Tell us about your PDF experience.

Processes and Threads

07/14/2025

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

Article • 01/07/2021

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.

Expand table

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.

Function	Description
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.
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.

Expand table

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

Function	Description
	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.

Expand table

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.
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

Function	Description
	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

Expand table

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.
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

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

About Processes and Threads

Article • 01/07/2021

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

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Multitasking

Article • 01/07/2021

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

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Advantages of Multitasking

Article • 01/07/2021

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.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

When to Use Multitasking

Article • 01/07/2021

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.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Multitasking Considerations

Article • 01/07/2021

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.

Feedback

Was this page helpful? Yes No

Scheduling

Article • 01/07/2021

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

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Scheduling Priorities

Article • 01/07/2021

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.

Expand table

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

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Context Switches

Article • 01/07/2021

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.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Priority Boosts

Article • 01/07/2021

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.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Priority Inversion

Article • 03/12/2024

A phenomenon known as **priority inversion** occurs when a high–priority thread is indefinitely delayed while awaiting a resource held by a low–priority thread that cannot proceed due to the presence of an unrelated medium–priority thread. As a result, the high–priority thread is effectively denied access to the CPU by the lower medium–priority thread.

For example, a thread T1 running at priority 4 gets preempted by a higher-priority thread T2 with a priority of 8 after acquiring a lock. Subsequently, a thread T3 with a priority of 12 arrives, preempts T2, and gets blocked trying to acquire the lock held by T1. At this point, both T1 and T2 are ready to run, but since T2 has a higher priority, it continues to execute, effectively preventing T3, a higher–priority thread, from making progress because T1 is unable to run and release the lock.

The thread scheduler addresses this issue through a feature called AutoBoost. AutoBoost automatically tracks resource reservations and adjusts thread priorities by applying priority floors that a thread must never fall below. For example, if a low–priority thread acquires a critical section and a higher–priority thread is blocked waiting for the critical section, the priority of the owner is raised to the maximum priority of the waiter until it releases the resource.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Multiple Processors

Article • 01/07/2021

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

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

NUMA Architecture

Article • 05/24/2022

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.

PowerShell

```
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.

C++

```
//
// 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));

    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.

Expand table

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.
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.

Function	Description
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.
VirtualAllocExNuma	Reserves or commits a region of memory within the virtual address space of the specified process, and specifies the

Function	Description
	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

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Thread Ordering Service

Article • 01/07/2021

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.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Multimedia Class Scheduler Service

Article • 03/03/2021

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.

Expand table

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.
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.

Expand table

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.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Real-Time Work Queue API

Article • 01/07/2021

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

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

IRtwqAsyncCallback interface (rtworkq.h)

Article02/22/2024

Callback interface to notify the application when an asynchronous method completes.

Inheritance

The **IRtwqAsyncCallback** interface inherits from the IUnknown interface. **IRtwqAsyncCallback** also has these types of members:

Methods

The **IRtwqAsyncCallback** interface has these methods.

Expand table

IRtwqAsyncCallback::GetParameters Provides configuration information to the dispatching thread for a callback. (IRtwqAsyncCallback.GetParameters)
IRtwqAsyncCallback::Invoke Called when an asynchronous operation is completed. (IRtwqAsyncCallback.Invoke)

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8.1 [desktop apps only]
Minimum supported server	Windows Server 2012 R2 [desktop apps only]
Target Platform	Windows
Header	rtworkq.h

IRtwqAsyncCallback::GetParameters method (rtworkq.h)

Article07/27/2022

Provides configuration information to the dispatching thread for a callback.

Syntax

```
C++

HRESULT GetParameters(
  [out] DWORD *pdwFlags,
  [out] DWORD *pdwQueue
);
```

Parameters

`[out] pdwFlags`

Receives a flag indicating the behavior of the callback object's IRtwqAsyncCallback::Invoke method. The following values are defined. The default value is zero.

Expand table

Value	Meaning
Zero	The callback does not take a long time to complete, but has no specific restrictions on what system calls it makes. The callback generally takes less than 30 milliseconds to complete.

`[out] pdwQueue`

Receives the identifier of the work queue on which the callback is dispatched.

Return value

The method returns an **HRESULT**. Possible values include, but are not limited to, those in the following table.

Expand table

Return code	Description
S_OK	The method succeeded.
E_NOTIMPL	Not implemented. Assume the default behavior.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8.1 [desktop apps only]
Minimum supported server	Windows Server 2012 R2 [desktop apps only]
Target Platform	Windows
Header	rtworkq.h
Library	Rtworkq.lib
DLL	RTWorkQ.dll

See also

IRtwqAsyncCallback

IRtwqAsyncCallback::Invoke method (rtworkq.h)

Article02/22/2024

Called when an asynchronous operation is completed.

Syntax

```C++
HRESULT Invoke(
  [in] IRtwqAsyncResult *pAsyncResult
);
```

Parameters

`[in] pAsyncResult`

Pointer to the IRtwqAsyncResult interface. Pass this pointer to the asynchronous **End...** method to complete the asynchronous call.

Return value

The method returns an **HRESULT**. Possible values include, but are not limited to, those in the following table.

Expand table

Return code	Description
S_OK	The method succeeded.

Remarks

Within your implementation of Invoke, call the corresponding **End...** method.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8.1 [desktop apps only]
Minimum supported server	Windows Server 2012 R2 [desktop apps only]
Target Platform	Windows
Header	rtworkq.h
Library	Rtworkq.lib
DLL	RTWorkQ.dll

See also

IRtwqAsyncCallback

IRtwqAsyncResult interface (rtworkq.h)

Article02/22/2024

Provides information about the result of an asynchronous operation.

Inheritance

The **IRtwqAsyncResult** interface inherits from the IUnknown interface. **IRtwqAsyncResult** also has these types of members:

Methods

The **IRtwqAsyncResult** interface has these methods.

Expand table

IRtwqAsyncResult::GetObject Returns an object associated with the asynchronous operation. The type of object, if any, depends on the asynchronous method that was called. (IRtwqAsyncResult.GetObject)
IRtwqAsyncResult::GetState Returns the state object specified by the caller in the asynchronous Begin method. (IRtwqAsyncResult.GetState)
IRtwqAsyncResult::GetStateNoAddRef Returns the state object specified by the caller in the asynchronous Begin method, without incrementing the object's reference count. (IRtwqAsyncResult.GetStateNoAddRef)
IRtwqAsyncResult::GetStatus Returns the status of the asynchronous operation. (IRtwqAsyncResult.GetStatus)
IRtwqAsyncResult::SetStatus Sets the status of the asynchronous operation. (IRtwqAsyncResult.SetStatus)

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8.1 [desktop apps only]
Minimum supported server	Windows Server 2012 R2 [desktop apps only]
Target Platform	Windows
Header	rtworkq.h

IRtwqAsyncResult::GetState method (rtworkq.h)

Article02/22/2024

Returns the state object specified by the caller in the asynchronous **Begin** method.

Syntax

```
C++

HRESULT GetState(
  [out] IUnknown **ppunkState
);
```

Parameters

`[out] ppunkState`

Receives a pointer to the state object's **IUnknown** interface. If the value is not **NULL**, the caller must release the interface.

Return value

The method returns an **HRESULT**. Possible values include, but are not limited to, those in the following table.

Expand table

Return code	Description
S_OK	The method succeeded.
E_POINTER	There is no state object associated with this asynchronous result.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8.1 [desktop apps only]
Minimum supported server	Windows Server 2012 R2 [desktop apps only]
Target Platform	Windows
Header	rtworkq.h
Library	Rtworkq.lib
DLL	RTWorkQ.dll

See also

IRtwqAsyncResult

IRtwqAsyncResult::GetStatus method (rtworkq.h)

Article02/22/2024

Returns the status of the asynchronous operation.

Syntax

```
C++

HRESULT GetStatus();
```

Return value

The method returns an **HRESULT**. Possible values include, but are not limited to, those in the following table.

Expand table

Return code	Description
S_OK	The operation completed successfully.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8.1 [desktop apps only]
Minimum supported server	Windows Server 2012 R2 [desktop apps only]
Target Platform	Windows
Header	rtworkq.h
Library	Rtworkq.lib
DLL	RTWorkQ.dll

See also

IRtwqAsyncResult

IRtwqAsyncResult::SetStatus method (rtworkq.h)

Article02/22/2024

Sets the status of the asynchronous operation.

Syntax

```C++
HRESULT SetStatus(
  [in] HRESULT hrStatus
);
```

Parameters

`[in] hrStatus`

The status of the asynchronous operation.

Return value

The method returns an **HRESULT**. Possible values include, but are not limited to, those in the following table.

Expand table

Return code	Description
S_OK	The method succeeded.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8.1 [desktop apps only]
Minimum supported server	Windows Server 2012 R2 [desktop apps only]

Requirement	Value
Target Platform	Windows
Header	rtworkq.h
Library	Rtworkq.lib
DLL	RTWorkQ.dll

See also

IRtwqAsyncResult

IRtwqAsyncResult::GetObject method (rtworkq.h)

Article02/22/2024

Returns an object associated with the asynchronous operation. The type of object, if any, depends on the asynchronous method that was called.

Syntax

```
C++

HRESULT GetObject(
  [out] IUnknown **ppObject
);
```

Parameters

`[out] ppObject`

Receives a pointer to the object's **IUnknown** interface. If no object is associated with the operation, this parameter receives the value **NULL**. If the value is not **NULL**, the caller must release the interface.

Return value

The method returns an **HRESULT**. Possible values include, but are not limited to, those in the following table.

Expand table

Return code	Description
S_OK	The method succeeded.
E_POINTER	There is no object associated with this asynchronous result.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8.1 [desktop apps only]
Minimum supported server	Windows Server 2012 R2 [desktop apps only]
Target Platform	Windows
Header	rtworkq.h
Library	Rtworkq.lib
DLL	RTWorkQ.dll

See also

IRtwqAsyncResult

IRtwqAsyncResult::GetStateNoAddRef method (rtworkq.h)

Article02/22/2024

Returns the state object specified by the caller in the asynchronous **Begin** method, without incrementing the object's reference count.

Syntax

```
C++

IUnknown * GetStateNoAddRef();
```

Return value

Returns a pointer to the state object's **IUnknown** interface, or **NULL** if no object was set. This pointer does not have an outstanding reference count. If you store this pointer, you must call **AddRef** on the pointer.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8.1 [desktop apps only]
Minimum supported server	Windows Server 2012 R2 [desktop apps only]
Target Platform	Windows
Header	rtworkq.h
Library	Rtworkq.lib
DLL	RTWorkQ.dll

See also

IRtwqAsyncResult

IRtwqPlatformEvents interface (rtworkq.h)

Article07/22/2021

Provides events related platform work queue.

Inheritance

The **IRtwqPlatformEvents** interface inherits from the IUnknown interface. **IRtwqPlatformEvents** also has these types of members:

Methods

The **IRtwqPlatformEvents** interface has these methods.

Expand table

IRtwqPlatformEvents::InitializationComplete Called after the platform has been initialized.
IRtwqPlatformEvents::ShutdownComplete Called after the platform has shutdown.
IRtwqPlatformEvents::ShutdownStart Called before the platform is about to shutdown.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8.1 [desktop apps only]
Minimum supported server	Windows Server 2012 R2 [desktop apps only]
Target Platform	Windows
Header	rtworkq.h

IRtwqPlatformEvents::InitializationComplete method (rtworkq.h)

Article02/22/2024

Called after the platform has been initialized.

Syntax

```
C++

HRESULT InitializationComplete();
```

Return value

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

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8.1 [desktop apps only]
Minimum supported server	Windows Server 2012 R2 [desktop apps only]
Target Platform	Windows
Header	rtworkq.h
Library	Rtworkq.lib
DLL	RTWorkQ.dll

See also

IRtwqPlatformEvents

IRtwqPlatformEvents::ShutdownStart method (rtworkq.h)

Article02/22/2024

Called before the platform is about to shutdown.

Syntax

```
C++

HRESULT ShutdownStart();
```

Return value

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

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8.1 [desktop apps only]
Minimum supported server	Windows Server 2012 R2 [desktop apps only]
Target Platform	Windows
Header	rtworkq.h
Library	Rtworkq.lib
DLL	RTWorkQ.dll

See also

IRtwqPlatformEvents

IRtwqPlatformEvents::ShutdownComplete method (rtworkq.h)

Article02/22/2024

Called after the platform has shutdown.

Syntax

```
C++

HRESULT ShutdownComplete();
```

Return value

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

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8.1 [desktop apps only]
Minimum supported server	Windows Server 2012 R2 [desktop apps only]
Target Platform	Windows
Header	rtworkq.h
Library	Rtworkq.lib
DLL	RTWorkQ.dll

See also

IRtwqPlatformEvents

RTWQ_WORKQUEUE_TYPE enumeration (rtworkq.h)

Article02/22/2024

Specifies the type of work queue for the RtwqAllocateWorkQueue function to create.

Syntax

```C++
typedef enum {
  RTWQ_STANDARD_WORKQUEUE = 0,
  RTWQ_WINDOW_WORKQUEUE = 1,
  RTWQ_MULTITHREADED_WORKQUEUE = 2
} RTWQ_WORKQUEUE_TYPE;
```

Constants

Expand table

`RTWQ_STANDARD_WORKQUEUE` Value: *0* Create a work queue without a message loop.
`RTWQ_WINDOW_WORKQUEUE` Value: *1* Create a work queue with a message loop.
`RTWQ_MULTITHREADED_WORKQUEUE` Value: *2* Create a multithreaded work queue. This type of work queue uses a thread pool to dispatch work items. The caller is responsible for serializing the work items.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8.1 [desktop apps only]

Requirement	Value
Minimum supported server	Windows Server 2012 R2 [desktop apps only]
Header	rtworkq.h

RtwqAddPeriodicCallback function (rtworkq.h)

Article07/27/2022

Sets a callback function to be called at a fixed interval.

Syntax

```C++
HRESULT RtwqAddPeriodicCallback(
  [in]            RTWQPERIODICCALLBACK Callback,
                  IUnknown             *context,
  [out, optional] DWORD                *key
);
```

Parameters

`[in] Callback`

Pointer to the callback function.

`context`

Pointer to a caller-provided object that implements IUnknown, or **NULL**. This parameter is passed to the callback function.

`[out, optional] key`

Receives a key that can be used to cancel the callback. To cancel the callback, call RtwqRemovePeriodicCallback and pass this key as the *dwKey* parameter.

Return value

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

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8.1 [desktop apps only]
Minimum supported server	Windows Server 2012 R2 [desktop apps only]
Target Platform	Windows
Header	rtworkq.h
Library	Rtworkq.lib
DLL	RTWorkQ.dll

RtwqAllocateSerialWorkQueue function (rtworkq.h)

Article10/13/2021

Creates a virtual work queue on top of another work queue that is guaranteed to serialize work items. The serial work queue wraps an existing multithreaded work queue. The serial work queue enforces a first-in, first-out (FIFO) execution order.

Syntax

```
C++

HRESULT RtwqAllocateSerialWorkQueue(
  [in]  DWORD workQueueIdIn,
  [out] DWORD *workQueueIdOut
);
```

Parameters

`[in] workQueueIdIn`

The identifier of an existing work queue. This must be either a multithreaded queue or another serial work queue. Any of the following can be used:

- The default work queue (**RTWQ_STANDARD_WORKQUEUE**). See RTWQ_WORKQUEUE_TYPE.
- The platform multithreaded queue (**RTWQ_MULTITHREADED_WORKQUEUE**). See RTWQ_WORKQUEUE_TYPE.
- A multithreaded queue returned by the RtwqLockSharedWorkQueue function.
- A serial queue created by the **RtwqAllocateSerialWorkQueue** function.

`[out] workQueueIdOut`

Receives an identifier for the new serial work queue. Use this identifier when queuing work items.

Return value

This function can return one of these values.

Expand table

Return code	Description
S_OK	The function succeeded.
E_FAIL	The application exceeded the maximum number of work queues.
RTWQ_E_SHUTDOWN	The application did not call RtwqStartup, or the application has already called RtwqShutdown.

Remarks

When you are done using the work queue, call RtwqUnlockWorkQueue.

Multithreaded queues use a thread pool, which can reduce the total number of threads in the pipeline. However, they do not serialize work items. A serial work queue enables the application to get the benefits of the thread pool, without needing to perform manual serialization of its own work items.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8.1 [desktop apps only]
Minimum supported server	Windows Server 2012 R2 [desktop apps only]
Target Platform	Windows
Header	rtworkq.h
Library	Rtworkq.lib
DLL	RTWorkQ.dll

See also

RTWQ_WORKQUEUE_TYPE

RtwqAllocateWorkQueue function (rtworkq.h)

Article02/22/2024

Creates a new work queue.

Syntax

```
C++

HRESULT RtwqAllocateWorkQueue(
  [in]  RTWQ_WORKQUEUE_TYPE WorkQueueType,
  [out] DWORD               *workQueueId
);
```

Parameters

`[in] WorkQueueType`

A member of the RTWQ_WORKQUEUE_TYPE enumeration, specifying the type of work queue to create.

Expand table

Value	Meaning
RTWQ_MULTITHREADED_WORKQUEUE	Create a multithreaded work queue. Generally, applications should not create private multithreaded queues. Use the platform multithreaded queues instead.
RTWQ_STANDARD_WORKQUEUE	Create a work queue without a message loop. Using this flag is equivalent to calling **RtwqAllocateWorkQueue**.
RTWQ_WINDOW_WORKQUEUE	Create a work queue with a message loop. The thread that dispatches the work items for this queue will also call PeekMessage and DispatchMessage. Use this option if your callback performs any actions that require a message loop.

`[out] workQueueId`

Receives an identifier for the work queue that was created.

Return value

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

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8.1 [desktop apps only]
Minimum supported server	Windows Server 2012 R2 [desktop apps only]
Target Platform	Windows
Header	rtworkq.h
Library	Rtworkq.lib
DLL	RTWorkQ.dll

RtwqBeginRegisterWorkQueueWithMMCSS function (rtworkq.h)

Article02/22/2024

Associates a work queue with a Multimedia Class Scheduler Service (MMCSS) task.

Syntax

```
C++

HRESULT RtwqBeginRegisterWorkQueueWithMMCSS(
  [in] DWORD              workQueueId,
  [in] PCWSTR             usageClass,
  [in] DWORD              dwTaskId,
  [in] LONG               lPriority,
  [in] IRtwqAsyncCallback *doneCallback,
  [in] IUnknown           *doneState
);
```

Parameters

`[in] workQueueId`

The identifier of the work queue. For private work queues, the identifier is returned by the RtwqAllocateWorkQueue function.

`[in] usageClass`

The name of the MMCSS task.

`[in] dwTaskId`

The unique task identifier. To obtain a new task identifier, set this value to zero.

`[in] lPriority`

The base relative priority for the work-queue threads. For more information, see AvSetMmThreadPriority.

`[in] doneCallback`

A pointer to the IRtwqAsyncCallback interface of a callback object. The caller must implement this interface.

`[in] doneState`

A pointer to the **IUnknown** interface of a state object, defined by the caller. This parameter can be **NULL**. You can use this object to hold state information. The object is returned to the caller when the callback is invoked.

Return value

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

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8.1 [desktop apps only]
Minimum supported server	Windows Server 2012 R2 [desktop apps only]
Target Platform	Windows
Header	rtworkq.h
Library	Rtworkq.lib
DLL	RTWorkQ.dll

RtwqBeginUnregisterWorkQueueWithMMCSS function (rtworkq.h)

Article02/22/2024

Unregisters a work queue from a Multimedia Class Scheduler Service (MMCSS) task.

Syntax

```
C++

HRESULT RtwqBeginUnregisterWorkQueueWithMMCSS(
  [in] DWORD              workQueueId,
  [in] IRtwqAsyncCallback *doneCallback,
  [in] IUnknown           *doneState
);
```

Parameters

`[in] workQueueId`

The identifier of the work queue. For private work queues, the identifier is returned by the RtwqAllocateWorkQueue function.

`[in] doneCallback`

Pointer to the IRtwqAsyncCallback interface of a callback object. The caller must implement this interface.

`[in] doneState`

Pointer to the **IUnknown** interface of a state object, defined by the caller. This parameter can be **NULL**. You can use this object to hold state information. The object is returned to the caller when the callback is invoked.

Return value

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

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8.1 [desktop apps only]
Minimum supported server	Windows Server 2012 R2 [desktop apps only]
Target Platform	Windows
Header	rtworkq.h
Library	Rtworkq.lib
DLL	RTWorkQ.dll

RtwqCancelDeadline function (rtworkq.h)

Article02/22/2024

Cancels a deadline that was previously set with RtwqSetDeadline.

Syntax

```
C++

HRESULT RtwqCancelDeadline(
  [in] HANDLE pRequest
);
```

Parameters

`[in] pRequest`

Receives a handle to the request that can be used to cancel the request by calling **RtwqCancelDeadline**.

