CurrentAffinity->Mask);

            ProcessorIndex = KeGetProcessorIndexFromNumber(&ProcessorNumber);

            ProcessorNodeContexts[ProcessorIndex] = NodeContexts[CurrentNode];

            CurrentAffinity->Mask &= ~((KAFFINITY)1 << ProcessorNumber.Number);
        }
    }
}

//
// Resolution using KeQueryLogicalProcessorRelationship.
//

ULONG ProcessorCount;
ULONG ProcessorIndex;
SYSTEM_LOGICAL_PROCESSOR_INFORMATION_EX ProcessorInformation;
ULONG ProcessorInformationSize;
PROCESSOR_NUMBER ProcessorNumber;
NTSTATUS Status;

ProcessorCount = KeQueryActiveProcessorCountEx(ALL_PROCESSOR_GROUPS);

for (ProcessorIndex = 0; ProcessorIndex < ProcessorCount; ProcessorIndex += 1) {

    Status = KeGetProcessorNumberFromIndex(ProcessorIndex, &ProcessorNumber);
    NT_ASSERT(NT_SUCCESS(Status));

    ProcessorInformationSize = sizeof(ProcessorInformation);
    Status = KeQueryLogicalProcessorRelationship(&ProcessorNumber,
                                                 RelationNumaNode,
                                                 &ProcessorInformation,
                                                 &ProcesorInformationSize);
    NT_ASSERT(NT_SUCCESS(Status));

    NodeNumber = ProcessorInformation.NumaNode.NodeNumber;

    ProcessorNodeContexts[ProcessorIndex] = NodeContexts[NodeNumber];
}
```

NUMA API

The following table describes the NUMA API.

FUNCTION	DESCRIPTION
AllocateUserPhysicalPagesNuma	Allocates physical memory pages to be mapped and unmapped within any Address Windowing Extensions (AWE) region of a specified process and specifies the NUMA node for the physical memory.
CreateFileMappingNuma	Creates or opens a named or unnamed file mapping object for a specified file, and specifies the NUMA node for the physical memory.

FUNCTION	DESCRIPTION
GetLogicalProcessorInformation	Updated in Windows 10 Build 20348. Retrieves information about logical processors and related hardware.
GetLogicalProcessorInformationEx	Updated in Windows 10 Build 20348. Retrieves information about the relationships of logical processors and related hardware.
GetNumaAvailableMemoryNode	Retrieves the amount of memory available in the specified node.
GetNumaAvailableMemoryNodeEx	Retrieves the amount of memory available in a node specified as a **USHORT** value.
GetNumaHighestNodeNumber	Retrieves the node that currently has the highest number.
GetNumaNodeProcessorMask	Updated in Windows 10 Build 20348. Retrieves the processor mask for the specified node.
GetNumaNodeProcessorMask2	New in Windows 10 Build 20348. Retrieves the multi-group processor mask of the specified node.
GetNumaNodeProcessorMaskEx	Updated in Windows 10 Build 20348. Retrieves the processor mask for a node specified as a **USHORT** value.
GetNumaProcessorNode	Retrieves the node number for the specified processor.
GetNumaProcessorNodeEx	Retrieves the node number as a **USHORT** value for the specified processor.
GetNumaProximityNode	Retrieves the node number for the specified proximity identifier.
GetNumaProximityNodeEx	Retrieves the node number as a **USHORT** value for the specified proximity identifier.
GetProcessDefaultCpuSetMasks	New in Windows 10 Build 20348. Retrieves the list of CPU Sets in the process default set that was set by SetProcessDefaultCpuSetMasks or SetProcessDefaultCpuSets.
GetThreadSelectedCpuSetMasks	New in Windows 10 Build 20348. Sets the selected CPU Sets assignment for the specified thread. This assignment overrides the process default assignment, if one is set.
MapViewOfFileExNuma	Maps a view of a file mapping into the address space of a calling process, and specifies the NUMA node for the physical memory.
SetProcessDefaultCpuSetMasks	New in Windows 10 Build 20348. Sets the default CPU Sets assignment for threads in the specified process.
SetThreadSelectedCpuSetMasks	New in Windows 10 Build 20348. Sets the selected CPU Sets assignment for the specified thread. This assignment overrides the process default assignment, if one is set.

FUNCTION	DESCRIPTION
VirtualAllocExNuma	Reserves or commits a region of memory within the virtual address space of the specified process, and specifies the NUMA node for the physical memory.

The **QueryWorkingSetEx** function can be used to retrieve the NUMA node on which a page is allocated. For an example, see Allocating Memory from a NUMA Node.

Related topics

Allocating Memory from a NUMA Node

Multiple Processors

Processor Groups

Thread Ordering Service

9/16/2022 • 2 minutes to read • Edit Online

The *thread ordering service* controls the execution of one or more client threads. It ensures that each client thread runs once during the specified period and in relative order.

Windows Server 2003 and Windows XP: The thread ordering service is available starting with Windows Vista and Windows Server 2008.

The thread ordering service is off by default and must be started by the user. While the thread ordering service is running, it is activated every 5 seconds to check whether there is a new request, even if the system is idle. This prevents the system from sleeping for longer than 5 seconds, causing the system to consume more power. If energy efficiency is critical to the application, it is better not to use the thread ordering service and instead allow the system scheduler to manage execution of threads.

Each client thread belongs to a *thread ordering group*. The *parent thread* creates one or more thread ordering groups by calling the **AvRtCreateThreadOrderingGroup** function. The parent thread uses this function to specify the period for the thread ordering group and a time-out interval.

Additional client threads call the **AvRtJoinThreadOrderingGroup** function to join an existing thread ordering group. These threads indicate whether they are to be a predecessor or successor to the parent thread in the execution order. Each client thread is expected to complete a certain amount of processing each period. All threads within the group should complete their execution within the period plus the time-out interval.

The threads of a thread ordering group enclose their processing code within a loop that is controlled by the **AvRtWaitOnThreadOrderingGroup** function. First, the predecessor threads are executed one at a time in the order that they joined the group, while the parent and successor threads are blocked on their calls to **AvRtWaitOnThreadOrderingGroup**. When each predecessor thread is finished with its processing, control of execution returns to the top of its processing loop and the thread calls **AvRtWaitOnThreadOrderingGroup** again to block until its next turn. After all predecessor threads have called this function, the thread ordering service can schedule the parent thread. Finally, when the parent thread finishes its processing and calls **AvRtWaitOnThreadOrderingGroup** again, the thread ordering service can schedule the successor threads one at a time in the order that they joined the group. If all threads complete their execution before a period ends, all threads wait until the remainder of the period elapses before any are executed again.

When the client need no longer run as part of the thread ordering group, it calls the **AvRtLeaveThreadOrderingGroup** function to remove itself from the group. Note that the parent thread should not remove itself from a thread ordering group. If a thread does not complete its execution before the period plus the time-out interval elapses, then it is deleted from the group.

The parent thread calls the **AvRtDeleteThreadOrderingGroup** function to delete the thread ordering group. The thread ordering group is also destroyed if the parent thread does not complete its execution before the period plus the time-out interval elapses. When the thread ordering group is destroyed, any calls to **AvRtWaitOnThreadOrderingGroup** from threads of that group fail or time out.

Multimedia Class Scheduler Service

9/16/2022 • 3 minutes to read • Edit Online

The Multimedia Class Scheduler service (MMCSS) enables multimedia applications to ensure that their time-sensitive processing receives prioritized access to CPU resources. This service enables multimedia applications to utilize as much of the CPU as possible without denying CPU resources to lower-priority applications.

MMCSS uses information stored in the registry to identify supported tasks and determine the relative priority of threads performing these tasks. Each thread that is performing work related to a particular task calls the **AvSetMmMaxThreadCharacteristics** or **AvSetMmThreadCharacteristics** function to inform MMCSS that it is working on that task.

For an example of a program that uses MMCSS, see Exclusive-Mode Streams.

Windows Server 2003 and Windows XP: MMCSS is not available.

Registry Settings

The MMCSS settings are stored in the following registry key:

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Multimedia\SystemProfile

This key contains a **REG_DWORD** value named **SystemResponsiveness** that determines the percentage of CPU resources that should be guaranteed to low-priority tasks. For example, if this value is 20, then 20% of CPU resources are reserved for low-priority tasks. Note that values that are not evenly divisible by 10 are rounded up to the nearest multiple of 10. A value of 0 is also treated as 10.

The key also contains a subkey named **Tasks** that contains the list of tasks. By default, Windows supports the following tasks:

- **Audio**
- **Capture**
- **Distribution**
- **Games**
- **Playback**
- **Pro Audio**
- **Window Manager**

OEMs can add additional tasks as required.

Each task key contains the following set of values that represent characteristics to be applied to threads that are associated with the task.

VALUE	FORMAT	POSSIBLE VALUES
Affinity	**REG_DWORD**	A bitmask that indicates the processor affinity. Both 0x00 and 0xFFFFFFFF indicate that processor affinity is not used.

VALUE	FORMAT	POSSIBLE VALUES
Background Only	**REG_SZ**	Indicates whether this is a background task (no user interface). The threads of a background task do not change because of a change in window focus. This value can be set to True or False.
BackgroundPriority	**REG_DWORD**	The background priority. The range of values is 1-8.
Clock Rate	**REG_DWORD**	A hint used by MMCSS to determine the granularity of processor resource scheduling.**Windows Server 2008 and Windows Vista:** The maximum guaranteed clock rate the system uses if a thread joins this task, in 100-nanosecond intervals. Starting with Windows 7 and Windows Server 2008 R2, this guarantee was removed to reduce system power consumption.
GPU Priority	**REG_DWORD**	The GPU priority. The range of values is 0-31. This priority is not yet used.
Priority	**REG_DWORD**	The task priority. The range of values is 1 (low) to 8 (high).For tasks with a **Scheduling Category** of High, this value is always treated as 2.
Scheduling Category	**REG_SZ**	The scheduling category. This value can be set to High, Medium, or Low.
SFIO Priority	**REG_SZ**	The scheduled I/O priority. This value can be set to Idle, Low, Normal, or High. This value is not used.

NOTE

To conserve power, applications should not set the resolution of the system-wide timer to a small value unless absolutely necessary. For more information, see Performance in the Windows 7 Developers Guide.

Thread Priorities

The MMCSS boosts the priority of threads that are working on high-priority multimedia tasks.

MMCSS determines the priority of a thread using the following factors:

- The base priority of the task.
- The *Priority* parameter of the **AvSetMmThreadPriority** function.
- Whether the application is in the foreground.
- How much CPU time is being consumed by the threads in each category.

MMCSS sets the priority of client threads depending on their scheduling category.

CATEGORY	PRIORITY	DESCRIPTION
High	23-26	These threads run at a thread priority that is lower than only certain system-level tasks. This category is designed for Pro Audio tasks.
Medium	16-22	These threads are part of the application that is in the foreground.
Low	8-15	This category contains the remainder of the threads. They are guaranteed a minimum percentage of the CPU resources if required.
	1-7	These threads have used their quota of CPU resource. They can continue to run if no low-priority threads are ready to run.

Real-Time Work Queue API

9/16/2022 • 2 minutes to read • Edit Online

These API are used for managing real-time work queues.

RtwqEndUnregisterWorkQueueWithMMCSS function

9/16/2022 • 2 minutes to read • Edit Online

Completes an asynchronous request to unregister a work queue with a Multimedia Class Scheduler Service (MMCSS) task.

Syntax

```
HRESULT WINAPI RtwqEndUnregisterWorkQueueWithMMCSS(
    IMFAsyncResult  *pResult
);
```

Parameters

pResult

Pointer to the **IMFAsyncResult** interface. Pass in the same pointer that your callback object received in the **IRtwqAsyncCallback::Invoke** method.

Return value

If this function succeeds, it returns **S_OK**. Otherwise, it returns an **HRESULT** error code.

Requirements

REQUIREMENT	VALUE
Minimum supported client	Windows 8.1 [desktop apps only]
Minimum supported server	Windows Server 2012 R2 [desktop apps only]
Header	None
Library	Rtworkq.lib
DLL	RTWorkQ.dll

Processor Groups

9/16/2022 • 5 minutes to read • Edit Online

The 64-bit versions of Windows 7 and Windows Server 2008 R2 and later versions of Windows support more than 64 logical processors on a single computer. This functionality is not available on 32-bit versions of Windows.

Systems with more than one physical processor or systems with physical processors that have multiple cores provide the operating system with multiple logical processors. A *logical processor* is one logical computing engine from the perspective of the operating system, application or driver. A *core* is one processor unit, which can consist of one or more logical processors. A *physical processor* can consist of one or more cores. A physical processor is the same as a processor package, a socket, or a CPU.

Support for systems that have more than 64 logical processors is based on the concept of a *processor group*, which is a static set of up to 64 logical processors that is treated as a single scheduling entity. Processor groups are numbered starting with 0. Systems with fewer than 64 logical processors always have a single group, Group 0.

Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP: Processor groups are not supported.

When the system starts, the operating system creates processor groups and assigns logical processors to the groups. If the system is capable of hot-adding processors, the operating system allows space in groups for processors that might arrive while the system is running. The operating system minimizes the number of groups in a system. For example, a system with 128 logical processors would have two processor groups with 64 processors in each group, not four groups with 32 logical processors in each group.

For better performance, the operating system takes physical locality into account when assigning logical processors to groups. All of the logical processors in a core, and all of the cores in a physical processor, are assigned to the same group, if possible. Physical processors that are physically close to one another are assigned to the same group. A NUMA node is assigned to a single group unless the capacity of the node exceeds the maximum group size. For more information, see NUMA Support.

On systems with 64 or fewer processors, existing applications will operate correctly without modification. Applications that do not call any functions that use processor affinity masks or processor numbers will operate correctly on all systems, regardless of the number of processors. To operate correctly on systems with more than 64 logical processors, the following kinds of applications might require modification:

- Applications that manage, maintain, or display per-processor information for the entire system must be modified to support more than 64 logical processors. An example of such an application is Windows Task Manager, which displays the workload of each processor in the system.
- Applications for which performance is critical and that can scale efficiently beyond 64 logical processors must be modified to run on such systems. For example, database applications might benefit from modifications.
- If an application uses a DLL that has per-processor data structures, and the DLL has not been modified to support more than 64 logical processors, all threads in the application that call functions exported by the DLL must be assigned to the same group.

By default, an application is constrained to a single group, which should provide ample processing capability for the typical application. The operating system initially assigns each process to a single group in a round-robin manner across the groups in the system. A process begins its execution assigned to one group. The first thread of a process initially runs in the group to which the process is assigned. Each newly created thread is assigned to

the same group as the thread that created it.

An application that requires the use of multiple groups so that it can run on more than 64 processors must explicitly determine where to run its threads and is responsible for setting the threads' processor affinities to the desired groups. The INHERIT_PARENT_AFFINITY flag can be used to specify a parent process (which can be different than the current process) from which to generate the affinity for a new process. If the process is running in a single group, it can read and modify its affinity using **GetProcessAffinityMask** and **SetProcessAffinityMask** while remaining in the same group; if the process affinity is modified, the new affinity is applied to its threads.

A thread's affinity can be specified at creation using the **PROC_THREAD_ATTRIBUTE_GROUP_AFFINITY** extended attribute with the **CreateRemoteThreadEx** function. After the thread is created, its affinity can be changed by calling **SetThreadAffinityMask** or **SetThreadGroupAffinity**. If a thread is assigned to a different group than the process, the process's affinity is updated to include the thread's affinity and the process becomes a multi-group process. Further affinity changes must be made for individual threads; a multi-group process's affinity cannot be modified using **SetProcessAffinityMask**. The **GetProcessGroupAffinity** function retrieves the set of groups to which a process and its threads are assigned.

To specify affinity for all processes associated with a job object, use the **SetInformationJobObject** function with the **JobObjectGroupInformation** or **JobObjectGroupInformationEx** information class.

A logical processor is identified by its group number and its group-relative processor number. This is represented by a **PROCESSOR_NUMBER** structure. Numeric processor numbers used by legacy functions are group-relative.

For a discussion of operating system architecture changes to support more than 64 processors, see the white paper Supporting Systems That Have More Than 64 Processors.

For a list of new functions and structures that support processor groups, see What's New in Processes and Threads.

Behavior starting with Windows 11 and Windows Server 2022

> **NOTE**
> Starting with Windows 11 and Windows Server 2022, it is no longer the case that applications are constrained by default to a single processor group. Instead, processes and their threads have processor affinities that by default span all processors in the system, across multiple groups on machines with more than 64 processors.

In order for applications to automatically take advantage of all the processors in a machine with more than 64 processors, starting with Windows 11 and Windows Server 2022 the OS has changed to make processes and their threads span all processors in the system, across all processor groups, by default. This means that applications no longer need to explicitly set their threads' affinities in order to access multiple processor groups.

For compatibility reasons, the OS uses a new **Primary Group** concept for both processes and threads. Each process is assigned a primary group at creation, and by default all of its threads' primary group is the same. Each thread's ideal processor is in the thread's primary group, so threads will preferentially be scheduled to processors on their primary group, but they are able to be scheduled to processors on any other group. Affinity APIs that are not group-aware or operate on a single group implicitly use the primary group as the process/thread processor group; for more information on the new behaviors check the Remarks sections for the following:

- **GetProcessAffinityMask**
- **SetProcessAffinityMask**
- **SetThreadAffinityMask**
- **GetProcessGroupAffinity**

- GetThreadGroupAffinity
- SetThreadGroupAffinity
- SetThreadIdealProcessor
- SetThreadIdealProcessorEx

Applications can use CPU Sets to effectively manage a process' or thread's affinity over multiple processor groups.

Related topics

Multiple Processors

NUMA Support

Quality of Service

9/16/2022 • 2 minutes to read • Edit Online

The Quality of Service (QoS) associated with a thread is used to indicate the desired performance and power efficiency. Each thread is assigned to a QoS level. While scheduling priority remains the main metric by which the system determines which thread to schedule next, QoS can influence core selection and processor power management. On platforms with heterogeneous processors, the QoS of a thread may restrict scheduling to a subset of processors, or indicate a preference for a particular class of processor.

Developers may already be employing other facilities to control when to execute, such as when the user is not present, only on AC/charging, or depending on the battery level. QoS provides a facility to influence how to execute. This facility can help to improve CPU efficiency and thus extend battery life. In addition, this process can assist in reducing CPU power consumption while operating on AC power to reduce thermal output which can lead to high fan noise, or even thermal throttling.

Quality of Service levels

The system maintains multiple QoS levels, each with differentiated performance and power efficiency. Windows provides standard default settings for scheduling and processor power management for each QoS level. The precise tuning of each QoS level for processor power management and heterogeneous scheduling can be modified through Windows Provisioning. For more information on performance tuning and provisioning, see Processor power management options.

QOS LEVEL	DESCRIPTION	PERFORMANCE AND POWER	RELEASE
High	Windowed applications that are in the foreground and in focus, or audible, and explicitly tag processes with SetProcessInformation or threads with SetThreadInformation	Standard high performance.	1709
Medium	Windowed applications that may be visible to the end user but are not in focus.	Varies by platform, between High and Low.	1709
Low	Windowed applications that are not visible or audible to the end user.	On battery, selects most efficient CPU frequency and schedules to efficient core.	1709
Utility	Background services	On battery, selects most efficient CPU frequency and schedules to efficient cores.	Windows 11 22H2
Eco	Applications that explicitly tag processes with SetProcessInformation or threads with SetThreadInformation.	Always selects most efficient CPU frequency and schedules to efficient cores.	Windows 11

QOS LEVEL	DESCRIPTION	PERFORMANCE AND POWER	RELEASE
Media	Threads explicitly tagged by the Multimedia Class Scheduler Service to denote multimedia batch buffering.	CPU frequency reduced for efficient batch processing.	2004
Deadline	Threads explicitly tagged by Multimedia Class Scheduler Service to denote that audio threads require performance to meet deadlines.	High performance to meet media deadlines.	2004

Quality of Service classification

The following table shows the supported QoS classifications.

SOURCE	DESCRIPTION
Multimedia Foundation	The Multimedia Class Scheduler Service prioritizes CPU resources for multimedia scenarios. The service tags specific threads responsible for multimedia processing using the Media and Deadline QoS levels to provide power efficiency while meeting performance deadlines.
API	SetProcessInformation lets developers explicitly tag a process as HighQoS or EcoQoS by toggling the `PROCESS_POWER_THROTTLING_EXECUTION_SPEED` feature in **ProcessPowerThrottling**. SetThreadInformation lets developers explicitly tag a thread as HighQoS or EcoQoS by toggling the `THREAD_POWER_THROTTLING_EXECUTION_SPEED` feature in **ThreadPowerThrottling** .
Audible	Processes which are determined to be playing audio are HighQoS.
Visible	Processes which directly own a window (or are descendants of window owning processes) are assigned a QoS level according to their visibility and focus state: WINDOW STATE / QUALITY OF SERVICE In Focus: High Visible: Medium Minimized, or Fully Occluded: Low
Heuristic	Threads which are not classified by the above sources are automatically assigned a QoS level by the system. These heuristics include (but are not limited to) thread priority, where threads running with reduced thread priority can imply a lower QoS level.

Multiple Threads

9/16/2022 • 2 minutes to read • Edit Online

A *thread* is the entity within a process that can be scheduled for execution. All threads of a process share its virtual address space and system resources. Each process is started with a single thread, but can create additional threads from any of its threads.

For more information, see the following topics:

- Creating Threads
- Thread Stack Size
- Thread Handles and Identifiers
- Suspending Thread Execution
- Synchronizing Execution of Multiple Threads
- Multiple Threads and GDI Objects
- Thread Local Storage
- Creating Windows in Threads
- Terminating a Thread
- Thread Security and Access Rights

Thread Stack Size

9/16/2022 • 2 minutes to read • Edit Online

Each new thread or fiber receives its own stack space consisting of both reserved and initially committed memory. The reserved memory size represents the total stack allocation in virtual memory. As such, the reserved size is limited to the virtual address range. The initially committed pages do not utilize physical memory until they are referenced; however, they do remove pages from the system total commit limit, which is the size of the page file plus the size of the physical memory. The system commits additional pages from the reserved stack memory as they are needed, until either the stack reaches the reserved size minus one page (which is used as a guard page to prevent stack overflow) or the system is so low on memory that the operation fails.

It is best to choose as small a stack size as possible and commit the stack that is needed for the thread or fiber to run reliably. Every page that is reserved for the stack cannot be used for any other purpose.

A stack is freed when its thread exits. It is not freed if the thread is terminated by another thread.

The default size for the reserved and initially committed stack memory is specified in the executable file header. Thread or fiber creation fails if there is not enough memory to reserve or commit the number of bytes requested. The default stack reservation size used by the linker is 1 MB. To specify a different default stack reservation size for all threads and fibers, use the STACKSIZE statement in the module definition (.def) file. The operating system rounds up the specified size to the nearest multiple of the system's allocation granularity (typically 64 KB). To retrieve the allocation granularity of the current system, use the **GetSystemInfo** function.

To change the initially committed stack space, use the *dwStackSize* parameter of the **CreateThread**, **CreateRemoteThread**, or **CreateFiber** function. This value is rounded up to the nearest page. Generally, the reserve size is the default reserve size specified in the executable header. However, if the initially committed size specified by *dwStackSize* is larger than or equal to the default reserve size, the reserve size is this new commit size rounded up to the nearest multiple of 1 MB.

To change the reserved stack size, set the *dwCreationFlags* parameter of **CreateThread** or **CreateRemoteThread** to STACK_SIZE_PARAM_IS_A_RESERVATION and use the *dwStackSize* parameter. In this case, the initially committed size is the default size specified in the executable header. For fibers, use the *dwStackReserveSize* parameter of **CreateFiberEx**. The committed size is specified in the *dwStackCommitSize* parameter.

The **SetThreadStackGuarantee** function sets the minimum size of the stack associated with the calling thread or fiber that will be available during any stack overflow exceptions.

Thread Handles and Identifiers

9/16/2022 • 2 minutes to read • Edit Online

When a new thread is created by the **CreateThread** or **CreateRemoteThread** function, a handle to the thread is returned. By default, this handle has full access rights, and—subject to security access checking—can be used in any of the functions that accept a thread handle. This handle can be inherited by child processes, depending on the inheritance flag specified when it is created. The handle can be duplicated by **DuplicateHandle**, which enables you to create a thread handle with a subset of the access rights. The handle is valid until closed, even after the thread it represents has been terminated.

The **CreateThread** and **CreateRemoteThread** functions also return an identifier that uniquely identifies the thread throughout the system. A thread can use the **GetCurrentThreadId** function to get its own thread identifier. The identifiers are valid from the time the thread is created until the thread has been terminated. Note that no thread identifier will ever be 0.

If you have a thread identifier, you can get the thread handle by calling the **OpenThread** function. **OpenThread** enables you to specify the handle's access rights and whether it can be inherited.

A thread can use the **GetCurrentThread** function to retrieve a *pseudo handle* to its own thread object. This pseudo handle is valid only for the calling process; it cannot be inherited or duplicated for use by other processes. To get the real handle to the thread, given a pseudo handle, use the **DuplicateHandle** function.

To enumerate the threads of a process, use the **Thread32First** and **Thread32Next** functions.

Suspending Thread Execution

9/16/2022 • 2 minutes to read • Edit Online

A thread can suspend and resume the execution of another thread. While a thread is suspended, it is not scheduled for time on the processor.

If a thread is created in a suspended state (with the **CREATE_SUSPENDED** flag), it does not begin to execute until another thread calls the **ResumeThread** function with a handle to the suspended thread. This can be useful for initializing the thread's state before it begins to execute. Suspending a thread at creation can be useful for one-time synchronization, because this ensures that the suspended thread will execute the starting point of its code when you call **ResumeThread**.

The **SuspendThread** function is not intended to be used for thread synchronization because it does not control the point in the code at which the thread's execution is suspended. This function is primarily designed for use by debuggers.

A thread can temporarily yield its execution for a specified interval by calling the **Sleep** or **SleepEx** functions This is useful particularly in cases where the thread responds to user interaction, because it can delay execution long enough to allow users to observe the results of their actions. During the sleep interval, the thread is not scheduled for time on the processor.

The **SwitchToThread** function is similar to **Sleep** and **SleepEx**, except that you cannot specify the interval. **SwitchToThread** allows the thread to give up its time slice.

Synchronizing Execution of Multiple Threads

9/16/2022 • 2 minutes to read • Edit Online

To avoid race conditions and deadlocks, it is necessary to synchronize access by multiple threads to shared resources. Synchronization is also necessary to ensure that interdependent code is executed in the proper sequence.

There are a number of objects whose handles can be used to synchronize multiple threads. These objects include:

- Console input buffers
- Events
- Mutexes
- Processes
- Semaphores
- Threads
- Timers

The state of each of these objects is either signaled or not signaled. When you specify a handle to any of these objects in a call to one of the wait functions, the execution of the calling thread is blocked until the state of the specified object becomes signaled.

Some of these objects are useful in blocking a thread until some event occurs. For example, a console input buffer handle is signaled when there is unread input, such as a keystroke or mouse button click. Process and thread handles are signaled when the process or thread terminates. This allows a process, for example, to create a child process and then block its own execution until the new process has terminated.

Other objects are useful in protecting shared resources from simultaneous access. For example, multiple threads can each have a handle to a mutex object. Before accessing a shared resource, the threads must call one of the wait functions to wait for the state of the mutex to be signaled. When the mutex becomes signaled, only one waiting thread is released to access the resource. The state of the mutex is immediately reset to not signaled so any other waiting threads remain blocked. When the thread is finished with the resource, it must set the state of the mutex to signaled to allow other threads to access the resource.

For the threads of a single process, critical-section objects provide a more efficient means of synchronization than mutexes. A critical section is used like a mutex to enable one thread at a time to use the protected resource. A thread can use the **EnterCriticalSection** function to request ownership of a critical section. If it is already owned by another thread, the requesting thread is blocked. A thread can use the **TryEnterCriticalSection** function to request ownership of a critical section, without blocking upon failure to obtain the critical section. After it receives ownership, the thread is free to use the protected resource. The execution of the other threads of the process is not affected unless they attempt to enter the same critical section.

The **WaitForInputIdle** function makes a thread wait until a specified process is initialized and waiting for user input with no input pending. Calling **WaitForInputIdle** can be useful for synchronizing parent and child processes, because **CreateProcess** returns without waiting for the child process to complete its initialization.

For more information, see Synchronization.

Multiple Threads and GDI Objects

9/16/2022 • 2 minutes to read • Edit Online

To enhance performance, access to graphics device interface (GDI) objects (such as palettes, device contexts, regions, and the like) is not serialized. This creates a potential danger for processes that have multiple threads sharing these objects. For example, if one thread deletes a GDI object while another thread is using it, the results are unpredictable. This danger can be avoided simply by not sharing GDI objects. If sharing is unavoidable (or desirable), the application must provide its own mechanisms for synchronizing access. For more information about synchronizing access, see Synchronizing Execution of Multiple Threads.

Thread Local Storage

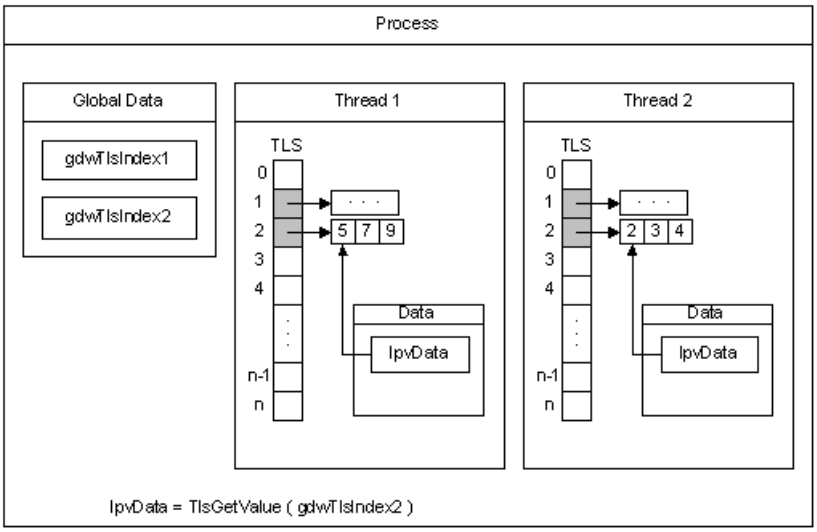
9/16/2022 • 2 minutes to read • Edit Online

All threads of a process share its virtual address space. The local variables of a function are unique to each thread that runs the function. However, the static and global variables are shared by all threads in the process. With *thread local storage* (TLS), you can provide unique data for each thread that the process can access using a global index. One thread allocates the index, which can be used by the other threads to retrieve the unique data associated with the index.

The constant TLS_MINIMUM_AVAILABLE defines the minimum number of TLS indexes available in each process. This minimum is guaranteed to be at least 64 for all systems. The maximum number of indexes per process is 1,088.

When the threads are created, the system allocates an array of **LPVOID** values for TLS, which are initialized to NULL. Before an index can be used, it must be allocated by one of the threads. Each thread stores its data for a TLS index in a *TLS slot* in the array. If the data associated with an index will fit in an **LPVOID** value, you can store the data directly in the TLS slot. However, if you are using a large number of indexes in this way, it is better to allocate separate storage, consolidate the data, and minimize the number of TLS slots in use.

The following diagram illustrates how TLS works. For a code example illustrating the use of thread local storage, see Using Thread Local Storage.



The process has two threads, Thread 1 and Thread 2. It allocates two indexes for use with TLS, gdwTlsIndex1 and gdwTlsIndex2. Each thread allocates two memory blocks (one for each index) in which to store the data, and stores the pointers to these memory blocks in the corresponding TLS slots. To access the data associated with an index, the thread retrieves the pointer to the memory block from the TLS slot and stores it in the lpvData local variable.

It is ideal to use TLS in a dynamic-link library (DLL). For an example, see Using Thread Local Storage in a Dynamic Link Library.

Related topics

Thread Local Storage (Visual C++)

Using Thread Local Storage

Using Thread Local Storage in a Dynamic Link Library

Creating Windows in Threads

9/16/2022 • 2 minutes to read • Edit Online

Any thread can create a window. The thread that creates the window owns the window and its associated message queue. Therefore, the thread must provide a message loop to process the messages in its message queue. In addition, you must use **MsgWaitForMultipleObjects** or **MsgWaitForMultipleObjectsEx** in that thread, rather than the other wait functions, so that it can process messages. Otherwise, the system can become deadlocked when the thread is sent a message while it is waiting.

The **AttachThreadInput** function can be used to allow a set of threads to share the same input state. By sharing input state, the threads share their concept of the active window. By doing this, one thread can always activate another thread's window. This function is also useful for sharing focus state, mouse capture state, keyboard state, and window Z-order state among windows created by different threads whose input state is shared.

For information about creating windows, see Windows Classes.

Terminating a Thread

9/16/2022 • 2 minutes to read • Edit Online

Terminating a thread has the following results:

- Any resources owned by the thread, such as windows and hooks, are freed.
- The thread exit code is set.
- The thread object is signaled.
- If the thread is the only active thread in the process, the process is terminated. For more information, see Terminating a Process.

The **GetExitCodeThread** function returns the termination status of a thread. While a thread is executing, its termination status is STILL_ACTIVE. When a thread terminates, its termination status changes from STILL_ACTIVE to the exit code of the thread.

When a thread terminates, the state of the thread object changes to signaled, releasing any other threads that had been waiting for the thread to terminate. For more about synchronization, see Synchronizing Execution of Multiple Threads.

When a thread terminates, its thread object is not freed until all open handles to the thread are closed.

How Threads are Terminated

A thread executes until one of the following events occurs:

- The thread calls the **ExitThread** function.
- Any thread of the process calls the **ExitProcess** function.
- The thread function returns.
- Any thread calls the **TerminateThread** function with a handle to the thread.
- Any thread calls the **TerminateProcess** function with a handle to the process.

The exit code for a thread is either the value specified in the call to **ExitThread**, **ExitProcess**, **TerminateThread**, or **TerminateProcess**, or the value returned by the thread function.

If a thread is terminated by **ExitThread**, the system calls the entry-point function of each attached DLL with a value indicating that the thread is detaching from the DLL (unless you call the **DisableThreadLibraryCalls** function). If a thread is terminated by **ExitProcess**, the DLL entry-point functions are invoked once, to indicate that the process is detaching. DLLs are not notified when a thread is terminated by **TerminateThread** or **TerminateProcess**. For more information about DLLs, see Dynamic-Link Libraries.

The **TerminateThread** and **TerminateProcess** functions should be used only in extreme circumstances, since they do not allow threads to clean up, do not notify attached DLLs, and do not free the initial stack. In addition, handles to objects owned by the thread are not closed until the process terminates. The following steps provide a better solution:

- Create an event object using the **CreateEvent** function.
- Create the threads.
- Each thread monitors the event state by calling the **WaitForSingleObject** function. Use a wait time-out interval of zero.
- Each thread terminates its own execution when the event is set to the signaled state (**WaitForSingleObject** returns WAIT_OBJECT_0).

Thread Security and Access Rights

9/16/2022 • 3 minutes to read • Edit Online

Microsoft Windows enables you to control access to thread objects. For more information about security, see Access-Control Model.

You can specify a security descriptor for a thread when you call the **CreateProcess**, **CreateProcessAsUser**, **CreateProcessWithLogonW**, **CreateThread**, or **CreateRemoteThread** function. If you specify **NULL**, the thread gets a default security descriptor. The ACLs in the default security descriptor for a thread come from the primary or impersonation token of the creator.

To retrieve a thread's security descriptor, call the **GetSecurityInfo** function. To change a thread's security descriptor, call the **SetSecurityInfo** function.

The handle returned by the **CreateThread** function has **THREAD_ALL_ACCESS** access to the thread object. When you call the **GetCurrentThread** function, the system returns a pseudohandle with the maximum access that the thread's security descriptor allows the caller.

The valid access rights for thread objects include the standard access rights and some thread-specific access rights. The following table lists the standard access rights used by all objects.

VALUE	MEANING
DELETE (0x00010000L)	Required to delete the object.
READ_CONTROL (0x00020000L)	Required to read information in the security descriptor for the object, not including the information in the SACL. To read or write the SACL, you must request the **ACCESS_SYSTEM_SECURITY** access right. For more information, see SACL Access Right.
SYNCHRONIZE (0x00100000L)	The right to use the object for synchronization. This enables a thread to wait until the object is in the signaled state.
WRITE_DAC (0x00040000L)	Required to modify the DACL in the security descriptor for the object.
WRITE_OWNER (0x00080000L)	Required to change the owner in the security descriptor for the object.

The following table lists the thread-specific access rights.

VALUE	MEANING
SYNCHRONIZE (0x00100000L)	Enables the use of the thread handle in any of the wait functions.

VALUE	MEANING
THREAD_ALL_ACCESS	All possible access rights for a thread object.**Windows Server 2003 and Windows XP:** The value of the **THREAD_ALL_ACCESS** flag increased on Windows Server 2008 and Windows Vista. If an application compiled for Windows Server 2008 and Windows Vista is run on Windows Server 2003 or Windows XP, the **THREAD_ALL_ACCESS** flag contains access bits that are not supported and the function specifying this flag fails with **ERROR_ACCESS_DENIED**. To avoid this problem, specify the minimum set of access rights required for the operation. If **THREAD_ALL_ACCESS** must be used, set **_WIN32_WINNT** to the minimum operating system targeted by your application (for example, `#define _WIN32_WINNT _WIN32_WINNT_WINXP`). For more information, see Using the Windows Headers.
THREAD_DIRECT_IMPERSONATION (0x0200)	Required for a server thread that impersonates a client.
THREAD_GET_CONTEXT (0x0008)	Required to read the context of a thread using **GetThreadContext**.
THREAD_IMPERSONATE (0x0100)	Required to use a thread's security information directly without calling it by using a communication mechanism that provides impersonation services.
THREAD_QUERY_INFORMATION (0x0040)	Required to read certain information from the thread object, such as the exit code (see **GetExitCodeThread**).
THREAD_QUERY_LIMITED_INFORMATION (0x0800)	Required to read certain information from the thread objects (see **GetProcessIdOfThread**). A handle that has the **THREAD_QUERY_INFORMATION** access right is automatically granted **THREAD_QUERY_LIMITED_INFORMATION**.**Windows Server 2003 and Windows XP:** This access right is not supported.
THREAD_SET_CONTEXT (0x0010)	Required to write the context of a thread using **SetThreadContext**.
THREAD_SET_INFORMATION (0x0020)	Required to set certain information in the thread object.
THREAD_SET_LIMITED_INFORMATION (0x0400)	Required to set certain information in the thread object. A handle that has the **THREAD_SET_INFORMATION** access right is automatically granted **THREAD_SET_LIMITED_INFORMATION**.**Windows Server 2003 and Windows XP:** This access right is not supported.
THREAD_SET_THREAD_TOKEN (0x0080)	Required to set the impersonation token for a thread using **SetThreadToken**.
THREAD_SUSPEND_RESUME (0x0002)	Required to suspend or resume a thread (see **SuspendThread** and **ResumeThread**).
THREAD_TERMINATE (0x0001)	Required to terminate a thread using **TerminateThread**.

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

Protected Processes

Protected processes enhance support for Digital Rights Management. The system restricts access to protected processes and the threads of protected processes.

Windows Server 2003 and Windows XP: Protected processes were added starting with Windows Vista.

The following specific access rights are not allowed from a process to the threads of a protected process:

THREAD_ALL_ACCESS
THREAD_DIRECT_IMPERSONATION
THREAD_GET_CONTEXT
THREAD_IMPERSONATE
THREAD_QUERY_INFORMATION
THREAD_SET_CONTEXT
THREAD_SET_INFORMATION
THREAD_SET_TOKEN
THREAD_TERMINATE

The **THREAD_QUERY_LIMITED_INFORMATION** right was introduced to provide access to a subset of the information available through **THREAD_QUERY_INFORMATION**.

Child Processes

9/16/2022 • 2 minutes to read • Edit Online

Each *process* provides the resources needed to execute a program. A *child process* is a process that is created by another process, called the *parent process*.

For more information, see the following topics:

- Creating Processes
- Setting Window Properties Using STARTUPINFO
- Process Handles and Identifiers
- Process Enumeration
- Obtaining Additional Process Information
- Inheritance
- Environment Variables
- Terminating a Process
- Process Working Set
- Process Security and Access Rights

Setting Window Properties Using STARTUPINFO

9/16/2022 • 2 minutes to read • Edit Online

A parent process can specify properties associated with the main window of its child process. The **CreateProcess** function takes a pointer to a **STARTUPINFO** structure as one of its parameters. Use the members of this structure to specify characteristics of the child process's main window. The **dwFlags** member contains a bitfield that determines which other members of the structure are used. This allows you to specify values for any subset of the window properties. The system uses default values for the properties you do not specify. The **dwFlags** member can also force a feedback cursor to be displayed during the initialization of the new process.

For GUI processes, the **STARTUPINFO** structure specifies the default values to be used the first time the new process calls the **CreateWindow** and **ShowWindow** functions to create and display an overlapped window. The following default values can be specified:

- The width and height, in pixels, of the window created by **CreateWindow**.
- The location, in screen coordinates of the window created by **CreateWindow**.
- The *nCmdShow* parameter of **ShowWindow**.

For console processes, use the **STARTUPINFO** structure to specify window properties only when creating a new console (either using **CreateProcess** with CREATE_NEW_CONSOLE or with the **AllocConsole** function). The **STARTUPINFO** structure can be used to specify the following console window properties:

- The size of the new console window, in character cells.
- The location of the new console window, in screen coordinates.
- The size, in character cells, of the new console's screen buffer.
- The text and background color attributes of the new console's screen buffer.
- The title of the new console's window.

Process Handles and Identifiers

9/16/2022 • 2 minutes to read • Edit Online

When a new process is created by the **CreateProcess** function, handles of the new process and its primary thread are returned. These handles are created with full access rights, and — subject to security access checking — can be used in any of the functions that accept thread or process handles. These handles can be inherited by child processes, depending on the inheritance flag specified when they are created. The handles are valid until closed, even after the process or thread they represent has been terminated.

The **CreateProcess** function also returns an identifier that uniquely identifies the process throughout the system. A process can use the **GetCurrentProcessId** function to get its own process identifier (also known as the process ID or PID). The identifier is valid from the time the process is created until the process has been terminated. A process can use the **Process32First** function to obtain the process identifier of its parent process.

If you have a process identifier, you can get the process handle by calling the **OpenProcess** function. **OpenProcess** enables you to specify the handle's access rights and whether it can be inherited.

A process can use the **GetCurrentProcess** function to retrieve a pseudo handle to its own process object. This pseudo handle is valid only for the calling process; it cannot be inherited or duplicated for use by other processes. To get the real handle to the process, call the **DuplicateHandle** function.

Process Enumeration

9/16/2022 • 2 minutes to read • Edit Online

All users have read access to the list of processes in the system and there are a number of different functions that enumerate the active processes. The function you should use will depend on factors such as desired platform support.

The following functions are used to enumerate processes.

FUNCTION	DESCRIPTION
EnumProcesses	Retrieves the process identifier for each process object in the system.
Process32First	Retrieves information about the first process encountered in a system snapshot.
Process32Next	Retrieves information about the next process recorded in a system snapshot.
WTSEnumerateProcesses	Retrieves information about the active processes on the specified terminal server.

The toolhelp functions and EnumProcesses enumerate all process. To list the processes that are running in a specific user account, use WTSEnumerateProcesses and filter on the user SID. You can filter on the session ID to hide processes running in other terminal server sessions.

You can also filter processes by user account, regardless of the enumeration function, by calling OpenProcess, OpenProcessToken, and GetTokenInformation with **TokenUser**. However, you cannot open a process that is protected by a security descriptor unless you have been granted access.

Obtaining Additional Process Information

9/16/2022 • 2 minutes to read • Edit Online

There are a variety of functions for obtaining information about processes. Some of these functions can be used only for the calling process, because they do not take a process handle as a parameter. You can use functions that take a process handle to obtain information about other processes.

- To obtain the command-line string for the current process, use the **GetCommandLine** function.
- To retrieve the **STARTUPINFO** structure specified when the current process was created, use the **GetStartupInfo** function.
- To obtain the version information from the executable header, use the **GetProcessVersion** function.
- To obtain the full path and file name for the executable file containing the process code, use the **GetModuleFileName** function.
- To obtain the count of handles to graphical user interface (GUI) objects in use, use the **GetGuiResources** function.
- To determine whether a process is being debugged, use the **IsDebuggerPresent** function.
- To retrieve accounting information for all I/O operations performed by the process, use the **GetProcessIoCounters** function.

Inheritance

9/16/2022 • 2 minutes to read • Edit Online

A child process can inherit several properties and resources from its parent process. You can also prevent a child process from inheriting properties from its parent process. The following can be inherited:

- Open handles returned by the **CreateFile** function. This includes handles to files, console input buffers, console screen buffers, named pipes, serial communication devices, and mailslots.
- Open handles to process, thread, mutex, event, semaphore, named-pipe, anonymous-pipe, and file-mapping objects. These are returned by the **CreateProcess**, **CreateThread**, **CreateMutex**, **CreateEvent**, **CreateSemaphore**, **CreateNamedPipe**, **CreatePipe**, and **CreateFileMapping** functions, respectively.
- Environment variables.
- The current directory.
- The console, unless the process is detached or a new console is created. A child console process can also inherit the parent's standard handles, as well as access to the input buffer and the active screen buffer.
- The error mode, as set by the **SetErrorMode** function.
- The processor affinity mask.
- The association with a job.

The child process does not inherit the following:

- Priority class.
- Handles returned by **LocalAlloc**, **GlobalAlloc**, **HeapCreate**, and **HeapAlloc**.
- Pseudo handles, as in the handles returned by the **GetCurrentProcess** or **GetCurrentThread** function. These handles are valid only for the calling process.
- DLL module handles returned by the **LoadLibrary** function.
- GDI or USER handles, such as **HBITMAP** or **HMENU**.

Inheriting Handles

A child process can inherit some of its parent's handles, but not inherit others. To cause a handle to be inherited, you must do two things:

- Specify that the handle is to be inherited when you create, open, or duplicate the handle. Creation functions typically use the **bInheritHandle** member of a **SECURITY_ATTRIBUTES** structure for this purpose. **DuplicateHandle** uses the *bInheritHandles* parameter.
- Specify that inheritable handles are to be inherited by setting the *bInheritHandles* parameter to TRUE when calling the **CreateProcess** function. Additionally, to inherit the standard input, standard output, and standard error handles, the **dwFlags** member of the **STARTUPINFO** structure must include STARTF_USESTDHANDLES.

To specify a list of the handles that should be inherited by a specific child process, call the **UpdateProcThreadAttribute** function with the *PROC_THREAD_ATTRIBUTE_HANDLE_LIST* flag.

An inherited handle refers to the same object in the child process as it does in the parent process. It also has the same value and access privileges. Therefore, when one process changes the state of the object, the change affects both processes. To use a handle, the child process must retrieve the handle value and "know" the object to which it refers. Usually, the parent process communicates this information to the child process through its command line, environment block, or some form of interprocess communication.

Use the **SetHandleInformation** function to control if an existing handle is inheritable or not.

Inheriting Environment Variables

A child process inherits the environment variables of its parent process by default. However, **CreateProcess** enables the parent process to specify a different block of environment variables. For more information, see Environment Variables.

Inheriting the Current Directory

The **GetCurrentDirectory** function retrieves the current directory of the calling process. A child process inherits the current directory of its parent process by default. However, **CreateProcess** enables the parent process to specify a different current directory for the child process. To change the current directory of the calling process, use the **SetCurrentDirectory** function.

Environment Variables

9/16/2022 • 2 minutes to read • Edit Online

Every process has an environment block that contains a set of environment variables and their values. There are two types of environment variables: user environment variables (set for each user) and system environment variables (set for everyone).

By default, a child process inherits the environment variables of its parent process. Programs started by the command processor inherit the command processor's environment variables. To specify a different environment for a child process, create a new environment block and pass a pointer to it as a parameter to the **CreateProcess** function.

The command processor provides the **set** command to display its environment block or to create new environment variables. You can also view or modify the environment variables by selecting **System** from the **Control Panel**, selecting **Advanced system settings**, and clicking **Environment Variables**.

Each environment block contains the environment variables in the following format:

Var1=*Value1*\0
Var2=*Value2*\0
Var3=*Value3*\0
...
VarN=*ValueN*\0\0

The name of an environment variable cannot include an equal sign (=).

The **GetEnvironmentStrings** function returns a pointer to the environment block of the calling process. This should be treated as a read-only block; do not modify it directly. Instead, use the **SetEnvironmentVariable** function to change an environment variable. When you are finished with the environment block obtained from **GetEnvironmentStrings**, call the **FreeEnvironmentStrings** function to free the block.

Calling **SetEnvironmentVariable** has no effect on the system environment variables. To programmatically add or modify system environment variables, add them to the **HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Session Manager\Environment** registry key, then broadcast a **WM_SETTINGCHANGE** message with *lParam* set to the string "Environment". This allows applications, such as the shell, to pick up your updates.

The maximum size of a user-defined environment variable is 32,767 characters. There is no technical limitation on the size of the environment block. However, there are practical limits depending on the mechanism used to access the block. For example, a batch file cannot set a variable that is longer than the maximum command line length.

Windows Server 2003 and Windows XP: The maximum size of the environment block for the process is 32,767 characters. Starting with Windows Vista and Windows Server 2008, there is no technical limitation on the size of the environment block.

The **GetEnvironmentVariable** function determines whether a specified variable is defined in the environment of the calling process, and, if so, what its value is.

To retrieve a copy of the environment block for a given user, use the **CreateEnvironmentBlock** function.

To expand environment-variable strings, use the **ExpandEnvironmentStrings** function.

Related topics

Changing Environment Variables

User Environment Variables

Terminating a Process

9/16/2022 • 3 minutes to read • Edit Online

Terminating a process has the following results:

- Any remaining threads in the process are marked for termination.
- Any resources allocated by the process are freed.
- All kernel objects are closed.
- The process code is removed from memory.
- The process exit code is set.
- The process object is signaled.

While open handles to kernel objects are closed automatically when a process terminates, the objects themselves exist until all open handles to them are closed. Therefore, an object will remain valid after a process that is using it terminates if another process has an open handle to it.

The **GetExitCodeProcess** function returns the termination status of a process. While a process is executing, its termination status is STILL_ACTIVE. When a process terminates, its termination status changes from STILL_ACTIVE to the exit code of the process.

When a process terminates, the state of the process object becomes signaled, releasing any threads that had been waiting for the process to terminate. For more about synchronization, see Synchronizing Execution of Multiple Threads.

When the system is terminating a process, it does not terminate any child processes that the process has created. Terminating a process does not generate notifications for WH_CBT hook procedures.

Use the **SetProcessShutdownParameters** function to specify certain aspects of the process termination at system shutdown, such as when a process should terminate relative to the other processes in the system.

How Processes are Terminated

A process executes until one of the following events occurs:

- Any thread of the process calls the **ExitProcess** function. Note that some implementation of the C run-time library (CRT) call **ExitProcess** if the primary thread of the process returns.
- The last thread of the process terminates.
- Any thread calls the **TerminateProcess** function with a handle to the process.
- For console processes, the default console control handler calls **ExitProcess** when the console receives a CTRL+C or CTRL+BREAK signal.
- The user shuts down the system or logs off.

Do not terminate a process unless its threads are in known states. If a thread is waiting on a kernel object, it will not be terminated until the wait has completed. This can cause the application to stop responding.

The primary thread can avoid terminating other threads by directing them to call **ExitThread** before causing the process to terminate (for more information, see Terminating a Thread). The primary thread can still call **ExitProcess** afterwards to ensure that all threads are terminated.

The exit code for a process is either the value specified in the call to **ExitProcess** or **TerminateProcess**, or the value returned by the main or **WinMain** function of the process. If a process is terminated due to a fatal exception, the exit code is the value of the exception that caused the termination. In addition, this value is used as

the exit code for all the threads that were executing when the exception occurred.

If a process is terminated by **ExitProcess**, the system calls the entry-point function of each attached DLL with a value indicating that the process is detaching from the DLL. DLLs are not notified when a process is terminated by **TerminateProcess**. For more information about DLLs, see Dynamic-Link Libraries.

If a process is terminated by **TerminateProcess**, all threads of the process are terminated immediately with no chance to run additional code. This means that the thread does not execute code in termination handler blocks. In addition, no attached DLLs are notified that the process is detaching. If you need to have one process terminate another process, the following steps provide a better solution:

- Have both processes call the **RegisterWindowMessage** function to create a private message.
- One process can terminate the other process by broadcasting a private message using the **BroadcastSystemMessage** function as follows:

```
DWORD dwRecipients = BSM_APPLICATIONS;
   UINT uMessage = PM_MYMSG;
   WPARAM wParam = 0;
   LPARAM lParam = 0;

   BroadcastSystemMessage(
       BSF_IGNORECURRENTTASK, // do not send message to this process
       &dwRecipients,         // broadcast only to applications
       uMessage,              // registered private message
       wParam,                // message-specific value
       lParam );              // message-specific value
```

- The process receiving the private message calls **ExitProcess** to terminate its execution.

The execution of the **ExitProcess**, **ExitThread**, **CreateThread**, **CreateRemoteThread**, and **CreateProcess** functions is serialized within an address space. The following restrictions apply:

- During process startup and DLL initialization routines, new threads can be created, but they do not begin execution until DLL initialization is finished for the process.
- Only one thread at a time can be in a DLL initialization or detach routine.
- The **ExitProcess** function does not return until there are no threads are in their DLL initialization or detach routines.

Process Working Set

9/16/2022 • 2 minutes to read • Edit Online

The *working set* of a program is a collection of those pages in its virtual address space that have been recently referenced. It includes both shared and private data. The shared data includes pages that contain all instructions your application executes, including those in your DLLs and the system DLLs. As the working set size increases, memory demand increases.

A process has an associated minimum working set size and maximum working set size. Each time you call **CreateProcess**, it reserves the minimum working set size for the process. The virtual memory manager attempts to keep enough memory for the minimum working set resident when the process is active, but keeps no more than the maximum size.

To get the requested minimum and maximum sizes of the working set for your application, call the **GetProcessWorkingSetSize** function.

The system sets the default working set sizes. You can also modify the working set sizes using the **SetProcessWorkingSetSize** function. Setting these values is not a guarantee that the memory will be reserved or resident. Be careful about requesting too large a minimum or maximum working set size, because doing so can degrade system performance.

To obtain the current or peak size of the working set for your process, use the **GetProcessMemoryInfo** function.

Related topics

Memory Performance Information

Working Set

Process Security and Access Rights

9/16/2022 • 4 minutes to read • Edit Online

The Microsoft Windows security model enables you to control access to process objects. For more information about security, see Access-Control Model.

When a user logs in, the system collects a set of data that uniquely identifies the user during the authentication process, and stores it in an access token. This access token describes the security context of all processes associated with the user. The security context of a process is the set of credentials given to the process or the user account that created the process.

You can use a token to specify the current security context for a process using the **CreateProcessWithTokenW** function. You can specify a security descriptor for a process when you call the **CreateProcess**, **CreateProcessAsUser**, or **CreateProcessWithLogonW** function. If you specify **NULL**, the process gets a default security descriptor. The ACLs in the default security descriptor for a process come from the primary or impersonation token of the creator.

To retrieve a process's security descriptor, call the **GetSecurityInfo** function. To change a process's security descriptor, call the **SetSecurityInfo** function.

The valid access rights for process objects include the standard access rights and some process-specific access rights. The following table lists the standard access rights used by all objects.

VALUE	MEANING
DELETE (0x00010000L)	Required to delete the object.
READ_CONTROL (0x00020000L)	Required to read information in the security descriptor for the object, not including the information in the SACL. To read or write the SACL, you must request the **ACCESS_SYSTEM_SECURITY** access right. For more information, see SACL Access Right.
SYNCHRONIZE (0x00100000L)	The right to use the object for synchronization. This enables a thread to wait until the object is in the signaled state.
WRITE_DAC (0x00040000L)	Required to modify the DACL in the security descriptor for the object.
WRITE_OWNER (0x00080000L)	Required to change the owner in the security descriptor for the object.

The following table lists the process-specific access rights.

VALUE	MEANING

VALUE	MEANING
PROCESS_ALL_ACCESS (STANDARD_RIGHTS_REQUIRED (0x000F0000L) | SYNCHRONIZE (0x00100000L) | 0xFFFF)	All possible access rights for a process object.**Windows Server 2003 and Windows XP:** The size of the **PROCESS_ALL_ACCESS** flag increased on Windows Server 2008 and Windows Vista. If an application compiled for Windows Server 2008 and Windows Vista is run on Windows Server 2003 or Windows XP, the **PROCESS_ALL_ACCESS** flag is too large and the function specifying this flag fails with **ERROR_ACCESS_DENIED**. To avoid this problem, specify the minimum set of access rights required for the operation. If **PROCESS_ALL_ACCESS** must be used, set _WIN32_WINNT to the minimum operating system targeted by your application (for example, `#define _WIN32_WINNT _WIN32_WINNT_WINXP`). For more information, see Using the Windows Headers.
PROCESS_CREATE_PROCESS (0x0080)	Required to use this process as the parent process with PROC_THREAD_ATTRIBUTE_PARENT_PROCESS.
PROCESS_CREATE_THREAD (0x0002)	Required to create a thread in the process.
PROCESS_DUP_HANDLE (0x0040)	Required to duplicate a handle using DuplicateHandle.
PROCESS_QUERY_INFORMATION (0x0400)	Required to retrieve certain information about a process, such as its token, exit code, and priority class (see OpenProcessToken).
PROCESS_QUERY_LIMITED_INFORMATION (0x1000)	Required to retrieve certain information about a process (see GetExitCodeProcess, GetPriorityClass, IsProcessInJob, QueryFullProcessImageName). A handle that has the **PROCESS_QUERY_INFORMATION** access right is automatically granted **PROCESS_QUERY_LIMITED_INFORMATION**.**Windows Server 2003 and Windows XP:** This access right is not supported.
PROCESS_SET_INFORMATION (0x0200)	Required to set certain information about a process, such as its priority class (see SetPriorityClass).
PROCESS_SET_QUOTA (0x0100)	Required to set memory limits using SetProcessWorkingSetSize.
PROCESS_SUSPEND_RESUME (0x0800)	Required to suspend or resume a process.
PROCESS_TERMINATE (0x0001)	Required to terminate a process using TerminateProcess.
PROCESS_VM_OPERATION (0x0008)	Required to perform an operation on the address space of a process (see VirtualProtectEx and WriteProcessMemory).
PROCESS_VM_READ (0x0010)	Required to read memory in a process using ReadProcessMemory.
PROCESS_VM_WRITE (0x0020)	Required to write to memory in a process using WriteProcessMemory.

VALUE	MEANING
SYNCHRONIZE (0x00100000L)	Required to wait for the process to terminate using the wait functions.

To open a handle to another process and obtain full access rights, you must enable the **SeDebugPrivilege** privilege. For more information, see Changing Privileges in a Token.

The handle returned by the **CreateProcess** function has **PROCESS_ALL_ACCESS** access to the process object. When you call the **OpenProcess** function, the system checks the requested access rights against the DACL in the process's security descriptor. When you call the **GetCurrentProcess** function, the system returns a pseudohandle with the maximum access that the DACL allows to the caller.

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

> **WARNING**
> A process that has some of the access rights noted here can use them to gain other access rights. For example, if process A has a handle to process B with **PROCESS_DUP_HANDLE** access, it can duplicate the pseudo handle for process B. This creates a handle that has maximum access to process B. For more information on pseudo handles, see **GetCurrentProcess**.

Protected Processes

Windows Vista introduces *protected processes* to enhance support for Digital Rights Management. The system restricts access to protected processes and the threads of protected processes.

The following standard access rights are not allowed from a process to a protected process:

- **DELETE**
- **READ_CONTROL**
- **WRITE_DAC**
- **WRITE_OWNER**

The following specific access rights are not allowed from a process to a protected process:

- **PROCESS_ALL_ACCESS**
- **PROCESS_CREATE_PROCESS**
- **PROCESS_CREATE_THREAD**
- **PROCESS_DUP_HANDLE**
- **PROCESS_QUERY_INFORMATION**
- **PROCESS_SET_INFORMATION**
- **PROCESS_SET_QUOTA**
- **PROCESS_VM_OPERATION**
- **PROCESS_VM_READ**
- **PROCESS_VM_WRITE**

The **PROCESS_QUERY_LIMITED_INFORMATION** right was introduced to provide access to a subset of the information available through **PROCESS_QUERY_INFORMATION**.

Thread Pools

9/16/2022 • 4 minutes to read • Edit Online

A *thread pool* is a collection of worker threads that efficiently execute asynchronous callbacks on behalf of the application. The thread pool is primarily used to reduce the number of application threads and provide management of the worker threads. Applications can queue work items, associate work with waitable handles, automatically queue based on a timer, and bind with I/O.

Thread Pool Architecture

The following applications can benefit from using a thread pool:

- An application that is highly parallel and can dispatch a large number of small work items asynchronously (such as distributed index search or network I/O).
- An application that creates and destroys a large number of threads that each run for a short time. Using the thread pool can reduce the complexity of thread management and the overhead involved in thread creation and destruction.
- An application that processes independent work items in the background and in parallel (such as loading multiple tabs).
- An application that must perform an exclusive wait on kernel objects or block on incoming events on an object. Using the thread pool can reduce the complexity of thread management and increase performance by reducing the number of context switches.
- An application that creates custom waiter threads to wait on events.

The original thread pool has been completely rearchitected in Windows Vista. The new thread pool is improved because it provides a single worker thread type (supports both I/O and non-I/O), does not use a timer thread, provides a single timer queue, and provides a dedicated persistent thread. It also provides clean-up groups, higher performance, multiple pools per process that are scheduled independently, and a new thread pool API.

The thread pool architecture consists of the following:

- Worker threads that execute the callback functions
- Waiter threads that wait on multiple wait handles
- A work queue
- A default thread pool for each process
- A worker factory that manages the worker threads

Best Practices

The new thread pool API provides more flexibility and control than the original thread pool API. However, there are a few subtle but important differences. In the original API, the wait reset was automatic; in the new API, the wait must be explicitly reset each time. The original API handled impersonation automatically, transferring the security context of the calling process to the thread. With the new API, the application must explicitly set the security context.

The following are best practices when using a thread pool:

- The threads of a process share the thread pool. A single worker thread can execute multiple callback functions, one at a time. These worker threads are managed by the thread pool. Therefore, do not terminate a thread from the thread pool by calling **TerminateThread** on the thread or by calling

ExitThread from a callback function.

- An I/O request can run on any thread in the thread pool. Canceling I/O on a thread pool thread requires synchronization because the cancel function might run on a different thread than the one that is handling the I/O request, which can result in cancellation of an unknown operation. To avoid this, always provide the OVERLAPPED structure with which an I/O request was initiated when calling CancelIoEx for asynchronous I/O, or use your own synchronization to ensure that no other I/O can be started on the target thread before calling either the CancelSynchronousIo or CancelIoEx function.
- Clean up all resources created in the callback function before returning from the function. These include TLS, security contexts, thread priority, and COM registration. Callback functions must also restore the thread state before returning.
- Keep wait handles and their associated objects alive until the thread pool has signaled that it is finished with the handle.
- Mark all threads that are waiting on lengthy operations (such as I/O flushes or resource cleanup) so that the thread pool can allocate new threads instead of waiting for this one.
- Before unloading a DLL that uses the thread pool, cancel all work items, I/O, wait operations, and timers, and wait for executing callbacks to complete.
- Avoid deadlocks by eliminating dependencies between work items and between callbacks, by ensuring a callback is not waiting for itself to complete, and by preserving the thread priority.
- Do not queue too many items too quickly in a process with other components using the default thread pool. There is one default thread pool per process, including Svchost.exe. By default, each thread pool has a maximum of 500 worker threads. The thread pool attempts to create more worker threads when the number of worker threads in the ready/running state must be less than the number of processors.
- Avoid the COM single-threaded apartment model, as it is incompatible with the thread pool. STA creates thread state which can affect the next work item for the thread. STA is generally long-lived and has thread affinity, which is the opposite of the thread pool.
- Create a new thread pool to control thread priority and isolation, create custom characteristics, and possibly improve responsiveness. However, additional thread pools require more system resources (threads, kernel memory). Too many pools increases the potential for CPU contention.
- If possible, use a waitable object instead of an APC-based mechanism to signal a thread pool thread. APCs do not work as well with thread pool threads as other signaling mechanisms because the system controls the lifetime of thread pool threads, so it is possible for a thread to be terminated before the notification is delivered.
- Use the thread pool debugger extension, !tp. This command has the following usage:
  - pool *address flags*
  - obj *address flags*
  - tqueue *address flags*
  - waiter *address*
  - worker *address*

  For pool, waiter, and worker, if the address is zero, the command dumps all objects. For waiter and worker, omitting the address dumps the current thread. The following flags are defined: 0x1 (single-line output), 0x2 (dump members), and 0x4 (dump pool work queue).

Related topics

Thread Pool API

Thread Pool API

9/16/2022 • 2 minutes to read • Edit Online

The thread pool application programming interface (API) uses an object-based design. Each of the following objects is represented by a user-mode data structure:

- A pool object is a set of worker threads that can be used to perform work. Each process can create multiple isolated pools with different characteristics as necessary. There is also a default pool for each process.
- A clean-up group is associated with a set of callback-generating objects. Functions exist to wait on and release all objects that are members of each clean-up group. This frees the application from keeping track of all the objects it has created.
- A work object is assigned to a pool and optionally to a clean-up group. It can be posted, causing a worker thread from the pool to execute its callback. A work object can have multiple posts outstanding; each generates a callback. The post operation cannot fail due to lack of resources.
- A timer object controls the scheduling of callbacks. Each time a timer expires, its callback is posted to its worker pool. Setting a timer cannot fail due to lack of resources.
- A wait object causes a waiter thread to wait on a waitable handle. After the wait is satisfied or the time-out period expires, the waiter thread posts the wait objects' callback to the wait's worker pool. Setting a wait cannot fail due to lack of resources.
- An I/O object associates a file handle with the I/O completion port for the thread pool. When an asynchronous I/O operation completes, a worker thread picks up the status of the operation and calls the I/O object's callback.

The following table describes the features of the original and current thread pool APIs.

FEATURE	ORIGINAL API	CURRENT API
Synch	RegisterWaitForSingleObject UnregisterWaitEx	CloseThreadpoolWait CreateThreadpoolWait SetThreadpoolWait WaitForThreadpoolWaitCallbacks
Work	QueueUserWorkItem	CloseThreadpoolWork CreateThreadpoolWork SubmitThreadpoolWork TrySubmitThreadpoolCallback WaitForThreadpoolWorkCallbacks
Timer	CreateTimerQueue CreateTimerQueueTimer ChangeTimerQueueTimer DeleteTimerQueueTimer DeleteTimerQueueEx	CloseThreadpoolTimer CreateThreadpoolTimer IsThreadpoolTimerSet SetThreadpoolTimer WaitForThreadpoolTimerCallbacks
I/O	BindIoCompletionCallback	CancelThreadpoolIo CloseThreadpoolIo CreateThreadpoolIo StartThreadpoolIo WaitForThreadpoolIoCallbacks

FEATURE	ORIGINAL API	CURRENT API
Clean-up group		CloseThreadpoolCleanupGroup CloseThreadpoolCleanupGroupMembers CreateThreadpoolCleanupGroup
Pool		CloseThreadpool CreateThreadpool SetThreadpoolThreadMaximum SetThreadpoolThreadMinimum
Callback environment		DestroyThreadpoolEnvironment InitializeThreadpoolEnvironment SetThreadpoolCallbackCleanupGroup SetThreadpoolCallbackLibrary SetThreadpoolCallbackPool SetThreadpoolCallbackPriority SetThreadpoolCallbackRunsLong
Callback		CallbackMayRunLong
Callback clean up		DisassociateCurrentThreadFromCallback FreeLibraryWhenCallbackReturns LeaveCriticalSectionWhenCallbackReturns ReleaseMutexWhenCallbackReturns ReleaseSemaphoreWhenCallbackReturns SetEventWhenCallbackReturns

Related topics

Thread Pools

Using the Thread Pool Functions

Thread Pooling

9/16/2022 • 2 minutes to read • Edit Online

There are many applications that create threads that spend a great deal of time in the sleeping state waiting for an event to occur. Other threads may enter a sleeping state only to be awakened periodically to poll for a change or update status information. *Thread pooling* enables you to use threads more efficiently by providing your application with a pool of worker threads that are managed by the system. At least one thread monitors the status of all wait operations queued to the thread pool. When a wait operation has completed, a worker thread from the thread pool executes the corresponding callback function.

This topic describes the original thread pool API. The thread pool API introduced in Windows Vista is simpler, more reliable, has better performance, and provides more flexibility for developers. For information on the current thread pool API, see Thread Pools.

You can also queue work items that are not related to a wait operation to the thread pool. To request that a work item be handled by a thread in the thread pool, call the **QueueUserWorkItem** function. This function takes a parameter to the function that will be called by the thread selected from the thread pool. There is no way to cancel a work item after it has been queued.

Timer-queue timers and registered wait operations also use the thread pool. Their callback functions are queued to the thread pool. You can also use the **BindIoCompletionCallback** function to post asynchronous I/O operations. On completion of the I/O , the callback is executed by a thread pool thread.

The thread pool is created the first time you call **QueueUserWorkItem** or **BindIoCompletionCallback**, or when a timer-queue timer or registered wait operation queues a callback function. By default, the number of threads that can be created in the thread pool is about 500. Each thread uses the default stack size and runs at the default priority.

There are two types of worker threads in the thread pool: I/O and non-I/O. An *I/O worker thread* is a thread that waits in an alertable wait state. Work items are queued to I/O worker threads as asynchronous procedure calls (APC). You should queue a work item to an I/O worker thread if it should be executed in a thread that waits in an alertable state.

A *non-I/O worker thread* waits on I/O completion ports. Using non-I/O worker threads is more efficient than using I/O worker threads. Therefore, you should use non-I/O worker threads whenever possible. Both I/O and non-I/O worker threads do not exit if there are pending asynchronous I/O requests. Both types of threads can be used by work items that initiate asynchronous I/O completion requests. However, avoid posting asynchronous I/O completion requests in non-I/O worker threads if they could take a long time to complete.

To use thread pooling, the work items and all the functions they call must be thread-pool safe. A safe function does not assume that the thread executing it is a dedicated or persistent thread. In general, you should avoid using thread local storage or making an asynchronous call that requires a persistent thread, such as the **RegNotifyChangeKeyValue** function. However, such functions can be called on a dedicated thread (created by the application) or queued to a persistent worker thread (using **QueueUserWorkItem** with the WT_EXECUTEINPERSISTENTTHREAD option).

Related topics

Alertable I/O

Asynchronous Procedure Calls

I/O Completion Ports

Thread Pools

Job Objects

9/16/2022 • 6 minutes to read • Edit Online

A *job object* allows groups of processes to be managed as a unit. Job objects are namable, securable, sharable objects that control attributes of the processes associated with them. Operations performed on a job object affect all processes associated with the job object. Examples include enforcing limits such as working set size and process priority or terminating all processes associated with a job.

Creating Jobs

To create a job object, use the **CreateJobObject** function. When the job is created, no processes are associated with the job.

To associate a process with a job, use the **AssignProcessToJobObject** function. After a process is associated with a job, the association cannot be broken. A process can be associated with more than one job in a hierarchy of nested jobs. For more information, see Nested Jobs.

Windows 7, Windows Server 2008 R2, Windows XP with SP3, Windows Server 2008, Windows Vista and Windows Server 2003: A process can be associated with only one job. Jobs cannot be nested. The ability to nest jobs was added in Windows 8 and Windows Server 2012.

You can specify a security descriptor for a job object when you call the **CreateJobObject** function. For more information, see Job Object Security and Access Rights.

Managing Processes in Jobs

After a process is associated with a job, by default any child processes it creates using **CreateProcess** are also associated with the job. (Child processes created using **Win32_Process.Create** are not associated with the job.) This default behavior can be changed by setting the extended limit JOB_OBJECT_LIMIT_BREAKAWAY_OK or JOB_OBJECT_LIMIT_SILENT_BREAKAWAY_OK for the job.

- If the job has the extended limit JOB_OBJECT_LIMIT_BREAKAWAY_OK and the parent process was created with the CREATE_BREAKAWAY_FROM_JOB flag, then child processes of the parent process are not associated with the job.
- If the job has the extended limit JOB_OBJECT_LIMIT_SILENT_BREAKAWAY_OK, then child processes of any parent process associated with the job are not associated with the job. It is not necessary for parent processes to be created with the CREATE_BREAKAWAY_FROM_JOB flag.

If the job is nested, the breakaway settings of parent jobs in the hierarchy affect whether child processes are associated with another job in the hierarchy. For more information, see Nested Jobs.

To determine if a process is running in a job, use the **IsProcessInJob** function.

To terminate all processes currently associated with a job object, use the **TerminateJobObject** function.

Job Limits and Notifications

A job can enforce limits such as working set size, process priority, and end-of-job time limit on each process that is associated with the job. If a process associated with a job attempts to increase its working set size or process priority from the limit established by the job, the function calls succeed but are silently ignored. A job can also set limits that trigger a notification when they are exceeded but allow the job to continue to run.

To set limits for a job, use the **SetInformationJobObject** function. For a list of possible limits that can be set for a job, see the following topics:

- **JOBOBJECT_BASIC_LIMIT_INFORMATION**
- **JOBOBJECT_BASIC_UI_RESTRICTIONS**
- **JOBOBJECT_CPU_RATE_CONTROL_INFORMATION**
- **JOBOBJECT_EXTENDED_LIMIT_INFORMATION**
- **JOBOBJECT_NOTIFICATION_LIMIT_INFORMATION**

Security limits must be set individually for each process associated with a job object. For more information, see Process Security and Access Rights.

Windows XP with SP3 and Windows Server 2003: The **SetInformationJobObject** function can be used to set security limitations for all processes associated with a job object. Starting with Windows Vista, security limits must be set individually for each process associated with a job object.

If the job is nested, parent jobs in the hierarchy influence the limit that is enforced for the job. For more information, see Nested Jobs.

If the job has an associated I/O completion port, it can receive notifications when certain job limits are exceeded. The system sends messages to the completion port when a limit is exceeded or certain other events occur. To associate a completion port with a job, use the **SetInformationJobObject** function with the job object information class **JobObjectAssociateCompletionPortInformation** and a pointer to a **JOBOBJECT_ASSOCIATE_COMPLETION_PORT** structure. It is best to do this when the job is inactive, to reduce the chance of missing notifications for processes whose states change during the association of the completion port.

All messages are sent directly from the job as if the job had called the **PostQueuedCompletionStatus** function. A thread must monitor the completion port using the **GetQueuedCompletionStatus** function to pick up the messages. Note that, with the exception of limits set with the **JobObjectNotificationLimitInformation** information class, delivery of messages to the completion port is not guaranteed; failure of a message to arrive does not necessarily mean that the event did not occur. Notifications for limits set with **JobObjectNotificationLimitInformation** are guaranteed to arrive at the completion port. For a list of possible messages, see **JOBOBJECT_ASSOCIATE_COMPLETION_PORT**.

Resource Accounting for Jobs

The job object records basic accounting information for all its associated processes, including those that have terminated. To retrieve this accounting information, use the **QueryInformationJobObject** function. For a list of the accounting information that is maintained for a job, see the following topics:

- **JOBOBJECT_BASIC_ACCOUNTING_INFORMATION**
- **JOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION**

If the job object is nested, accounting information for each child job is aggregated in its parent job. For more information, see Nested Jobs.

Managing Job Objects

The state of a job object is set to signaled when all of its processes are terminated because the specified end-of-job time limit has been exceeded. Use **WaitForSingleObject** or **WaitForSingleObjectEx** to monitor the job object for this event.

To obtain a handle for an existing job object, use the **OpenJobObject** function and specify the name given to the object when it was created. Only named job objects can be opened.

To close a job object handle, use the **CloseHandle** function. The job is destroyed when its last handle has been closed and all associated processes have been terminated. However, if the job has the JOB_OBJECT_LIMIT_KILL_ON_JOB_CLOSE flag specified, closing the last job object handle terminates all associated processes and then destroys the job object itself. If a nested job has the JOB_OBJECT_LIMIT_KILL_ON_JOB_CLOSE flag specified, closing the last job object handle terminates all processes associated with the job and its child jobs in the hierarchy.

Managing a Process Tree that Uses Job Objects

Starting with Windows 8 and Windows Server 2012, an application can use nested jobs to manage a process tree that uses more than one job object. However, an application that must run on Windows 7, Windows Server 2008 R2, or earlier versions of Windows that do not support nested jobs must manage the process tree in other ways.

If a tool must manage a process tree that uses job objects and it is not possible to use nested jobs, both the tool and the members of the process tree must cooperate. Use one of the following options:

- Use the JOB_OBJECT_LIMIT_SILENT_BREAKAWAY_OK limit. If the tool uses this limit, it cannot monitor an entire process tree. The tool can monitor only the processes it adds to the job. If these processes create child processes, they are not associated with the job. In this option, child processes can be associated with other job objects.