Return value

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

Requirements

Expand table

Requirement	Value
Target Platform	Windows
Header	rtworkq.h

See also

RtwqSetDeadline

RtwqCancelWorkItem function (rtworkq.h)

Article02/22/2024

Attempts to cancel an asynchronous operation that was scheduled with RtwqScheduleWorkItem.

Syntax

```
C++

HRESULT RtwqCancelWorkItem(
  [in] RTWQWORKITEM_KEY Key
);
```

Parameters

`[in] Key`

The key that was received in the *key* parameter of the RtwqScheduleWorkItem.

Return value

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

Remarks

Because work items are asynchronous, the work-item callback might still be invoked after **RtwqCancelWorkItem** is called.

Requirements

Expand table

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

Requirement	Value
Header	rtworkq.h
Library	Rtworkq.lib
DLL	RTWorkQ.dll

RtwqCreateAsyncResult function (rtworkq.h)

Article02/22/2024

Creates an asynchronous result object. Use this function if you are implementing an asynchronous method.

Syntax

```
C++

HRESULT RtwqCreateAsyncResult(
  [in]  IUnknown           *appObject,
  [in]  IRtwqAsyncCallback *callback,
  [in]  IUnknown           *appState,
  [out] IRtwqAsyncResult   **asyncResult
);
```

Parameters

`[in] appObject`

Pointer to the object stored in the asynchronous result. This pointer is returned by the IRtwqAsyncResult::GetObject method. This parameter can be **NULL**.

`[in] callback`

Pointer to the IRtwqAsyncCallback interface. This interface is implemented by the caller of the asynchronous method.

`[in] appState`

Pointer to the **IUnknown** interface of a state object. This value is provided by the caller of the asynchronous method. This parameter can be **NULL**.

`[out] asyncResult`

Receives a pointer to the IRtwqAsyncResult interface. The caller must release the interface.

Return value

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

Remarks

To invoke the callback specified in *pCallback*, call the RtwqInvokeCallback function.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8.1 [desktop apps only]
Minimum supported server	Windows Server 2012 R2 [desktop apps only]
Target Platform	Windows
Header	rtworkq.h
Library	Rtworkq.lib
DLL	RTWorkQ.dll

RtwqEndRegisterWorkQueueWithMMCSS function (rtworkq.h)

Article02/22/2024

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

Syntax

```
C++

HRESULT RtwqEndRegisterWorkQueueWithMMCSS(
  [in]  IRtwqAsyncResult *result,
  [out] DWORD            *taskId
);
```

Parameters

`[in] result`

Pointer to the asynchronous result. Pass in the same pointer that your callback object received in the IRtwqAsyncCallback::Invoke method.

`[out] taskId`

The unique task identifier.

Return value

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

Remarks

Call this function when the RtwqBeginRegisterWorkQueueWithMMCSSEx function completes asynchronously.

To unregister the work queue from the MMCSS class, call RtwqBeginUnregisterWorkQueueWithMMCSS.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8.1 [desktop apps only]
Minimum supported server	Windows Server 2012 R2 [desktop apps only]
Target Platform	Windows
Header	rtworkq.h
Library	Rtworkq.lib
DLL	RTWorkQ.dll

RtwqEndUnregisterWorkQueueWithMMCSS function

Article • 01/07/2021

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

Syntax

```
C++

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

Feedback

Was this page helpful? Yes No

Get help at Microsoft Q&A

RtwqGetWorkQueueMMCSSClass function (rtworkq.h)

Article02/22/2024

Retrieves the Multimedia Class Scheduler Service (MMCSS) class currently associated with this work queue.

Syntax

```
C++

HRESULT RtwqGetWorkQueueMMCSSClass(
  [in]      DWORD workQueueId,
  [out]     PWSTR usageClass,
  [in, out] DWORD *usageClassLength
);
```

Parameters

`[in] workQueueId`

Identifier for the work queue. The identifier is retrieved by the RtwqAllocateWorkQueue function.

`[out] usageClass`

Pointer to a buffer that receives the name of the MMCSS class. This parameter can be **NULL**.

`[in, out] usageClassLength`

On input, specifies the size of the *usageClass* buffer, in characters. On output, receives the required size of the buffer, in characters. The size includes the terminating null character.

Return value

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

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8.1 [desktop apps only]
Minimum supported server	Windows Server 2012 R2 [desktop apps only]
Target Platform	Windows
Header	rtworkq.h
Library	Rtworkq.lib
DLL	RTWorkQ.dll

RtwqGetWorkQueueMMCSSPriority function (rtworkq.h)

Article02/22/2024

Gets the relative thread priority of a work queue.

Syntax

```
C++

HRESULT RtwqGetWorkQueueMMCSSPriority(
  [in]  DWORD workQueueId,
  [out] LONG  *priority
);
```

Parameters

`[in] workQueueId`

The identifier of the work queue. For private work queues, the identifier is returned by the RtwqAllocateWorkQueue function.

`[out] priority`

Receives the relative thread priority.

Return value

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

Remarks

This function returns the relative thread priority set by the RtwqBeginRegisterWorkQueueWithMMCSS function.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8.1 [desktop apps only]
Minimum supported server	Windows Server 2012 R2 [desktop apps only]
Target Platform	Windows
Header	rtworkq.h
Library	Rtworkq.lib
DLL	RTWorkQ.dll

RtwqGetWorkQueueMMCSSTaskId function (rtworkq.h)

Article02/22/2024

Retrieves the Multimedia Class Scheduler Service (MMCSS) task identifier currently associated with this work queue.

Syntax

```
C++

HRESULT RtwqGetWorkQueueMMCSSTaskId(
  [in]  DWORD workQueueId,
  [out] DWORD *taskId
);
```

Parameters

`[in] workQueueId`

Identifier for the work queue. The identifier is retrieved by the RtwqAllocateWorkQueue function.

`[out] taskId`

Receives the task identifier.

Return value

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

Remarks

To associate a work queue with an MMCSS task, call <a href="RtwqBeginRegisterWorkQueueWithMMCSSEx">RtwqBeginRegisterWorkQueueWithMMCSSEx.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8.1 [desktop apps only]
Minimum supported server	Windows Server 2012 R2 [desktop apps only]
Target Platform	Windows
Header	rtworkq.h
Library	Rtworkq.lib
DLL	RTWorkQ.dll

RtwqInvokeCallback function (rtworkq.h)

Article02/22/2024

Invokes a callback method to complete an asynchronous operation.

Syntax

```
C++

HRESULT RtwqInvokeCallback(
  IRtwqAsyncResult *result
);
```

Parameters

`result`

The asynchronous result. To create this object, call RtwqCreateAsyncResult.

Return value

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

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8.1 [desktop apps only]
Minimum supported server	Windows Server 2012 R2 [desktop apps only]
Target Platform	Windows
Header	rtworkq.h
Library	Rtworkq.lib
DLL	RTWorkQ.dll

RtwqJoinWorkQueue function (rtworkq.h)

Article02/22/2024

Associates a work queue with an input/output (I/O) handle.

Syntax

```
C++

HRESULT RtwqJoinWorkQueue(
  [in]  DWORD  workQueueId,
  [in]  HANDLE hFile,
  [out] HANDLE *out
);
```

Parameters

`[in] workQueueId`

The ID of the work queue to redirect the I/O handle into.

`[in] hFile`

The network I/O handle.

`[out] out`

A cookie that represents the association between the network and I/O handles.

Return value

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

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8.1 [desktop apps only]
Minimum supported server	Windows Server 2012 R2 [desktop apps only]

Requirement	Value
Target Platform	Windows
Header	rtworkq.h
Library	Rtworkq.lib
DLL	RTWorkQ.dll

RtwqLockPlatform function (rtworkq.h)

Article02/22/2024

Adds a reference to indicate to the platform that there are still pending asynchronous items. Blocks the RtwqShutdown function if there are active asynchronous items.

Syntax

```
C++

HRESULT RtwqLockPlatform();
```

Return value

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

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8.1 [desktop apps only]
Minimum supported server	Windows Server 2012 R2 [desktop apps only]
Target Platform	Windows
Header	rtworkq.h
Library	Rtworkq.lib
DLL	RTWorkQ.dll

RtwqLockSharedWorkQueue function (rtworkq.h)

Article02/22/2024

Obtains and locks a shared work queue.

Syntax

```
C++

HRESULT RtwqLockSharedWorkQueue(
  [in]      PCWSTR usageClass,
  [in]      LONG   basePriority,
  [in, out] DWORD  *taskId,
  [out]     DWORD  *id
);
```

Parameters

`[in] usageClass`

The name of the Multimedia Class Scheduler Service (MMCSS) task.

`[in] basePriority`

The base priority of the work-queue threads. If the regular-priority queue is being used (`usageClass=""`), then the value 0 must be passed in.

`[in, out] taskId`

The MMCSS task identifier. On input, specify an existing MCCSS task group ID, or use the value zero to create a new task group. If the regular priority queue is being used (`usageClass=""`), then **NULL** must be passed in. On output, receives the actual task group ID.

`[out] id`

Receives an identifier for the new work queue. Use this identifier when queuing work items.

Return value

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

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8.1 [desktop apps only]
Minimum supported server	Windows Server 2012 R2 [desktop apps only]
Target Platform	Windows
Header	rtworkq.h
Library	Rtworkq.lib
DLL	RTWorkQ.dll

RtwqLockWorkQueue function (rtworkq.h)

Article02/22/2024

Locks a work queue.

Syntax

```
C++

HRESULT RtwqLockWorkQueue(
  [in] DWORD workQueueId
);
```

Parameters

`[in] workQueueId`

The identifier for the work queue. The identifier is returned by the RtwqAllocateWorkQueue function.

Return value

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

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8.1 [desktop apps only]
Minimum supported server	Windows Server 2012 R2 [desktop apps only]
Target Platform	Windows
Header	rtworkq.h
Library	Rtworkq.lib
DLL	RTWorkQ.dll

RtwqPutWaitingWorkItem function (rtworkq.h)

Article02/22/2024

Queues a work item that waits for an event to be signaled.

Syntax

```
C++

HRESULT RtwqPutWaitingWorkItem(
  [in]            HANDLE           hEvent,
  [in]            LONG             lPriority,
  [in]            IRtwqAsyncResult *result,
  [out, optional] RTWQWORKITEM_KEY *key
);
```

Parameters

`[in] hEvent`

A handle to an event object, such as an event or timer. To create an event object, call CreateEvent or CreateEventEx.

`[in] lPriority`

The priority of the work item. Work items are performed in order of priority.

`[in] result`

A pointer to the IRtwqAsyncResult interface of an asynchronous result object. To create the result object, call RtwqCreateAsyncResult.

`[out, optional] key`

Receives a key that can be used to cancel the wait. To cancel the wait, call RtwqCancelWorkItem and pass this key in the *Key* parameter. This parameter can be **NULL**.

Return value

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

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8.1 [desktop apps only]
Minimum supported server	Windows Server 2012 R2 [desktop apps only]
Target Platform	Windows
Header	rtworkq.h
Library	Rtworkq.lib
DLL	RTWorkQ.dll

RtwqPutWorkItem function (rtworkq.h)

Article02/22/2024

Puts an asynchronous operation on a work queue.

Syntax

```
C++

HRESULT RtwqPutWorkItem(
  [in] DWORD            dwQueue,
  [in] LONG             lPriority,
  [in] IRtwqAsyncResult *result
);
```

Parameters

`[in] dwQueue`

The identifier for the work queue. This value can specify one of the standard work queues, or a work queue created by the app. To access to a work queue, call RtwqLockSharedWorkQueue.

`[in] lPriority`

The priority of the work item. Work items are performed in order of priority. This value should be -1, 0, or 1, where -1 is the lowest priority and 1 is the highest priority.

`[in] result`

A pointer to the callback . The caller must implement this interface.

Return value

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

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8.1 [desktop apps only]

Requirement	Value
Minimum supported server	Windows Server 2012 R2 [desktop apps only]
Target Platform	Windows
Header	rtworkq.h
Library	Rtworkq.lib
DLL	RTWorkQ.dll

RtwqRegisterPlatformEvents function (rtworkq.h)

Article02/22/2024

Enables an app to listen to the RtwqStartup and RtwqShutdown functions.

Syntax

```
C++

HRESULT RtwqRegisterPlatformEvents(
  [in] IRtwqPlatformEvents *platformEvents
);
```

Parameters

`[in] platformEvents`

Pointer to the IRtwqPlatformEvents object which provides the events.

Return value

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

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8.1 [desktop apps only]
Minimum supported server	Windows Server 2012 R2 [desktop apps only]
Target Platform	Windows
Header	rtworkq.h
Library	Rtworkq.lib
DLL	RTWorkQ.dll

RtwqRegisterPlatformWithMMCSS function (rtworkq.h)

Article02/22/2024

Registers the standard platform work queues with the Multimedia Class Scheduler Service (MMCSS).

Syntax

```
C++

HRESULT RtwqRegisterPlatformWithMMCSS(
  [in]      PCWSTR usageClass,
  [in, out] DWORD  *taskId,
  [in]      LONG   lPriority
);
```

Parameters

`[in] usageClass`

The name of the MMCSS task.

`[in, out] taskId`

The MMCSS task identifier. On input, specify an existing MCCSS task group ID, or use the value zero to create a new task group. On output, receives the actual task group ID.

`[in] lPriority`

The base priority of the work-queue threads.

Return value

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

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8.1 [desktop apps only]
Minimum supported server	Windows Server 2012 R2 [desktop apps only]
Target Platform	Windows
Header	rtworkq.h
Library	Rtworkq.lib
DLL	RTWorkQ.dll

RtwqRemovePeriodicCallback function (rtworkq.h)

Article02/22/2024

Cancels a callback function that was set by the RtwqAddPeriodicCallback function.

Syntax

```
C++

HRESULT RtwqRemovePeriodicCallback(
  [in] DWORD dwKey
);
```

Parameters

`[in] dwKey`

Key that identifies the callback. This value is retrieved by the RtwqAddPeriodicCallback function.

Return value

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

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8.1 [desktop apps only]
Minimum supported server	Windows Server 2012 R2 [desktop apps only]
Target Platform	Windows
Header	rtworkq.h
Library	Rtworkq.lib
DLL	RTWorkQ.dll

RtwqScheduleWorkItem function (rtworkq.h)

Article02/22/2024

Schedules an asynchronous operation to be completed after a specified interval.

Syntax

```
C++

HRESULT RtwqScheduleWorkItem(
  [in]            IRtwqAsyncResult *result,
  [in]            INT64            Timeout,
  [out, optional] RTWQWORKITEM_KEY *key
);
```

Parameters

`[in] result`

A pointer to the callback. The caller must implement this interface.

`[in] Timeout`

Time-out interval, in milliseconds. Set this parameter to a negative value. The callback is invoked after −*Timeout* milliseconds. For example, if *Timeout* is −5000, the callback is invoked after 5000 milliseconds.

`[out, optional] key`

Receives a key that can be used to cancel the timer. To cancel the wait, call RtwqCancelWorkItem and pass this key in the *Key* parameter.

Return value

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

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8.1 [desktop apps only]
Minimum supported server	Windows Server 2012 R2 [desktop apps only]
Target Platform	Windows
Header	rtworkq.h
Library	Rtworkq.lib
DLL	RTWorkQ.dll

RtwqSetDeadline function (rtworkq.h)

Article02/22/2024

Sets a deadline by which the work in a work queue must be completed.

Syntax

```
C++

HRESULT RtwqSetDeadline(
  [in]      DWORD    workQueueId,
  [in]      LONGLONG deadlineInHNS,
  [in, out] HANDLE   *pRequest
);
```

Parameters

`[in] workQueueId`

The identifier for the work queue. The identifier is returned by the RtwqAllocateWorkQueue function.

`[in] deadlineInHNS`

The deadline for the work in the queue to be completed, in hundred-nanosecond units. For example, if `deadlineInHNS` is 9600, that represents 9600 hundred-nanoseconds, which is equal to 960 microseconds, or 0.96 milliseconds.

`[in, out] pRequest`

Receives a handle to the request that can be used to cancel the request by calling RtwqCancelDeadline.

Return value

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

Remarks

Update a deadline by creating a new deadline and releasing the old one.

Cancel a deadline by calling RtwqCancelDeadline.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 10 [desktop apps only]
Minimum supported server	Windows Server 2016 [desktop apps only]
Target Platform	Windows
Header	rtworkq.h
Library	Rtworkq.lib
DLL	RTWorkQ.dll

See also

RtwqCancelDeadline

RtwqSetDeadline2 function (rtworkq.h)

Article02/22/2024

Sets a deadline by which the work in a work queue must be completed.

Syntax

```
C++

HRESULT RtwqSetDeadline2(
  [in]      DWORD    workQueueId,
  [in]      LONGLONG deadlineInHNS,
  [in]      LONGLONG preDeadlineInHNS,
  [in, out] HANDLE   *pRequest
);
```

Parameters

`[in] workQueueId`

The identifier for the work queue. The identifier is returned by the RtwqAllocateWorkQueue function.

`[in] deadlineInHNS`

The deadline for the work in the queue to be completed, in hundred-nanosecond units. For example, if `deadlineInHNS` is 9600, that represents 9600 hundred-nanoseconds, which is equal to 960 microseconds, or 0.96 milliseconds.

`[in] preDeadlineInHNS`

The pre-deadline for the work in the queue to be completed, in hundred-nanosecond units. For example, if `preDeadlineInHNS` is 9600, that represents 9600 hundred-nanoseconds, which is equal to 960 microseconds, or 0.96 milliseconds.

`[in, out] pRequest`

Receives a handle to the request that can be used to cancel the request by calling RtwqCancelDeadline.

Return value

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

Remarks

Update a deadline by creating a new deadline and releasing the old one.

Cancel a deadline by calling RtwqCancelDeadline.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 10 [desktop apps only]
Minimum supported server	Windows Server 2016 [desktop apps only]
Target Platform	Windows
Header	rtworkq.h
Library	Rtworkq.lib
DLL	RTWorkQ.dll

See also

RtwqCancelDeadline

RtwqSetLongRunning function (rtworkq.h)

Article02/22/2024

Indicates that the app will be submitting a hint that long running work will occur on this work queue.

Syntax

```
C++

HRESULT RtwqSetLongRunning(
  [in] DWORD workQueueId,
  [in] BOOL  enable
);
```

Parameters

`[in] workQueueId`

The ID of the work queue.

`[in] enable`

true if the app will be submitting the hint; otherwise, **false**. The default is **false**.

Return value

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

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8.1 [desktop apps only]
Minimum supported server	Windows Server 2012 R2 [desktop apps only]
Target Platform	Windows
Header	rtworkq.h

Requirement	Value
Library	Rtworkq.lib
DLL	RTWorkQ.dll

RtwqShutdown function (rtworkq.h)

Article02/22/2024

Shuts down the platform. Call this function once for every call to RtwqStartup. Do not call this function from work queue threads.

Syntax

```
C++

HRESULT RtwqShutdown();
```

Return value

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

Remarks

In general, apps should not have pending work after they call this function.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8.1 [desktop apps only]
Minimum supported server	Windows Server 2012 R2 [desktop apps only]
Target Platform	Windows
Header	rtworkq.h
Library	Rtworkq.lib
DLL	RTWorkQ.dll

RtwqStartup function (rtworkq.h)

Article02/22/2024

Initializes the platform.

Syntax

```
C++

HRESULT RtwqStartup();
```

Return value

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

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8.1 [desktop apps only]
Minimum supported server	Windows Server 2012 R2 [desktop apps only]
Target Platform	Windows
Header	rtworkq.h
Library	Rtworkq.lib
DLL	RTWorkQ.dll

RtwqUnjoinWorkQueue function (rtworkq.h)

Article02/22/2024

Disassociates a work queue from an input/output (I/O) handle.

Syntax

```C++
HRESULT RtwqUnjoinWorkQueue(
  [in] DWORD  workQueueId,
  [in] HANDLE hFile
);
```

Parameters

`[in] workQueueId`

The ID of the work queue to disassociate.

`[in] hFile`

The associated handle returned by the RtwqJoinWorkQueue function.

Return value

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

Requirements

Expand table

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

Requirement	Value
Header	rtworkq.h
Library	Rtworkq.lib
DLL	RTWorkQ.dll

RtwqUnlockPlatform function (rtworkq.h)

Article02/22/2024

Unlocks the platform after it was locked by a call to the RtwqLockPlatform function.

Syntax

```
C++

HRESULT RtwqUnlockPlatform();
```

Return value

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

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8.1 [desktop apps only]
Minimum supported server	Windows Server 2012 R2 [desktop apps only]
Target Platform	Windows
Header	rtworkq.h
Library	Rtworkq.lib
DLL	RTWorkQ.dll

RtwqUnlockWorkQueue function (rtworkq.h)

Article02/22/2024

Unlocks a work queue.

Syntax