- Use the JOB_OBJECT_LIMIT_BREAKAWAY_OK limit. If the tool uses this limit, it can monitor the entire process tree, except for those processes that any member of the tree explicitly breaks away from the tree. A member of the tree can create a child process in a new job object by calling the **CreateProcess** function with the CREATE_BREAKAWAY_FROM_JOB flag, then calling the **AssignProcessToJobObject** function. Otherwise, the member must handle cases in which **AssignProcessToJobObject** fails.

  The CREATE_BREAKAWAY_FROM_JOB flag has no effect if the tree is not being monitored by the tool. Therefore, this is the preferred option, but it requires advance knowledge of the processes being monitored.

- Prevent breakaways of any kind by setting neither the JOB_OBJECT_LIMIT_BREAKAWAY_OK nor the JOB_OBJECT_LIMIT_SILENT_BREAKAWAY_OK limit. In this option, the tool can monitor the entire process tree. However, if a child process attempts to associate itself or another child process with a job by calling **AssignProcessToJobObject**, the call will fail. If the process was designed to be associated with a specific job, this failure may prevent the process from working properly.

Nested Jobs

9/16/2022 • 5 minutes to read • Edit Online

An application can use nested jobs to manage subsets of processes. Nested jobs also enable an application that uses jobs to host other applications that also use jobs.

Windows 7, Windows Server 2008 R2, Windows XP with SP3, Windows Server 2008, Windows Vista and Windows Server 2003: A process can be associated with only a single job. Nested jobs were introduced in Windows 8 and Windows Server 2012.

This topic provides an overview of job nesting and behavior of nested jobs:

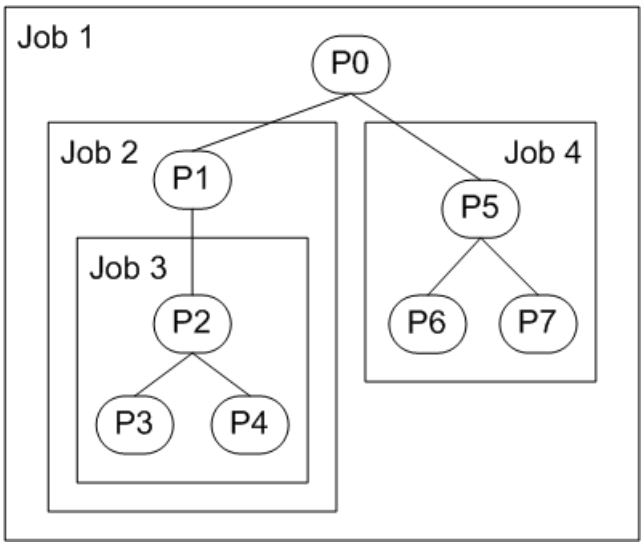
- Nested Job Hierarchies
- Creating a Nested Job Hierarchy
- Job Limits and Notifications for Nested Jobs
- Resource Accounting for Nested Jobs
- Termination of Nested Jobs

For general information about jobs and job objects, see Job Objects.

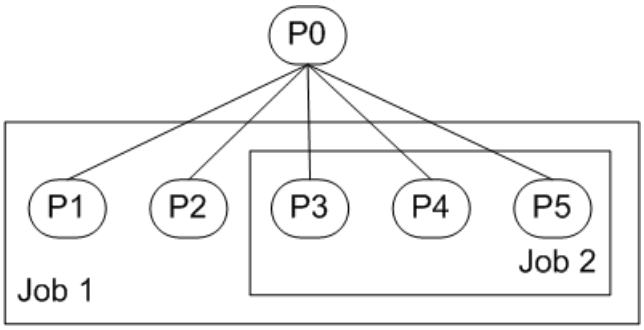
Nested Job Hierarchies

Nested jobs have a parent-child relationship in which each child job contains a subset of the processes in its parent job. If a process that is already in a job is added to another job, the jobs are nested by default if the system can form a valid job hierarchy and neither job sets UI limits (**SetInformationJobObject** with **JobObjectBasicUIRestrictions**).

Figure 1 shows a job hierarchy that contains a tree of processes labeled P0 through P7. Job 1 is the *parent job* of Job 2 and Job 4, and it is an *ancestor* of Job 3. Job 2 is the *immediate parent* of Job 3; Job 3 is the *immediate child* of Job 2. Jobs 1, 2, and 3 form a *job chain* in which Jobs 1 and 2 are the *parent job chain* of Job 3. The end job in a job chain is the *immediate job* of the processes in that job. In Figure 1, Job 3 is the immediate job of processes P2, P3, and P4.



Nested jobs can also be used to manage groups of peer processes. In the job hierarchy shown in Figure 2, Job 1 is the parent job of Job 2. Note that a job hierarchy might contain only part of a process tree. In Figure 2, P0 is not in the hierarchy, but its child processes P1 through P5 are.



Creating a Nested Job Hierarchy

Processes in a job hierarchy are either explicitly associated with a job object using the **AssignProcessToJobObject** function or implicitly associated during process creation, same as for standalone jobs. The order in which jobs are created and processes are assigned determines whether a hierarchy can be created.

To build a job hierarchy using explicit association, all job objects must be created using **CreateJobObject**, then **AssignProcessToJobObject** must be called multiple times for each process to associate the process with each job it should belong to. To ensure that the job hierarchy is valid, first assign all processes to the job at the root of the hierarchy, then assign a subset of processes to the immediate child job object, and so on. If processes are assigned to jobs in this order, a child job will always have a subset of processes in its parent job while the hierarchy is being created, which is required for nesting. If processes are assigned to jobs in random order, at some point a child job will have processes that are not in its parent job. This is not allowed by nesting and it will cause **AssignProcessToJobObject** to fail.

When processes are implicitly associated with a job during process creation, a child process is associated with every job in the job chain of its parent process. If the immediate job object allows breakaway, the child process breaks away from the immediate job object and from each job in the parent job chain, moving up the hierarchy until it reaches a job that does not permit breakaway. If the immediate job object does not allow breakaway, the child process does not break away even if jobs in its parent job chain allow it.

Job Limits and Notifications for Nested Jobs

For certain resource limits, the limit set for jobs in a parent job chain determine the *effective limit* that is enforced for a child job. The effective limit for child job can be more restrictive than the limit of its parent, but it cannot be less restrictive. For example, if a child job's priority class is ABOVE_NORMAL_PRIORITY_CLASS and its parent job's priority class is NORMAL_PRIORITY_CLASS, the effective limit for processes in the child job is NORMAL_PRIORITY_CLASS. However, if the child job's priority class is BELOW_NORMAL_PRIORITY_CLASS, the effective limit for processes in the child job is BELOW_NORMAL_PRIORITY_CLASS. Effective limits are enforced for priority class, affinity, commit charge, per-process execution time limit, scheduling class limit, and working set minimum and maximum. For more information about specific resource limits, see **SetInformationJobObject**.

When certain events occur, such as new process creation or resource limit violation, a message is sent to the I/O completion port associated with a job. A job can also register to receive notifications when certain limits are exceeded. For a non-nested job, the message is sent to the I/O completion port associated with the job. For a nested job, the message is sent to every I/O completion port associated with any job in the parent job chain of the job that triggered the message. A child job does not need to have an associated I/O completion port for messages it triggers to be sent to the I/O completion ports of parent jobs higher in the job chain. For more information about specific messages, see **JOBOBJECT_ASSOCIATE_COMPLETION_PORT**.

Resource Accounting for Nested Jobs

Resource accounting information for a nested job describes the usage of every process associated with that job,

including processes in child jobs. Each job in a job chain therefore represents the aggregated resources used by its own processes and the processes of every child job below it in the job chain.

Termination of Nested Jobs

When a job in a job hierarchy is terminated, the system terminates processes in that job and all of its child jobs, starting with the child job at the bottom of the hierarchy. Outstanding resources used by each terminated process are charged to the parent job.

The job handle must have the JOB_OBJECT_TERMINATE access right, same as for standalone jobs.

Job Object Security and Access Rights

9/16/2022 • 2 minutes to read • Edit Online

The Microsoft Windows security model enables you to control access to job objects. For more information about security, see Access-Control Model.

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

To get or set the security descriptor for a job object, call the **GetNamedSecurityInfo**, **SetNamedSecurityInfo**, **GetSecurityInfo**, or **SetSecurityInfo** function.

The valid access rights for job objects include the standard access rights and some job-specific access rights. The following table lists the standard access rights used by all objects.

VALUE	MEANING
DELETE (0x00010000L)	Required to delete the object.
READ_CONTROL (0x00020000L)	Required to read information in the security descriptor for the object, not including the information in the SACL. To read or write the SACL, you must request the **ACCESS_SYSTEM_SECURITY** access right. For more information, see SACL Access Right.
SYNCHRONIZE (0x00100000L)	The right to use the object for synchronization. This enables a thread to wait until the object is in the signaled state.
WRITE_DAC (0x00040000L)	Required to modify the DACL in the security descriptor for the object.
WRITE_OWNER (0x00080000L)	Required to change the owner in the security descriptor for the object.

The following table lists the job-specific access rights.

VALUE	MEANING
JOB_OBJECT_ALL_ACCESS (0x1F001F)	Combines all valid job object access rights.
JOB_OBJECT_ASSIGN_PROCESS (0x0001)	Required to call the **AssignProcessToJobObject** function to assign processes to the job object.
JOB_OBJECT_QUERY (0x0004)	Required to retrieve certain information about a job object, such as attributes and accounting information (see **QueryInformationJobObject** and **IsProcessInJob**).
JOB_OBJECT_SET_ATTRIBUTES (0x0002)	Required to call the **SetInformationJobObject** function to set the attributes of the job object.

VALUE	MEANING
JOB_OBJECT_SET_SECURITY_ATTRIBUTES (0x0010)	This flag is not supported. You must set security limitations individually for each process associated with a job object.**Windows Server 2003 and Windows XP:** Required to call the **SetInformationJobObject** function with the **JobObjectSecurityLimitInformation** information class to set security limitations for the processes associated with the job object. Support for this flag was removed in Windows Vista and Windows Server 2008.
JOB_OBJECT_TERMINATE (0x0008)	Required to call the **TerminateJobObject** function to terminate all processes in the job object.

The handle returned by **CreateJobObject** has **JOB_OBJECT_ALL_ACCESS** access to the job object. When you call the **OpenJobObject** function, the system checks the requested access rights against the object's security descriptor. If a job object is in a hierarchy of nested jobs, a caller with access to the job object implicitly has access to all of its child jobs in the hierarchy.

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

You must set security limitations individually for each process associated with a job object, rather than setting them for the job object itself. For information, see Process Security and Access Rights.

Windows Server 2003 and Windows XP: You can use the **SetInformationJobObject** function to set security limitations for the job object. This capability was removed in Windows Vista and Windows Server 2008.

CPU Sets

9/16/2022 • 2 minutes to read • Edit Online

CPU Sets provide APIs to declare application affinity in a 'soft' manner that is compatible with OS power management. Additionally, the API provides applications with the ability to reaffinitize all background threads in the process to a subset of processors using the **Process Default** mechanism to avoid interference from OS threads within the process. Some versions of Windows support Core Reservation policies, in which a subset of the system's CPU Sets can be devoted to the exclusive use of individual applications and workloads.

The CPU Set API works with CPU Set IDs, which are associated with virtual processor affinities. On most systems, and under most conditions, each CPU Set ID will map directly to a single **home** logical processor. A thread affinitized to a given CPU Set will typically execute on one of the processors in its list of selected CPU Set IDs.

CPU Sets that are reserved can be determined by inspecting the **Allocated** flag in the SYSTEM_CPU_SET_INFORMATION. The system controls access to reserved CPU Sets and the assignment can be queried using the **AllocatedToTargetProcess** flag of the SYSTEM_CPU_SET_INFORMATION structure. If a process attempts to use a CPU Set assignment which is allocated exclusively to other processes, its request is ignored and threads assigned to disallowed CPU sets are scheduled elsewhere. CPU Sets can be assigned at two levels. The Process Default CPU sets are assigned to all threads in a process that do not have an assignment at the Thread Selected level. If a thread or process has a restrictive affinity mask set, the affinity mask is respected above any conflicting CPU Set assignment. On multi-group systems, CPU assignments are ignored if they are in groups that do not match the group in the thread's affinity mask. If a thread is assigned to multiple valid CPU Sets, it will run on one of the corresponding processors according to its priorities and the priorities of competing threads on those processors.

CPU Set Functions/Enumerations/Structures

- GetProcessDefaultCpuSets function
- GetProcessDefaultCpuSetMasks function
- GetSystemCpuSetInformation function
- GetThreadSelectedCpuSets function
- GetThreadSelectedCpuSetMasks function
- SetProcessDefaultCpuSets function
- SetProcessDefaultCpuSetMasks function
- SetThreadSelectedCpuSets function
- SetThreadSelectedCpuSetMasks function
- CPU_SET_INFORMATION_TYPE enumeration
- SYSTEM_CPU_SET_INFORMATION structure

Fibers

9/16/2022 • 2 minutes to read • Edit Online

A *fiber* is a unit of execution that must be manually scheduled by the application. Fibers run in the context of the threads that schedule them. Each thread can schedule multiple fibers. In general, fibers do not provide advantages over a well-designed multithreaded application. However, using fibers can make it easier to port applications that were designed to schedule their own threads.

From a system standpoint, operations performed by a fiber are considered to have been performed by the thread that runs it. For example, if a fiber accesses thread local storage (TLS), it is accessing the thread local storage of the thread that is running it. In addition, if a fiber calls the **ExitThread** function, the thread that is running it exits. However, a fiber does not have all the same state information associated with it as that associated with a thread. The only state information maintained for a fiber is its stack, a subset of its registers, and the fiber data provided during fiber creation. The saved registers are the set of registers typically preserved across a function call.

Fibers are not preemptively scheduled. You schedule a fiber by switching to it from another fiber. The system still schedules threads to run. When a thread running fibers is preempted, its currently running fiber is preempted but remains selected. The selected fiber runs when its thread runs.

Before scheduling the first fiber, call the **ConvertThreadToFiber** function to create an area in which to save fiber state information. The calling thread is now the currently executing fiber. The stored state information for this fiber includes the fiber data passed as an argument to **ConvertThreadToFiber**.

The **CreateFiber** function is used to create a new fiber from an existing fiber; the call requires the stack size, the starting address, and the fiber data. The starting address is typically a user-supplied function, called the fiber function, that takes one parameter (the fiber data) and does not return a value. If your fiber function returns, the thread running the fiber exits. To execute any fiber created with **CreateFiber**, call the **SwitchToFiber** function. You can call **SwitchToFiber** with the address of a fiber created by a different thread. To do this, you must have the address returned to the other thread when it called **CreateFiber** and you must use proper synchronization.

A fiber can retrieve the fiber data by calling the **GetFiberData** macro. A fiber can retrieve the fiber address at any time by calling the **GetCurrentFiber** macro.

Fiber Local Storage

A fiber can use *fiber local storage* (FLS) to create a unique copy of a variable for each fiber. If no fiber switching occurs, FLS acts exactly the same as thread local storage. The FLS functions (**FlsAlloc**, **FlsFree**, **FlsGetValue**, and **FlsSetValue**) manipulate the FLS associated with the current thread. If the thread is executing a fiber and the fiber is switched, the FLS is also switched.

To clean up the data associated with a fiber, call the **DeleteFiber** function. This data includes the stack, a subset of the registers, and the fiber data. If the currently running fiber calls **DeleteFiber**, its thread calls **ExitThread** and terminates. However, if the selected fiber of a thread is deleted by a fiber running in another thread, the thread with the deleted fiber is likely to terminate abnormally because the fiber stack has been freed.

Related topics

Using Fibers

User-Mode Scheduling

9/16/2022 • 8 minutes to read • Edit Online

WARNING

As of Windows 11, user-mode scheduling is not supported. All calls fail with the error `ERROR_NOT_SUPPORTED`.

User-mode scheduling (UMS) is a lightweight mechanism that applications can use to schedule their own threads. An application can switch between UMS threads in user mode without involving the system scheduler and regain control of the processor if a UMS thread blocks in the kernel. UMS threads differ from fibers in that each UMS thread has its own thread context instead of sharing the thread context of a single thread. The ability to switch between threads in user mode makes UMS more efficient than thread pools for managing large numbers of short-duration work items that require few system calls.

UMS is recommended for applications with high performance requirements that need to efficiently run many threads concurrently on multiprocessor or multicore systems. To take advantage of UMS, an application must implement a scheduler component that manages the application's UMS threads and determines when they should run. Developers should consider whether their application performance requirements justify the work involved in developing such a component. Applications with moderate performance requirements might be better served by allowing the system scheduler to schedule their threads.

UMS is available for 64-bit applications running on AMD64 and Itanium versions of Windows 7 and Windows Server 2008 R2 through Windows 10 Version 21H2 and Windows Server 2022. This feature is not available on Arm64, 32-bit versions of Windows or on Windows 11.

For details, see the following sections:

- UMS Scheduler
- UMS Scheduler Thread
- UMS Worker Threads, Thread Contexts, and Completion Lists
- UMS Scheduler Entry Point Function
- UMS Thread Execution
- UMS Best Practices

UMS Scheduler

An application's UMS scheduler is responsible for creating, managing, and deleting UMS threads and determining which UMS thread to run. An application's scheduler performs the following tasks:

- Creates one UMS scheduler thread for each processor on which the application will run UMS worker threads.
- Creates UMS worker threads to perform the work of the application.
- Maintains its own ready-thread queue of worker threads that are ready to run, and selects threads to run based on the application's scheduling policies.
- Creates and monitors one or more completion lists where the system queues threads after they finish processing in the kernel. These include newly created worker threads and threads previously blocked on a system call that become unblocked.
- Provides a scheduler entry point function to handle notifications from the system. The system calls the entry point function when a scheduler thread is created, a worker thread blocks on a system call, or a worker thread explicitly yields control.

- Performs cleanup tasks for worker threads that have finished running.
- Performs an orderly shutdown of the scheduler when requested by the application.

UMS Scheduler Thread

A UMS scheduler thread is an ordinary thread that has converted itself to UMS by calling the **EnterUmsSchedulingMode** function. The system scheduler determines when the UMS scheduler thread runs based on its priority relative to other ready threads. The processor on which the scheduler thread runs is influenced by the thread's affinity, same as for non-UMS threads.

The caller of **EnterUmsSchedulingMode** specifies a completion list and a *UmsSchedulerProc* entry point function to associate with the UMS scheduler thread. The system calls the specified entry point function when it is finished converting the calling thread to UMS. The scheduler entry point function is responsible for determining the appropriate next action for the specified thread. For more information, see UMS Scheduler Entry Point Function later in this topic.

An application might create one UMS scheduler thread for each processor that will be used to run UMS threads. The application might also set the affinity of each UMS scheduler thread for a specific logical processor, which tends to exclude unrelated threads from running on that processor, effectively reserving it for that scheduler thread. Be aware that setting thread affinity in this way can affect overall system performance by starving other processes that may be running on the system. For more information about thread affinity, see Multiple Processors.

UMS Worker Threads, Thread Contexts, and Completion Lists

A UMS worker thread is created by calling **CreateRemoteThreadEx** with the PROC_THREAD_ATTRIBUTE_UMS_THREAD attribute and specifying a UMS thread context and a completion list.

A UMS thread context represents the UMS thread state of a worker thread and is used to identify the worker thread in UMS function calls. It is created by calling **CreateUmsThreadContext**.

A completion list is created by calling the **CreateUmsCompletionList** function. A completion list receives UMS worker threads that have completed execution in the kernel and are ready to run in user mode. Only the system can queue worker threads to a completion list. New UMS worker threads are automatically queued to the completion list specified when the threads were created. Previously blocked worker threads are also queued to the completion list when they are no longer blocked.

Each UMS scheduler thread is associated with a single completion list. However, the same completion list can be associated with any number of UMS scheduler threads, and a scheduler thread can retrieve UMS contexts from any completion list for which it has a pointer.

Each completion list has an associated event that is signaled by the system when it queues one or more worker threads to an empty list. The **GetUmsCompletionListEvent** function retrieves a handle to the event for a specified completion list. An application can wait on more than one completion list event along with other events that make sense for the application.

UMS Scheduler Entry Point Function

An application's scheduler entry point function is implemented as a *UmsSchedulerProc* function. The system calls the application's scheduler entry point function at the following times:

- When a non-UMS thread is converted to a UMS scheduler thread by calling **EnterUmsSchedulingMode**.
- When a UMS worker thread calls **UmsThreadYield**.
- When a UMS worker thread blocks on a system service such as a system call or a page fault.

The *Reason* parameter of the *UmsSchedulerProc* function specifies the reason that the entry point function was

called. If the entry point function was called because a new UMS scheduler thread was created, the *SchedulerParam* parameter contains data specified by the caller of **EnterUmsSchedulingMode**. If the entry point function was called because a UMS worker thread yielded, the *SchedulerParam* parameter contains data specified by the caller of **UmsThreadYield**. If the entry point function was called because a UMS worker thread blocked in the kernel, the *SchedulerParam* parameter is NULL.

The scheduler entry point function is responsible for determining the appropriate next action for the specified thread. For example, if a worker thread is blocked, the scheduler entry point function might run the next available ready UMS worker thread.

When the scheduler entry point function is called, the application's scheduler should attempt to retrieve all of the items in its associated completion list by calling the **DequeueUmsCompletionListItems** function. This function retrieves a list of UMS thread contexts that have finished processing in the kernel and are ready to run in user mode. The application's scheduler should not run UMS threads directly from this list because this can cause unpredictable behavior in the application. Instead, the scheduler should retrieve all UMS thread contexts by calling the **GetNextUmsListItem** function once for each context, insert the UMS thread contexts in the scheduler's ready thread queue, and only then run UMS threads from the ready thread queue.

If the scheduler does not need to wait on multiple events, it should call **DequeueUmsCompletionListItems** with a nonzero timeout parameter so the function waits on the completion list event before returning. If the scheduler does need to wait on multiple completion list events, it should call **DequeueUmsCompletionListItems** with a timeout parameter of zero so the function returns immediately, even if the completion list is empty. In this case, the scheduler can wait explicitly on completion list events, for example, by using **WaitForMultipleObjects**.

UMS Thread Execution

A newly created UMS worker thread is queued to the specified completion list and does not begin running until the application's UMS scheduler selects it to run. This differs from non-UMS threads, which the system scheduler automatically schedules to run unless the caller explicitly creates the thread suspended.

The scheduler runs a worker thread by calling **ExecuteUmsThread** with the worker thread's UMS context. A UMS worker thread runs until it yields by calling the **UmsThreadYield** function, blocks, or terminates.

UMS Best Practices

Applications that implement UMS should follow these best practices:

- The underlying structures for UMS thread contexts are managed by the system and should not be modified directly. Instead, use **QueryUmsThreadInformation** and **SetUmsThreadInformation** to retrieve and set information about a UMS worker thread.
- To help prevent deadlocks, the UMS scheduler thread should not share locks with UMS worker threads. This includes both application-created locks and system locks that are acquired indirectly by operations such as allocating from the heap or loading DLLs. For example, suppose the scheduler runs a UMS worker thread that loads a DLL. The worker thread acquires the loader lock and blocks. The system calls the scheduler entry point function, which then loads a DLL. This causes a deadlock, because the loader lock is already held and cannot be released until the first thread unblocks. To help avoid this problem, delegate work that might share locks with UMS worker threads to a dedicated UMS worker thread or a non-UMS thread.
- UMS is most efficient when most processing is done in user mode. Whenever possible, avoid making system calls in UMS worker threads.
- UMS worker threads should not assume the system scheduler is being used. This assumption can have subtle effects; for example, if a thread in the unknown code sets a thread priority or affinity, the UMS scheduler might still override it. Code that assumes the system scheduler is being used may not behave as expected and may break when called by a UMS thread.

- The system may need to lock the thread context of a UMS worker thread. For example, a kernel-mode asynchronous procedure call (APC) might change the context of the UMS thread, so the thread context must be locked. If the scheduler tries to execute the UMS thread context while it is locked, the call will fail. This behavior is by design, and the scheduler should be designed to retry access to the UMS thread context.

Using Processes and Threads

9/16/2022 • 2 minutes to read • Edit Online

The following examples demonstrate the process, thread, and fiber functions:

- Creating processes
- Creating threads
- Creating a child process with redirected input and output
- Isolated User Mode (IUM) Processes
- Changing environment variables
- Using thread local storage
- Using fibers
- Using the thread pool functions

Creating Processes

9/16/2022 • 2 minutes to read • Edit Online

The CreateProcess function creates a new process, which runs independently of the creating process. However, for simplicity, the relationship is referred to as a parent-child relationship.

The following code demonstrates how to create a process.

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

void _tmain( int argc, TCHAR *argv[] )
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    ZeroMemory( &si, sizeof(si) );
    si.cb = sizeof(si);
    ZeroMemory( &pi, sizeof(pi) );

    if( argc != 2 )
    {
        printf("Usage: %s [cmdline]\n", argv[0]);
        return;
    }

    // Start the child process.
    if( !CreateProcess( NULL,   // No module name (use command line)
        argv[1],        // Command line
        NULL,           // Process handle not inheritable
        NULL,           // Thread handle not inheritable
        FALSE,          // Set handle inheritance to FALSE
        0,              // No creation flags
        NULL,           // Use parent's environment block
        NULL,           // Use parent's starting directory
        &si,            // Pointer to STARTUPINFO structure
        &pi )           // Pointer to PROCESS_INFORMATION structure
    )
    {
        printf( "CreateProcess failed (%d).\n", GetLastError() );
        return;
    }

    // Wait until child process exits.
    WaitForSingleObject( pi.hProcess, INFINITE );

    // Close process and thread handles.
    CloseHandle( pi.hProcess );
    CloseHandle( pi.hThread );
}
```

If CreateProcess succeeds, it returns a PROCESS_INFORMATION structure containing handles and identifiers for the new process and its primary thread. The thread and process handles are created with full access rights, although access can be restricted if you specify security descriptors. When you no longer need these handles, close them by using the CloseHandle function.

You can also create a process using the CreateProcessAsUser or CreateProcessWithLogonW function. This allows you to specify the security context of the user account in which the process will execute.

Creating Threads

9/16/2022 • 4 minutes to read • Edit Online

The **CreateThread** function creates a new thread for a process. The creating thread must specify the starting address of the code that the new thread is to execute. Typically, the starting address is the name of a function defined in the program code (for more information, see *ThreadProc*). This function takes a single parameter and returns a **DWORD** value. A process can have multiple threads simultaneously executing the same function.

The following is a simple example that demonstrates how to create a new thread that executes the locally defined function, `MyThreadFunction`.

The calling thread uses the **WaitForMultipleObjects** function to persist until all worker threads have terminated. The calling thread blocks while it is waiting; to continue processing, a calling thread would use **WaitForSingleObject** and wait for each worker thread to signal its wait object. Note that if you were to close the handle to a worker thread before it terminated, this does not terminate the worker thread. However, the handle will be unavailable for use in subsequent function calls.

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

#define MAX_THREADS 3
#define BUF_SIZE 255

DWORD WINAPI MyThreadFunction( LPVOID lpParam );
void ErrorHandler(LPTSTR lpszFunction);

// Sample custom data structure for threads to use.
// This is passed by void pointer so it can be any data type
// that can be passed using a single void pointer (LPVOID).
typedef struct MyData {
    int val1;
    int val2;
} MYDATA, *PMYDATA;


int _tmain()
{
    PMYDATA pDataArray[MAX_THREADS];
    DWORD   dwThreadIdArray[MAX_THREADS];
    HANDLE  hThreadArray[MAX_THREADS];

    // Create MAX_THREADS worker threads.

    for( int i=0; i<MAX_THREADS; i++ )
    {
        // Allocate memory for thread data.

        pDataArray[i] = (PMYDATA) HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY,
                sizeof(MYDATA));

        if( pDataArray[i] == NULL )
        {
           // If the array allocation fails, the system is out of memory
           // so there is no point in trying to print an error message.
           // Just terminate execution.
            ExitProcess(2);
        }

        // Generate unique data for each thread to work with
```

```
        // Generate unique data for each thread to work with.

        pDataArray[i]->val1 = i;
        pDataArray[i]->val2 = i+100;

        // Create the thread to begin execution on its own.

        hThreadArray[i] = CreateThread(
            NULL,                   // default security attributes
            0,                      // use default stack size
            MyThreadFunction,       // thread function name
            pDataArray[i],          // argument to thread function
            0,                      // use default creation flags
            &dwThreadIdArray[i]);   // returns the thread identifier


        // Check the return value for success.
        // If CreateThread fails, terminate execution.
        // This will automatically clean up threads and memory.

        if (hThreadArray[i] == NULL)
        {
           ErrorHandler(TEXT("CreateThread"));
           ExitProcess(3);
        }
    } // End of main thread creation loop.

    // Wait until all threads have terminated.

    WaitForMultipleObjects(MAX_THREADS, hThreadArray, TRUE, INFINITE);

    // Close all thread handles and free memory allocations.

    for(int i=0; i<MAX_THREADS; i++)
    {
        CloseHandle(hThreadArray[i]);
        if(pDataArray[i] != NULL)
        {
            HeapFree(GetProcessHeap(), 0, pDataArray[i]);
            pDataArray[i] = NULL;    // Ensure address is not reused.
        }
    }

    return 0;
}


DWORD WINAPI MyThreadFunction( LPVOID lpParam )
{
    HANDLE hStdout;
    PMYDATA pDataArray;

    TCHAR msgBuf[BUF_SIZE];
    size_t cchStringSize;
    DWORD dwChars;

    // Make sure there is a console to receive output results.

    hStdout = GetStdHandle(STD_OUTPUT_HANDLE);
    if( hStdout == INVALID_HANDLE_VALUE )
        return 1;

    // Cast the parameter to the correct data type.
    // The pointer is known to be valid because
    // it was checked for NULL before the thread was created.

    pDataArray = (PMYDATA)lpParam;

    // Print the parameter values using thread-safe functions.
```

```
    StringCchPrintf(msgBuf, BUF_SIZE, TEXT("Parameters = %d, %d\n"),
        pDataArray->val1, pDataArray->val2);
    StringCchLength(msgBuf, BUF_SIZE, &cchStringSize);
    WriteConsole(hStdout, msgBuf, (DWORD)cchStringSize, &dwChars, NULL);

    return 0;
}



void ErrorHandler(LPTSTR lpszFunction)
{
    // Retrieve the system error message for the last-error code.

    LPVOID lpMsgBuf;
    LPVOID lpDisplayBuf;
    DWORD dw = GetLastError();

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

    // Display the error message.

    lpDisplayBuf = (LPVOID)LocalAlloc(LMEM_ZEROINIT,
        (lstrlen((LPCTSTR) lpMsgBuf) + lstrlen((LPCTSTR) lpszFunction) + 40) * sizeof(TCHAR));
    StringCchPrintf((LPTSTR)lpDisplayBuf,
        LocalSize(lpDisplayBuf) / sizeof(TCHAR),
        TEXT("%s failed with error %d: %s"),
        lpszFunction, dw, lpMsgBuf);
    MessageBox(NULL, (LPCTSTR) lpDisplayBuf, TEXT("Error"), MB_OK);

    // Free error-handling buffer allocations.

    LocalFree(lpMsgBuf);
    LocalFree(lpDisplayBuf);
}
```

The `MyThreadFunction` function avoids the use of the C run-time library (CRT), as many of its functions are not thread-safe, particularly if you are not using the multithreaded CRT. If you would like to use the CRT in a `ThreadProc` function, use the **_beginthreadex** function instead.

It is risky to pass the address of a local variable if the creating thread exits before the new thread, because the pointer becomes invalid. Instead, either pass a pointer to dynamically allocated memory or make the creating thread wait for the new thread to terminate. Data can also be passed from the creating thread to the new thread using global variables. With global variables, it is usually necessary to synchronize access by multiple threads. For more information about synchronization, see Synchronizing Execution of Multiple Threads.

The creating thread can use the arguments to **CreateThread** to specify the following:

- The security attributes for the handle to the new thread. These security attributes include an inheritance flag that determines whether the handle can be inherited by child processes. The security attributes also include a security descriptor, which the system uses to perform access checks on all subsequent uses of the thread's handle before access is granted.
- The initial stack size of the new thread. The thread's stack is allocated automatically in the memory space of the process; the system increases the stack as needed and frees it when the thread terminates. For more information, see Thread Stack Size.

- A creation flag that enables you to create the thread in a suspended state. When suspended, the thread does not run until the **ResumeThread** function is called.

You can also create a thread by calling the **CreateRemoteThread** function. This function is used by debugger processes to create a thread that runs in the address space of the process being debugged.

Related topics

Terminating a Thread

Creating a Child Process with Redirected Input and Output

9/16/2022 • 5 minutes to read • Edit Online

The example in this topic demonstrates how to create a child process using the **CreateProcess** function from a console process. It also demonstrates a technique for using anonymous pipes to redirect the child process's standard input and output handles. Note that named pipes can also be used to redirect process I/O.

The **CreatePipe** function uses the **SECURITY_ATTRIBUTES** structure to create inheritable handles to the read and write ends of two pipes. The read end of one pipe serves as standard input for the child process, and the write end of the other pipe is the standard output for the child process. These pipe handles are specified in the **STARTUPINFO** structure, which makes them the standard handles inherited by the child process.

The parent process uses the opposite ends of these two pipes to write to the child process's input and read from the child process's output. As specified in the **SECURITY_ATTRIBUTES** structure, these handles are also inheritable. However, these handles must not be inherited. Therefore, before creating the child process, the parent process uses the **SetHandleInformation** function to ensure that the write handle for the child process's standard input and the read handle for the child process's standard output cannot be inherited. For more information, see Pipes.

The following is the code for the parent process. It takes a single command-line argument: the name of a text file.

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

#define BUFSIZE 4096

HANDLE g_hChildStd_IN_Rd = NULL;
HANDLE g_hChildStd_IN_Wr = NULL;
HANDLE g_hChildStd_OUT_Rd = NULL;
HANDLE g_hChildStd_OUT_Wr = NULL;

HANDLE g_hInputFile = NULL;

void CreateChildProcess(void);
void WriteToPipe(void);
void ReadFromPipe(void);
void ErrorExit(PTSTR);

int _tmain(int argc, TCHAR *argv[])
{
   SECURITY_ATTRIBUTES saAttr;

   printf("\n->Start of parent execution.\n");

// Set the bInheritHandle flag so pipe handles are inherited.

   saAttr.nLength = sizeof(SECURITY_ATTRIBUTES);
   saAttr.bInheritHandle = TRUE;
   saAttr.lpSecurityDescriptor = NULL;

// Create a pipe for the child process's STDOUT.

   if ( ! CreatePipe(&g_hChildStd_OUT_Rd, &g_hChildStd_OUT_Wr, &saAttr, 0) )
```

```
      ErrorExit(TEXT("StdoutRd CreatePipe"));

// Ensure the read handle to the pipe for STDOUT is not inherited.

   if ( ! SetHandleInformation(g_hChildStd_OUT_Rd, HANDLE_FLAG_INHERIT, 0) )
      ErrorExit(TEXT("Stdout SetHandleInformation"));

// Create a pipe for the child process's STDIN.

   if (! CreatePipe(&g_hChildStd_IN_Rd, &g_hChildStd_IN_Wr, &saAttr, 0))
      ErrorExit(TEXT("Stdin CreatePipe"));

// Ensure the write handle to the pipe for STDIN is not inherited.

   if ( ! SetHandleInformation(g_hChildStd_IN_Wr, HANDLE_FLAG_INHERIT, 0) )
      ErrorExit(TEXT("Stdin SetHandleInformation"));

// Create the child process.

   CreateChildProcess();

// Get a handle to an input file for the parent.
// This example assumes a plain text file and uses string output to verify data flow.

   if (argc == 1)
      ErrorExit(TEXT("Please specify an input file.\n"));

   g_hInputFile = CreateFile(
       argv[1],
       GENERIC_READ,
       0,
       NULL,
       OPEN_EXISTING,
       FILE_ATTRIBUTE_READONLY,
       NULL);

   if ( g_hInputFile == INVALID_HANDLE_VALUE )
      ErrorExit(TEXT("CreateFile"));

// Write to the pipe that is the standard input for a child process.
// Data is written to the pipe's buffers, so it is not necessary to wait
// until the child process is running before writing data.

   WriteToPipe();
   printf( "\n->Contents of %S written to child STDIN pipe.\n", argv[1]);

// Read from pipe that is the standard output for child process.

   printf( "\n->Contents of child process STDOUT:\n\n");
   ReadFromPipe();

   printf("\n->End of parent execution.\n");

// The remaining open handles are cleaned up when this process terminates.
// To avoid resource leaks in a larger application, close handles explicitly.

   return 0;
}

void CreateChildProcess()
// Create a child process that uses the previously created pipes for STDIN and STDOUT.
{
   TCHAR szCmdline[]=TEXT("child");
   PROCESS_INFORMATION piProcInfo;
   STARTUPINFO siStartInfo;
   BOOL bSuccess = FALSE;

// Set up members of the PROCESS_INFORMATION structure.
```

```
   ZeroMemory( &piProcInfo, sizeof(PROCESS_INFORMATION) );

// Set up members of the STARTUPINFO structure.
// This structure specifies the STDIN and STDOUT handles for redirection.

   ZeroMemory( &siStartInfo, sizeof(STARTUPINFO) );
   siStartInfo.cb = sizeof(STARTUPINFO);
   siStartInfo.hStdError = g_hChildStd_OUT_Wr;
   siStartInfo.hStdOutput = g_hChildStd_OUT_Wr;
   siStartInfo.hStdInput = g_hChildStd_IN_Rd;
   siStartInfo.dwFlags |= STARTF_USESTDHANDLES;

// Create the child process.

   bSuccess = CreateProcess(NULL,
      szCmdline,     // command line
      NULL,          // process security attributes
      NULL,          // primary thread security attributes
      TRUE,          // handles are inherited
      0,             // creation flags
      NULL,          // use parent's environment
      NULL,          // use parent's current directory
      &siStartInfo,  // STARTUPINFO pointer
      &piProcInfo);  // receives PROCESS_INFORMATION

   // If an error occurs, exit the application.
   if ( ! bSuccess )
      ErrorExit(TEXT("CreateProcess"));
   else
   {
      // Close handles to the child process and its primary thread.
      // Some applications might keep these handles to monitor the status
      // of the child process, for example.

      CloseHandle(piProcInfo.hProcess);
      CloseHandle(piProcInfo.hThread);

      // Close handles to the stdin and stdout pipes no longer needed by the child process.
      // If they are not explicitly closed, there is no way to recognize that the child process has ended.

      CloseHandle(g_hChildStd_OUT_Wr);
      CloseHandle(g_hChildStd_IN_Rd);
   }
}

void WriteToPipe(void)

// Read from a file and write its contents to the pipe for the child's STDIN.
// Stop when there is no more data.
{
   DWORD dwRead, dwWritten;
   CHAR chBuf[BUFSIZE];
   BOOL bSuccess = FALSE;

   for (;;)
   {
      bSuccess = ReadFile(g_hInputFile, chBuf, BUFSIZE, &dwRead, NULL);
      if ( ! bSuccess || dwRead == 0 ) break;

      bSuccess = WriteFile(g_hChildStd_IN_Wr, chBuf, dwRead, &dwWritten, NULL);
      if ( ! bSuccess ) break;
   }

// Close the pipe handle so the child process stops reading.

   if ( ! CloseHandle(g_hChildStd_IN_Wr) )
      ErrorExit(TEXT("StdInWr CloseHandle"));
}
```

```
void ReadFromPipe(void)

// Read output from the child process's pipe for STDOUT
// and write to the parent process's pipe for STDOUT.
// Stop when there is no more data.
{
   DWORD dwRead, dwWritten;
   CHAR chBuf[BUFSIZE];
   BOOL bSuccess = FALSE;
   HANDLE hParentStdOut = GetStdHandle(STD_OUTPUT_HANDLE);

   for (;;)
   {
      bSuccess = ReadFile( g_hChildStd_OUT_Rd, chBuf, BUFSIZE, &dwRead, NULL);
      if( ! bSuccess || dwRead == 0 ) break;

      bSuccess = WriteFile(hParentStdOut, chBuf,
                           dwRead, &dwWritten, NULL);
      if (! bSuccess ) break;
   }
}

void ErrorExit(PTSTR lpszFunction)

// Format a readable error message, display a message box,
// and exit from the application.
{
    LPVOID lpMsgBuf;
    LPVOID lpDisplayBuf;
    DWORD dw = GetLastError();

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

    lpDisplayBuf = (LPVOID)LocalAlloc(LMEM_ZEROINIT,
        (lstrlen((LPCTSTR)lpMsgBuf)+lstrlen((LPCTSTR)lpszFunction)+40)*sizeof(TCHAR));
    StringCchPrintf((LPTSTR)lpDisplayBuf,
        LocalSize(lpDisplayBuf) / sizeof(TCHAR),
        TEXT("%s failed with error %d: %s"),
        lpszFunction, dw, lpMsgBuf);
    MessageBox(NULL, (LPCTSTR)lpDisplayBuf, TEXT("Error"), MB_OK);

    LocalFree(lpMsgBuf);
    LocalFree(lpDisplayBuf);
    ExitProcess(1);
}
```

The following is the code for the child process. It uses the inherited handles for STDIN and STDOUT to access the pipe created by the parent. The parent process reads from its input file and writes the information to a pipe. The child receives text through the pipe using STDIN and writes to the pipe using STDOUT. The parent reads from the read end of the pipe and displays the information to its STDOUT.

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

#define BUFSIZE 4096

int main(void)
{
   CHAR chBuf[BUFSIZE];
   DWORD dwRead, dwWritten;
   HANDLE hStdin, hStdout;
   BOOL bSuccess;

   hStdout = GetStdHandle(STD_OUTPUT_HANDLE);
   hStdin = GetStdHandle(STD_INPUT_HANDLE);
   if (
       (hStdout == INVALID_HANDLE_VALUE) ||
       (hStdin == INVALID_HANDLE_VALUE)
      )
      ExitProcess(1);

   // Send something to this process's stdout using printf.
   printf("\n ** This is a message from the child process. ** \n");

   // This simple algorithm uses the existence of the pipes to control execution.
   // It relies on the pipe buffers to ensure that no data is lost.
   // Larger applications would use more advanced process control.

   for (;;)
   {
   // Read from standard input and stop on error or no data.
      bSuccess = ReadFile(hStdin, chBuf, BUFSIZE, &dwRead, NULL);

      if (! bSuccess || dwRead == 0)
         break;

   // Write to standard output and stop on error.
      bSuccess = WriteFile(hStdout, chBuf, dwRead, &dwWritten, NULL);

      if (! bSuccess)
         break;
   }
   return 0;
}
```

Isolated User Mode (IUM) Processes

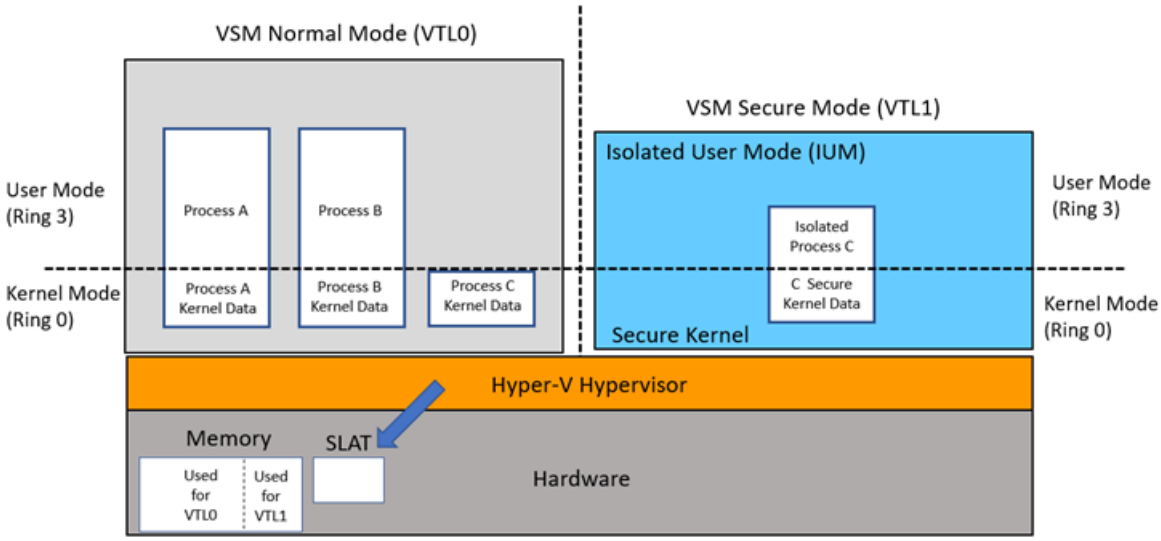
9/16/2022 • 3 minutes to read • Edit Online

Windows 10 introduced a new security feature named Virtual Secure Mode (VSM). VSM leverages the Hyper-V Hypervisor and Second Level Address Translation (SLAT) to create a set of modes called Virtual Trust Levels (VTLs). This new software architecture creates a security boundary to prevent processes running in one VTL from accessing the memory of another VTL. The benefit of this isolation includes additional mitigation from kernel exploits while protecting assets such as password hashes and Kerberos keys.

Diagram 1 depicts the traditional model of Kernel mode and User mode code running in CPU ring 0 and ring 3, respectively. In this new model, the code running in the traditional model executes in VTL0 and it cannot access the higher privileged VTL1, where the Secure Kernel and Isolated User Mode (IUM) execute code. The VTLs are hierarchal meaning any code running in VTL1 is more privileged than code running in VTL0.

The VTL isolation is created by the Hyper-V Hypervisor which assigns memory at boot time using Second Level Address Translation (SLAT). It continues this dynamically as the system runs, protecting memory the Secure Kernel specifies needing protection from VTL0 because it will be used to contain secrets. As separate blocks of memory are allocated for the two VTLs, a secure runtime environment is created for VTL1 by assigning exclusive memory blocks to VTL1 and VTL0 with the appropriate access permissions.

Diagram 1 - IUM Architecture



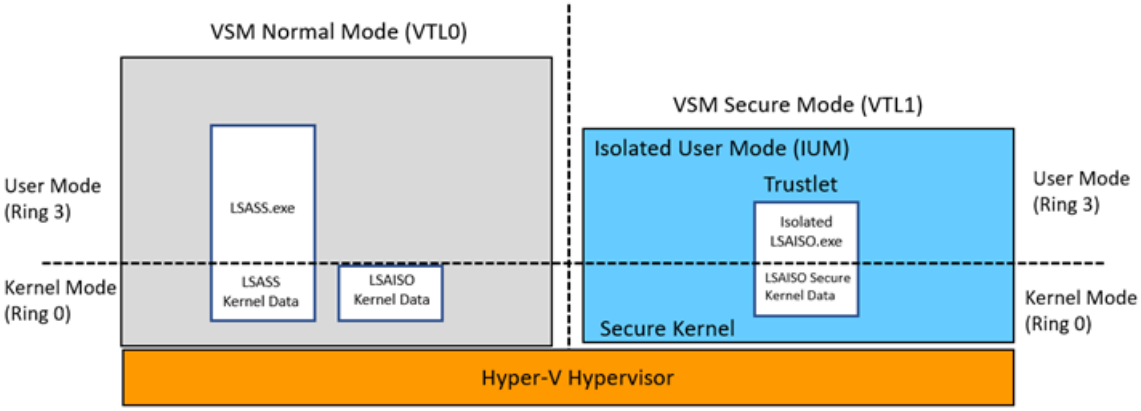
Trustlets

Trustlets (also known as trusted processes, secure processes, or IUM processes) are programs running as IUM processes in VSM. They complete system calls by marshalling them over to the Windows kernel running in VTL0 ring 0. VSM creates a small execution environment that includes the small Secure Kernel executing in VTL1 (isolated from the kernel and drivers running in VTL0). The clear security benefit is isolation of trustlet user mode pages in VTL1 from drivers running in the VTL0 kernel. Even if kernel mode of VTL0 is compromised by malware, it will not have access to the IUM process pages.

With VSM enabled, the Local Security Authority (LSASS) environment runs as a trustlet. LSASS manages the local system policy, user authentication, and auditing while handling sensitive security data such as password hashes and Kerberos keys. To leverage the security benefits of VSM, a trustlet named LSAISO.exe (LSA Isolated) runs in VTL1 and communicates with LSASS.exe running in VTL0 through an RPC channel. The LSAISO secrets

are encrypted before sending them over to LSASS running in VSM Normal Mode and the pages of LSAISO are protected from malicious code running in VTL0.

Diagram 2 – LSASS Trustlet Design



Isolated User Mode (IUM) Implications

It is not possible to attach to an IUM process, inhibiting the ability to debug VTL1 code. This includes post mortem debugging of memory dumps and attaching the Debugging Tools for live debugging. It also includes attempts by privileged accounts or kernel drivers to load a DLL into an IUM process, to inject a thread, or deliver a user-mode APC. Such attempts may result in destabilization of the entire system. Windows APIs that would compromise the security of a Trustlet may fail in unexpected ways. For example, loading a DLL into a Trustlet will make it available in VTL0 but not VTL1. QueueUserApc may silently fail if the target thread is in a Trustlet. Other APIs, such as CreateRemoteThread, VirtualAllocEx, and Read/WriteProcessMemory will also not work as expected when used against Trustlets.

Use the sample code below to prevent calling any functions which attempt to attach or inject code into an IUM process. This includes kernel drivers that queue APCs for execution of code in a trustlet.

Remarks

If the return status of IsSecureProcess is success, examine the SecureProcess _Out_ parameter to determine if the process is an IUM process. IUM processes are marked by the system to be “Secure Processes”. A Boolean result of TRUE means the target process is of type IUM.