```
C++

HRESULT RtwqUnlockWorkQueue(
  [in] DWORD workQueueId
);
```

Parameters

`[in] workQueueId`

Identifier for the work queue to be unlocked. This identifier is returned by the RtwqAllocateSerialWorkQueue, RtwqAllocateWorkQueue, or RtwqLockSharedWorkQueue functions.

Return value

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

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8.1 [desktop apps only]
Minimum supported server	Windows Server 2012 R2 [desktop apps only]
Target Platform	Windows
Header	rtworkq.h
Library	Rtworkq.lib

Requirement	Value
DLL	RTWorkQ.dll

RtwqUnregisterPlatformEvents function (rtworkq.h)

Article02/22/2024

Unregisters a listener event from the callback platform.

Syntax

```C++
HRESULT RtwqUnregisterPlatformEvents(
  IRtwqPlatformEvents *platformEvents
);
```

Parameters

`platformEvents`

Pointer to the IRtwqPlatformEvents object which provides the events.

Return value

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

Remarks

The app should use the same pointer that was passed to RtwqRegisterPlatformEvents.

Requirements

Expand table

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

Requirement	Value
Header	rtworkq.h
Library	Rtworkq.lib
DLL	RTWorkQ.dll

RtwqUnregisterPlatformFromMMCSS function (rtworkq.h)

Article02/22/2024

Unregisters the platform work queues from a Multimedia Class Scheduler Service (MMCSS) task.

Syntax

```
C++

HRESULT RtwqUnregisterPlatformFromMMCSS();
```

Return value

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

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8.1 [desktop apps only]
Minimum supported server	Windows Server 2012 R2 [desktop apps only]
Target Platform	Windows
Header	rtworkq.h
Library	Rtworkq.lib
DLL	RTWorkQ.dll

Processor Groups

Article • 12/30/2021

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

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Quality of Service

Article • 07/15/2022

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.

Expand table

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

QoS level	Description	Performance and power	Release
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
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.

Expand table

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.

Source	Description
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: Expand table **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.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Multiple Threads

Article • 01/07/2021

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

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Thread Stack Size

Article • 01/07/2021

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.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Thread Handles and Identifiers

Article • 01/07/2021

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.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Suspending Thread Execution

Article • 01/07/2021

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.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Synchronizing Execution of Multiple Threads

Article • 01/07/2021

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.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Multiple Threads and GDI Objects

Article • 01/07/2021

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.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Thread Local Storage

Article • 06/20/2022

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

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Creating Windows in Threads

Article • 01/07/2021

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.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Terminating a Thread

Article • 01/07/2021

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).

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Thread Security and Access Rights

Article • 01/07/2021

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.

Expand table

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.

Expand table

Value	Meaning
SYNCHRONIZE (0x00100000L)	Enables the use of the thread handle in any of the wait functions.
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**

Value	Meaning
	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**.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Child Processes

Article • 01/07/2021

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

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Setting Window Properties Using STARTUPINFO

Article • 01/07/2021

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.

Feedback

Was this page helpful? Yes No

Process Handles and Identifiers

Article • 01/07/2021

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.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Process Enumeration

Article • 01/07/2021

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.

Expand table

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.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Obtaining Additional Process Information

Article • 01/07/2021

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.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Inheritance

Article • 10/20/2020

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.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Environment Variables

Article • 01/07/2021

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

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Terminating a Process

Article • 01/07/2021

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:

syntax

```
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.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Process Working Set

Article • 06/04/2021

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

Feedback

Was this page helpful? Yes No

Process Security and Access Rights

Article • 01/07/2022

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.

Expand table

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.

Value	Meaning
WRITE_OWNER (0x00080000L)	Required to change the owner in the security descriptor for the object.

The following table lists the process-specific access rights.

Expand table

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**

Value	Meaning
	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**.
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**.

Feedback

Was this page helpful? Yes No

Thread Pools

Article • 01/07/2021

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

Using the Thread Pool Functions

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Thread Pool API

Article • 01/07/2021

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.

Expand table

Feature	Original API	Current API
Synch	RegisterWaitForSingleObject UnregisterWaitEx	CloseThreadpoolWait CreateThreadpoolWait SetThreadpoolWait WaitForThreadpoolWaitCallbacks
Work	QueueUserWorkItem	CloseThreadpoolWork CreateThreadpoolWork SubmitThreadpoolWork TrySubmitThreadpoolCallback WaitForThreadpoolWorkCallbacks

Feature	Original API	Current API
Timer	CreateTimerQueue CreateTimerQueueTimer ChangeTimerQueueTimer DeleteTimerQueueTimer DeleteTimerQueueEx	CloseThreadpoolTimer CreateThreadpoolTimer IsThreadpoolTimerSet SetThreadpoolTimer WaitForThreadpoolTimerCallbacks
I/O	BindIoCompletionCallback	CancelThreadpoolIo CloseThreadpoolIo CreateThreadpoolIo StartThreadpoolIo WaitForThreadpoolIoCallbacks
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

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Thread Pooling

Article • 01/07/2021

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

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Job Objects

Article • 01/07/2021

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
- Managing Processes in Jobs
- Job Limits and Notifications
- Resource Accounting for Jobs
- Managing Job Objects
- Managing a Process Tree that Uses Job Objects

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.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Nested Jobs

Article • 01/07/2021

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.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Job Object Security and Access Rights

Article • 01/25/2021

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.

Expand table

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.

Expand table

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.
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.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

CPU Sets

Article • 12/30/2021

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

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Fibers

Article • 04/12/2022

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

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

User-Mode Scheduling

Article • 03/17/2022

> ⚠ **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.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Using Processes and Threads

Article • 01/07/2021

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

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Create processes

Article • 02/09/2023

The CreateProcess function creates a new process that runs independently of the creating process. For simplicity, this relationship is called a parent-child relationship.

The following code demonstrates how to create a process.

C++

```
#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 that contains handles and identifiers for the new process and its primary thread. The thread and process handles are created with full access rights, although you can restrict access 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 by using the CreateProcessAsUser or CreateProcessWithLogonW functions. These functions let you specify the security context of the user account in which the process runs.

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Creating Threads

Article • 12/08/2022

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.

C++

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

#define MAX_THREADS 3
#define BUF_SIZE 255

DWORD WINAPI MyThreadFunction( LPVOID lpParam );
void ErrorHandler(LPCTSTR 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.

        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(LPCTSTR 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

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Creating a Child Process with Redirected Input and Output

Article • 02/01/2021

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.

C++

```
#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(PCTSTR);

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(PCTSTR 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.

C++

```
#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;
}
```

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Isolated User Mode (IUM) Processes

Article • 01/07/2021

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.

C++

```
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.

C++

```
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;
```


Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Changing Environment Variables

Article • 01/07/2021

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.

C++

```
#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.

C++

```
#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.

C++

```
#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;
}
```

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Using Thread Local Storage

Article • 01/07/2021

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.

C++

```
#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

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Using Fibers

Article • 01/07/2021

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.

C++

```
#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

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Using the Thread Pool Functions

Article • 01/07/2021

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

C++

```
#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

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Process and Thread Reference

Article • 01/07/2021

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

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

Process and Thread Enumerations

Article • 10/20/2020

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

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

DISPATCHERQUEUE_THREAD_APARTMENTTYPE enumeration (dispatcherqueue.h)

Article02/22/2024

Defines constants that specify options around type of COM apartment for a new DispatcherQueueController.

Syntax

C++

```
typedef enum DISPATCHERQUEUE_THREAD_APARTMENTTYPE {
  DQTAT_COM_NONE = 0,
  DQTAT_COM_ASTA = 1,
  DQTAT_COM_STA = 2
} ;
```

Constants

Expand table

`DQTAT_COM_NONE` Value: *0* No COM threading apartment type specified.
`DQTAT_COM_ASTA` Value: *1* Specifies an application single-threaded apartment (ASTA) COM threading apartment.
`DQTAT_COM_STA` Value: *2* Specifies a single-threaded apartment (STA) COM threading apartment.

Remarks

This value is relevant when DispatcherQueueOptions.threadType is **DQTYPE_THREAD_DEDICATED**. Use **DQTAT_COM_NONE** when **DispatcherQueueOptions.threadType** is **DQTYPE_THREAD_CURRENT**.

Also see Remarks for CreateDispatcherQueueController.

Introduced in Windows 10, version 1709.

Requirements

Expand table

Requirement	Value
Header	dispatcherqueue.h

See also

CreateDispatcherQueueController

DispatcherQueueOptions

DISPATCHERQUEUE_THREAD_TYPE enumeration (dispatcherqueue.h)

Article02/22/2024

Defines constants that specify options around threading affinity for a new DispatcherQueueController.

Syntax

C++

```
typedef enum DISPATCHERQUEUE_THREAD_TYPE {
  DQTYPE_THREAD_DEDICATED = 1,
  DQTYPE_THREAD_CURRENT = 2
} ;
```

Constants

Expand table

`DQTYPE_THREAD_DEDICATED` Value: *1* Specifies that the DispatcherQueue for the DispatcherQueueController will be created on a dedicated thread.
`DQTYPE_THREAD_CURRENT` Value: *2* Specifies that the DispatcherQueue for the DispatcherQueueController will be created on the caller's thread.

Remarks

See Remarks for CreateDispatcherQueueController.

Introduced in Windows 10, version 1709.

Requirements

Expand table

Requirement	Value
Header	dispatcherqueue.h

See also

CreateDispatcherQueueController

DispatcherQueueOptions

CPU_SET_INFORMATION_TYPE enumeration

Article • 01/07/2021

Represents the type of information in the SYSTEM_CPU_SET_INFORMATION structure.

Syntax

C++

```
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)

Feedback

Was this page helpful? Yes No

Get help at Microsoft Q&A

LOGICAL_PROCESSOR_RELATIONSHIP enumeration (winnt.h)

Article03/06/2023

Represents the relationship between the processor set identified in the corresponding SYSTEM_LOGICAL_PROCESSOR_INFORMATION_EX structure.

Syntax

C++

```
typedef enum _LOGICAL_PROCESSOR_RELATIONSHIP {
  RelationProcessorCore,
  RelationNumaNode,
  RelationCache,
  RelationProcessorPackage,
  RelationGroup,
  RelationProcessorDie,
  RelationNumaNodeEx,
  RelationProcessorModule,
  RelationAll = 0xffff
} LOGICAL_PROCESSOR_RELATIONSHIP;
```

Constants

Expand table

`RelationProcessorCore` The specified logical processors share a single processor core.
`RelationNumaNode` The specified logical processors are part of the same NUMA node.
`RelationCache` The specified logical processors share a cache. **Windows Server 2003:** This value is not supported until Windows Server 2003 with SP1 and Windows XP Professional x64 Edition.
`RelationProcessorPackage` The specified logical processors share a physical package (a single package socketed or soldered onto a motherboard may contain multiple processor cores or threads, each of which is treated as

a separate processor by the operating system). **Windows Server 2003:** This value is not supported until Windows Server 2003 with SP1 and Windows XP Professional x64 Edition.	
`RelationGroup` The specified logical processors share a single processor group. **Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP Professional x64 Edition:** This value is not supported until Windows Server 2008 R2.	
`RelationProcessorDie` The specified logical processors share a single processor die.	
`RelationNumaNodeEx` **Introduced in Windows Server 2022 (21H2, build 20348).** Requests that the full affinity be returned. Unlike the other relation types, **RelationNumaNodeEx** is not used on input. It is simply a request for **RelationNumaNode** with full group information.	
`RelationAll` Value: *0xffff* On input, retrieves information about all possible relationship types. This value is not used on output. **Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP Professional x64 Edition:** This value is not supported until Windows Server 2008 R2.	

Remarks

The value specified by this enumeration indicates the relationship represented in the corresponding SYSTEM_LOGICAL_PROCESSOR_INFORMATION_EX structure.

Examples

For an example, see GetLogicalProcessorInformation.

Requirements

Expand table

Requirement	Value
Minimum supported	Windows Vista, Windows XP Professional x64 Edition [desktop apps

Requirement	Value
client	only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	winnt.h (include Windows.h)

See also

GetLogicalProcessorInformation

GetLogicalProcessorInformationEx

SYSTEM_LOGICAL_PROCESSOR_INFORMATION

SYSTEM_LOGICAL_PROCESSOR_INFORMATION_EX

Feedback

Was this page helpful? Yes No

JOB_OBJECT_NET_RATE_CONTROL_FLAGS enumeration (winnt.h)

Article01/31/2022

Specifies types of scheduling policies for network rate control.

Syntax

```
C++

typedef enum JOB_OBJECT_NET_RATE_CONTROL_FLAGS {
  JOB_OBJECT_NET_RATE_CONTROL_ENABLE = 0x1,
  JOB_OBJECT_NET_RATE_CONTROL_MAX_BANDWIDTH = 0x2,
  JOB_OBJECT_NET_RATE_CONTROL_DSCP_TAG = 0x4,
  JOB_OBJECT_NET_RATE_CONTROL_VALID_FLAGS = =
} ;
```

Constants

Expand table

`JOB_OBJECT_NET_RATE_CONTROL_ENABLE` Value: *0x1* Turns on the control of the network traffic. You must set this value if you also set either **JOB_OBJECT_NET_RATE_CONTROL_MAX_BANDWIDTH** or **JOB_OBJECT_NET_RATE_CONTROL_DSCP_TAG**.
`JOB_OBJECT_NET_RATE_CONTROL_MAX_BANDWIDTH` Value: *0x2*
`JOB_OBJECT_NET_RATE_CONTROL_DSCP_TAG` Value: *0x4* Sets the DSCP field in the packet header to the value of the **DscpTag** member of the JOBOBJECT_NET_RATE_CONTROL_INFORMATION structure. For information about DSCP, see Differentiated Services.
`JOB_OBJECT_NET_RATE_CONTROL_VALID_FLAGS` Value: = The combination of all of the valid flags for the JOB_OBJECT_NET_RATE_CONTROL_FLAGS enumeration.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 10 [desktop apps only]
Minimum supported server	Windows Server 2016 [desktop apps only]
Header	winnt.h

See also

JOBOBJECT_NET_RATE_CONTROL_INFORMATION

PROCESS_MEMORY_EXHAUSTION_TYPE enumeration (processthreadsapi.h)

Article02/22/2024

Represents the different memory exhaustion types.

Syntax

C++

```
typedef enum _PROCESS_MEMORY_EXHAUSTION_TYPE {
  PMETypeFailFastOnCommitFailure,
  PMETypeMax
} PROCESS_MEMORY_EXHAUSTION_TYPE, *PPROCESS_MEMORY_EXHAUSTION_TYPE;
```

Constants

Expand table

`PMETypeFailFastOnCommitFailure` Anytime memory management fails an allocation due to an inability to commit memory, it will cause the process to trigger a Windows Error Reporting report and then terminate immediately with **STATUS_COMMITMENT_LIMIT**. The failure cannot be caught and handled by the app.
`PMETypeMax` The maximum value for this enumeration. This value may change in a future version.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 10, version 1511 [desktop apps | UWP apps]
Minimum supported server	Windows Server 2016 [desktop apps | UWP apps]
Header	processthreadsapi.h (include Windows.h)

Feedback

Was this page helpful? Yes No

PROCESS_MITIGATION_POLICY enumeration (winnt.h)

Article05/27/2022

Represents the different process mitigation policies.

Syntax

C++

```
typedef enum _PROCESS_MITIGATION_POLICY {
  ProcessDEPPolicy,
  ProcessASLRPolicy,
  ProcessDynamicCodePolicy,
  ProcessStrictHandleCheckPolicy,
  ProcessSystemCallDisablePolicy,
  ProcessMitigationOptionsMask,
  ProcessExtensionPointDisablePolicy,
  ProcessControlFlowGuardPolicy,
  ProcessSignaturePolicy,
  ProcessFontDisablePolicy,
  ProcessImageLoadPolicy,
  ProcessSystemCallFilterPolicy,
  ProcessPayloadRestrictionPolicy,
  ProcessChildProcessPolicy,
  ProcessSideChannelIsolationPolicy,
  ProcessUserShadowStackPolicy,
  ProcessRedirectionTrustPolicy,
  ProcessUserPointerAuthPolicy,
  ProcessSEHOPPolicy,
  MaxProcessMitigationPolicy
} PROCESS_MITIGATION_POLICY, *PPROCESS_MITIGATION_POLICY;
```

Constants

Expand table

`ProcessDEPPolicy` The data execution prevention (DEP) policy of the process.
`ProcessASLRPolicy` The Address Space Layout Randomization (ASLR) policy of the process.

`ProcessDynamicCodePolicy` The policy that turns off the ability of the process to generate dynamic code or modify existing executable code.
`ProcessStrictHandleCheckPolicy` The process will receive a fatal error if it manipulates an invalid handle. Useful for preventing downstream problems in a process due to handle misuse.
`ProcessSystemCallDisablePolicy` Disables the ability to use NTUser/GDI functions at the lowest layer.
`ProcessMitigationOptionsMask` Returns the mask of valid bits for all the mitigation options on the system. An application can set many mitigation options without querying the operating system for mitigation options by combining bitwise with the mask to exclude all non-supported bits at once.
`ProcessExtensionPointDisablePolicy` The policy that prevents some built-in third party extension points from being turned on, which prevents legacy extension point DLLs from being loaded into the process.
`ProcessControlFlowGuardPolicy` The Control Flow Guard (CFG) policy of the process.
`ProcessSignaturePolicy` The policy of a process that can restrict image loading to those images that are either signed by Microsoft, by the Windows Store, or by Microsoft, the Windows Store and the Windows Hardware Quality Labs (WHQL).
`ProcessFontDisablePolicy` The policy that turns off the ability of the process to load non-system fonts.
`ProcessImageLoadPolicy` The policy that turns off the ability of the process to load images from some locations, such a remote devices or files that have the low mandatory label.
`ProcessSystemCallFilterPolicy` The system call filter policy of the process.
`ProcessPayloadRestrictionPolicy` The payload restriction policy of the process.
`ProcessChildProcessPolicy` The child process policy of the process.
`ProcessSideChannelIsolationPolicy` The side channel isolation policy of the process.
`ProcessUserShadowStackPolicy` Windows 10, version 2004 and above: The policy regarding user-mode Hardware-enforced Stack

	Protection for the process.
`ProcessRedirectionTrustPolicy`	The RedirectionGuard policy of the process.
`ProcessUserPointerAuthPolicy`	The user pointer authentication policy of the process.
`ProcessSEHOPPolicy`	The Structured Exception Handling Overwrite Protection (SEHOP) policy of the process.
`MaxProcessMitigationPolicy`	Ends the enumeration.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]
Header	winnt.h (include Windows.h)

See also

GetProcessMitigationPolicy function, SetProcessMitigationPolicy function

Feedback

Was this page helpful? Yes No

PROCESSOR_CACHE_TYPE enumeration (winnt.h)

Article02/22/2024

Represents the type of processor cache identified in the corresponding CACHE_DESCRIPTOR structure.

Syntax

C++

```
typedef enum _PROCESSOR_CACHE_TYPE {
  CacheUnified,
  CacheInstruction,
  CacheData,
  CacheTrace,
  CacheUnknown
} PROCESSOR_CACHE_TYPE, *PPROCESSOR_CACHE_TYPE;
```

Constants

Expand table

`CacheUnified` The cache is unified.
`CacheInstruction` The cache is for processor instructions.
`CacheData` The cache is for data.
`CacheTrace` The cache is for traces.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista, Windows XP Professional x64 Edition [desktop apps only]
Minimum supported server	Windows Server 2008, Windows Server 2003 with SP1 [desktop apps only]
Header	winnt.h (include Windows.h)

See also

CACHE_DESCRIPTOR

GetLogicalProcessorInformation

SYSTEM_LOGICAL_PROCESSOR_INFORMATION

Feedback

Was this page helpful? Yes No

UMS_SYSTEM_THREAD_INFORMATION structure (winbase.h)

Article02/22/2024

Specifies a UMS scheduler thread, UMS worker thread, or non-UMS thread.

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

Syntax

C++

```
typedef struct _UMS_SYSTEM_THREAD_INFORMATION {
  ULONG UmsVersion;
  union {
    struct {
      ULONG IsUmsSchedulerThread : 1;
      ULONG IsUmsWorkerThread : 1;
    } DUMMYSTRUCTNAME;
    ULONG ThreadUmsFlags;
  } DUMMYUNIONNAME;
} UMS_SYSTEM_THREAD_INFORMATION, *PUMS_SYSTEM_THREAD_INFORMATION;
```

Members

`UmsVersion`

The UMS version.

You must set this member to UMS_VERSION before calling the GetUmsSystemThreadInformation function.

`DUMMYUNIONNAME`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.IsUmsSchedulerThread`

A bitfield that specifies that the thread is a UMS scheduler thread.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.IsUmsWorkerThread`

A bitfield that specifies that the thread is a UMS worker thread.

`DUMMYUNIONNAME.ThreadUmsFlags`

Remarks

Used by the GetUmsSystemThreadInformation function.

At most one of **IsUmsSchedulerThread** and **IsUmsWorkerThread** will be set.

If both **IsUmsSchedulerThread** and **IsUmsWorkerThread** are clear then the thread is a non-UMS thread.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 with SP1 [desktop apps only],Windows 7 (64-bit only) and Windows Server 2008 R2 (64-bit only) with KB977165 installed
Minimum supported server	Windows Server 2008 R2 with SP1 [desktop apps only]
Header	winbase.h (include Windows.h)

Feedback

Was this page helpful? Yes No

Process and Thread Functions

Article • 04/12/2022

This topic describes the process and thread functions.

- Dispatch Queue Function
- Process Functions
- Process Enumeration Functions
- Policy Functions
- Thread Functions
- Process and Thread Extended Attribute Functions
- WOW64 Functions
- Job Object Functions
- Thread Pool Functions
- Thread Ordering Service Functions
- Multimedia Class Scheduler Service Functions
- Fiber Functions
- NUMA Support Functions
- Processor Functions
- User-Mode Scheduling Functions
- Obsolete Functions

Dispatch Queue Function

The following function creates a DispatcherQueueController.

Expand table

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.

Expand table

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.
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.

Function	Description
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.
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.

Function	Description
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.
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.

Expand table

Function	Description
EnumProcesses	Retrieves the process identifier for each process object in the system.

Function	Description
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.

Expand table

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.

Expand table

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.

Function	Description
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.
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.

Function	Description
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.
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.

Expand table

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.

Expand table

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.

Expand table

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.

Expand table

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.

Function	Description
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.
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.

Function	Description
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.
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.

Function	Description
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.
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.

Expand table

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.

Expand table

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.

Expand table

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.

Expand table

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.
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.

Function	Description
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.

Expand table

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.
GetNumaProcessorNode	Retrieves the node number for the specified processor.
GetNumaProcessorNodeEx	Retrieves the node number of the specified logical processor as a USHORT value.

Function	Description
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.

Expand table

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.

Function	Description
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).

Expand table

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.

Function	Description
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

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

AssignProcessToJobObject function (jobapi2.h)

Article10/13/2021

Assigns a process to an existing job object.

Syntax

```
C++

BOOL AssignProcessToJobObject(
  [in] HANDLE hJob,
  [in] HANDLE hProcess
);
```

Parameters

`[in] hJob`

A handle to the job object to which the process will be associated. The CreateJobObject or OpenJobObject function returns this handle. The handle must have the JOB_OBJECT_ASSIGN_PROCESS access right. For more information, see Job Object Security and Access Rights.

`[in] hProcess`

A handle to the process to associate with the job object. The handle must have the PROCESS_SET_QUOTA and PROCESS_TERMINATE access rights. For more information, see Process Security and Access Rights.

If the process is already associated with a job, the job specified by *hJob* must be empty or it must be in the hierarchy of nested jobs to which the process already belongs, and it cannot have UI limits set (SetInformationJobObject with **JobObjectBasicUIRestrictions**). For more information, see Remarks.

Windows 7, Windows Server 2008 R2, Windows XP with SP3, Windows Server 2008, Windows Vista and Windows Server 2003: The process must not already be assigned to a job; if it is, the function fails with ERROR_ACCESS_DENIED. This behavior changed starting in Windows 8 and Windows Server 2012.

Terminal Services: All processes within a job must run within the same session as the job.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

After you associate a process with a job object using **AssignProcessToJobObject**, the process is subject to the limits set for the job. To set limits for a job, use the SetInformationJobObject function.

If the job has a user-mode time limit, and the time limit has been exhausted, **AssignProcessToJobObject** fails and the specified process is terminated. If the time limit would be exceeded by associating the process, **AssignProcessToJobObject** still succeeds. However, the time limit violation will be reported. If the job has an active process limit, and the limit would be exceeded by associating this process, **AssignProcessToJobObject** fails, and the specified process is terminated.

Memory operations performed by a process associated with a job that has a memory limit are subject to the memory limit. Memory operations performed by the process before it was associated with the job are not examined by **AssignProcessToJobObject**.

If the process is already running and the job has security limitations, **AssignProcessToJobObject** may fail. For example, if the primary token of the process contains the local administrators group, but the job object has the security limitation JOB_OBJECT_SECURITY_NO_ADMIN, the function fails. If the job has the security limitation JOB_OBJECT_SECURITY_ONLY_TOKEN, the process must be created suspended. To create a suspended process, call the CreateProcess function with the CREATE_SUSPENDED flag.

A process can be associated with more than one job in a hierarchy of nested jobs. For priority class, affinity, commit charge, per-process execution time limit, scheduling class limit, and working set minimum and maximum, the process inherits an effective limit which is the most restrictive limit of all the jobs in its parent job chain. For other resource limits, the process inherits limits from its immediate job in the hierarchy. Accounting information is added to the immediate job and aggregated in each parent job in the job chain. By default, all child processes are associated with the immediate job and every job in the parent job chain. To create a child process that is not part of the same job chain, call the CreateProcess function with the CREATE_BREAKAWAY_FROM_JOB flag. The child process breaks away from every job in the job chain unless a job in the chain does not allow breakaway. In this case, the child process

does not break away from that job or any job above it in the job chain. 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 only with a single job. A process inherits limits from the job it is associated with and adds its accounting information to the job. If a process is associated with a job, all child processes it creates are associated with that job by default. To create a child process that is not part of the same job, call the CreateProcess function with the CREATE_BREAKAWAY_FROM_JOB flag. A process can be associated with more than one job starting in Windows 8 and Windows Server 2012.

Windows 7, Windows Server 2008 R2, Windows Server 2008 and Windows Vista: If the process is being monitored by the Program Compatibility Assistant (PCA), it is placed into a compatibility job. Therefore, the process must be created using CREATE_BREAKAWAY_FROM_JOB before it can be placed in another job. Alternatively, you can embed an application manifest that specifies a User Account Control (UAC) level in your application and PCA will not add the process to the compatibility job. For more information, see Application Development Requirements for User Account Control Compatibility.

If the job or any of its parent jobs in the job chain is terminating when **AssignProcessToJob** is called, the function fails.

To compile an application that uses this function, define _WIN32_WINNT as 0x0500 or later. For more information, see Using the Windows Headers.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	jobapi2.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

CreateJobObject

CreateProcess

Job Objects

OpenJobObject

Process and Thread Functions

SetInformationJobObject

AttachThreadInput function (winuser.h)

10/13/2021

Attaches or detaches the input processing mechanism of one thread to that of another thread.

Syntax

```C++
BOOL AttachThreadInput(
  [in] DWORD idAttach,
  [in] DWORD idAttachTo,
  [in] BOOL  fAttach
);
```

Parameters

`[in] idAttach`

The identifier of the thread to be attached to another thread. The thread to be attached cannot be a system thread.

`[in] idAttachTo`

The identifier of the thread to which *idAttach* will be attached. This thread cannot be a system thread.

A thread cannot attach to itself. Therefore, *idAttachTo* cannot equal *idAttach*.

`[in] fAttach`

If this parameter is **TRUE**, the two threads are attached. If the parameter is **FALSE**, the threads are detached.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Windows Server 2003 and Windows XP: There is no extended error information; do not call GetLastError. This behavior changed as of Windows Vista.

Remarks

By using the **AttachThreadInput** function, a thread can share its input states (such as keyboard states and the current focus window) with another thread. Keyboard and mouse events received by both threads are processed in the order they were received until the threads are detached by calling **AttachThreadInput** a second time and specifying **FALSE** for the *fAttach* parameter.

The **AttachThreadInput** function fails if either of the specified threads does not have a message queue. The system creates a thread's message queue when the thread makes its first call to one of the USER or GDI functions. The **AttachThreadInput** function also fails if a journal record hook is installed. Journal record hooks attach all input queues together.

Note that key state, which can be ascertained by calls to the GetKeyState or GetKeyboardState function, is reset after a call to **AttachThreadInput**. You cannot attach a thread to a thread in another desktop.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-misc-l1-2-0 (introduced in Windows 8.1)

See also

GetCurrentThreadId

GetKeyState

GetKeyboardState

GetWindowThreadProcessId

Process and Thread Functions

SetFocus

Threads

AvQuerySystemResponsiveness function (avrt.h)

Article02/22/2024

Retrieves the system responsiveness setting used by the multimedia class scheduler service.

Syntax

```
C++

AVRTAPI BOOL AvQuerySystemResponsiveness(
  [in]  HANDLE AvrtHandle,
  [out] PULONG SystemResponsivenessValue
);
```

Parameters

`[in] AvrtHandle`

A handle to the task. This handle is returned by the AvSetMmThreadCharacteristics or AvSetMmMaxThreadCharacteristics function.

`[out] SystemResponsivenessValue`

The system responsiveness value. This value can range from 10 to 100 percent.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Requirements

Expand table

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

Requirement	Value
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	avrt.h
Library	Avrt.lib
DLL	Avrt.dll

See also

Multimedia Class Scheduler Service

AvRevertMmThreadCharacteristics function (avrt.h)

Article02/22/2024

Indicates that a thread is no longer performing work associated with the specified task.

Syntax

```
C++

AVRTAPI BOOL AvRevertMmThreadCharacteristics(
  [in] HANDLE AvrtHandle
);
```

Parameters

`[in] AvrtHandle`

A handle to the task. This handle is returned by the AvSetMmThreadCharacteristics or AvSetMmMaxThreadCharacteristics function.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

This function must be called from the same thread that called the AvSetMmThreadCharacteristics or AvSetMmMaxThreadCharacteristics function to create the handle. Otherwise, the function will fail.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	avrt.h
Library	Avrt.lib
DLL	Avrt.dll

See also

Multimedia Class Scheduler Service

AvRtCreateThreadOrderingGroup function (avrt.h)

Article10/13/2021

Creates a thread ordering group.

To prevent the server thread for the thread ordering group from being starved if higher priority client threads are running, use the AvRtCreateThreadOrderingGroupEx function.

Syntax

```
C++

AVRTAPI BOOL AvRtCreateThreadOrderingGroup(
  [out]          PHANDLE        Context,
  [in]           PLARGE_INTEGER Period,
  [in, out]      GUID           *ThreadOrderingGuid,
  [in, optional] PLARGE_INTEGER Timeout
);
```

Parameters

`[out] Context`

A pointer to a context handle.

`[in] Period`

A pointer to a value, in 100-nanosecond increments, that specifies the period for the thread ordering group. Each thread in the thread ordering group runs one time during this period. 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.

The possible values for this parameter depend on the platform, but this parameter can be as low as 500 microseconds or as high as 0x1FFFFFFFFFFFFFFF. If this parameter is less than 500 microseconds, then it is set to 500 microseconds. If this parameter is greater than the maximum, then it is set to 0x1FFFFFFFFFFFFFFF.

`[in, out] ThreadOrderingGuid`

A pointer to the unique identifier for the thread ordering group to be created. If this value is not unique to the thread ordering service, the function fails.

If the identifier is GUID_NULL on input, the thread ordering service generates and returns a unique identifier.

`[in, optional] Timeout`

A pointer to a time-out value. All threads within the group should complete their execution within *Period* plus *Timeout*.

If a thread fails to complete its processing within the period plus this time-out interval, it is removed from the thread ordering group. If the parent fails to complete its processing within the period plus the time-out interval, the thread ordering group is destroyed.

The possible values for this parameter depend on the platform, but can be as low as 500 microseconds or as high as 0x1FFFFFFFFFFFFFFF. If this parameter is less than 500 microseconds, then it is set to 500 microseconds. If this parameter is greater than the maximum, then it is set to 0x1FFFFFFFFFFFFFFF.

If this parameter is **NULL** or 0, the default is five times the value of *Period*.

If this parameter is THREAD_ORDER_GROUP_INFINITE_TIMEOUT, the group is created with an infinite time-out interval. This can be useful for debugging purposes.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

If a thread ordering group with the specified identifier already exists, the function fails and sets the last error to ERROR_ALREADY_EXISTS.

Remarks

The calling thread is considered to be the parent thread. Each thread ordering group has one parent thread. Each parent thread can have zero or more predecessor threads and zero or more successor threads. A client thread can join a thread ordering group and specify whether it is a predecessor or successor using the AvRtJoinThreadOrderingGroup function.

The parent thread encloses the code to be executed during each period within a loop that is controlled by the AvRtWaitOnThreadOrderingGroup function.

To delete the thread ordering group, call the AvRtDeleteThreadOrderingGroup function.

A thread can create more than one thread ordering group and join more than one thread ordering group. However, a thread cannot join the same thread ordering group more than one time.

Examples

The following snippet creates a thread ordering group.

C++

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

#pragma comment(lib, "Avrt.lib")

#define _100NS_IN_1MS 10000

int main( void )
{
    HANDLE hContext = NULL;
    LARGE_INTEGER period, timeout;
    GUID guid = { 0 };
    BOOL bRes;

    period.QuadPart = Int32x32To64(_100NS_IN_1MS, 1000);   // 1 second
    timeout.QuadPart = Int32x32To64(_100NS_IN_1MS, 10000); // 10 seconds

    bRes = AvRtCreateThreadOrderingGroup(
            &hContext,
            &period,
            &guid,
            &timeout );

    if( bRes != TRUE )
    {
        printf("Error creating group (%d)\n", GetLastError());
            return 1;
    }

    return 0;
}
```

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	avrt.h
Library	Avrt.lib
DLL	Avrt.dll

See also

Thread Ordering Service

AvRtCreateThreadOrderingGroupExA function (avrt.h)

Article02/09/2023

Creates a thread ordering group and associates the server thread with a task.

Syntax

```C++
AVRTAPI BOOL AvRtCreateThreadOrderingGroupExA(
  [out]          PHANDLE        Context,
  [in]           PLARGE_INTEGER Period,
  [in, out]      GUID           *ThreadOrderingGuid,
  [in, optional] PLARGE_INTEGER Timeout,
  [in]           LPCSTR         TaskName
);
```

Parameters

`[out] Context`

A pointer to a context handle.

`[in] Period`

A pointer to a value, in 100-nanosecond increments, that specifies the period for the thread ordering group. Each thread in the thread ordering group runs one time during this period. 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.

The possible values for this parameter depend on the platform, but this parameter can be as low as 500 microseconds or as high as 0x1FFFFFFFFFFFFFFF. If this parameter is less than 500 microseconds, then it is set to 500 microseconds. If this parameter is greater than the maximum, then it is set to 0x1FFFFFFFFFFFFFFF.

`[in, out] ThreadOrderingGuid`

A pointer to the unique identifier for the thread ordering group to be created. If this value is not unique to the thread ordering service, the function fails.

If the identifier is GUID_NULL on input, the thread ordering service generates and returns a unique identifier.

`[in, optional] Timeout`

A pointer to a time-out value. All threads within the group should complete their execution within *Period* plus *Timeout*.

If a thread fails to complete its processing within the period plus this time-out interval, it is removed from the thread ordering group. If the parent fails to complete its processing within the period plus the time-out interval, the thread ordering group is destroyed.

The possible values for this parameter depend on the platform, but can be as low as 500 microseconds or as high as 0x1FFFFFFFFFFFFFFF. If this parameter is less than 500 microseconds, then it is set to 500 microseconds. If this parameter is greater than the maximum, then it is set to 0x1FFFFFFFFFFFFFFF.

If this parameter is **NULL** or 0, the default is five times the value of *Period*.

If this parameter is THREAD_ORDER_GROUP_INFINITE_TIMEOUT, the group is created with an infinite time-out interval. This can be useful for debugging purposes.

`[in] TaskName`

The name of the task.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

If a thread ordering group with the specified identifier already exists, the function fails and sets the last error to ERROR_ALREADY_EXISTS.

Remarks

The calling thread is considered to be the parent thread. Each thread ordering group has one parent thread. Each parent thread can have zero or more predecessor threads and zero or more successor threads. A client thread can join a thread ordering group and specify whether it is a predecessor or successor using the AvRtJoinThreadOrderingGroup function.

The parent thread encloses the code to be executed during each period within a loop that is controlled by the AvRtWaitOnThreadOrderingGroup function.

To delete the thread ordering group, call the AvRtDeleteThreadOrderingGroup function.

A thread can create more than one thread ordering group and join more than one thread ordering group. However, a thread cannot join the same thread ordering group more than one time.

The parent and client threads of a thread ordering group run at high priorities. However, the server thread that manages the thread ordering group runs at normal priority. Therefore, there can be a delay switching from one client thread to another if there are other high-priority threads running. The *TaskName* parameter of this function specifies the task to be associated with the server thread.

Examples

The following snippet creates a thread ordering group.

C++

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

#pragma comment(lib, "Avrt.lib")

#define _100NS_IN_1MS 10000

int main( void )
{
    HANDLE hContext = NULL;
    LARGE_INTEGER period, timeout;
    GUID guid = { 0 };
    BOOL bRes;

    period.QuadPart = Int32x32To64(_100NS_IN_1MS, 1000);   // 1 second
    timeout.QuadPart = Int32x32To64(_100NS_IN_1MS, 10000); // 10 seconds

    bRes = AvRtCreateThreadOrderingGroupEx(
            &hContext,
            &period,
            &guid,
            &timeout,
            TEXT("Audio") );

    if( bRes != TRUE )
    {
        printf("Error creating group (%d)\n", GetLastError());
            return 1;
    }

    return 0;
```

```
}
```

> **Note**
>
> The avrt.h header defines AvRtCreateThreadOrderingGroupEx as an alias that automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	avrt.h
Library	Avrt.lib
DLL	Avrt.dll

See also

Thread Ordering Service

AvRtDeleteThreadOrderingGroup function (avrt.h)

Article02/22/2024

Deletes the specified thread ordering group created by the caller. It cleans up resources for the thread ordering group, including the context information, and returns.

Syntax

```
C++

AVRTAPI BOOL AvRtDeleteThreadOrderingGroup(
  [in] HANDLE Context
);
```

Parameters

`[in] Context`

A context handle. This handle is returned by the AvRtCreateThreadOrderingGroup function when creating the group.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

This function can only be called successfully by the parent thread for the thread ordering group. If a thread other than the parent thread calls this function, the function fails with a last error code of ERROR_INVALID_FUNCTION.

If the parent thread times out and attempts to call this function, the function fails with a last error code of ERROR_INVALID_PARAMETER.

Examples

The following code deletes a thread ordering group.