```
NTSTATUS
IsSecureProcess(
   _In_ HANDLE ProcessHandle,
   _Out_ BOOLEAN *SecureProcess
   )
{
   NTSTATUS status;

       // definition included in ntddk.h
   PROCESS_EXTENDED_BASIC_INFORMATION extendedInfo = {0};

   PAGED_CODE();

   extendedInfo.Size = sizeof(extendedInfo);

   // Query for the process information
   status = ZwQueryInformationProcess(
                  ProcessHandle, ProcessBasicInformation, &extendedInfo,
                  sizeof(extendedInfo), NULL);

   if (NT_SUCCESS(status)) {
      *SecureProcess = (BOOLEAN)(extendedInfo.IsSecureProcess != 0);
   }

          return status;
}
```

The WDK for Windows 10, "Windows Driver Kit - Windows 10.0.15063.0", contains the required definition of the PROCESS_EXTENDED_BASIC_INFORMATION structure. The updated version of the structure is defined in ntddk.h with the new IsSecureProcess field.

```
typedef struct _PROCESS_EXTENDED_BASIC_INFORMATION {
    SIZE_T Size;    // Ignored as input, written with structure size on output
    PROCESS_BASIC_INFORMATION BasicInfo;
    union {
        ULONG Flags;
        struct {
            ULONG IsProtectedProcess : 1;
            ULONG IsWow64Process : 1;
            ULONG IsProcessDeleting : 1;
            ULONG IsCrossSessionCreate : 1;
            ULONG IsFrozen : 1;
            ULONG IsBackground : 1;
            ULONG IsStronglyNamed : 1;
            ULONG IsSecureProcess : 1;
            ULONG IsSubsystemProcess : 1;
            ULONG SpareBits : 23;
        } DUMMYSTRUCTNAME;
    } DUMMYUNIONNAME;
} PROCESS_EXTENDED_BASIC_INFORMATION, *PPROCESS_EXTENDED_BASIC_INFORMATION;
```

Changing Environment Variables

9/16/2022 • 3 minutes to read • Edit Online

Each process has an environment block associated with it. The environment block consists of a null-terminated block of null-terminated strings (meaning there are two null bytes at the end of the block), where each string is in the form:

name=*value*

All strings in the environment block must be sorted alphabetically by name. The sort is case-insensitive, Unicode order, without regard to locale. Because the equal sign is a separator, it must not be used in the name of an environment variable.

Example 1

By default, a child process inherits a copy of the environment block of the parent process. The following example demonstrates how to create a new environment block to pass to a child process using **CreateProcess**.

This example uses the code in example three as the child process, Ex3.exe.

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

#define BUFSIZE 4096

int _tmain()
{
    TCHAR chNewEnv[BUFSIZE];
    LPTSTR lpszCurrentVariable;
    DWORD dwFlags=0;
    TCHAR szAppName[]=TEXT("ex3.exe");
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    BOOL fSuccess;

    // Copy environment strings into an environment block.

    lpszCurrentVariable = (LPTSTR) chNewEnv;
    if (FAILED(StringCchCopy(lpszCurrentVariable, BUFSIZE, TEXT("MySetting=A"))))
    {
        printf("String copy failed\n");
        return FALSE;
    }

    lpszCurrentVariable += lstrlen(lpszCurrentVariable) + 1;
    if (FAILED(StringCchCopy(lpszCurrentVariable, BUFSIZE, TEXT("MyVersion=2"))))
    {
        printf("String copy failed\n");
        return FALSE;
    }

    // Terminate the block with a NULL byte.

    lpszCurrentVariable += lstrlen(lpszCurrentVariable) + 1;
    *lpszCurrentVariable = (TCHAR)0;

    // Create the child process, specifying a new environment block.

    SecureZeroMemory(&si, sizeof(STARTUPINFO));
    si.cb = sizeof(STARTUPINFO);

#ifdef UNICODE
    dwFlags = CREATE_UNICODE_ENVIRONMENT;
#endif

    fSuccess = CreateProcess(szAppName, NULL, NULL, NULL, TRUE, dwFlags,
        (LPVOID) chNewEnv,   // new environment block
        NULL, &si, &pi);

    if (! fSuccess)
    {
        printf("CreateProcess failed (%d)\n", GetLastError());
        return FALSE;
    }
    WaitForSingleObject(pi.hProcess, INFINITE);
    return TRUE;
}
```

Example 2

Altering the environment variables of a child process during process creation is the only way one process can directly change the environment variables of another process. A process can never directly change the environment variables of another process that is not a child of that process.

If you want the child process to inherit most of the parent's environment with only a few changes, retrieve the current values using **GetEnvironmentVariable**, save these values, create an updated block for the child process to inherit, create the child process, and then restore the saved values using **SetEnvironmentVariable**, as shown in the following example.

This example uses the code in example three as the child process, Ex3.exe.

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

#define BUFSIZE 4096
#define VARNAME TEXT("MyVariable")

int _tmain()
{
    DWORD dwRet, dwErr;
    LPTSTR pszOldVal;
    TCHAR szAppName[]=TEXT("ex3.exe");
    DWORD dwFlags=0;
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    BOOL fExist, fSuccess;

    // Retrieves the current value of the variable if it exists.
    // Sets the variable to a new value, creates a child process,
    // then uses SetEnvironmentVariable to restore the original
    // value or delete it if it did not exist previously.

    pszOldVal = (LPTSTR) malloc(BUFSIZE*sizeof(TCHAR));
    if(NULL == pszOldVal)
    {
        printf("Out of memory\n");
        return FALSE;
    }

    dwRet = GetEnvironmentVariable(VARNAME, pszOldVal, BUFSIZE);

    if(0 == dwRet)
    {
        dwErr = GetLastError();
        if( ERROR_ENVVAR_NOT_FOUND == dwErr )
        {
            printf("Environment variable does not exist.\n");
            fExist=FALSE;
        }
    }
    else if(BUFSIZE < dwRet)
    {
        pszOldVal = (LPTSTR) realloc(pszOldVal, dwRet*sizeof(TCHAR));
        if(NULL == pszOldVal)
        {
            printf("Out of memory\n");
            return FALSE;
        }
        dwRet = GetEnvironmentVariable(VARNAME, pszOldVal, dwRet);
        if(!dwRet)
        {
            printf("GetEnvironmentVariable failed (%d)\n", GetLastError());
            return FALSE;
        }
        else fExist=TRUE;
    }
    else fExist=TRUE;

    // Set a value for the child process to inherit.
```

```
    if (! SetEnvironmentVariable(VARNAME, TEXT("Test")))
    {
        printf("SetEnvironmentVariable failed (%d)\n", GetLastError());
        return FALSE;
    }

    // Create a child process.

    SecureZeroMemory(&si, sizeof(STARTUPINFO));
    si.cb = sizeof(STARTUPINFO);

#ifdef UNICODE
    dwFlags = CREATE_UNICODE_ENVIRONMENT;
#endif

    fSuccess = CreateProcess(szAppName, NULL, NULL, NULL, TRUE, dwFlags,
        NULL,     // inherit parent's environment
        NULL, &si, &pi);
    if (! fSuccess)
    {
        printf("CreateProcess failed (%d)\n", GetLastError());
    }
    WaitForSingleObject(pi.hProcess, INFINITE);

    // Restore the original environment variable.

    if(fExist)
    {
        if (! SetEnvironmentVariable(VARNAME, pszOldVal))
        {
            printf("SetEnvironmentVariable failed (%d)\n", GetLastError());
            return FALSE;
        }
    }
    else SetEnvironmentVariable(VARNAME, NULL);

    free(pszOldVal);

    return fSuccess;
}
```

Example 3

The following example retrieves the process's environment block using **GetEnvironmentStrings** and prints the contents to the console.

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

int _tmain()
{
    LPTSTR lpszVariable;
    LPTCH lpvEnv;

    // Get a pointer to the environment block.

    lpvEnv = GetEnvironmentStrings();

    // If the returned pointer is NULL, exit.
    if (lpvEnv == NULL)
    {
        printf("GetEnvironmentStrings failed (%d)\n", GetLastError());
        return 0;
    }

    // Variable strings are separated by NULL byte, and the block is
    // terminated by a NULL byte.

    lpszVariable = (LPTSTR) lpvEnv;

    while (*lpszVariable)
    {
        _tprintf(TEXT("%s\n"), lpszVariable);
        lpszVariable += lstrlen(lpszVariable) + 1;
    }
    FreeEnvironmentStrings(lpvEnv);
    return 1;
}
```

Using Thread Local Storage

9/16/2022 • 2 minutes to read • Edit Online

Thread local storage (TLS) enables multiple threads of the same process to use an index allocated by the **TlsAlloc** function to store and retrieve a value that is local to the thread. In this example, an index is allocated when the process starts. When each thread starts, it allocates a block of dynamic memory and stores a pointer to this memory in the TLS slot using the **TlsSetValue** function. The CommonFunc function uses the **TlsGetValue** function to access the data associated with the index that is local to the calling thread. Before each thread terminates, it releases its dynamic memory. Before the process terminates, it calls **TlsFree** to release the index.

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

#define THREADCOUNT 4
DWORD dwTlsIndex;

VOID ErrorExit (LPCSTR message);

VOID CommonFunc(VOID)
{
   LPVOID lpvData;

// Retrieve a data pointer for the current thread.

   lpvData = TlsGetValue(dwTlsIndex);
   if ((lpvData == 0) && (GetLastError() != ERROR_SUCCESS))
      ErrorExit("TlsGetValue error");

// Use the data stored for the current thread.

   printf("common: thread %d: lpvData=%lx\n",
      GetCurrentThreadId(), lpvData);

   Sleep(5000);
}

DWORD WINAPI ThreadFunc(VOID)
{
   LPVOID lpvData;

// Initialize the TLS index for this thread.

   lpvData = (LPVOID) LocalAlloc(LPTR, 256);
   if (! TlsSetValue(dwTlsIndex, lpvData))
      ErrorExit("TlsSetValue error");

   printf("thread %d: lpvData=%lx\n", GetCurrentThreadId(), lpvData);

   CommonFunc();

// Release the dynamic memory before the thread returns.

   lpvData = TlsGetValue(dwTlsIndex);
   if (lpvData != 0)
      LocalFree((HLOCAL) lpvData);

   return 0;
}
```

```
int main(VOID)
{
   DWORD IDThread;
   HANDLE hThread[THREADCOUNT];
   int i;

// Allocate a TLS index.

   if ((dwTlsIndex = TlsAlloc()) == TLS_OUT_OF_INDEXES)
      ErrorExit("TlsAlloc failed");

// Create multiple threads.

   for (i = 0; i < THREADCOUNT; i++)
   {
      hThread[i] = CreateThread(NULL, // default security attributes
         0,                           // use default stack size
         (LPTHREAD_START_ROUTINE) ThreadFunc, // thread function
         NULL,                    // no thread function argument
         0,                       // use default creation flags
         &IDThread);              // returns thread identifier

   // Check the return value for success.
      if (hThread[i] == NULL)
         ErrorExit("CreateThread error\n");
   }

   for (i = 0; i < THREADCOUNT; i++)
      WaitForSingleObject(hThread[i], INFINITE);

   TlsFree(dwTlsIndex);

   return 0;
}

VOID ErrorExit (LPCSTR message)
{
   fprintf(stderr, "%s\n", message);
   ExitProcess(0);
}
```

Related topics

Using Thread Local Storage in a Dynamic-Link Library

Using Fibers

9/16/2022 • 5 minutes to read • Edit Online

The CreateFiber function creates a new fiber for a thread. The creating thread must specify the starting address of the code that the new fiber is to execute. Typically, the starting address is the name of a user-supplied function. Multiple fibers can execute the same function.

The following example demonstrates how to create, schedule, and delete fibers. The fibers execute the locally defined functions ReadFiberFunc and WriteFiberFunc. This example implements a fiber-based file copy operation. When running the example, you must specify the source and destination files. Note that there are many other ways to copy file programmatically; this example exists primarily to illustrate the use of the fiber functions.

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

VOID
__stdcall
ReadFiberFunc(LPVOID lpParameter);

VOID
__stdcall
WriteFiberFunc(LPVOID lpParameter);

void DisplayFiberInfo(void);

typedef struct
{
   DWORD dwParameter;          // DWORD parameter to fiber (unused)
   DWORD dwFiberResultCode;    // GetLastError() result code
   HANDLE hFile;               // handle to operate on
   DWORD dwBytesProcessed;     // number of bytes processed
} FIBERDATASTRUCT, *PFIBERDATASTRUCT, *LPFIBERDATASTRUCT;

#define RTN_OK 0
#define RTN_USAGE 1
#define RTN_ERROR 13

#define BUFFER_SIZE 32768   // read/write buffer size
#define FIBER_COUNT 3       // max fibers (including primary)

#define PRIMARY_FIBER 0 // array index to primary fiber
#define READ_FIBER 1    // array index to read fiber
#define WRITE_FIBER 2   // array index to write fiber

LPVOID g_lpFiber[FIBER_COUNT];
LPBYTE g_lpBuffer;
DWORD g_dwBytesRead;

int __cdecl _tmain(int argc, TCHAR *argv[])
{
   LPFIBERDATASTRUCT fs;

   if (argc != 3)
   {
      printf("Usage: %s <SourceFile> <DestinationFile>\n", argv[0]);
      return RTN_USAGE;
   }
```

```
    //
    // Allocate storage for our fiber data structures
    //
    fs = (LPFIBERDATASTRUCT) HeapAlloc(
                               GetProcessHeap(), 0,
                               sizeof(FIBERDATASTRUCT) * FIBER_COUNT);

    if (fs == NULL)
    {
       printf("HeapAlloc error (%d)\n", GetLastError());
       return RTN_ERROR;
    }

    //
    // Allocate storage for the read/write buffer
    //
    g_lpBuffer = (LPBYTE)HeapAlloc(GetProcessHeap(), 0, BUFFER_SIZE);
    if (g_lpBuffer == NULL)
    {
       printf("HeapAlloc error (%d)\n", GetLastError());
       return RTN_ERROR;
    }

    //
    // Open the source file
    //
    fs[READ_FIBER].hFile = CreateFile(
                                    argv[1],
                                    GENERIC_READ,
                                    FILE_SHARE_READ,
                                    NULL,
                                    OPEN_EXISTING,
                                    FILE_FLAG_SEQUENTIAL_SCAN,
                                    NULL
                                    );

    if (fs[READ_FIBER].hFile == INVALID_HANDLE_VALUE)
    {
       printf("CreateFile error (%d)\n", GetLastError());
       return RTN_ERROR;
    }

    //
    // Open the destination file
    //
    fs[WRITE_FIBER].hFile = CreateFile(
                                     argv[2],
                                     GENERIC_WRITE,
                                     0,
                                     NULL,
                                     CREATE_NEW,
                                     FILE_FLAG_SEQUENTIAL_SCAN,
                                     NULL
                                     );

    if (fs[WRITE_FIBER].hFile == INVALID_HANDLE_VALUE)
    {
       printf("CreateFile error (%d)\n", GetLastError());
       return RTN_ERROR;
    }

    //
    // Convert thread to a fiber, to allow scheduling other fibers
    //
    g_lpFiber[PRIMARY_FIBER]=ConvertThreadToFiber(&fs[PRIMARY_FIBER]);

    if (g_lpFiber[PRIMARY_FIBER] == NULL)
    {
       printf("ConvertThreadToFiber error (%d)\n", GetLastError());
```

```
   return RTN_ERROR;
}

//
// Initialize the primary fiber data structure.  We don't use
// the primary fiber data structure for anything in this sample.
//
fs[PRIMARY_FIBER].dwParameter = 0;
fs[PRIMARY_FIBER].dwFiberResultCode = 0;
fs[PRIMARY_FIBER].hFile = INVALID_HANDLE_VALUE;

//
// Create the Read fiber
//
g_lpFiber[READ_FIBER]=CreateFiber(0,ReadFiberFunc,&fs[READ_FIBER]);

if (g_lpFiber[READ_FIBER] == NULL)
{
   printf("CreateFiber error (%d)\n", GetLastError());
   return RTN_ERROR;
}

fs[READ_FIBER].dwParameter = 0x12345678;

//
// Create the Write fiber
//
g_lpFiber[WRITE_FIBER]=CreateFiber(0,WriteFiberFunc,&fs[WRITE_FIBER]);

if (g_lpFiber[WRITE_FIBER] == NULL)
{
   printf("CreateFiber error (%d)\n", GetLastError());
   return RTN_ERROR;
}

fs[WRITE_FIBER].dwParameter = 0x54545454;

//
// Switch to the read fiber
//
SwitchToFiber(g_lpFiber[READ_FIBER]);

//
// We have been scheduled again. Display results from the
// read/write fibers
//
printf("ReadFiber: result code is %lu, %lu bytes processed\n",
fs[READ_FIBER].dwFiberResultCode, fs[READ_FIBER].dwBytesProcessed);

printf("WriteFiber: result code is %lu, %lu bytes processed\n",
fs[WRITE_FIBER].dwFiberResultCode, fs[WRITE_FIBER].dwBytesProcessed);

//
// Delete the fibers
//
DeleteFiber(g_lpFiber[READ_FIBER]);
DeleteFiber(g_lpFiber[WRITE_FIBER]);

//
// Close handles
//
CloseHandle(fs[READ_FIBER].hFile);
CloseHandle(fs[WRITE_FIBER].hFile);

//
// Free allocated memory
//
HeapFree(GetProcessHeap(), 0, g_lpBuffer);
HeapFree(GetProcessHeap(), 0, fs);
```

```
   return RTN_OK;
}

VOID
__stdcall
ReadFiberFunc(
             LPVOID lpParameter
             )
{
   LPFIBERDATASTRUCT fds = (LPFIBERDATASTRUCT)lpParameter;

   //
   // If this fiber was passed NULL for fiber data, just return,
   // causing the current thread to exit
   //
   if (fds == NULL)
   {
      printf("Passed NULL fiber data; exiting current thread.\n");
      return;
   }

   //
   // Display some information pertaining to the current fiber
   //
   DisplayFiberInfo();

   fds->dwBytesProcessed = 0;

   while (1)
   {
      //
      // Read data from file specified in the READ_FIBER structure
      //
      if (!ReadFile(fds->hFile, g_lpBuffer, BUFFER_SIZE,
         &g_dwBytesRead, NULL))
      {
         break;
      }

      //
      // if we reached EOF, break
      //
      if (g_dwBytesRead == 0) break;

      //
      // Update number of bytes processed in the fiber data structure
      //
      fds->dwBytesProcessed += g_dwBytesRead;

      //
      // Switch to the write fiber
      //
      SwitchToFiber(g_lpFiber[WRITE_FIBER]);
   } // while

   //
   // Update the fiber result code
   //
   fds->dwFiberResultCode = GetLastError();

   //
   // Switch back to the primary fiber
   //
   SwitchToFiber(g_lpFiber[PRIMARY_FIBER]);
}

VOID
  stdcall
```

```
WriteFiberFunc(
              LPVOID lpParameter
              )
{
   LPFIBERDATASTRUCT fds = (LPFIBERDATASTRUCT)lpParameter;
   DWORD dwBytesWritten;

   //
   // If this fiber was passed NULL for fiber data, just return,
   // causing the current thread to exit
   //
   if (fds == NULL)
   {
      printf("Passed NULL fiber data; exiting current thread.\n");
      return;
   }

   //
   // Display some information pertaining to the current fiber
   //
   DisplayFiberInfo();

   //
   // Assume all writes succeeded.  If a write fails, the fiber
   // result code will be updated to reflect the reason for failure
   //
   fds->dwBytesProcessed = 0;
   fds->dwFiberResultCode = ERROR_SUCCESS;

   while (1)
   {
      //
      // Write data to the file specified in the WRITE_FIBER structure
      //
      if (!WriteFile(fds->hFile, g_lpBuffer, g_dwBytesRead,
         &dwBytesWritten, NULL))
      {
         //
         // If an error occurred writing, break
         //
         break;
      }

      //
      // Update number of bytes processed in the fiber data structure
      //
      fds->dwBytesProcessed += dwBytesWritten;

      //
      // Switch back to the read fiber
      //
      SwitchToFiber(g_lpFiber[READ_FIBER]);
   }  // while

   //
   // If an error occurred, update the fiber result code...
   //
   fds->dwFiberResultCode = GetLastError();

   //
   // ...and switch to the primary fiber
   //
   SwitchToFiber(g_lpFiber[PRIMARY_FIBER]);
}

void
DisplayFiberInfo(
                void
                )
```

```
{
   LPFIBERDATASTRUCT fds = (LPFIBERDATASTRUCT)GetFiberData();
   LPVOID lpCurrentFiber = GetCurrentFiber();

   //
   // Determine which fiber is executing, based on the fiber address
   //
   if (lpCurrentFiber == g_lpFiber[READ_FIBER])
      printf("Read fiber entered");
   else
   {
      if (lpCurrentFiber == g_lpFiber[WRITE_FIBER])
         printf("Write fiber entered");
      else
      {
         if (lpCurrentFiber == g_lpFiber[PRIMARY_FIBER])
            printf("Primary fiber entered");
         else
            printf("Unknown fiber entered");
      }
   }

   //
   // Display dwParameter from the current fiber data structure
   //
   printf(" (dwParameter is 0x%lx)\n", fds->dwParameter);
}
```

This example makes use of a fiber data structure which is used to determine the behavior and state of the fiber. One data structure exists for each fiber; the pointer to the data structure is passed to the fiber at fiber creation time using the parameter of the *FiberProc* function.

The calling thread calls the **ConvertThreadToFiber** function, which enables fibers to be scheduled by the caller. This also allows the fiber to be scheduled by another fiber. Next, the thread creates two additional fibers, one that performs read operations against a specified file, and another that performs the write operations against a specified file.

The primary fiber calls the **SwitchToFiber** function to schedule the read fiber. After a successful read, the read fiber schedules the write fiber. After a successful write in the write fiber, the write fiber schedules the read fiber. When the read/write cycle has completed, the primary fiber is scheduled, which results in the display of the read/write status. If an error occurs during the read or write operations, the primary fiber is scheduled and example displays the status of the operation.

Prior to process termination, the process frees the fibers using the **DeleteFiber** function, closes the file handles, and frees the allocated memory.