```C++
#include <windows.h>
#include <avrt.h>
#include <stdio.h>

#pragma comment(lib, "Avrt.lib")

HANDLE Context;

int main( void )
{
    if(!AvRtDeleteThreadOrderingGroup(Context))
    {
        printf("Error deleting group (%d)\n", GetLastError());
        return 1;
    }

    return 0;
}
```

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	avrt.h
Library	Avrt.lib
DLL	Avrt.dll

See also

Thread Ordering Service

AvRtJoinThreadOrderingGroup function (avrt.h)

Article02/22/2024

Joins client threads to a thread ordering group.

Syntax

```
C++

AVRTAPI BOOL AvRtJoinThreadOrderingGroup(
  [out] PHANDLE Context,
  [in]  GUID    *ThreadOrderingGuid,
  [in]  BOOL    Before
);
```

Parameters

`[out] Context`

A pointer to a context handle.

`[in] ThreadOrderingGuid`

A pointer to the unique identifier for the thread ordering group.

`[in] Before`

The thread order. If this parameter is **TRUE**, the thread is a predecessor thread that is scheduled to run before the parent thread. If this parameter is **FALSE**, the thread is a successor thread that is scheduled to run after the parent thread.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The thread encloses the code to be executed during each period within a loop that is controlled by the AvRtWaitOnThreadOrderingGroup function.

A thread can create more than one thread ordering group and join more than one thread ordering group. However, a thread cannot join the same thread ordering group more than one time.

The number of threads that can join a group is limited only by available system resources.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	avrt.h
Library	Avrt.lib
DLL	Avrt.dll

See also

Thread Ordering Service

AvRtLeaveThreadOrderingGroup function (avrt.h)

Article02/22/2024

Enables client threads to leave a thread ordering group.

Syntax

```
C++

AVRTAPI BOOL AvRtLeaveThreadOrderingGroup(
  [in] HANDLE Context
);
```

Parameters

`[in] Context`

A context handle. This handle is returned by the AvRtJoinThreadOrderingGroup function.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The parent thread for a thread ordering group should not remove itself from the group.

If a thread times out and attempts to call this function, the function fails with a last error code of ERROR_INVALID_PARAMETER.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	avrt.h
Library	Avrt.lib
DLL	Avrt.dll

See also

Thread Ordering Service

AvRtWaitOnThreadOrderingGroup function (avrt.h)

Article10/13/2021

Enables client threads of a thread ordering group to wait until they should execute.

Syntax

```
C++

AVRTAPI BOOL AvRtWaitOnThreadOrderingGroup(
  [in] HANDLE Context
);
```

Parameters

`[in] Context`

A context handle. This handle is returned by the AvRtCreateThreadOrderingGroup or AvRtJoinThreadOrderingGroup function.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

When this function returns, the thread should complete its processing for the period and then call the function again.

If the thread fails to complete its processing during the time-out interval specified by the parent thread when creating the group, it is deleted from the thread ordering group. Therefore, when the thread finishes its processing loop, the next call to **AvRtWaitOnThreadOrderingGroup** fails and the last error code is set to ERROR_ACCESS_DENIED.

If the thread ordering group is deleted during the wait, this function eventually times out and return ERROR_ACCESS_DENIED.

Examples

```C++
#include <windows.h>
#include <avrt.h>
#include <stdio.h>

#pragma comment(lib, "Avrt.lib")

HANDLE Context;

int main( void )
{
    while(AvRtWaitOnThreadOrderingGroup(Context))
    {
        // Complete task for this period.
    }

return 0;
}
```

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	avrt.h
Library	Avrt.lib
DLL	Avrt.dll

See also

Thread Ordering Service

AvSetMmMaxThreadCharacteristicsA function (avrt.h)

Article02/09/2023

Associates the calling thread with the specified tasks.

Syntax

```
C++

AVRTAPI HANDLE AvSetMmMaxThreadCharacteristicsA(
  [in]      LPCSTR  FirstTask,
  [in]      LPCSTR  SecondTask,
  [in, out] LPDWORD TaskIndex
);
```

Parameters

`[in] FirstTask`

The name of the first task to be performed. This name must match the name of one of the subkeys of the following key **HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Multimedia\SystemProfile\Tasks**.

`[in] SecondTask`

The name of the second task to be performed. This name must match the name of one of the subkeys of the following key **HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Multimedia\SystemProfile\Tasks**.

`[in, out] TaskIndex`

The unique task identifier. The first time this function is called, this value must be 0 on input. The index value is returned on output and can be used as input in subsequent calls.

Return value

If the function succeeds, it returns a handle to the task.

If the function fails, it returns 0. To retrieve extended error information, call GetLastError.

The following are possible error codes.

Expand table

Return code	Description
ERROR_INVALID_TASK_INDEX	Either *TaskIndex* is not 0 on the first call or is not recognized value (on subsequent calls).
ERROR_INVALID_TASK_NAME	The specified task does not match any of the tasks stored in the registry.
ERROR_PRIVILEGE_NOT_HELD	The caller does not have sufficient privilege.

Remarks

The resulting characteristics of the thread performing the tasks reflect the task with the highest priority.

When the task is completed, call the AvRevertMmThreadCharacteristics function.

> **Note**
>
> The avrt.h header defines AvSetMmMaxThreadCharacteristics as an alias that automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	avrt.h
Library	Avrt.lib
DLL	Avrt.dll

See also

Multimedia Class Scheduler Service

AvSetMmThreadCharacteristicsA function (avrt.h)

Article11/20/2024

Associates the calling thread with the specified task.

Syntax

```
C++

AVRTAPI HANDLE AvSetMmThreadCharacteristicsA(
  [in]      LPCSTR  TaskName,
  [in, out] LPDWORD TaskIndex
);
```

Parameters

`[in] TaskName`

The name of the task to be performed. This name must match the name of one of the subkeys of the following key **HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Multimedia\SystemProfile\Tasks**.

`[in, out] TaskIndex`

The unique task identifier. The first time this function is called, this value must be 0 on input. The index value is returned on output and can be used as input in subsequent calls.

Return value

If the function succeeds, it returns a handle to the task.

If the function fails, it returns 0. To retrieve extended error information, call GetLastError.

The following are possible error codes.

Expand table

Return code	Description
ERROR_INVALID_TASK_INDEX	Either *TaskIndex* is not 0 on the first call or is not recognized value (on subsequent calls).

ERROR_INVALID_TASK_NAME	The specified task does not match any of the tasks stored in the registry.
ERROR_PRIVILEGE_NOT_HELD	The caller does not have sufficient privilege.

Remarks

When the task is completed, call the AvRevertMmThreadCharacteristics function.

> **Note**
>
> The avrt.h header defines AvSetMmThreadCharacteristics as an alias that automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	avrt.h
Library	Avrt.lib
DLL	Avrt.dll

See also

Multimedia Class Scheduler Service

AvSetMmThreadPriority function (avrt.h)

Article10/13/2021

Adjusts the thread priority of the calling thread relative to other threads performing the same task.

Syntax

```
C++

AVRTAPI BOOL AvSetMmThreadPriority(
  [in] HANDLE        AvrtHandle,
  [in] AVRT_PRIORITY Priority
);
```

Parameters

`[in] AvrtHandle`

A handle to the task. This handle is returned by the AvSetMmThreadCharacteristics or AvSetMmMaxThreadCharacteristics function.

`[in] Priority`

The relative thread priority of this thread to other threads performing a similar task. This parameter can be one of the following values.

AVRT_PRIORITY_CRITICAL (2)

AVRT_PRIORITY_HIGH (1)

AVRT_PRIORITY_LOW (-1)

AVRT_PRIORITY_NORMAL (0)

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	avrt.h
Library	Avrt.lib
DLL	Avrt.dll

See also

Multimedia Class Scheduler Service

BindIoCompletionCallback function (winbase.h)

03/11/2023

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.

Syntax

```C++
BOOL BindIoCompletionCallback(
  [in] HANDLE                          FileHandle,
  [in] LPOVERLAPPED_COMPLETION_ROUTINE Function,
  [in] ULONG                           Flags
);
```

Parameters

`[in] FileHandle`

A handle to the file opened for overlapped I/O completion. This handle is returned by the CreateFile function, with the **FILE_FLAG_OVERLAPPED** flag.

`[in] Function`

A pointer to the callback function to be executed in a non-I/O worker thread when the I/O operation is complete. This callback function must not call the TerminateThread function.

For more information about the completion routine, see FileIOCompletionRoutine.

`[in] Flags`

This parameter must be zero.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call the GetLastError function.

Remarks

The callback function might not be executed if the process issues an asynchronous request on the file specified by the *FileHandle* parameter but the request returns immediately with an error code other than ERROR_IO_PENDING.

Be sure that the thread that initiates the asynchronous I/O request does not terminate before the request is completed. Also, if a function in a DLL is queued to a worker thread, be sure that the function in the DLL has completed execution before the DLL is unloaded.

The thread pool maintains an I/O completion port. When you call **BindIoCompletionCallback**, it associates the specified file with the thread pool's I/O completion port. Asynchronous requests on that file object will complete by posting to the completion port, where they will be picked up by thread pool worker threads. For callbacks that must issue an I/O request that completes as an asynchronous procedure call, the thread pool provides an I/O worker pool. The I/O worker threads do not wait on the completion port; they sleep in an alertable wait state so that I/O request packets that complete can wake them up. Both types of worker threads check whether there is I/O pending on them and if there is, they do not exit. For more information, see Asynchronous Procedure Calls.

To compile an application that uses this function, define _WIN32_WINNT as 0x0500 or later. For more information, see Using the Windows Headers.

Requirements

Expand table

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

See also

FileIOCompletionRoutine

Process and Thread Functions

Thread Pooling

CallbackMayRunLong function (threadpoolapiset.h)

Article09/23/2022

Indicates that the callback may not return quickly.

Syntax

```C++
BOOL CallbackMayRunLong(
  [in, out] PTP_CALLBACK_INSTANCE pci
);
```

Parameters

`[in, out] pci`

A pointer to a **TP_CALLBACK_INSTANCE** structure that defines the callback instance. The pointer is passed to the callback function.

Return value

The function returns TRUE if another thread in the thread pool is available for processing callbacks or the thread pool was able to create a new thread. In this case, the current callback function may use the current thread indefinitely.

The function returns FALSE if another thread in the thread pool is not available to process callbacks and the thread pool was not able to create a new thread. The thread pool will attempt to create a new thread after a delay, but if the current callback function runs long, the thread pool may lose efficiency.

Remarks

The thread pool may use this information to better determine when a new thread should be created.

The **CallbackMayRunLong** function should be called only by the thread that is processing the callback. Calling this function from another thread may cause a race condition.

The **CallbackMayRunLong** function always marks the callback as long-running, whether or not a thread is available for processing callbacks or the threadpool is able to allocate a new thread. Therefore, this function should be called only once, even if it returns FALSE.

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

Requirements

Expand table

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

See also

DisassociateCurrentThreadFromCallback

FreeLibraryWhenCallbackReturns

LeaveCriticalSectionWhenCallbackReturns

ReleaseMutexWhenCallbackReturns

ReleaseSemaphoreWhenCallbackReturns

SetEventWhenCallbackReturns

SetThreadpoolCallbackRunsLong

Thread Pools

TrySubmitThreadpoolCallback

CancelThreadpoolIo function (threadpoolapiset.h)

02/22/2024

Cancels the notification from the StartThreadpoolIo function.

Syntax

```C++
VOID CancelThreadpoolIo(
  [in, out] PTP_IO pio
);
```

Parameters

`[in, out] pio`

A pointer to a **TP_IO** structure that defines the I/O completion object. The CreateThreadpoolIo function returns this pointer.

Return value

None

Remarks

To prevent memory leaks, you must call the **CancelThreadpoolIo** function for either of the following scenarios:

- An overlapped (asynchronous) I/O operation fails (that is, the asynchronous I/O function call returns failure with an error code other than ERROR_IO_PENDING).
- An asynchronous I/O operation returns immediately with success and the file handle associated with the I/O completion object has the notification mode FILE_SKIP_COMPLETION_PORT_ON_SUCCESS. The file handle will not notify the I/O completion port and the associated I/O callback function will not be called.

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

Requirements

Expand table

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

See also

CloseThreadpoolIo

CreateThreadpoolIo

StartThreadpoolIo

Thread Pools

WaitForThreadpoolIoCallbacks

CloseThreadpool function (threadpoolapiset.h)

02/22/2024

Closes the specified thread pool.

Syntax

```
C++

VOID CloseThreadpool(
  [in, out] PTP_POOL ptpp
);
```

Parameters

`[in, out] ptpp`

A pointer to a **TP_POOL** structure that defines the thread pool. The CreateThreadpool function returns this pointer.

Return value

None

Remarks

The thread pool is closed immediately if there are no outstanding work, I/O, timer, or wait objects that are bound to the pool; otherwise, the thread pool is released asynchronously after the outstanding objects are freed.

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

Examples

For an example, see Using the Thread Pool Functions.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps | UWP apps]
Target Platform	Windows
Header	threadpoolapiset.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

CreateThreadpool

SetThreadpoolThreadMaximum

SetThreadpoolThreadMinimum

Thread Pools

CloseThreadpoolCleanupGroup function (threadpoolapiset.h)

02/22/2024

Closes the specified cleanup group.

Syntax

```C++
VOID CloseThreadpoolCleanupGroup(
  [in, out] PTP_CLEANUP_GROUP ptpcg
);
```

Parameters

`[in, out] ptpcg`

A pointer to a **TP_CLEANUP_GROUP** structure that defines the cleanup group. The CreateThreadpoolCleanupGroup returns this pointer.

Return value

None

Remarks

The cleanup group must have no members when you call this function. For information on removing members of the group, see CloseThreadpoolCleanupGroupMembers.

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

Examples

For an example, see Using the Thread Pool Functions.

Requirements

Expand table

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

See also

CloseThreadpoolCleanupGroupMembers

CreateThreadpoolCleanupGroup

Thread Pools

CloseThreadpoolCleanupGroupMembers function (threadpoolapiset.h)

10/13/2021

Releases the members of the specified cleanup group, waits for all callback functions to complete, and optionally cancels any outstanding callback functions.

Syntax

```
C++

VOID CloseThreadpoolCleanupGroupMembers(
  [in, out]           PTP_CLEANUP_GROUP ptpcg,
  [in]                BOOL              fCancelPendingCallbacks,
  [in, out, optional] PVOID             pvCleanupContext
);
```

Parameters

`[in, out] ptpcg`

A pointer to a **TP_CLEANUP_GROUP** structure that defines the cleanup group. The CreateThreadpoolCleanupGroup function returns this pointer.

`[in] fCancelPendingCallbacks`

If this parameter is TRUE, the function cancels outstanding callbacks that have not yet started. If this parameter is FALSE, the function waits for outstanding callback functions to complete.

`[in, out, optional] pvCleanupContext`

The application-defined data to pass to the application's cleanup group callback function. You can specify the callback function when you call SetThreadpoolCallbackCleanupGroup.

Return value

None

Remarks

The **CloseThreadpoolCleanupGroupMembers** function simplifies cleanup of thread pool callback objects by releasing, in a single operation, all work objects, wait objects, and timer objects that are members of the cleanup group. An object becomes a member of a cleanup group when the object is created with the threadpool callback environment that was specified when the cleanup group was created. For more information, see CreateThreadpoolCleanupGroup.

The **CloseThreadpoolCleanupGroupMembers** function blocks until all currently executing callback functions finish. If *fCancelPendingCallbacks* is TRUE, outstanding callbacks are canceled; otherwise, the function blocks until all outstanding callbacks also finish. After the **CloseThreadpoolCleanupGroupMembers** function returns, an application should not use any object that was a member of the cleanup group at the time **CloseThreadpoolCleanupGroupMembers** was called. Also, an application should not release any of the objects individually by calling a function such as CloseThreadpoolWork, because the objects have already been released.

The **CloseThreadpoolCleanupGroupMembers** function does not close the cleanup group itself. Instead, the cleanup group persists until the CloseThreadpoolCleanupGroup function is called. Also, closing a cleanup group does not affect the associated threadpool callback environment. The callback environment persists until it is destroyed by calling DestroyThreadpoolEnvironment.

As long as a cleanup group persists, new objects created with the cleanup group's associated threadpool callback environment are added to the cleanup group. This allows an application to reuse the cleanup group. However, it can lead to errors if the application does not synchronize code that calls **CloseThreadpoolCleanupGroupMembers** with code that creates new objects. For example, suppose a thread creates two threadpool work objects, Work1 and Work2. Another thread calls **CloseThreadpoolCleanupGroupMembers**. Depending on when the threads run, any of the following might occur:

- Work1 and Work2 are added to the cleanup group after its existing members were released. Code that submits Work1 and Work2 will succeed.
- Work1 is added to the cleanup group before its existing members are released, causing Work1 to be released along with other members. Then Work2 is added. Code that submits Work1 will generate an exception; code that submits Work2 will succeed.
- Work1 and Work2 are added to the cleanup group before its existing members are released, causing both Work1 and Work2 to be released. Code that submits Work1 or Work2 will generate an exception.

To simply wait for or cancel pending work items without releasing them, use one of the threadpool callback functions: WaitForThreadpoolIoCallbacks,

WaitForThreadpoolTimerCallbacks, WaitForThreadpoolWaitCallbacks, or WaitForThreadpoolWorkCallbacks.

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

Examples

For an example, see Using the Thread Pool Functions.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps | UWP apps]
Target Platform	Windows
Header	threadpoolapiset.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

CloseThreadpoolCleanupGroup

CreateThreadpoolCleanupGroup

SetThreadpoolCallbackCleanupGroup

Thread Pools

CloseThreadpoolIo function (threadpoolapiset.h)

02/22/2024

Releases the specified I/O completion object.

Syntax

```C++
VOID CloseThreadpoolIo(
  [in, out] PTP_IO pio
);
```

Parameters

`[in, out] pio`

A pointer to a **TP_IO** structure that defines the I/O completion object. The CreateThreadpoolIo function returns this pointer.

Return value

None

Remarks

The I/O completion object is freed immediately if there are no outstanding callbacks; otherwise, the I/O completion object is freed asynchronously after the outstanding callbacks complete.

You should close the associated file handle and wait for all outstanding overlapped I/O operations to complete before calling this function. You must not cause any more overlapped I/O operations to occur after calling this function.

It may be necessary to cancel threadpool I/O notifications to prevent memory leaks. For more information, see CancelThreadpoolIo.

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

Requirements

Expand table

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

See also

CancelThreadpoolIo

CreateThreadpoolIo

StartThreadpoolIo

Thread Pools

WaitForThreadpoolIoCallbacks

CloseThreadpoolTimer function (threadpoolapiset.h)

02/22/2024

Releases the specified timer object.

Syntax

```C++
VOID CloseThreadpoolTimer(
  [in, out] PTP_TIMER pti
);
```

Parameters

`[in, out] pti`

A pointer to **TP_TIMER** structure that defines the timer object. The CreateThreadpoolTimer function returns this pointer.

Return value

None

Remarks

The timer object is freed immediately if there are no outstanding callbacks; otherwise, the timer object is freed asynchronously after the outstanding callback functions complete.

In some cases, callback functions might run after **CloseThreadpoolTimer** has been called. To prevent this behavior:

- Call the SetThreadpoolTimer function or SetThreadpoolTimerEx function with the *pftDueTime* parameter set to NULL and the *msPeriod* and *msWindowLength* parameters set to 0.
- Call the WaitForThreadpoolTimerCallbacks function with the *fCancelPendingCallbacks* parameter set to TRUE.
- Call **CloseThreadpoolTimer**.

If there is a cleanup group associated with the timer object, it is not necessary to call this function; calling the CloseThreadpoolCleanupGroupMembers function releases the work, wait, and timer objects associated with the cleanup group.

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

Requirements

Expand table

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

See also

CreateThreadpoolTimer

IsThreadpoolTimerSet

SetThreadpoolTimer

SetThreadpoolTimerEx

Thread Pools

WaitForThreadpoolTimerCallbacks

CloseThreadpoolWait function (threadpoolapiset.h)

09/23/2022

Releases the specified wait object.

Syntax

```
C++

VOID CloseThreadpoolWait(
  [in, out] PTP_WAIT pwa
);
```

Parameters

`[in, out] pwa`

A pointer to a **TP_WAIT** structure that defines the wait object. The CreateThreadpoolWait function returns this pointer.

Return value

None

Remarks

The wait object is freed immediately if there are no outstanding callbacks; otherwise, the timer object is freed asynchronously after the outstanding callbacks complete.

In some cases, callback functions might run after **CloseThreadpoolWait** has been called. To prevent this behavior:

- Call the SetThreadpoolWait function or SetThreadpoolWaitEx function with the *h* parameter set to NULL.
- Call the WaitForThreadpoolWaitCallbacks function with the *fCancelPendingCallbacks* parameter set to TRUE.
- Call **CloseThreadpoolWait**.

If there is a cleanup group associated with the wait object, it is not necessary to call this function; calling the CloseThreadpoolCleanupGroupMembers function releases the work, wait, and timer objects associated with the cleanup group.

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

Examples

For an example, see Using the Thread Pool Functions.

Requirements

Expand table

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

See also

CreateThreadpoolWait

SetThreadpoolWait

SetThreadpoolWaitEx

Thread Pools

WaitForThreadpoolWaitCallbacks

CloseThreadpoolWork function (threadpoolapiset.h)

02/22/2024

Releases the specified work object.

Syntax

```C++
VOID CloseThreadpoolWork(
  [in, out] PTP_WORK pwk
);
```

Parameters

`[in, out] pwk`

A pointer to a **TP_WORK** structure that defines the work object. The CreateThreadpoolWork function returns this pointer.

Return value

None

Remarks

The work object is freed immediately if there are no outstanding callbacks; otherwise, the work object is freed asynchronously after the outstanding callbacks complete.

If there is a cleanup group associated with the work object, it is not necessary to call this function; calling the CloseThreadpoolCleanupGroupMembers function releases the work, wait, and timer objects associated with the cleanup group.

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

Requirements

Expand table

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

See also

CreateThreadpoolWork

SubmitThreadpoolWork

Thread Pools

WaitForThreadpoolWorkCallbacks

ConvertFiberToThread function (winbase.h)

Article02/22/2024

Converts the current fiber into a thread.

Syntax

```
C++

BOOL ConvertFiberToThread();
```

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The function releases the resources allocated by the ConvertThreadToFiber function. After calling this function, you cannot call any of the fiber functions from the thread.

To compile an application that uses this function, define _WIN32_WINNT as _WIN32_WINNT_WS03 (0x0502) or later. For more information, see Using the Windows Headers.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib

Requirement	Value
DLL	Kernel32.dll

See also

ConvertThreadToFiber

Fibers

Process and Thread Functions

ConvertThreadToFiber function (winbase.h)

Article02/22/2024

Converts the current thread into a fiber. You must convert a thread into a fiber before you can schedule other fibers.

Syntax

```
C++

LPVOID ConvertThreadToFiber(
  [in, optional] LPVOID lpParameter
);
```

Parameters

`[in, optional] lpParameter`

A pointer to a variable that is passed to the fiber. The fiber can retrieve this data by using the GetFiberData macro.

Return value

If the function succeeds, the return value is the address of the fiber.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Remarks

Only fibers can execute other fibers. If a thread needs to execute a fiber, it must call **ConvertThreadToFiber** or ConvertThreadToFiberEx to create an area in which to save fiber state information. The thread is now the current fiber. The state information for this fiber includes the fiber data specified by *lpParameter*.

To compile an application that uses this function, define _WIN32_WINNT as 0x0400 or later. For more information, see Using the Windows Headers.

Examples

For an example, see Using Fibers.

Requirements

Expand table

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

See also

ConvertFiberToThread

ConvertThreadToFiberEx

Fibers

GetFiberData

Process and Thread Functions

ConvertThreadToFiberEx function (winbase.h)

Article02/22/2024

Converts the current thread into a fiber. You must convert a thread into a fiber before you can schedule other fibers.

Syntax

```
C++

LPVOID ConvertThreadToFiberEx(
  [in, optional] LPVOID lpParameter,
  [in]           DWORD  dwFlags
);
```

Parameters

`[in, optional] lpParameter`

A pointer to a variable that is passed to the fiber. The fiber can retrieve this data by using the GetFiberData macro.

`[in] dwFlags`

If this parameter is zero, the floating-point state on x86 systems is not switched and data can be corrupted if a fiber uses floating-point arithmetic. If this parameter is FIBER_FLAG_FLOAT_SWITCH, the floating-point state is switched for the fiber.

Return value

If the function succeeds, the return value is the address of the fiber.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Remarks

Only fibers can execute other fibers. If a thread needs to execute a fiber, it must call ConvertTheadToFiber or **ConvertThreadToFiberEx** to create an area in which to save fiber state

information. The thread is now the current fiber. The state information for this fiber includes the fiber data specified by *lpParameter*.

To compile an application that uses this function, define _WIN32_WINNT as 0x0400 or later. For more information, see Using the Windows Headers.

Requirements

Expand table

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

See also

ConvertFiberToThread

Fibers

GetFiberData

Process and Thread Functions

CreateDispatcherQueueController function (dispatcherqueue.h)

Article10/27/2023

Creates a DispatcherQueueController. Use the created DispatcherQueueController to create and manage the lifetime of a DispatcherQueue to run queued tasks in priority order on the dispatcher queue's thread.

Syntax

```
C++

HRESULT CreateDispatcherQueueController(
  [in]  DispatcherQueueOptions     options,
  [out] PDISPATCHERQUEUECONTROLLER *dispatcherQueueController
);
```

Parameters

`[in] options`

The threading affinity and type of COM apartment for the created DispatcherQueueController. See remarks for details.

`[out] dispatcherQueueController`

The created dispatcher queue controller.

> **Important** The **DispatcherQueueController** is a WinRT object.

Return value

S_OK for success; otherwise a failure code.

Remarks

Introduced in Windows 10, version 1709.

If *options.threadType* is **DQTYPE_THREAD_DEDICATED**, then this function creates a thread, initializes it with the specified COM apartment, and associates a DispatcherQueue with that thread. The dispatcher queue event loop runs on the new dedicated thread until the dispatcher queue is explicitly shut down. To avoid thread and memory leaks, call DispatcherQueueController.ShutdownQueueAsync when you are finished with the dispatcher queue.

If *options.threadType* is **DQTYPE_THREAD_CURRENT**, then a DispatcherQueue is created and associated with the current thread. An error results if there is already a **DispatcherQueue** associated with the current thread. The current thread must pump messages to allow the dispatcher queue to dispatch tasks. Before the current thread exits, it must call DispatcherQueueController.ShutdownQueueAsync, and continue pumping messages until the **IAsyncAction** completes.

This call does not return until the DispatcherQueueController and new thread (if any) are created.

> **Important** The **DispatcherQueueController**, and its associated **DispatcherQueue**, are WinRT objects. See their documentation for usage details.

Requirements

Expand table

Requirement	Value
Target Platform	Windows
Header	dispatcherqueue.h
Library	CoreMessaging.lib
DLL	CoreMessaging.dll

See also

DispatcherQueue

DispatcherQueueController

CreateFiber function (winbase.h)

Article07/27/2022

Allocates a fiber object, assigns it a stack, and sets up execution to begin at the specified start address, typically the fiber function. This function does not schedule the fiber.

To specify both a commit and reserve stack size, use the CreateFiberEx function.

Syntax

```C++
LPVOID CreateFiber(
  [in]           SIZE_T                dwStackSize,
  [in]           LPFIBER_START_ROUTINE lpStartAddress,
  [in, optional] LPVOID                lpParameter
);
```

Parameters

`[in] dwStackSize`

The initial committed size of the stack, in bytes. If this parameter is zero, the new fiber uses the default commit stack size for the executable. For more information, see Thread Stack Size.

`[in] lpStartAddress`

A pointer to the application-defined function to be executed by the fiber and represents the starting address of the fiber. Execution of the newly created fiber does not begin until another fiber calls the SwitchToFiber function with this address. For more information of the fiber callback function, see FiberProc.

`[in, optional] lpParameter`

A pointer to a variable that is passed to the fiber. The fiber can retrieve this data by using the GetFiberData macro.

Return value

If the function succeeds, the return value is the address of the fiber.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Remarks

The number of fibers a process can create is limited by the available virtual memory. For example, if you create each fiber with 1 megabyte of reserved stack space, you can create at most 2028 fibers. If you reduce the default stack size by using the STACKSIZE statement in the module definition (.def) file or by using CreateFiberEx, you can create more fibers. However, your application will have better performance if you use an alternate strategy for processing requests instead of creating such a large number of fibers.

Before a thread can schedule a fiber using the SwitchToFiber function, it must call the ConvertThreadToFiber function so there is a fiber associated with the thread.

To compile an application that uses this function, define _WIN32_WINNT as 0x0400 or later. For more information, see Using the Windows Headers.

Examples

For an example, see Using Fibers.

Requirements

Expand table

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

See also

ConvertThreadToFiber

CreateFiberEx

FiberProc

Fibers

GetFiberData

Process and Thread Functions

SwitchToFiber

CreateFiberEx function (winbase.h)

Article07/27/2022

Allocates a fiber object, assigns it a stack, and sets up execution to begin at the specified start address, typically the fiber function. This function does not schedule the fiber.

Syntax

```C++
LPVOID CreateFiberEx(
  [in]           SIZE_T                dwStackCommitSize,
  [in]           SIZE_T                dwStackReserveSize,
  [in]           DWORD                 dwFlags,
  [in]           LPFIBER_START_ROUTINE lpStartAddress,
  [in, optional] LPVOID                lpParameter
);
```

Parameters

`[in] dwStackCommitSize`

The initial commit size of the stack, in bytes. If this parameter is zero, the new fiber uses the default commit stack size for the executable. For more information, see Thread Stack Size.

`[in] dwStackReserveSize`

The initial reserve size of the stack, in bytes. If this parameter is zero, the new fiber uses the default reserved stack size for the executable. For more information, see Thread Stack Size.

`[in] dwFlags`

If this parameter is zero, the floating-point state on x86 systems is not switched and data can be corrupted if a fiber uses floating-point arithmetic. If this parameter is **FIBER_FLAG_FLOAT_SWITCH**, the floating-point state is switched for the fiber.

Windows XP: The **FIBER_FLAG_FLOAT_SWITCH** flag is not supported.

`[in] lpStartAddress`

A pointer to the application-defined function to be executed by the fiber and represents the starting address of the fiber. Execution of the newly created fiber does not begin until another fiber calls the SwitchToFiber function with this address. For more information on the fiber callback function, see FiberProc.

`[in, optional] lpParameter`

A pointer to a variable that is passed to the fiber. The fiber can retrieve this data by using the GetFiberData macro.

Return value

If the function succeeds, the return value is the address of the fiber.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Remarks

The number of fibers a process can create is limited by the available virtual memory. By default, every fiber has 1 megabyte of reserved stack space. Therefore, you can create at most 2028 fibers. If you reduce the default stack size, you can create more fibers. However, your application will have better performance if you use an alternate strategy for processing requests.

Before a thread can schedule a fiber using the SwitchToFiber function, it must call the ConvertThreadToFiber function so there is a fiber associated with the thread.

To compile an application that uses this function, define **_WIN32_WINNT** as 0x0400 or later. For more information, see Using the Windows Headers.

Requirements

Expand table

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

See also

ConvertThreadToFiber

FiberProc

Fibers

GetFiberData

Process and Thread Functions

SwitchToFiber

CreateJobObjectA function (winbase.h)

Article07/27/2022

Creates or opens a job object.

Syntax

```
C++

HANDLE CreateJobObjectA(
  [in, optional] LPSECURITY_ATTRIBUTES lpJobAttributes,
  [in, optional] LPCSTR                lpName
);
```

Parameters

`[in, optional] lpJobAttributes`

A pointer to a SECURITY_ATTRIBUTES structure that specifies the security descriptor for the job object and determines whether child processes can inherit the returned handle. If *lpJobAttributes* is **NULL**, the job object gets a default security descriptor and the handle cannot be inherited. The ACLs in the default security descriptor for a job object come from the primary or impersonation token of the creator.

`[in, optional] lpName`

The name of the job. The name is limited to **MAX_PATH** characters. Name comparison is case-sensitive.

If *lpName* is **NULL**, the job is created without a name.

If *lpName* matches the name of an existing event, semaphore, mutex, waitable timer, or file-mapping object, the function fails and the GetLastError function returns **ERROR_INVALID_HANDLE**. This occurs because these objects share the same namespace.

The object can be created in a private namespace. For more information, see Object Namespaces.

Terminal Services: The name can have a "Global" or "Local" prefix to explicitly create the object in the global or session namespace. The remainder of the name can contain any character except the backslash character (\). For more information, see Kernel Object Namespaces.

Return value

If the function succeeds, the return value is a handle to the job object. The handle has the **JOB_OBJECT_ALL_ACCESS** access right. If the object existed before the function call, the function returns a handle to the existing job object and GetLastError returns **ERROR_ALREADY_EXISTS**.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Remarks

When a job is created, its accounting information is initialized to zero, all limits are inactive, and there are no associated processes. To assign a process to a job object, use the AssignProcessToJobObject function. To set limits for a job, use the SetInformationJobObject function. To query accounting information, use the QueryInformationJobObject function.

All processes associated with a job must run in the same session. A job is associated with the session of the first process to be assigned to the job.

Windows Server 2003 and Windows XP: A job is associated with the session of the process that created it.

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 exited. 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.

To compile an application that uses this function, define **_WIN32_WINNT** as 0x0500 or later. For more information, see Using the Windows Headers.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows

Requirement	Value
Header	winbase.h (include Windows.h, Jobapi2.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

AssignProcessToJobObject

CloseHandle

Job Objects

Process and Thread Functions

QueryInformationJobObject

SECURITY_ATTRIBUTES

SetInformationJobObject

CreateProcessA function (processthreadsapi.h)

02/09/2023

Creates a new process and its primary thread. The new process runs in the security context of the calling process.

If the calling process is impersonating another user, the new process uses the token for the calling process, not the impersonation token. To run the new process in the security context of the user represented by the impersonation token, use the CreateProcessAsUserA function or CreateProcessWithLogonW function.

Syntax

C++

```
BOOL CreateProcessA(
  [in, optional]      LPCSTR                lpApplicationName,
  [in, out, optional] LPSTR                 lpCommandLine,
  [in, optional]      LPSECURITY_ATTRIBUTES lpProcessAttributes,
  [in, optional]      LPSECURITY_ATTRIBUTES lpThreadAttributes,
  [in]                BOOL                  bInheritHandles,
  [in]                DWORD                 dwCreationFlags,
  [in, optional]      LPVOID                lpEnvironment,
  [in, optional]      LPCSTR                lpCurrentDirectory,
  [in]                LPSTARTUPINFOA        lpStartupInfo,
  [out]               LPPROCESS_INFORMATION lpProcessInformation
);
```

Parameters

`[in, optional] lpApplicationName`

The name of the module to be executed. This module can be a Windows-based application. It can be some other type of module (for example, MS-DOS or OS/2) if the appropriate subsystem is available on the local computer.

The string can specify the full path and file name of the module to execute or it can specify a partial name. In the case of a partial name, the function uses the current drive and current directory to complete the specification. The function will not use the search path. This parameter must include the file name extension; no default extension is assumed.

The *lpApplicationName* parameter can be **NULL**. In that case, the module name must be the first white space–delimited token in the *lpCommandLine* string. If you are using a long file name that contains a space, use quoted strings to indicate where the file name ends and the arguments begin; otherwise, the file name is ambiguous. For example, consider the string "c:\program files\sub dir\program name". This string can be interpreted in a number of ways. The system tries to interpret the possibilities in the following order:

1. **c:\program.exe**
2. **c:\program files\sub.exe**
3. **c:\program files\sub dir\program.exe**
4. **c:\program files\sub dir\program name.exe**

If the executable module is a 16-bit application, *lpApplicationName* should be **NULL**, and the string pointed to by *lpCommandLine* should specify the executable module as well as its arguments.

To run a batch file, you must start the command interpreter; set *lpApplicationName* to cmd.exe and set *lpCommandLine* to the following arguments: /c plus the name of the batch file.

> ⓘ **Important**
>
> The MSRC engineering team advises against this. See [MS14-019 – Fixing a binary hijacking via .cmd or .bat file] for more details.

`[in, out, optional] lpCommandLine`

The command line to be executed.

The maximum length of this string is 32,767 characters, including the Unicode terminating null character. If *lpApplicationName* is **NULL**, the module name portion of *lpCommandLine* is limited to **MAX_PATH** characters.

The Unicode version of this function, **CreateProcessW**, can modify the contents of this string. Therefore, this parameter cannot be a pointer to read-only memory (such as a **const** variable or a literal string). If this parameter is a constant string, the function may cause an access violation.

The *lpCommandLine* parameter can be NULL. In that case, the function uses the string pointed to by *lpApplicationName* as the command line.

If both *lpApplicationName* and *lpCommandLine* are non-**NULL**, the null-terminated string pointed to by *lpApplicationName* specifies the module to execute, and the null-terminated string pointed to by *lpCommandLine* specifies the command line. The new process can use

GetCommandLine to retrieve the entire command line. Console processes written in C can use the *argc* and *argv* arguments to parse the command line. Because argv[0] is the module name, C programmers generally repeat the module name as the first token in the command line.

If *lpApplicationName* is NULL, the first white space–delimited token of the command line specifies the module name. If you are using a long file name that contains a space, use quoted strings to indicate where the file name ends and the arguments begin (see the explanation for the *lpApplicationName* parameter). If the file name does not contain an extension, .exe is appended. Therefore, if the file name extension is .com, this parameter must include the .com extension. If the file name ends in a period (.) with no extension, or if the file name contains a path, .exe is not appended. If the file name does not contain a directory path, the system searches for the executable file in the following sequence:

- The directory from which the application loaded.
- The current directory for the parent process.
- The 32-bit Windows system directory. Use the GetSystemDirectoryA function function to get the path of this directory.
- The 16-bit Windows system directory. There is no function that obtains the path of this directory, but it is searched. The name of this directory is System.
- The Windows directory. Use the GetWindowsDirectoryA function to get the path of this directory.
- The directories that are listed in the PATH environment variable. Note that this function does not search the per-application path specified by the **App Paths** registry key. To include this per-application path in the search sequence, use the ShellExecute function.

The system adds a terminating null character to the command-line string to separate the file name from the arguments. This divides the original string into two strings for internal processing.

`[in, optional] lpProcessAttributes`

A pointer to a SECURITY_ATTRIBUTES structure that determines whether the returned handle to the new process object can be inherited by child processes. If *lpProcessAttributes* is **NULL**, the handle cannot be inherited.

The **lpSecurityDescriptor** member of the structure specifies a security descriptor for the new process. If *lpProcessAttributes* is NULL or **lpSecurityDescriptor** is **NULL**, the process gets a default security descriptor. The ACLs in the default security descriptor for a process come from the primary token of the creator. **Windows XP:** The ACLs in the default security descriptor for a process come from the primary or impersonation token of the creator. This behavior changed with Windows XP with SP2 and Windows Server 2003.

`[in, optional] lpThreadAttributes`

A pointer to a SECURITY_ATTRIBUTES structure that specifies a security descriptor for the new thread and determines whether child processes can inherit the returned handle. If *lpThreadAttributes* is NULL, the thread gets a default security descriptor and the handle cannot be inherited. The access control lists (ACL) in the default security descriptor for a thread come from the primary token of the creator.

`[in] bInheritHandles`

If this parameter is TRUE, each inheritable handle in the calling process is inherited by the new process. If the parameter is FALSE, the handles are not inherited. Note that inherited handles have the same value and access rights as the original handles. For additional discussion of inheritable handles, see Remarks.

Terminal Services: You cannot inherit handles across sessions. Additionally, if this parameter is TRUE, you must create the process in the same session as the caller.

Protected Process Light (PPL) processes: The generic handle inheritance is blocked when a PPL process creates a non-PPL process since PROCESS_DUP_HANDLE is not allowed from a non-PPL process to a PPL process. See Process Security and Access Rights

Windows 7: STD_INPUT_HANDLE, STD_OUTPUT_HANDLE, and STD_ERROR_HANDLE are inherited, even when the parameter is FALSE.

`[in] dwCreationFlags`

The flags that control the priority class and the creation of the process. For a list of values, see Process Creation Flags.

This parameter also controls the new process's priority class, which is used to determine the scheduling priorities of the process's threads. For a list of values, see GetPriorityClass. If none of the priority class flags is specified, the priority class defaults to **NORMAL_PRIORITY_CLASS** unless the priority class of the creating process is **IDLE_PRIORITY_CLASS** or **BELOW_NORMAL_PRIORITY_CLASS**. In this case, the child process receives the default priority class of the calling process.

If the dwCreationFlags parameter has a value of 0:

- The process inherits both the error mode of the caller and the parent's console.
- The environment block for the new process is assumed to contain ANSI characters (see *lpEnvironment* parameter for additional information).
- A 16-bit Windows-based application runs in a shared Virtual DOS machine (VDM).

`[in, optional] lpEnvironment`

A pointer to the environment block for the new process. If this parameter is **NULL**, the new process uses the environment of the calling process.

An environment block consists of a null-terminated block of null-terminated strings. Each string is in the following form:

name=*value*\0

Because the equal sign is used as a separator, it must not be used in the name of an environment variable.

An environment block can contain either Unicode or ANSI characters. If the environment block pointed to by *lpEnvironment* contains Unicode characters, be sure that *dwCreationFlags* includes **CREATE_UNICODE_ENVIRONMENT**.

The ANSI version of this function, **CreateProcessA** fails if the total size of the environment block for the process exceeds 32,767 characters.

Note that an ANSI environment block is terminated by two zero bytes: one for the last string, one more to terminate the block. A Unicode environment block is terminated by four zero bytes: two for the last string, two more to terminate the block.

`[in, optional] lpCurrentDirectory`

The full path to the current directory for the process. The string can also specify a UNC path.

If this parameter is **NULL**, the new process will have the same current drive and directory as the calling process. (This feature is provided primarily for shells that need to start an application and specify its initial drive and working directory.)

`[in] lpStartupInfo`

A pointer to a STARTUPINFO or STARTUPINFOEX structure.

To set extended attributes, use a STARTUPINFOEX structure and specify EXTENDED_STARTUPINFO_PRESENT in the *dwCreationFlags* parameter.

Handles in STARTUPINFO or STARTUPINFOEX must be closed with CloseHandle when they are no longer needed.

> ⓘ **Important**
>
> The caller is responsible for ensuring that the standard handle fields in STARTUPINFO contain valid handle values. These fields are copied unchanged to the child process without validation, even when the **dwFlags** member specifies **STARTF_USESTDHANDLES**.

> Incorrect values can cause the child process to misbehave or crash. Use the Application Verifier runtime verification tool to detect invalid handles.

`[out] lpProcessInformation`

A pointer to a PROCESS_INFORMATION structure that receives identification information about the new process.

Handles in PROCESS_INFORMATION must be closed with CloseHandle when they are no longer needed.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Note that the function returns before the process has finished initialization. If a required DLL cannot be located or fails to initialize, the process is terminated. To get the termination status of a process, call GetExitCodeProcess.

Remarks

The process is assigned a process identifier. The identifier is valid until the process terminates. It can be used to identify the process, or specified in the OpenProcess function to open a handle to the process. The initial thread in the process is also assigned a thread identifier. It can be specified in the OpenThread function to open a handle to the thread. The identifier is valid until the thread terminates and can be used to uniquely identify the thread within the system. These identifiers are returned in the PROCESS_INFORMATION structure.

The name of the executable in the command line that the operating system provides to a process is not necessarily identical to that in the command line that the calling process gives to the **CreateProcess** function. The operating system may prepend a fully qualified path to an executable name that is provided without a fully qualified path.

The calling thread can use the WaitForInputIdle function to wait until the new process has finished its initialization and is waiting for user input with no input pending. This can be useful for synchronization between parent and child processes, because **CreateProcess** returns without waiting for the new process to finish its initialization. For example, the creating process would use **WaitForInputIdle** before trying to find a window associated with the new process.

The preferred way to shut down a process is by using the ExitProcess function, because this function sends notification of approaching termination to all DLLs attached to the process. Other means of shutting down a process do not notify the attached DLLs. Note that when a thread calls **ExitProcess**, other threads of the process are terminated without an opportunity to execute any additional code (including the thread termination code of attached DLLs). For more information, see Terminating a Process.

A parent process can directly alter the environment variables of a child process during process creation. This is the only situation when a process can directly change the environment settings of another process. For more information, see Changing Environment Variables.

If an application provides an environment block, the current directory information of the system drives is not automatically propagated to the new process. For example, there is an environment variable named =C: whose value is the current directory on drive C. An application must manually pass the current directory information to the new process. To do so, the application must explicitly create these environment variable strings, sort them alphabetically (because the system uses a sorted environment), and put them into the environment block. Typically, they will go at the front of the environment block, due to the environment block sort order.

One way to obtain the current directory information for a drive X is to make the following call: `GetFullPathName("X:", ...)`. That avoids an application having to scan the environment block. If the full path returned is X:, there is no need to pass that value on as environment data, since the root directory is the default current directory for drive X of a new process.

When a process is created with **CREATE_NEW_PROCESS_GROUP** specified, an implicit call to SetConsoleCtrlHandler(**NULL**,**TRUE**) is made on behalf of the new process; this means that the new process has CTRL+C disabled. This lets shells handle CTRL+C themselves, and selectively pass that signal on to sub-processes. CTRL+BREAK is not disabled, and may be used to interrupt the process/process group.

By default, passing **TRUE** as the value of the *bInheritHandles* parameter causes all inheritable handles to be inherited by the new process. This can be problematic for applications which create processes from multiple threads simultaneously yet desire each process to inherit different handles.

Applications can use the UpdateProcThreadAttributeList function with the **PROC_THREAD_ATTRIBUTE_HANDLE_LIST** parameter to provide a list of handles to be inherited by a particular process.

Security Remarks

The first parameter, *lpApplicationName*, can be **NULL**, in which case the executable name must be in the white space–delimited string pointed to by *lpCommandLine*. If the executable or path name has a space in it, there is a risk that a different executable could be run because of the way the function parses spaces. The following example is dangerous because the function will attempt to run "Program.exe", if it exists, instead of "MyApp.exe".

syntax