Related topics

Fibers

Using the Thread Pool Functions

9/16/2022 • 4 minutes to read • Edit Online

This example creates a custom thread pool, creates a work item and a thread pool timer, and associates them with a cleanup group. The pool consists of one persistent thread. It demonstrates the use of the following thread pool functions:

- CloseThreadpool
- CloseThreadpoolCleanupGroup
- CloseThreadpoolCleanupGroupMembers
- CloseThreadpoolWait
- CreateThreadpool
- CreateThreadpoolCleanupGroup
- CreateThreadpoolTimer
- CreateThreadpoolWait
- CreateThreadpoolWork
- InitializeThreadpoolEnvironment
- SetThreadpoolCallbackCleanupGroup
- SetThreadpoolCallbackPool
- SetThreadpoolThreadMaximum
- SetThreadpoolThreadMinimum
- SetThreadpoolTimer
- SetThreadpoolWait
- SubmitThreadpoolWork
- WaitForThreadpoolWaitCallbacks

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

//
// Thread pool wait callback function template
//
VOID
CALLBACK
MyWaitCallback(
    PTP_CALLBACK_INSTANCE Instance,
    PVOID                 Parameter,
    PTP_WAIT              Wait,
    TP_WAIT_RESULT        WaitResult
    )
{
    // Instance, Parameter, Wait, and WaitResult not used in this example.
    UNREFERENCED_PARAMETER(Instance);
    UNREFERENCED_PARAMETER(Parameter);
    UNREFERENCED_PARAMETER(Wait);
    UNREFERENCED_PARAMETER(WaitResult);

    //
    // Do something when the wait is over.
    //
    _tprintf(_T("MyWaitCallback: wait is over.\n"));
}
```

```
//
// Thread pool timer callback function template
//
VOID
CALLBACK
MyTimerCallback(
    PTP_CALLBACK_INSTANCE Instance,
    PVOID                 Parameter,
    PTP_TIMER             Timer
    )
{
    // Instance, Parameter, and Timer not used in this example.
    UNREFERENCED_PARAMETER(Instance);
    UNREFERENCED_PARAMETER(Parameter);
    UNREFERENCED_PARAMETER(Timer);

    //
    // Do something when the timer fires.
    //
    _tprintf(_T("MyTimerCallback: timer has fired.\n"));

}


//
// This is the thread pool work callback function.
//
VOID
CALLBACK
MyWorkCallback(
    PTP_CALLBACK_INSTANCE Instance,
    PVOID                 Parameter,
    PTP_WORK              Work
    )
{
    // Instance, Parameter, and Work not used in this example.
    UNREFERENCED_PARAMETER(Instance);
    UNREFERENCED_PARAMETER(Parameter);
    UNREFERENCED_PARAMETER(Work);

    BOOL bRet = FALSE;

    //
    // Do something when the work callback is invoked.
    //
     {
         _tprintf(_T("MyWorkCallback: Task performed.\n"));
     }

    return;
}

VOID
DemoCleanupPersistentWorkTimer()
{
    BOOL bRet = FALSE;
    PTP_WORK work = NULL;
    PTP_TIMER timer = NULL;
    PTP_POOL pool = NULL;
    PTP_WORK_CALLBACK workcallback = MyWorkCallback;
    PTP_TIMER_CALLBACK timercallback = MyTimerCallback;
    TP_CALLBACK_ENVIRON CallBackEnviron;
    PTP_CLEANUP_GROUP cleanupgroup = NULL;
    FILETIME FileDueTime;
    ULARGE_INTEGER ulDueTime;
    UINT rollback = 0;
```

```
    InitializeThreadpoolEnvironment(&CallBackEnviron);

    //
    // Create a custom, dedicated thread pool.
    //
    pool = CreateThreadpool(NULL);

    if (NULL == pool) {
        _tprintf(_T("CreateThreadpool failed. LastError: %u\n"),
                    GetLastError());
        goto main_cleanup;
    }

    rollback = 1; // pool creation succeeded

    //
    // The thread pool is made persistent simply by setting
    // both the minimum and maximum threads to 1.
    //
    SetThreadpoolThreadMaximum(pool, 1);

    bRet = SetThreadpoolThreadMinimum(pool, 1);

    if (FALSE == bRet) {
        _tprintf(_T("SetThreadpoolThreadMinimum failed. LastError: %u\n"),
                    GetLastError());
        goto main_cleanup;
    }

    //
    // Create a cleanup group for this thread pool.
    //
    cleanupgroup = CreateThreadpoolCleanupGroup();

    if (NULL == cleanupgroup) {
        _tprintf(_T("CreateThreadpoolCleanupGroup failed. LastError: %u\n"),
                    GetLastError());
        goto main_cleanup;
    }

    rollback = 2;  // Cleanup group creation succeeded

    //
    // Associate the callback environment with our thread pool.
    //
    SetThreadpoolCallbackPool(&CallBackEnviron, pool);

    //
    // Associate the cleanup group with our thread pool.
    // Objects created with the same callback environment
    // as the cleanup group become members of the cleanup group.
    //
    SetThreadpoolCallbackCleanupGroup(&CallBackEnviron,
                                      cleanupgroup,
                                      NULL);

    //
    // Create work with the callback environment.
    //
    work = CreateThreadpoolWork(workcallback,
                                NULL,
                                &CallBackEnviron);

    if (NULL == work) {
        _tprintf(_T("CreateThreadpoolWork failed. LastError: %u\n"),
                    GetLastError());
        goto main_cleanup;
    }
```

```
    rollback = 3;  // Creation of work succeeded

    //
    // Submit the work to the pool. Because this was a pre-allocated
    // work item (using CreateThreadpoolWork), it is guaranteed to execute.
    //
    SubmitThreadpoolWork(work);


    //
    // Create a timer with the same callback environment.
    //
    timer = CreateThreadpoolTimer(timercallback,
                                  NULL,
                                  &CallBackEnviron);


    if (NULL == timer) {
        _tprintf(_T("CreateThreadpoolTimer failed. LastError: %u\n"),
                    GetLastError());
        goto main_cleanup;
    }

    rollback = 4;  // Timer creation succeeded

    //
    // Set the timer to fire in one second.
    //
    ulDueTime.QuadPart = (ULONGLONG) -(1 * 10 * 1000 * 1000);
    FileDueTime.dwHighDateTime = ulDueTime.HighPart;
    FileDueTime.dwLowDateTime  = ulDueTime.LowPart;

    SetThreadpoolTimer(timer,
                       &FileDueTime,
                       0,
                       0);

    //
    // Delay for the timer to be fired
    //
    Sleep(1500);

    //
    // Wait for all callbacks to finish.
    // CloseThreadpoolCleanupGroupMembers also releases objects
    // that are members of the cleanup group, so it is not necessary
    // to call close functions on individual objects
    // after calling CloseThreadpoolCleanupGroupMembers.
    //
    CloseThreadpoolCleanupGroupMembers(cleanupgroup,
                                       FALSE,
                                       NULL);

    //
    // Already cleaned up the work item with the
    // CloseThreadpoolCleanupGroupMembers, so set rollback to 2.
    //
    rollback = 2;
    goto main_cleanup;

main_cleanup:
    //
    // Clean up any individual pieces manually
    // Notice the fall-through structure of the switch.
    // Clean up in reverse order.
    //

    switch (rollback) {
        case 4:
```

```
        case 3:
            // Clean up the cleanup group members.
            CloseThreadpoolCleanupGroupMembers(cleanupgroup,
                FALSE, NULL);
        case 2:
            // Clean up the cleanup group.
            CloseThreadpoolCleanupGroup(cleanupgroup);

        case 1:
            // Clean up the pool.
            CloseThreadpool(pool);

        default:
            break;
    }

    return;
}

VOID
DemoNewRegisterWait()
{
    PTP_WAIT Wait = NULL;
    PTP_WAIT_CALLBACK waitcallback = MyWaitCallback;
    HANDLE hEvent = NULL;
    UINT i = 0;
    UINT rollback = 0;

    //
    // Create an auto-reset event.
    //
    hEvent = CreateEvent(NULL, FALSE, FALSE, NULL);

    if (NULL == hEvent) {
        // Error Handling
        return;
    }

    rollback = 1; // CreateEvent succeeded

    Wait = CreateThreadpoolWait(waitcallback,
                                NULL,
                                NULL);

    if(NULL == Wait) {
        _tprintf(_T("CreateThreadpoolWait failed. LastError: %u\n"),
                     GetLastError());
        goto new_wait_cleanup;
    }

    rollback = 2; // CreateThreadpoolWait succeeded

    //
    // Need to re-register the event with the wait object
    // each time before signaling the event to trigger the wait callback.
    //
    for (i = 0; i < 5; i ++) {
        SetThreadpoolWait(Wait,
                          hEvent,
                          NULL);

        SetEvent(hEvent);

        //
        // Delay for the waiter thread to act if necessary.
        //
        Sleep(500);

        //
```

```
        //
        // Block here until the callback function is done executing.
        //

        WaitForThreadpoolWaitCallbacks(Wait, FALSE);
    }

new_wait_cleanup:
    switch (rollback) {
        case 2:
            // Unregister the wait by setting the event to NULL.
            SetThreadpoolWait(Wait, NULL, NULL);

            // Close the wait.
            CloseThreadpoolWait(Wait);

        case 1:
            // Close the event.
            CloseHandle(hEvent);

        default:
            break;
    }
    return;
}

int main( void)
{
    DemoNewRegisterWait();
    DemoCleanupPersistentWorkTimer();
    return 0;
}
```

Related topics

Thread Pools

Process and Thread Reference

9/16/2022 • 2 minutes to read • Edit Online

The following elements are used with processes and threads.

- Process and Thread Enumerations
- Process and Thread Functions
- Process and Thread Structures
- Process and Thread Macros
- Process Creation Flags

Process and Thread Enumerations

9/16/2022 • 2 minutes to read • Edit Online

The following enumerations are used with the process and thread functions:

- CPU_SET_INFORMATION_TYPE
- DISPATCHERQUEUE_THREAD_APARTMENTTYPE
- DISPATCHERQUEUE_THREAD_TYPE
- LOGICAL_PROCESSOR_RELATIONSHIP
- JOB_OBJECT_NET_RATE_CONTROL_FLAGS
- PROCESS_INFORMATION_CLASS
- PROCESS_MEMORY_EXHAUSTION_TYPE
- PROCESS_MITIGATION_POLICY
- PROCESSOR_CACHE_TYPE
- UMS_THREAD_INFO_CLASS

CPU_SET_INFORMATION_TYPE enumeration

9/16/2022 • 2 minutes to read • Edit Online

Represents the type of information in the SYSTEM_CPU_SET_INFORMATION structure.

Syntax

```
typedef enum _CPU_SET_INFORMATION_TYPE {
  CpuSetInformation
} CPU_SET_INFORMATION_TYPE, *PCPU_SET_INFORMATION_TYPE;
```

Constants

CpuSetInformation

The structure contains CPU Set information.

Requirements

REQUIREMENT	VALUE
Minimum supported client	Windows 10 [desktop apps only]
Minimum supported server	Windows Server 2016 [desktop apps only]
Header	Processthreadsapi.h (include Windows.h)

Process and Thread Functions

9/16/2022 • 16 minutes to read • Edit Online

This topic describes the process and thread functions.

Dispatch Queue Function

The following function creates a DispatcherQueueController.

FUNCTION	DESCRIPTION
CreateDispatcherQueueController	Creates a DispatcherQueueController which manages the lifetime of a DispatcherQueue that runs queued tasks in priority order on another thread.

Process Functions

The following functions are used with processes.

FUNCTION	DESCRIPTION
CreateProcess	Creates a new process and its primary thread.
CreateProcessAsUser	Creates a new process and its primary thread. The new process runs in the security context of the user represented by the specified token.

FUNCTION	DESCRIPTION
CreateProcessWithLogonW	Creates a new process and its primary thread. The new process then runs the specified executable file in the security context of the specified credentials (user, domain, and password).
CreateProcessWithTokenW	Creates a new process and its primary thread. The new process runs in the security context of the specified token.
ExitProcess	Ends the calling process and all its threads.
FlushProcessWriteBuffers	Flushes the write queue of each processor that is running a thread of the current process.
FreeEnvironmentStrings	Frees a block of environment strings.
GetCommandLine	Retrieves the command-line string for the current process.
GetCurrentProcess	Retrieves a pseudo handle for the current process.
GetCurrentProcessId	Retrieves the process identifier of the calling process.
GetCurrentProcessorNumber	Retrieves the number of the processor the current thread was running on during the call to this function.
GetEnvironmentStrings	Retrieves the environment block for the current process.
GetEnvironmentVariable	Retrieves the value of the specified variable from the environment block of the calling process.
GetExitCodeProcess	Retrieves the termination status of the specified process.
GetGuiResources	Retrieves the count of handles to graphical user interface (GUI) objects in use by the specified process.
GetLogicalProcessorInformation	Retrieves information about logical processors and related hardware.
GetPriorityClass	Retrieves the priority class for the specified process.
GetProcessAffinityMask	Retrieves a process affinity mask for the specified process and the system affinity mask for the system.
GetProcessGroupAffinity	Retrieves the processor group affinity of the specified process.
GetProcessHandleCount	Retrieves the number of open handles that belong to the specified process.
GetProcessId	Retrieves the process identifier of the specified process.
GetProcessIoCounters	Retrieves accounting information for all I/O operations performed by the specified process.

FUNCTION	DESCRIPTION
GetProcessMitigationPolicy	Retrieves mitigation policy settings for the calling process.
GetProcessPriorityBoost	Retrieves the priority boost control state of the specified process.
GetProcessShutdownParameters	Retrieves shutdown parameters for the currently calling process.
GetProcessTimes	Retrieves timing information about for the specified process.
GetProcessVersion	Retrieves the major and minor version numbers of the system on which the specified process expects to run.
GetProcessWorkingSetSize	Retrieves the minimum and maximum working set sizes of the specified process.
GetProcessWorkingSetSizeEx	Retrieves the minimum and maximum working set sizes of the specified process.
GetProcessorSystemCycleTime	Retrieves the cycle time each processor in the specified group spent executing deferred procedure calls (DPCs) and interrupt service routines (ISRs).
GetStartupInfo	Retrieves the contents of the STARTUPINFO structure that was specified when the calling process was created.
IsImmersiveProcess	Determines whether the process belongs to a Windows Store app.
NeedCurrentDirectoryForExePath	Determines whether the current directory should be included in the search path for the specified executable.
OpenProcess	Opens an existing local process object.
QueryFullProcessImageName	Retrieves the full name of the executable image for the specified process.
QueryProcessAffinityUpdateMode	Retrieves the affinity update mode of the specified process.
QueryProcessCycleTime	Retrieves the sum of the cycle time of all threads of the specified process.
SetEnvironmentVariable	Sets the value of an environment variable for the current process.
SetPriorityClass	Sets the priority class for the specified process.
SetProcessAffinityMask	Sets a processor affinity mask for the threads of a specified process.
SetProcessAffinityUpdateMode	Sets the affinity update mode of the specified process.
SetProcessInformation	Sets information for the specified process.

FUNCTION	DESCRIPTION
SetProcessMitigationPolicy	Sets the mitigation policy for the calling process.
SetProcessPriorityBoost	Disables the ability of the system to temporarily boost the priority of the threads of the specified process.
SetProcessRestrictionExemption	Exempts the calling process from restrictions preventing desktop processes from interacting with the Windows Store app environment. This function is used by development and debugging tools.
SetProcessShutdownParameters	Sets shutdown parameters for the currently calling process.
SetProcessWorkingSetSize	Sets the minimum and maximum working set sizes for the specified process.
SetProcessWorkingSetSizeEx	Sets the minimum and maximum working set sizes for the specified process.
TerminateProcess	Terminates the specified process and all of its threads.

Process Enumeration Functions

The following functions are used to enumerate processes.

FUNCTION	DESCRIPTION
EnumProcesses	Retrieves the process identifier for each process object in the system.
Process32First	Retrieves information about the first process encountered in a system snapshot.
Process32Next	Retrieves information about the next process recorded in a system snapshot.
WTSEnumerateProcesses	Retrieves information about the active processes on the specified terminal server.

Policy Functions

The following functions are used with process wide policy.

FUNCTION	DESCRIPTION
QueryProtectedPolicy	Queries the value associated with a protected policy.
SetProtectedPolicy	Sets a protected policy.

Thread Functions

The following functions are used with threads.

FUNCTION	DESCRIPTION
AttachThreadInput	Attaches the input processing mechanism of one thread to that of another thread.
CreateRemoteThread	Creates a thread that runs in the virtual address space of another process.
CreateRemoteThreadEx	Creates a thread that runs in the virtual address space of another process and optionally specifies extended attributes such as processor group affinity.
CreateThread	Creates a thread to execute within the virtual address space of the calling process.
ExitThread	Ends the calling thread.
GetCurrentThread	Retrieves a pseudo handle for the current thread.
GetCurrentThreadId	Retrieves the thread identifier of the calling thread.
GetExitCodeThread	Retrieves the termination status of the specified thread.
GetProcessIdOfThread	Retrieves the process identifier of the process associated with the specified thread.
GetThreadDescription	Retrieves the description that was assigned to a thread by calling SetThreadDescription.
GetThreadGroupAffinity	Retrieves the processor group affinity of the specified thread.
GetThreadId	Retrieves the thread identifier of the specified thread.
GetThreadIdealProcessorEx	Retrieves the processor number of the ideal processor for the specified thread.
GetThreadInformation	Retrieves information about the specified thread.
GetThreadIOPendingFlag	Determines whether a specified thread has any I/O requests pending.
GetThreadPriority	Retrieves the priority value for the specified thread.
GetThreadPriorityBoost	Retrieves the priority boost control state of the specified thread.
GetThreadTimes	Retrieves timing information for the specified thread.
OpenThread	Opens an existing thread object.

FUNCTION	DESCRIPTION
QueryIdleProcessorCycleTime	Retrieves the cycle time for the idle thread of each processor in the system.
QueryThreadCycleTime	Retrieves the cycle time for the specified thread.
ResumeThread	Decrements a thread's suspend count.
SetThreadAffinityMask	Sets a processor affinity mask for the specified thread.
SetThreadDescription	Assigns a description to a thread.
SetThreadGroupAffinity	Sets the processor group affinity for the specified thread.
SetThreadIdealProcessor	Specifies a preferred processor for a thread.
SetThreadIdealProcessorEx	Sets the ideal processor for the specified thread and optionally retrieves the previous ideal processor.
SetThreadInformation	Sets information for the specified thread.
SetThreadPriority	Sets the priority value for the specified thread.
SetThreadPriorityBoost	Disables the ability of the system to temporarily boost the priority of a thread.
SetThreadStackGuarantee	Sets the stack guarantee for the calling thread.
Sleep	Suspends the execution of the current thread for a specified interval.
SleepEx	Suspends the current thread until the specified condition is met.
SuspendThread	Suspends the specified thread.
SwitchToThread	Causes the calling thread to yield execution to another thread that is ready to run on the current processor.
TerminateThread	Terminates a thread.
ThreadProc	An application-defined function that serves as the starting address for a thread.
TlsAlloc	Allocates a thread local storage (TLS) index.
TlsFree	Releases a TLS index.
TlsGetValue	Retrieves the value in the calling thread's TLS slot for a specified TLS index.
TlsSetValue	Stores a value in the calling thread's TLS slot for a specified TLS index.

FUNCTION	DESCRIPTION
WaitForInputIdle	Waits until the specified process is waiting for user input with no input pending, or until the time-out interval has elapsed.

Process and Thread Extended Attribute Functions

The following functions are used to set extended attributes for process and thread creation.

FUNCTION	DESCRIPTION
DeleteProcThreadAttributeList	Deletes the specified list of attributes for process and thread creation.
InitializeProcThreadAttributeList	Initializes the specified list of attributes for process and thread creation.
UpdateProcThreadAttribute	Updates the specified attribute in the specified list of attributes for process and thread creation.

WOW64 Functions

The following functions are used with WOW64.

FUNCTION	DESCRIPTION
IsWow64Message	Determines whether the last message read from the current thread's queue originated from a WOW64 process.
IsWow64Process	Determines whether the specified process is running under WOW64.
IsWow64Process2	Determines whether the specified process is running under WOW64; also returns additional machine process and architecture information.
Wow64SuspendThread	Suspends the specified WOW64 thread.

Job Object Functions

The following functions are used with job objects.

FUNCTION	DESCRIPTION
AssignProcessToJobObject	Associates a process with an existing job object.
CreateJobObject	Creates or opens a job object.

FUNCTION	DESCRIPTION
IsProcessInJob	Determines whether the process is running in the specified job.
OpenJobObject	Opens an existing job object.
QueryInformationJobObject	Retrieves limit and job state information from the job object.
SetInformationJobObject	Set limits for a job object.
TerminateJobObject	Terminates all processes currently associated with the job.
UserHandleGrantAccess	Grants or denies access to a handle to a User object to a job that has a user-interface restriction.

Thread Pool Functions

The following functions are used with thread pools.

FUNCTION	DESCRIPTION
CallbackMayRunLong	Indicates that the callback may not return quickly.
CancelThreadpoolIo	Cancels the notification from the StartThreadpoolIo function.
CloseThreadpool	Closes the specified thread pool.
CloseThreadpoolCleanupGroup	Closes the specified cleanup group.
CloseThreadpoolCleanupGroupMembers	Releases the members of the specified cleanup group, waits for all callback functions to complete, and optionally cancels any outstanding callback functions.
CloseThreadpoolIo	Releases the specified I/O completion object.
CloseThreadpoolTimer	Releases the specified timer object.
CloseThreadpoolWait	Releases the specified wait object.
CloseThreadpoolWork	Releases the specified work object.
CreateThreadpool	Allocates a new pool of threads to execute callbacks.
CreateThreadpoolCleanupGroup	Creates a cleanup group that applications can use to track one or more thread pool callbacks.
CreateThreadpoolIo	Creates a new I/O completion object.
CreateThreadpoolTimer	Creates a new timer object.

FUNCTION	DESCRIPTION
CreateThreadpoolWait	Creates a new wait object.
CreateThreadpoolWork	Creates a new work object.
DestroyThreadpoolEnvironment	Deletes the specified callback environment. Call this function when the callback environment is no longer needed for creating new thread pool objects.
DisassociateCurrentThreadFromCallback	Removes the association between the currently executing callback function and the object that initiated the callback. The current thread will no longer count as executing a callback on behalf of the object.
FreeLibraryWhenCallbackReturns	Specifies the DLL that the thread pool will unload when the current callback completes.
InitializeThreadpoolEnvironment	Initializes a callback environment.
IsThreadpoolTimerSet	Determines whether the specified timer object is currently set.
LeaveCriticalSectionWhenCallbackReturns	Specifies the critical section that the thread pool will release when the current callback completes.
QueryThreadpoolStackInformation	Retrieves the stack reserve and commit sizes for threads in the specified thread pool.
ReleaseMutexWhenCallbackReturns	Specifies the mutex that the thread pool will release when the current callback completes.
ReleaseSemaphoreWhenCallbackReturns	Specifies the semaphore that the thread pool will release when the current callback completes.
SetEventWhenCallbackReturns	Specifies the event that the thread pool will set when the current callback completes.
SetThreadpoolCallbackCleanupGroup	Associates the specified cleanup group with the specified callback environment.
SetThreadpoolCallbackLibrary	Ensures that the specified DLL remains loaded as long as there are outstanding callbacks.
SetThreadpoolCallbackPersistent	Specifies that the callback should run on a persistent thread.
SetThreadpoolCallbackPool	Sets the thread pool to be used when generating callbacks.
SetThreadpoolCallbackPriority	Specifies the priority of a callback function relative to other work items in the same thread pool.
SetThreadpoolCallbackRunsLong	Indicates that callbacks associated with this callback environment may not return quickly.

FUNCTION	DESCRIPTION
SetThreadpoolStackInformation	Sets the stack reserve and commit sizes for new threads in the specified thread pool.
SetThreadpoolThreadMaximum	Sets the maximum number of threads that the specified thread pool can allocate to process callbacks.
SetThreadpoolThreadMinimum	Sets the minimum number of threads that the specified thread pool must make available to process callbacks.
SetThreadpoolTimerEx	Sets the timer object. A worker thread calls the timer object's callback after the specified timeout expires.
SetThreadpoolTimer	Sets the timer object. A worker thread calls the timer object's callback after the specified timeout expires.
SetThreadpoolWait	Sets the wait object. A worker thread calls the wait object's callback function after the handle becomes signaled or after the specified timeout expires.
SetThreadpoolWaitEx	Sets the wait object. A worker thread calls the wait object's callback function after the handle becomes signaled or after the specified timeout expires.
StartThreadpoolIo	Notifies the thread pool that I/O operations may possibly begin for the specified I/O completion object. A worker thread calls the I/O completion object's callback function after the operation completes on the file handle bound to this object.
SubmitThreadpoolWork	Posts a work object to the thread pool. A worker thread calls the work object's callback function.
TpInitializeCallbackEnviron	Initializes a callback environment for the thread pool.
TpDestroyCallbackEnviron	Deletes the specified callback environment. Call this function when the callback environment is no longer needed for creating new thread pool objects.
TpSetCallbackActivationContext	Assigns an activation context to the callback environment.
TpSetCallbackCleanupGroup	Associates the specified cleanup group with the specified callback environment.
TpSetCallbackFinalizationCallback	Indicates a function to call when the callback environment is finalized.
TpSetCallbackLongFunction	Indicates that callbacks associated with this callback environment may not return quickly.
TpSetCallbackNoActivationContext	Indicates that the callback environment has no activation context.
TpSetCallbackPersistent	Specifies that the callback should run on a persistent thread.

FUNCTION	DESCRIPTION
TpSetCallbackPriority	Specifies the priority of a callback function relative to other work items in the same thread pool.
TpSetCallbackRaceWithDll	Ensures that the specified DLL remains loaded as long as there are outstanding callbacks.
TpSetCallbackThreadpool	Assigns a thread pool to a callback environment.
TrySubmitThreadpoolCallback	Requests that a thread pool worker thread call the specified callback function.
WaitForThreadpoolIoCallbacks	Waits for outstanding I/O completion callbacks to complete and optionally cancels pending callbacks that have not yet started to execute.
WaitForThreadpoolTimerCallbacks	Waits for outstanding timer callbacks to complete and optionally cancels pending callbacks that have not yet started to execute.
WaitForThreadpoolWaitCallbacks	Waits for outstanding wait callbacks to complete and optionally cancels pending callbacks that have not yet started to execute.
WaitForThreadpoolWorkCallbacks	Waits for outstanding work callbacks to complete and optionally cancels pending callbacks that have not yet started to execute.

The following functions are part of the original thread pooling API.

FUNCTION	DESCRIPTION
BindIoCompletionCallback	Associates the I/O completion port owned by the thread pool with the specified file handle. On completion of an I/O request involving this file, a non-I/O worker thread will execute the specified callback function.
QueueUserWorkItem	Queues a work item to a worker thread in the thread pool.
RegisterWaitForSingleObject	Directs a wait thread in the thread pool to wait on the object.
UnregisterWaitEx	Waits until one or all of the specified objects are in the signaled state or the time-out interval elapses.

Thread Ordering Service Functions

The following functions are used with the thread ordering service.

FUNCTION	DESCRIPTION
AvQuerySystemResponsiveness	Retrieves the system responsiveness setting used by the multimedia class scheduler service.
AvRtCreateThreadOrderingGroup	Creates a thread ordering group.
AvRtCreateThreadOrderingGroupEx	Creates a thread ordering group and associates the server thread with a task.
AvRtDeleteThreadOrderingGroup	Deletes the specified thread ordering group created by the caller.
AvRtJoinThreadOrderingGroup	Joins client threads to a thread ordering group.
AvRtLeaveThreadOrderingGroup	Enables client threads to leave a thread ordering group.
AvRtWaitOnThreadOrderingGroup	Enables client threads of a thread ordering group to wait until they should execute.

Multimedia Class Scheduler Service Functions

The following functions are used with the multimedia class scheduler service.

FUNCTION	DESCRIPTION
AvRevertMmThreadCharacteristics	Indicates that a thread is no longer performing work associated with the specified task.
AvSetMmMaxThreadCharacteristics	Associates the calling thread with the specified tasks.
AvSetMmThreadCharacteristics	Associates the calling thread with the specified task.
AvSetMmThreadPriority	Adjusts the thread priority of the calling thread relative to other threads performing the same task.

Fiber Functions

The following functions are used with fibers.

FUNCTION	DESCRIPTION
ConvertFiberToThread	Converts the current fiber into a thread.
ConvertThreadToFiber	Converts the current thread into a fiber.
ConvertThreadToFiberEx	Converts the current thread into a fiber.
CreateFiber	Allocates a fiber object, assigns it a stack, and sets up execution to begin at the specified start address.

FUNCTION	DESCRIPTION
CreateFiberEx	Allocates a fiber object, assigns it a stack, and sets up execution to begin at the specified start address.
DeleteFiber	Deletes an existing fiber.
FiberProc	An application-defined function used with the CreateFiber function.
FlsAlloc	Allocates a fiber local storage (FLS) index.
FlsFree	Releases an FLS index.
FlsGetValue	Retrieves the value in the calling fiber's FLS slot for a specified FLS index.
FlsSetValue	Stores a value in the calling fiber's FLS slot for a specified FLS index.
IsThreadAFiber	Determines whether the current thread is a fiber.
SwitchToFiber	Schedules a fiber.

NUMA Support Functions

The following functions provide NUMA support.

FUNCTION	DESCRIPTION
AllocateUserPhysicalPagesNuma	Reserves or commits a region of memory within the virtual address space of the specified process, and specifies the NUMA node for the physical memory.
GetLogicalProcessorInformation	Retrieves information about logical processors and related hardware.
GetNumaAvailableMemoryNode	Retrieves the amount of memory available in the specified node.
GetNumaAvailableMemoryNodeEx	Retrieves the amount of memory that is available in the specified node as a USHORT value.
GetNumaHighestNodeNumber	Retrieves the node that currently has the highest number.
GetNumaNodeNumberFromHandle	Retrieves the NUMA node associated with the underlying device for a file handle.
GetNumaNodeProcessorMask	Retrieves the processor mask for the specified node.
GetNumaNodeProcessorMaskEx	Retrieves the processor mask for the specified NUMA node as a USHORT value.

FUNCTION	DESCRIPTION
GetNumaProcessorNode	Retrieves the node number for the specified processor.
GetNumaProcessorNodeEx	Retrieves the node number of the specified logical processor as a USHORT value.
GetNumaProximityNode	Retrieves the node number for the specified proximity identifier.
GetNumaProximityNodeEx	Retrieves the node number as a USHORT value for the specified proximity identifier.
VirtualAllocExNuma	Reserves or commits a region of memory within the virtual address space of the specified process, and specifies the NUMA node for the physical memory.

Processor Functions

The following functions are used with logical processors and processor groups.

FUNCTION	DESCRIPTION
GetActiveProcessorCount	Returns the number of active processors in a processor group or in the system.
GetActiveProcessorGroupCount	Returns the number of active processor groups in the system.
GetCurrentProcessorNumber	Retrieves the number of the processor the current thread was running on during the call to this function.
GetCurrentProcessorNumberEx	Retrieves the processor group and number of the logical processor in which the calling thread is running.
GetLogicalProcessorInformation	Retrieves information about logical processors and related hardware.
GetLogicalProcessorInformationEx	Retrieves information about the relationships of logical processors and related hardware.
GetMaximumProcessorCount	Returns the maximum number of logical processors that a processor group or the system can have.
GetMaximumProcessorGroupCount	Returns the maximum number of processor groups that the system can have.
QueryIdleProcessorCycleTime	Retrieves the cycle time for the idle thread of each processor in the system.
QueryIdleProcessorCycleTimeEx	Retrieves the accumulated cycle time for the idle thread on each logical processor in the specified processor group.

User-Mode Scheduling Functions

The following functions are used with user-mode scheduling (UMS).

FUNCTION	DESCRIPTION
CreateUmsCompletionList	Creates a UMS completion list.
CreateUmsThreadContext	Creates a UMS thread context to represent a UMS worker thread.
DeleteUmsCompletionList	Deletes the specified UMS completion list. The list must be empty.
DeleteUmsThreadContext	Deletes the specified UMS thread context. The thread must be terminated.
DequeueUmsCompletionListItems	Retrieves UMS worker threads from the specified UMS completion list.
EnterUmsSchedulingMode	Converts the calling thread into a UMS scheduler thread.
ExecuteUmsThread	Runs the specified UMS worker thread.
GetCurrentUmsThread	Returns the UMS thread context of the calling UMS thread.
GetNextUmsListItem	Returns the next UMS thread context in a list of UMS thread contexts.
GetUmsCompletionListEvent	Retrieves a handle to the event associated with the specified UMS completion list.
GetUmsSystemThreadInformation	Queries whether the specified thread is a UMS scheduler thread, a UMS worker thread, or a non-UMS thread.
QueryUmsThreadInformation	Retrieves information about the specified UMS worker thread.
SetUmsThreadInformation	Sets application-specific context information for the specified UMS worker thread.
UmsSchedulerProc	The application-defined UMS scheduler entry point function associated with a UMS completion list.
UmsThreadYield	Yields control to the UMS scheduler thread on which the calling UMS worker thread is running.

Obsolete Functions

- NtGetCurrentProcessorNumber
- NtQueryInformationProcess
- NtQueryInformationThread
- WinExec

- ZwQueryInformationProcess

GetSystemCpuSetInformation function

9/16/2022 • 2 minutes to read • Edit Online

Allows an application to query the available CPU Sets on the system, and their current state.

Syntax