```
LPTSTR szCmdline = _tcsdup(TEXT("C:\\Program Files\\MyApp -L -S"));
CreateProcess(NULL, szCmdline, /* ... */);
```

If a malicious user were to create an application called "Program.exe" on a system, any program that incorrectly calls **CreateProcess** using the Program Files directory will run this application instead of the intended application.

To avoid this problem, do not pass **NULL** for *lpApplicationName*. If you do pass **NULL** for *lpApplicationName*, use quotation marks around the executable path in *lpCommandLine*, as shown in the example below.

syntax

```
LPTSTR szCmdline[] = _tcsdup(TEXT("\"C:\\Program Files\\MyApp\" -L -S"));
CreateProcess(NULL, szCmdline, /*...*/);
```

Examples

For an example, see Creating Processes.

> **Note**
>
> The processthreadsapi.h header defines CreateProcess as an alias that automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Expand table

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

See also

CloseHandle

ShellExecuteA

CreateProcessAsUser

CreateProcessWithLogonW

ExitProcess

GetCommandLine

GetEnvironmentStrings

GetExitCodeProcess

GetFullPathName

GetStartupInfo

OpenProcess

PROCESS_INFORMATION

Process and Thread Functions

Processes

SECURITY_ATTRIBUTES

STARTUPINFO

STARTUPINFOEX

SetErrorMode

TerminateProcess

WaitForInputIdle

CreateProcessAsUserA function (processthreadsapi.h)

02/09/2023

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

Typically, the process that calls the **CreateProcessAsUser** function must have the **SE_INCREASE_QUOTA_NAME** privilege and may require the **SE_ASSIGNPRIMARYTOKEN_NAME** privilege if the token is not assignable. If this function fails with **ERROR_PRIVILEGE_NOT_HELD** (1314), use the CreateProcessWithLogonW function instead. **CreateProcessWithLogonW** requires no special privileges, but the specified user account must be allowed to log on interactively. Generally, it is best to use **CreateProcessWithLogonW** to create a process with alternate credentials.

Syntax

C++

```
BOOL CreateProcessAsUserA(
  [in, optional]      HANDLE                hToken,
  [in, optional]      LPCSTR                lpApplicationName,
  [in, out, optional] LPSTR                 lpCommandLine,
  [in, optional]      LPSECURITY_ATTRIBUTES lpProcessAttributes,
  [in, optional]      LPSECURITY_ATTRIBUTES lpThreadAttributes,
  [in]                BOOL                  bInheritHandles,
  [in]                DWORD                 dwCreationFlags,
  [in, optional]      LPVOID                lpEnvironment,
  [in, optional]      LPCSTR                lpCurrentDirectory,
  [in]                LPSTARTUPINFOA        lpStartupInfo,
  [out]               LPPROCESS_INFORMATION lpProcessInformation
);
```

Parameters

`[in, optional] hToken`

A handle to the primary token that represents a user. The handle must have the **TOKEN_QUERY**, **TOKEN_DUPLICATE**, and **TOKEN_ASSIGN_PRIMARY** access rights. For more information, see Access Rights for Access-Token Objects. The user represented by the token must have read and execute access to the application specified by the *lpApplicationName* or the *lpCommandLine* parameter.

To get a primary token that represents the specified user, call the LogonUser function. Alternatively, you can call the DuplicateTokenEx function to convert an impersonation token into a primary token. This allows a server application that is impersonating a client to create a process that has the security context of the client.

If *hToken* is a restricted version of the caller's primary token, the **SE_ASSIGNPRIMARYTOKEN_NAME** privilege is not required. If the necessary privileges are not already enabled, **CreateProcessAsUser** enables them for the duration of the call. For more information, see Running with Special Privileges.

Terminal Services: The process is run in the session specified in the token. By default, this is the same session that called LogonUser. To change the session, use the SetTokenInformation function.

`[in, optional] lpApplicationName`

The name of the module to be executed. This module can be a Windows-based application. It can be some other type of module (for example, MS-DOS or OS/2) if the appropriate subsystem is available on the local computer.

The string can specify the full path and file name of the module to execute or it can specify a partial name. In the case of a partial name, the function uses the current drive and current directory to complete the specification. The function will not use the search path. This parameter must include the file name extension; no default extension is assumed.

The *lpApplicationName* parameter can be **NULL**. In that case, the module name must be the first white space–delimited token in the *lpCommandLine* string. If you are using a long file name that contains a space, use quoted strings to indicate where the file name ends and the arguments begin; otherwise, the file name is ambiguous. For example, consider the string "c:\program files\sub dir\program name". This string can be interpreted in a number of ways. The system tries to interpret the possibilities in the following order:

c:\program.exe c:\program files\sub.exe c:\program files\sub dir\program.exe c:\program files\sub dir\program name.exe If the executable module is a 16-bit application, *lpApplicationName* should be **NULL**, and the string pointed to by *lpCommandLine* should specify the executable module as well as its arguments. By default, all 16-bit Windows-based applications created by **CreateProcessAsUser** are run in a separate VDM (equivalent to **CREATE_SEPARATE_WOW_VDM** in CreateProcess).

`[in, out, optional] lpCommandLine`

The command line to be executed. The maximum length of this string is 32K characters. If *lpApplicationName* is **NULL**, the module name portion of *lpCommandLine* is limited to **MAX_PATH** characters.

The Unicode version of this function, **CreateProcessAsUserW**, can modify the contents of this string. Therefore, this parameter cannot be a pointer to read-only memory (such as a **const** variable or a literal string). If this parameter is a constant string, the function may cause an access violation.

The *lpCommandLine* parameter can be **NULL**. In that case, the function uses the string pointed to by *lpApplicationName* as the command line.

If both *lpApplicationName* and *lpCommandLine* are non-**NULL**, **lpApplicationName* specifies the module to execute, and **lpCommandLine* specifies the command line. The new process can use GetCommandLine to retrieve the entire command line. Console processes written in C can use the *argc* and *argv* arguments to parse the command line. Because *argv*[0] is the module name, C programmers generally repeat the module name as the first token in the command line.

If *lpApplicationName* is **NULL**, the first white space–delimited token of the command line specifies the module name. If you are using a long file name that contains a space, use quoted strings to indicate where the file name ends and the arguments begin (see the explanation for the *lpApplicationName* parameter). If the file name does not contain an extension, .exe is appended. Therefore, if the file name extension is .com, this parameter must include the .com extension. If the file name ends in a period (.) with no extension, or if the file name contains a path, .exe is not appended. If the file name does not contain a directory path, the system searches for the executable file in the following sequence:

1. The directory from which the application loaded.
2. The current directory for the parent process.
3. The 32-bit Windows system directory. Use the GetSystemDirectory function to get the path of this directory.
4. The 16-bit Windows system directory. There is no function that obtains the path of this directory, but it is searched.
5. The Windows directory. Use the GetWindowsDirectory function to get the path of this directory.
6. The directories that are listed in the PATH environment variable. Note that this function does not search the per-application path specified by the **App Paths** registry key. To include this per-application path in the search sequence, use the ShellExecute function.

The system adds a null character to the command line string to separate the file name from the arguments. This divides the original string into two strings for internal processing.

`[in, optional] lpProcessAttributes`

A pointer to a SECURITY_ATTRIBUTES structure that specifies a security descriptor for the new process object and determines whether child processes can inherit the returned handle to the

process. If *lpProcessAttributes* is **NULL** or **lpSecurityDescriptor** is **NULL**, the process gets a default security descriptor and the handle cannot be inherited. The default security descriptor is that of the user referenced in the *hToken* parameter. This security descriptor may not allow access for the caller, in which case the process may not be opened again after it is run. The process handle is valid and will continue to have full access rights.

`[in, optional] lpThreadAttributes`

A pointer to a SECURITY_ATTRIBUTES structure that specifies a security descriptor for the new thread and determines whether child processes can inherit the returned handle. If *lpThreadAttributes* is **NULL** or **lpSecurityDescriptor** is **NULL**, the thread gets a default security descriptor and the handle cannot be inherited. The default security descriptor is that of the user referenced in the *hToken* parameter. This security descriptor may not allow access for the caller.

`[in] bInheritHandles`

If this parameter is **TRUE**, each inheritable handle in the calling process is inherited by the new process. If the parameter is **FALSE**, the handles are not inherited. Note that inherited handles have the same value and access rights as the original handles. For additional discussion of inheritable handles, see Remarks.

Terminal Services: You cannot inherit handles across sessions. Additionally, if this parameter is **TRUE**, you must create the process in the same session as the caller.

Protected Process Light (PPL) processes: The generic handle inheritance is blocked when a PPL process creates a non-PPL process since PROCESS_DUP_HANDLE is not allowed from a non-PPL process to a PPL process. See Process Security and Access Rights

`[in] dwCreationFlags`

The flags that control the priority class and the creation of the process. For a list of values, see Process Creation Flags.

This parameter also controls the new process's priority class, which is used to determine the scheduling priorities of the process's threads. For a list of values, see GetPriorityClass. If none of the priority class flags is specified, the priority class defaults to **NORMAL_PRIORITY_CLASS** unless the priority class of the creating process is **IDLE_PRIORITY_CLASS** or **BELOW_NORMAL_PRIORITY_CLASS**. In this case, the child process receives the default priority class of the calling process.

If the dwCreationFlags parameter has a value of 0:

- The process inherits both the error mode of the caller and the parent's console.

- The environment block for the new process is assumed to contain ANSI characters (see *lpEnvironment* parameter for additional information).
- A 16-bit Windows-based application runs in a shared Virtual DOS machine (VDM).

`[in, optional] lpEnvironment`

A pointer to an environment block for the new process. If this parameter is **NULL**, the new process uses the environment of the calling process.

An environment block consists of a null-terminated block of null-terminated strings. Each string is in the following form:

name=*value*\0

Because the equal sign is used as a separator, it must not be used in the name of an environment variable.

An environment block can contain either Unicode or ANSI characters. If the environment block pointed to by *lpEnvironment* contains Unicode characters, be sure that *dwCreationFlags* includes **CREATE_UNICODE_ENVIRONMENT**.

The ANSI version of this function, **CreateProcessAsUserA** fails if the total size of the environment block for the process exceeds 32,767 characters.

Note that an ANSI environment block is terminated by two zero bytes: one for the last string, one more to terminate the block. A Unicode environment block is terminated by four zero bytes: two for the last string, two more to terminate the block.

Windows Server 2003 and Windows XP: If the size of the combined user and system environment variable exceeds 8192 bytes, the process created by **CreateProcessAsUser** no longer runs with the environment block passed to the function by the parent process. Instead, the child process runs with the environment block returned by the CreateEnvironmentBlock function.

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

`[in, optional] lpCurrentDirectory`

The full path to the current directory for the process. The string can also specify a UNC path.

If this parameter is NULL, the new process will have the same current drive and directory as the calling process. (This feature is provided primarily for shells that need to start an application and specify its initial drive and working directory.)

`[in] lpStartupInfo`

A pointer to a STARTUPINFO or STARTUPINFOEX structure.

The user must have full access to both the specified window station and desktop. If you want the process to be interactive, specify winsta0\default. If the **lpDesktop** member is NULL, the new process inherits the desktop and window station of its parent process. If this member is an empty string, "", the new process connects to a window station using the rules described in Process Connection to a Window Station.

To set extended attributes, use a STARTUPINFOEX structure and specify **EXTENDED_STARTUPINFO_PRESENT** in the *dwCreationFlags* parameter.

Handles in STARTUPINFO or STARTUPINFOEX must be closed with CloseHandle when they are no longer needed.

> **Important** The caller is responsible for ensuring that the standard handle fields in **STARTUPINFO** contain valid handle values. These fields are copied unchanged to the child process without validation, even when the **dwFlags** member specifies **STARTF_USESTDHANDLES**. Incorrect values can cause the child process to misbehave or crash. Use the **Application Verifier** runtime verification tool to detect invalid handles.

`[out] lpProcessInformation`

A pointer to a PROCESS_INFORMATION structure that receives identification information about the new process.

Handles in PROCESS_INFORMATION must be closed with CloseHandle when they are no longer needed.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Note that the function returns before the process has finished initialization. If a required DLL cannot be located or fails to initialize, the process is terminated. To get the termination status of a process, call GetExitCodeProcess.

Remarks

CreateProcessAsUser must be able to open the primary token of the calling process with the **TOKEN_DUPLICATE** and **TOKEN_IMPERSONATE** access rights.

By default, **CreateProcessAsUser** creates the new process on a noninteractive window station with a desktop that is not visible and cannot receive user input. To enable user interaction with the new process, you must specify the name of the default interactive window station and desktop, "winsta0\default", in the **lpDesktop** member of the STARTUPINFO structure. In addition, before calling **CreateProcessAsUser**, you must change the discretionary access control list (DACL) of both the default interactive window station and the default desktop. The DACLs for the window station and desktop must grant access to the user or the logon session represented by the *hToken* parameter.

CreateProcessAsUser does not load the specified user's profile into the **HKEY_USERS** registry key. Therefore, to access the information in the **HKEY_CURRENT_USER** registry key, you must load the user's profile information into **HKEY_USERS** with the LoadUserProfile function before calling **CreateProcessAsUser**. Be sure to call UnloadUserProfile after the new process exits.

If the *lpEnvironment* parameter is NULL, the new process inherits the environment of the calling process. **CreateProcessAsUser** does not automatically modify the environment block to include environment variables specific to the user represented by *hToken*. For example, the USERNAME and USERDOMAIN variables are inherited from the calling process if *lpEnvironment* is NULL. It is your responsibility to prepare the environment block for the new process and specify it in *lpEnvironment*.

The CreateProcessWithLogonW and CreateProcessWithTokenW functions are similar to **CreateProcessAsUser**, except that the caller does not need to call the LogonUser function to authenticate the user and get a token.

CreateProcessAsUser allows you to access the specified directory and executable image in the security context of the caller or the target user. By default, **CreateProcessAsUser** accesses the directory and executable image in the security context of the caller. In this case, if the caller does not have access to the directory and executable image, the function fails. To access the directory and executable image using the security context of the target user, specify *hToken* in a call to the ImpersonateLoggedOnUser function before calling **CreateProcessAsUser**.

The process is assigned a process identifier. The identifier is valid until the process terminates. It can be used to identify the process, or specified in the OpenProcess function to open a handle to the process. The initial thread in the process is also assigned a thread identifier. It can be specified in the OpenThread function to open a handle to the thread. The identifier is valid until the thread terminates and can be used to uniquely identify the thread within the system. These identifiers are returned in the PROCESS_INFORMATION structure.

The calling thread can use the WaitForInputIdle function to wait until the new process has finished its initialization and is waiting for user input with no input pending. This can be useful for synchronization between parent and child processes, because **CreateProcessAsUser** returns without waiting for the new process to finish its initialization. For example, the creating process would use **WaitForInputIdle** before trying to find a window associated with the new process.

The preferred way to shut down a process is by using the ExitProcess function, because this function sends notification of approaching termination to all DLLs attached to the process. Other means of shutting down a process do not notify the attached DLLs. Note that when a thread calls **ExitProcess**, other threads of the process are terminated without an opportunity to execute any additional code (including the thread termination code of attached DLLs). For more information, see Terminating a Process.

By default, passing **TRUE** as the value of the *bInheritHandles* parameter causes all inheritable handles to be inherited by the new process. This can be problematic for applications which create processes from multiple threads simultaneously yet desire each process to inherit different handles. Applications can use the UpdateProcThreadAttributeList function with the **PROC_THREAD_ATTRIBUTE_HANDLE_LIST** parameter to provide a list of handles to be inherited by a particular process.

Security Remarks

The *lpApplicationName* parameter can be NULL, in which case the executable name must be the first white space–delimited string in *lpCommandLine*. If the executable or path name has a space in it, there is a risk that a different executable could be run because of the way the function parses spaces. The following example is dangerous because the function will attempt to run "Program.exe", if it exists, instead of "MyApp.exe".

syntax

```
LPTSTR szCmdline[] = _tcsdup(TEXT("C:\\Program Files\\MyApp"));
CreateProcessAsUser(hToken, NULL, szCmdline, /*...*/ );
```

If a malicious user were to create an application called "Program.exe" on a system, any program that incorrectly calls **CreateProcessAsUser** using the Program Files directory will run this application instead of the intended application.

To avoid this problem, do not pass NULL for *lpApplicationName*. If you do pass **NULL** for *lpApplicationName*, use quotation marks around the executable path in *lpCommandLine*, as shown in the example below.

syntax

```
LPTSTR szCmdline[] = _tcsdup(TEXT("\"C:\\Program Files\\MyApp\""));
CreateProcessAsUser(hToken, NULL, szCmdline, /*...*/);
```

PowerShell: When the **CreateProcessAsUser** function is used to implement a cmdlet in PowerShell version 2.0, the cmdlet operates correctly for both fan-in and fan-out remote sessions. Because of certain security scenarios, however, a cmdlet implemented with **CreateProcessAsUser** only operates correctly in PowerShell version 3.0 for fan-in remote sessions; fan-out remote sessions will fail because of insufficient client security privileges. To implement a cmdlet that works for both fan-in and fan-out remote sessions in PowerShell version 3.0, use the CreateProcess function.

Examples

For an example, see Starting an Interactive Client Process.

> **Note**
>
> The processthreadsapi.h header defines CreateProcessAsUser as an alias that automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	processthreadsapi.h (include Windows.h)
Library	Advapi32.lib
DLL	Advapi32.dll

See also

CloseHandle

CreateEnvironmentBlock

CreateProcess

CreateProcessWithLogonW

ExitProcess

GetEnvironmentStrings

GetExitCodeProcess

GetStartupInfo

ImpersonateLoggedOnUser

LoadUserProfile

PROCESS_INFORMATION

Process and Thread Functions

Processes

SECURITY_ATTRIBUTES

SHCreateProcessAsUserW

STARTUPINFO

STARTUPINFOEX

SetErrorMode

WaitForInputIdle

CreateProcessWithLogonW function (winbase.h)

Article02/02/2023

Creates a new process and its primary thread. Then the new process runs the specified executable file in the security context of the specified credentials (user, domain, and password). It can optionally load the user profile for a specified user.

This function is similar to the CreateProcessAsUser and CreateProcessWithTokenW functions, except that the caller does not need to call the LogonUser function to authenticate the user and get a token.

Syntax