```
BOOL WINAPI GetSystemCpuSetInformation(
  _Out_opt_  PSYSTEM_CPU_SET_INFORMATION  Information,
  _In_       ULONG                        BufferLength,
  _Out_      PULONG                       ReturnedLength,
  _In_opt_   HANDLE                       Process,
  _Reserved_ ULONG                        Flags
);
```

Parameters

Information [out, optional]

A pointer to a SYSTEM_CPU_SET_INFORMATION structure that receives the CPU Set data. Pass NULL with a buffer length of 0 to determine the required buffer size.

BufferLength [in]

The length, in bytes, of the output buffer passed as the Information argument.

ReturnedLength [out]

The length, in bytes, of the valid data in the output buffer if the buffer is large enough, or the required size of the output buffer. If no CPU Sets exist, this value will be 0.

Process [in, optional]

An optional handle to a process. This process is used to determine the value of the **AllocatedToTargetProcess** flag in the SYSTEM_CPU_SET_INFORMATION structure. If a CPU Set is allocated to the specified process, the flag is set. Otherwise, it is clear. This handle must have the PROCESS_QUERY_LIMITED_INFORMATION access right. The value returned by GetCurrentProcess may also be specified here.

Flags

Reserved, must be 0.

Return value

If the API succeeds it returns TRUE. If it fails, the error reason is available through **GetLastError**. If the Information buffer was NULL or not large enough, the error code ERROR_INSUFFICIENT_BUFFER is returned. This API cannot fail when passed valid parameters and a buffer that is large enough to hold all of the return data.

Requirements

REQUIREMENT	VALUE
Minimum supported client	Windows 10 [desktop apps | UWP apps]
Minimum supported server	Windows Server 2016 [desktop apps | UWP apps]
Header	Processthreadsapi.h
Library	Windows.h
DLL	Kernel32.dll

NtGetCurrentProcessorNumber function

9/16/2022 • 2 minutes to read • Edit Online

[**NtGetCurrentProcessorNumber** may be altered or unavailable in future versions of Windows. Applications should use the **GetCurrentProcessorNumber** function instead.]

Retrieves the number of the processor the current thread was running on during the call to this function.

Syntax

```
ULONG WINAPI NtGetCurrentProcessorNumber(void);
```

Parameters

This function has no parameters.

Return value

The function returns the current processor number.

Remarks

This function is used to provide information for estimating process performance.

This function has no associated import library. You must use the **LoadLibrary** and **GetProcAddress** functions to dynamically link to Ntdll.dll.

Requirements

REQUIREMENT	VALUE
DLL	Ntdll.dll

See also

Multiple Processors

Process and Thread Functions

Processes

ZwQueryInformationProcess function

9/16/2022 • 4 minutes to read • Edit Online

[**ZwQueryInformationProcess** may be altered or unavailable in future versions of Windows. Applications should use the alternate functions listed in this topic.]

Retrieves information about the specified process.

Syntax

```
NTSTATUS WINAPI ZwQueryInformationProcess(
  _In_      HANDLE           ProcessHandle,
  _In_      PROCESSINFOCLASS ProcessInformationClass,
  _Out_     PVOID            ProcessInformation,
  _In_      ULONG            ProcessInformationLength,
  _Out_opt_ PULONG           ReturnLength
);
```

Parameters

ProcessHandle [in]

A handle to the process for which information is to be retrieved.

ProcessInformationClass [in]

The type of process information to be retrieved. This parameter can be one of the following values from the **PROCESSINFOCLASS** enumeration.

VALUE	MEANING
ProcessBasicInformation 0	Retrieves a pointer to a PEB structure that can be used to determine whether the specified process is being debugged, and a unique value used by the system to identify the specified process. It is best to use the **CheckRemoteDebuggerPresent** and **GetProcessId** functions to obtain this information.
ProcessDebugPort 7	Retrieves a **DWORD_PTR** value that is the port number of the debugger for the process. A nonzero value indicates that the process is being run under the control of a ring 3 debugger. It is best to use the **CheckRemoteDebuggerPresent** or **IsDebuggerPresent** function.
ProcessWow64Information 26	Determines whether the process is running in the WOW64 environment (WOW64 is the x86 emulator that allows Win32-based applications to run on 64-bit Windows). It is best to use the **IsWow64Process** function to obtain this information.

VALUE	MEANING
ProcessImageFileName 27	Retrieves a **UNICODE_STRING** value containing the name of the image file for the process.
ProcessBreakOnTermination 29	Retrieves a **ULONG** value indicating whether the process is considered critical. [!Note] This value can be used starting in Windows XP with SP3. Starting in Windows 8.1, **IsProcessCritical** should be used instead.
ProcessProtectionInformation 61	Retrieves a BYTE value indicating the type of protected process and the protected process signer.

ProcessInformation [out]

A pointer to a buffer supplied by the calling application into which the function writes the requested information. The size of the information written varies depending on the value of the *ProcessInformationClass* parameter:

PROCESS_BASIC_INFORMATION

When the *ProcessInformationClass* parameter is **ProcessBasicInformation**, the buffer pointed to by the *ProcessInformation* parameter should be large enough to hold a single **PROCESS_BASIC_INFORMATION** structure having the following layout:

```
typedef struct _PROCESS_BASIC_INFORMATION {
    NTSTATUS ExitStatus;
    PPEB PebBaseAddress;
    ULONG_PTR AffinityMask;
    KPRIORITY BasePriority;
    ULONG_PTR UniqueProcessId;
    ULONG_PTR InheritedFromUniqueProcessId;
} PROCESS_BASIC_INFORMATION;
```

FIELD	MEANING
ExitStatus	Contains the same value that **GetExitCodeProcess** would return. However the use of **GetExitCodeProcess** is preferable for clarity and safety.
PebBaseAddress	Points to a **PEB** structure.
AffinityMask	Can be cast to a **DWORD** and contains the same value that **GetProcessAffinityMask** would return for the `lpProcessAffinityMask` parameter.
BasePriority	Contains the process priority as described in Scheduling Priorities.

FIELD	MEANING
UniqueProcessId	Can be cast to a **DWORD** and contains a unique identifier for this process. It is best to use the **GetProcessId** function to retrieve this information.
InheritedFromUniqueProcessId	Can be cast to a **DWORD** and contains a unique identifier for the parent process.

ULONG_PTR

When the *ProcessInformationClass* parameter is **ProcessWow64Information**, the buffer pointed to by the *ProcessInformation* parameter should be large enough to hold a **ULONG_PTR**. If this value is nonzero, the process is running in a WOW64 environment; otherwise, if the value is equal to zero, the process is not running in a WOW64 environment.

It is best to use the **IsWow64Process** function to determine whether a process is running in the WOW64 environment.

UNICODE_STRING

When the *ProcessInformationClass* parameter is **ProcessImageFileName**, the buffer pointed to by the *ProcessInformation* parameter should be large enough to hold a **UNICODE_STRING** structure as well as the string itself. The string stored in the **Buffer** member is the name of the image file.

If the buffer is too small, the function fails with the STATUS_INFO_LENGTH_MISMATCH error code and the *ReturnLength* parameter is set to the required buffer size.

PS_PROTECTION

When the *ProcessInformationClass* parameter is **ProcessProtectionInformation**, the buffer pointed to by the *ProcessInformation* parameter should be large enough to hold a single **PS_PROTECTION** structure having the following layout:

```
typedef struct _PS_PROTECTION {
    union {
        UCHAR Level;
        struct {
            UCHAR Type   : 3;
            UCHAR Audit  : 1;                  // Reserved
            UCHAR Signer : 4;
        };
    };
} PS_PROTECTION, *PPS_PROTECTION;
```

The first 3 bits contain the type of protected process:

```
typedef enum _PS_PROTECTED_TYPE {
    PsProtectedTypeNone = 0,
    PsProtectedTypeProtectedLight = 1,
    PsProtectedTypeProtected = 2
} PS_PROTECTED_TYPE, *PPS_PROTECTED_TYPE;
```

The top 4 bits contain the protected process signer:

```
typedef enum _PS_PROTECTED_SIGNER {
    PsProtectedSignerNone = 0,
    PsProtectedSignerAuthenticode,
    PsProtectedSignerCodeGen,
    PsProtectedSignerAntimalware,
    PsProtectedSignerLsa,
    PsProtectedSignerWindows,
    PsProtectedSignerWinTcb,
    PsProtectedSignerWinSystem,
    PsProtectedSignerApp,
    PsProtectedSignerMax
} PS_PROTECTED_SIGNER, *PPS_PROTECTED_SIGNER;
```

ProcessInformationLength [in]

The size of the buffer pointed to by the *ProcessInformation* parameter, in bytes.

ReturnLength [out, optional]

A pointer to a variable in which the function returns the size of the requested information. If the function was successful, this is the size of the information written to the buffer pointed to by the *ProcessInformation* parameter, but if the buffer was too small, this is the minimum size of buffer needed to receive the information successfully.

Return value

Returns an NTSTATUS success or error code.

The forms and significance of NTSTATUS error codes are listed in the Ntstatus.h header file available in the DDK, and are described in the DDK documentation under Kernel-Mode Driver Architecture / Design Guide / Driver Programming Techniques / Logging Errors.

Remarks

The **ZwQueryInformationProcess** function and the structures that it returns are internal to the operating system and subject to change from one release of Windows to another. To maintain the compatibility of your application, it is better to use public functions mentioned in the description of the *ProcessInformationClass* parameter instead.

If you do use **ZwQueryInformationProcess**, access the function through run-time dynamic linking. This gives your code an opportunity to respond gracefully if the function has been changed or removed from the operating system. Signature changes, however, may not be detectable.

This function has no associated import library. You must use the **LoadLibrary** and **GetProcAddress** functions to dynamically link to Ntdll.dll.

Requirements

REQUIREMENT	VALUE
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
DLL	Ntdll.dll

See also

CheckRemoteDebuggerPresent

GetProcessId

IsDebuggerPresent

IsWow64Process

Process and Thread Macros

9/16/2022 • 2 minutes to read • Edit Online

The following macros are used with fibers:

- GetCurrentFiber
- GetFiberData

Process and Thread Structures

9/16/2022 • 2 minutes to read • Edit Online

This topic lists structures that are used with processes, threads, processors, job objects, and user-mode scheduling (UMS).

Process and Thread Structures

The following structures are used with processes and threads:

- APP_MEMORY_INFORMATION
- AR_STATE
- CACHE_DESCRIPTOR
- IO_COUNTERS
- ORIENTATION_PREFERENCE
- PEB
- PEB_LDR_DATA
- PROCESS_INFORMATION
- PROCESS_MEMORY_EXHAUSTION_INFO
- PROCESS_MITIGATION_ASLR_POLICY
- PROCESS_MITIGATION_BINARY_SIGNATURE_POLICY
- PROCESS_MITIGATION_CONTROL_FLOW_GUARD_POLICY
- PROCESS_MITIGATION_DEP_POLICY
- PROCESS_MITIGATION_DYNAMIC_CODE_POLICY
- PROCESS_MITIGATION_EXTENSION_POINT_DISABLE_POLICY
- PROCESS_MITIGATION_FONT_DISABLE_POLICY
- PROCESS_MITIGATION_IMAGE_LOAD_POLICY
- PROCESS_MITIGATION_STRICT_HANDLE_CHECK_POLICY
- PROCESS_MITIGATION_SYSTEM_CALL_DISABLE_POLICY
- RTL_USER_PROCESS_PARAMETERS
- STARTUPINFO
- STARTUPINFOEX
- TEB

Processor Structures

The following structures are used with processors and processor groups:

- CACHE_RELATIONSHIP
- GROUP_AFFINITY
- GROUP_RELATIONSHIP
- NUMA_NODE_RELATIONSHIP
- PROCESSOR_GROUP_INFO
- PROCESSOR_NUMBER
- PROCESSOR_RELATIONSHIP
- SYSTEM_CPU_SET_INFORMATION

- SYSTEM_LOGICAL_PROCESSOR_INFORMATION
- SYSTEM_LOGICAL_PROCESSOR_INFORMATION_EX

Dispatcher Queue Structure

The following structure is used to create a DispatcherQueueController.

- DispatcherQueueOptions

Job Object Structures

The following structures are used with job objects:

- JOBOBJECT_ASSOCIATE_COMPLETION_PORT
- JOBOBJECT_BASIC_ACCOUNTING_INFORMATION
- JOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION
- JOBOBJECT_BASIC_LIMIT_INFORMATION
- JOBOBJECT_BASIC_PROCESS_ID_LIST
- JOBOBJECT_BASIC_UI_RESTRICTIONS
- JOBOBJECT_END_OF_JOB_TIME_INFORMATION
- JOBOBJECT_EXTENDED_LIMIT_INFORMATION
- JOBOBJECT_SECURITY_LIMIT_INFORMATION

User-Mode Scheduling Structures

The following structures are used with UMS:

- UMS_CREATE_THREAD_ATTRIBUTES
- UMS_SCHEDULER_STARTUP_INFO
- UMS_SYSTEM_THREAD_INFORMATION

Process Creation Flags

9/16/2022 • 5 minutes to read • Edit Online

The following process creation flags are used by the **CreateProcess**, **CreateProcessAsUser**, **CreateProcessWithLogonW**, and **CreateProcessWithTokenW** functions. They can be specified in any combination, except as noted.

Example

```
 BOOL creationResult;

creationResult = CreateProcess(
    NULL,                   // No module name (use command line)
    cmdLine,                // Command line
    NULL,                   // Process handle not inheritable
    NULL,                   // Thread handle not inheritable
    FALSE,                  // Set handle inheritance to FALSE
    NORMAL_PRIORITY_CLASS | CREATE_NEW_CONSOLE | CREATE_NEW_PROCESS_GROUP, // creation flags
    NULL,                   // Use parent's environment block
    NULL,                   // Use parent's starting directory
    &startupInfo,           // Pointer to STARTUPINFO structure
    &processInformation);   // Pointer to PROCESS_INFORMATION structure
```

Example from Windows classic samples on GitHub.

Flags

CONSTANT/VALUE	DESCRIPTION
CREATE_BREAKAWAY_FROM_JOB 0x01000000	The child processes of a process associated with a job are not associated with the job. If the calling process is not associated with a job, this constant has no effect. If the calling process is associated with a job, the job must set the **JOB_OBJECT_LIMIT_BREAKAWAY_OK** limit.
CREATE_DEFAULT_ERROR_MODE 0x04000000	The new process does not inherit the error mode of the calling process. Instead, the new process gets the default error mode. This feature is particularly useful for multithreaded shell applications that run with hard errors disabled. The default behavior is for the new process to inherit the error mode of the caller. Setting this flag changes that default behavior.
CREATE_NEW_CONSOLE 0x00000010	The new process has a new console, instead of inheriting its parent's console (the default). For more information, see Creation of a Console. This flag cannot be used with **DETACHED_PROCESS**.

CONSTANT/VALUE	DESCRIPTION
CREATE_NEW_PROCESS_GROUP 0x00000200	The new process is the root process of a new process group. The process group includes all processes that are descendants of this root process. The process identifier of the new process group is the same as the process identifier, which is returned in the *lpProcessInformation* parameter. Process groups are used by the **GenerateConsoleCtrlEvent** function to enable sending a CTRL+BREAK signal to a group of console processes. If this flag is specified, CTRL+C signals will be disabled for all processes within the new process group. This flag is ignored if specified with **CREATE_NEW_CONSOLE**.
CREATE_NO_WINDOW 0x08000000	The process is a console application that is being run without a console window. Therefore, the console handle for the application is not set. This flag is ignored if the application is not a console application, or if it is used with either **CREATE_NEW_CONSOLE** or **DETACHED_PROCESS**.
CREATE_PROTECTED_PROCESS 0x00040000	The process is to be run as a protected process. The system restricts access to protected processes and the threads of protected processes. For more information on how processes can interact with protected processes, see Process Security and Access Rights. To activate a protected process, the binary must have a special signature. This signature is provided by Microsoft but not currently available for non-Microsoft binaries. There are currently four protected processes: media foundation, audio engine, Windows error reporting, and system. Components that load into these binaries must also be signed. Multimedia companies can leverage the first two protected processes. For more information, see Overview of the Protected Media Path. **Windows Server 2003 and Windows XP:** This value is not supported.
CREATE_PRESERVE_CODE_AUTHZ_LEVEL 0x02000000	Allows the caller to execute a child process that bypasses the process restrictions that would normally be applied automatically to the process.
CREATE_SECURE_PROCESS 0x00400000	This flag allows secure processes, that run in the Virtualization-Based Security environment, to launch.

CONSTANT/VALUE	DESCRIPTION
CREATE_SEPARATE_WOW_VDM 0x00000800	This flag is valid only when starting a 16-bit Windows-based application. If set, the new process runs in a private Virtual DOS Machine (VDM). By default, all 16-bit Windows-based applications run as threads in a single, shared VDM. The advantage of running separately is that a crash only terminates the single VDM; any other programs running in distinct VDMs continue to function normally. Also, 16-bit Windows-based applications that are run in separate VDMs have separate input queues. That means that if one application stops responding momentarily, applications in separate VDMs continue to receive input. The disadvantage of running separately is that it takes significantly more memory to do so. You should use this flag only if the user requests that 16-bit applications should run in their own VDM.
CREATE_SHARED_WOW_VDM 0x00001000	The flag is valid only when starting a 16-bit Windows-based application. If the DefaultSeparateVDM switch in the Windows section of WIN.INI is **TRUE**, this flag overrides the switch. The new process is run in the shared Virtual DOS Machine.
CREATE_SUSPENDED 0x00000004	The primary thread of the new process is created in a suspended state, and does not run until the **ResumeThread** function is called.
CREATE_UNICODE_ENVIRONMENT 0x00000400	If this flag is set, the environment block pointed to by *lpEnvironment* uses Unicode characters. Otherwise, the environment block uses ANSI characters.
DEBUG_ONLY_THIS_PROCESS 0x00000002	The calling thread starts and debugs the new process. It can receive all related debug events using the **WaitForDebugEvent** function.
DEBUG_PROCESS 0x00000001	The calling thread starts and debugs the new process and all child processes created by the new process. It can receive all related debug events using the **WaitForDebugEvent** function. A process that uses **DEBUG_PROCESS** becomes the root of a debugging chain. This continues until another process in the chain is created with **DEBUG_PROCESS**. If this flag is combined with **DEBUG_ONLY_THIS_PROCESS**, the caller debugs only the new process, not any child processes.
DETACHED_PROCESS 0x00000008	For console processes, the new process does not inherit its parent's console (the default). The new process can call the **AllocConsole** function at a later time to create a console. For more information, see Creation of a Console. This value cannot be used with **CREATE_NEW_CONSOLE**.
EXTENDED_STARTUPINFO_PRESENT 0x00080000	The process is created with extended startup information; the *lpStartupInfo* parameter specifies a **STARTUPINFOEX** structure. **Windows Server 2003 and Windows XP:** This value is not supported.

CONSTANT/VALUE	DESCRIPTION
INHERIT_PARENT_AFFINITY 0x00010000	The process inherits its parent's affinity. If the parent process has threads in more than one processor group, the new process inherits the group-relative affinity of an arbitrary group in use by the parent. **Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP:** This value is not supported.

Remarks

On 32-bit Windows, 16-bit applications are simulated by ntvdm.exe, not run as individual processes. Therefore, the process creation flags apply to *ntvdm.exe*. Because ntvdm.exe persists after you run the first 16-bit application, when you launch another 16-bit application, the new creation flags are not applied, except for **CREATE_NEW_CONSOLE** and **CREATE_SEPARATE_WOW_VDM**, which create a new ntvdm.exe.

Requirements

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