```
C++

BOOL CreateProcessWithLogonW(
  [in]                LPCWSTR               lpUsername,
  [in, optional]      LPCWSTR               lpDomain,
  [in]                LPCWSTR               lpPassword,
  [in]                DWORD                 dwLogonFlags,
  [in, optional]      LPCWSTR               lpApplicationName,
  [in, out, optional] LPWSTR                lpCommandLine,
  [in]                DWORD                 dwCreationFlags,
  [in, optional]      LPVOID                lpEnvironment,
  [in, optional]      LPCWSTR               lpCurrentDirectory,
  [in]                LPSTARTUPINFOW        lpStartupInfo,
  [out]               LPPROCESS_INFORMATION lpProcessInformation
);
```

Parameters

`[in] lpUsername`

The name of the user. This is the name of the user account to log on to. If you use the UPN format, *user@DNS_domain_name*, the *lpDomain* parameter must be NULL.

The user account must have the Log On Locally permission on the local computer. This permission is granted to all users on workstations and servers, but only to administrators on domain controllers.

`[in, optional] lpDomain`

The name of the domain or server whose account database contains the *lpUsername* account. If this parameter is NULL, the user name must be specified in UPN format.

`[in] lpPassword`

The clear-text password for the *lpUsername* account.

`[in] dwLogonFlags`

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

Expand table

Value	Meaning
LOGON_WITH_PROFILE 0x00000001	Log on, then load the user profile in the **HKEY_USERS** registry key. The function returns after the profile is loaded. Loading the profile can be time-consuming, so it is best to use this value only if you must access the information in the **HKEY_CURRENT_USER** registry key. **Windows Server 2003:** The profile is unloaded after the new process is terminated, whether or not it has created child processes. **Windows XP:** The profile is unloaded after the new process and all child processes it has created are terminated.
LOGON_NETCREDENTIALS_ONLY 0x00000002	Log on, but use the specified credentials on the network only. The new process uses the same token as the caller, but the system creates a new logon session within LSA, and the process uses the specified credentials as the default credentials. This value can be used to create a process that uses a different set of credentials locally than it does remotely. This is useful in inter-domain scenarios where there is no trust relationship. The system does not validate the specified credentials. Therefore, the process can start, but it may not have access to network resources.

`[in, optional] lpApplicationName`

The name of the module to be executed. This module can be a Windows-based application. It can be some other type of module (for example, MS-DOS or OS/2) if the appropriate subsystem is available on the local computer.

The string can specify the full path and file name of the module to execute or it can specify a partial name. If it is a partial name, the function uses the current drive and current directory to complete the specification. The function does not use the search path. This parameter must include the file name extension; no default extension is assumed.

The *lpApplicationName* parameter can be NULL, and the module name must be the first white space–delimited token in the *lpCommandLine* string. If you are using a long file name that contains a space, use quoted strings to indicate where the file name ends and the arguments begin; otherwise, the file name is ambiguous.

For example, the following string can be interpreted in different ways:

"c:\program files\sub dir\program name"

The system tries to interpret the possibilities in the following order:

1. **c:\program.exe** files\sub dir\program name
2. **c:\program files\sub.exe** dir\program name
3. **c:\program files\sub dir\program.exe** name
4. **c:\program files\sub dir\program name.exe**

If the executable module is a 16-bit application, *lpApplicationName* should be NULL, and the string pointed to by *lpCommandLine* should specify the executable module and its arguments.

`[in, out, optional] lpCommandLine`

The command line to be executed. The maximum length of this string is 1024 characters. If *lpApplicationName* is **NULL**, the module name portion of *lpCommandLine* is limited to **MAX_PATH** characters.

The function can modify the contents of this string. Therefore, this parameter cannot be a pointer to read-only memory (such as a **const** variable or a literal string). If this parameter is a constant string, the function may cause an access violation.

The *lpCommandLine* parameter can be **NULL**, and the function uses the string pointed to by *lpApplicationName* as the command line.

If both *lpApplicationName* and *lpCommandLine* are non-**NULL**, **lpApplicationName* specifies the module to execute, and **lpCommandLine* specifies the command line. The new process can use GetCommandLine to retrieve the entire command line. Console processes written in C can use the *argc* and *argv* arguments to parse the command line. Because *argv[0]* is the module name, C programmers typically repeat the module name as the first token in the command line.

If *lpApplicationName* is **NULL**, the first white space–delimited token of the command line specifies the module name. If you are using a long file name that contains a space, use quoted strings to indicate where the file name ends and the arguments begin (see the explanation for the *lpApplicationName* parameter). If the file name does not contain an extension, .exe is appended. Therefore, if the file name extension is .com, this parameter must include the .com extension. If the file name ends in a period with no extension, or if the file name contains a path, .exe is not appended. If the file name does not contain a directory path, the system searches for the executable file in the following sequence:

1. The directory from which the application loaded.
2. The current directory for the parent process.
3. The 32-bit Windows system directory. Use the GetSystemDirectory function to get the path of this directory.
4. The 16-bit Windows system directory. There is no function that obtains the path of this directory, but it is searched.
5. The Windows directory. Use the GetWindowsDirectory function to get the path of this directory.
6. The directories that are listed in the PATH environment variable. Note that this function does not search the per-application path specified by the **App Paths** registry key. To include this per-application path in the search sequence, use the ShellExecute function.

The system adds a null character to the command line string to separate the file name from the arguments. This divides the original string into two strings for internal processing.

`[in] dwCreationFlags`

The flags that control how the process is created. The **CREATE_DEFAULT_ERROR_MODE**, **CREATE_NEW_CONSOLE**, and **CREATE_NEW_PROCESS_GROUP** flags are enabled by default. For a list of values, see Process Creation Flags.

This parameter also controls the new process's priority class, which is used to determine the scheduling priorities of the process's threads. For a list of values, see GetPriorityClass. If none of the priority class flags is specified, the priority class defaults to **NORMAL_PRIORITY_CLASS** unless the priority class of the creating process is **IDLE_PRIORITY_CLASS** or **BELOW_NORMAL_PRIORITY_CLASS**. In this case, the child process receives the default priority class of the calling process.

If the dwCreationFlags parameter has a value of 0:

- The process gets the default error mode, creates a new console and creates a new process group.
- The environment block for the new process is assumed to contain ANSI characters (see *lpEnvironment* parameter for additional information).

- A 16-bit Windows-based application runs in a shared Virtual DOS machine (VDM).

`[in, optional] lpEnvironment`

A pointer to an environment block for the new process. If this parameter is **NULL**, the new process uses an environment created from the profile of the user specified by *lpUsername*.

An environment block consists of a null-terminated block of null-terminated strings. Each string is in the following form:

name=value

Because the equal sign (=) is used as a separator, it must not be used in the name of an environment variable.

An environment block can contain Unicode or ANSI characters. If the environment block pointed to by *lpEnvironment* contains Unicode characters, ensure that *dwCreationFlags* includes **CREATE_UNICODE_ENVIRONMENT**.

An ANSI environment block is terminated by two 0 (zero) bytes: one for the last string and one more to terminate the block. A Unicode environment block is terminated by four zero bytes: two for the last string and two more to terminate the block.

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

`[in, optional] lpCurrentDirectory`

The full path to the current directory for the process. The string can also specify a UNC path.

If this parameter is **NULL**, the new process has the same current drive and directory as the calling process. This feature is provided primarily for shells that need to start an application, and specify its initial drive and working directory.

`[in] lpStartupInfo`

A pointer to a STARTUPINFO structure.

The application must add permission for the specified user account to the specified window station and desktop, even for WinSta0\Default.

If the **lpDesktop** member is **NULL** or an empty string, the new process inherits the desktop and window station of its parent process. The application must add permission for the specified user account to the inherited window station and desktop.

Windows XP: **CreateProcessWithLogonW** adds permission for the specified user account to the inherited window station and desktop.

Handles in STARTUPINFO must be closed with CloseHandle when they are no longer needed.

> **Important** If the **dwFlags** member of the **STARTUPINFO** structure specifies **STARTF_USESTDHANDLES**, the standard handle fields are copied unchanged to the child process without validation. The caller is responsible for ensuring that these fields contain valid handle values. Incorrect values can cause the child process to misbehave or crash. Use the **Application Verifier** runtime verification tool to detect invalid handles.

`[out] lpProcessInformation`

A pointer to a PROCESS_INFORMATION structure that receives identification information for the new process, including a handle to the process.

Handles in PROCESS_INFORMATION must be closed with the CloseHandle function when they are not needed.

Return value

If the function succeeds, the return value is nonzero.

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

Note that the function returns before the process has finished initialization. If a required DLL cannot be located or fails to initialize, the process is terminated. To get the termination status of a process, call GetExitCodeProcess.

Remarks

By default, **CreateProcessWithLogonW** does not load the specified user profile into the **HKEY_USERS** registry key. This means that access to information in the **HKEY_CURRENT_USER** registry key may not produce results that are consistent with a normal interactive logon. It is your responsibility to load the user registry hive into **HKEY_USERS** before calling **CreateProcessWithLogonW**, by using **LOGON_WITH_PROFILE**, or by calling the LoadUserProfile function.

If the *lpEnvironment* parameter is NULL, the new process uses an environment block created from the profile of the user specified by *lpUserName*. If the HOMEDRIVE and HOMEPATH variables are not set, **CreateProcessWithLogonW** modifies the environment block to use the drive and path of the user's working directory.

When created, the new process and thread handles receive full access rights (**PROCESS_ALL_ACCESS** and **THREAD_ALL_ACCESS**). For either handle, if a security descriptor is not provided, the handle can be used in any function that requires an object handle of that type. When a security descriptor is provided, an access check is performed on all subsequent uses of the handle before access is granted. If access is denied, the requesting process cannot use the handle to gain access to the process or thread.

To retrieve a security token, pass the process handle in the PROCESS_INFORMATION structure to the OpenProcessToken function.

The process is assigned a process identifier. The identifier is valid until the process terminates. It can be used to identify the process, or it can be specified in the OpenProcess function to open a handle to the process. The initial thread in the process is also assigned a thread identifier. It can be specified in the OpenThread function to open a handle to the thread. The identifier is valid until the thread terminates and can be used to uniquely identify the thread within the system. These identifiers are returned in PROCESS_INFORMATION.

The calling thread can use the WaitForInputIdle function to wait until the new process has completed its initialization and is waiting for user input with no input pending. This can be useful for synchronization between parent and child processes, because **CreateProcessWithLogonW** returns without waiting for the new process to finish its initialization. For example, the creating process would use **WaitForInputIdle** before trying to find a window that is associated with the new process.

The preferred way to shut down a process is by using the ExitProcess function, because this function sends notification of approaching termination to all DLLs attached to the process. Other means of shutting down a process do not notify the attached DLLs. Note that when a thread calls **ExitProcess**, other threads of the process are terminated without an opportunity to execute any additional code (including the thread termination code of attached DLLs). For more information, see Terminating a Process.

CreateProcessWithLogonW accesses the specified directory and executable image in the security context of the target user. If the executable image is on a network and a network drive letter is specified in the path, the network drive letter is not available to the target user, as network drive letters can be assigned for each logon. If a network drive letter is specified, this function fails. If the executable image is on a network, use the UNC path.

There is a limit to the number of child processes that can be created by this function and run simultaneously. For example, on Windows XP, this limit is **MAXIMUM_WAIT_OBJECTS***4. However, you may not be able to create this many processes due to system-wide quota limits.

Windows XP with SP2,Windows Server 2003, or later: You cannot call **CreateProcessWithLogonW** from a process that is running under the "LocalSystem" account, because the function uses the logon SID in the caller token, and the token for the "LocalSystem" account does not contain this SID. As an alternative, use the CreateProcessAsUser and LogonUser functions.

To compile an application that uses this function, define **_WIN32_WINNT** as 0x0500 or later. For more information, see Using the Windows Headers.

Security Remarks

The *lpApplicationName* parameter can be **NULL**, and the executable name must be the first white space–delimited string in *lpCommandLine*. If the executable or path name has a space in it, there is a risk that a different executable could be run because of the way the function parses spaces. Avoid the following example, because the function attempts to run "Program.exe", if it exists, instead of "MyApp.exe".

syntax

```
LPTSTR szCmdline[]=_tcsdup(TEXT("C:\\Program Files\\MyApp"));
CreateProcessWithLogonW(..., szCmdline, ...)
```

If a malicious user creates an application called "Program.exe" on a system, any program that incorrectly calls **CreateProcessWithLogonW** using the Program Files directory runs the malicious user application instead of the intended application.

To avoid this issue, do not pass **NULL** for *lpApplicationName*. If you pass **NULL** for *lpApplicationName*, use quotation marks around the executable path in *lpCommandLine*, as shown in the following example:

syntax

```
LPTSTR szCmdline[]=_tcsdup(TEXT("\"C:\\Program Files\\MyApp\""));
CreateProcessWithLogonW(..., szCmdline, ...)
```

Examples

The following example demonstrates how to call this function.

C++

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

void DisplayError(LPWSTR pszAPI)
{
    LPVOID lpvMessageBuffer;

    FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER |
        FORMAT_MESSAGE_FROM_SYSTEM,
        NULL, GetLastError(),
        MAKELANGID(LANG_NEUTRAL, SUBLANG_DEFAULT),
        (LPWSTR)&lpvMessageBuffer, 0, NULL);

    //
    //... now display this string
    //
    wprintf(L"ERROR: API        = %s.\n", pszAPI);
    wprintf(L"       error code = %d.\n", GetLastError());
    wprintf(L"       message    = %s.\n", (LPWSTR)lpvMessageBuffer);

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

    ExitProcess(GetLastError());
}

void wmain(int argc, WCHAR *argv[])
{
    DWORD     dwSize;
    HANDLE    hToken;
    LPVOID    lpvEnv;
    PROCESS_INFORMATION pi = {0};
    STARTUPINFO         si = {0};
    WCHAR               szUserProfile[256] = L"";

    si.cb = sizeof(STARTUPINFO);

    if (argc != 4)
    {
        wprintf(L"Usage: %s [user@domain] [password] [cmd]", argv[0]);
        wprintf(L"\n\n");
        return;
    }

    //
    // TO DO: change NULL to '.' to use local account database
    //
    if (!LogonUser(argv[1], NULL, argv[2], LOGON32_LOGON_INTERACTIVE,
```

```
            LOGON32_PROVIDER_DEFAULT, &hToken))
        DisplayError(L"LogonUser");

    if (!CreateEnvironmentBlock(&lpvEnv, hToken, TRUE))
        DisplayError(L"CreateEnvironmentBlock");

    dwSize = sizeof(szUserProfile)/sizeof(WCHAR);

    if (!GetUserProfileDirectory(hToken, szUserProfile, &dwSize))
        DisplayError(L"GetUserProfileDirectory");

    //
    // TO DO: change NULL to '.' to use local account database
    //
    if (!CreateProcessWithLogonW(argv[1], NULL, argv[2],
            LOGON_WITH_PROFILE, NULL, argv[3],
            CREATE_UNICODE_ENVIRONMENT, lpvEnv, szUserProfile,
            &si, &pi))
        DisplayError(L"CreateProcessWithLogonW");

    if (!DestroyEnvironmentBlock(lpvEnv))
        DisplayError(L"DestroyEnvironmentBlock");

    CloseHandle(hToken);
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Advapi32.lib
DLL	Advapi32.dll

See also

CloseHandle

CreateEnvironmentBlock

CreateProcessAsUser

ExitProcess

GetEnvironmentStrings

GetExitCodeProcess

OpenProcess

PROCESS_INFORMATION

Process and Thread Functions

Processes

STARTUPINFO

SetErrorMode

WaitForInputIdle

CreateProcessWithTokenW function (winbase.h)

Article02/02/2023

Creates a new process and its primary thread. The new process runs in the security context of the specified token. It can optionally load the user profile for the specified user.

The process that calls **CreateProcessWithTokenW** must have the SE_IMPERSONATE_NAME privilege. If this function fails with ERROR_PRIVILEGE_NOT_HELD (1314), use the CreateProcessAsUser or CreateProcessWithLogonW function instead. Typically, the process that calls
CreateProcessAsUser must have the SE_INCREASE_QUOTA_NAME privilege and may require the SE_ASSIGNPRIMARYTOKEN_NAME privilege if the token is not assignable.
CreateProcessWithLogonW requires no special privileges, but the specified user account must be allowed to log on interactively. Generally, it is best to use **CreateProcessWithLogonW** to create a process with alternate credentials.

Syntax

C++

```
BOOL CreateProcessWithTokenW(
  [in]                HANDLE                hToken,
  [in]                DWORD                 dwLogonFlags,
  [in, optional]      LPCWSTR               lpApplicationName,
  [in, out, optional] LPWSTR                lpCommandLine,
  [in]                DWORD                 dwCreationFlags,
  [in, optional]      LPVOID                lpEnvironment,
  [in, optional]      LPCWSTR               lpCurrentDirectory,
  [in]                LPSTARTUPINFOW        lpStartupInfo,
  [out]               LPPROCESS_INFORMATION lpProcessInformation
);
```

Parameters

`[in] hToken`

A handle to the primary token that represents a user. The handle must have the TOKEN_QUERY, TOKEN_DUPLICATE, and TOKEN_ASSIGN_PRIMARY access rights. For more information, see Access Rights for Access-Token Objects. The user represented by the token must have read and execute access to the application specified by the *lpApplicationName* or the *lpCommandLine* parameter.

To get a primary token that represents the specified user, call the LogonUser function. Alternatively, you can call the DuplicateTokenEx function to convert an impersonation token into a primary token. This allows a server application that is impersonating a client to create a process that has the security context of the client.

Terminal Services: The caller's process always runs in the caller's session, not in the session specified in the token. To run a process in the session specified in the token, use the CreateProcessAsUser function.

`[in] dwLogonFlags`

The logon option. This parameter can be zero or one of the following values.

Expand table

Value	Meaning
LOGON_WITH_PROFILE 0x00000001	Log on, then load the user's profile in the **HKEY_USERS** registry key. The function returns after the profile has been loaded. Loading the profile can be time-consuming, so it is best to use this value only if you must access the information in the **HKEY_CURRENT_USER** registry key. **Windows Server 2003:** The profile is unloaded after the new process has been terminated, regardless of whether it has created child processes.
LOGON_NETCREDENTIALS_ONLY 0x00000002	Log on, but use the specified credentials on the network only. The new process uses the same token as the caller, but the system creates a new logon session within LSA, and the process uses the specified credentials as the default credentials. This value can be used to create a process that uses a different set of credentials locally than it does remotely. This is useful in inter-domain scenarios where there is no trust relationship. The system does not validate the specified credentials. Therefore, the process can start, but it may not have access to network resources.

`[in, optional] lpApplicationName`

The name of the module to be executed. This module can be a Windows-based application. It can be some other type of module (for example, MS-DOS or OS/2) if the appropriate subsystem is available on the local computer.

The string can specify the full path and file name of the module to execute or it can specify a partial name. In the case of a partial name, the function uses the current drive and current directory to complete the specification. The function will not use the search path. This parameter must include the file name extension; no default extension is assumed.

The *lpApplicationName* parameter can be NULL. In that case, the module name must be the first white space–delimited token in the *lpCommandLine* string. If you are using a long file name that contains a space, use quoted strings to indicate where the file name ends and the arguments begin; otherwise, the file name is ambiguous. For example, consider the string "c:\program files\sub dir\program name". This string can be interpreted in a number of ways. The system tries to interpret the possibilities in the following order:

c:\program.exe c:\program files\sub.exe c:\program files\sub dir\program.exe c:\program files\sub dir\program name.exe If the executable module is a 16-bit application, *lpApplicationName* should be NULL, and the string pointed to by *lpCommandLine* should specify the executable module as well as its arguments.

`[in, out, optional] lpCommandLine`

The command line to be executed.

The maximum length of this string is 1024 characters. If *lpApplicationName* is NULL, the module name portion of *lpCommandLine* is limited to MAX_PATH characters.

The function can modify the contents of this string. Therefore, this parameter cannot be a pointer to read-only memory (such as a **const** variable or a literal string). If this parameter is a constant string, the function may cause an access violation.

The *lpCommandLine* parameter can be NULL. In that case, the function uses the string pointed to by *lpApplicationName* as the command line.

If both *lpApplicationName* and *lpCommandLine* are non-NULL, **lpApplicationName* specifies the module to execute, and **lpCommandLine* specifies the command line. The new process can use GetCommandLine to retrieve the entire command line. Console processes written in C can use the *argc* and *argv* arguments to parse the command line. Because argv[0] is the module name, C programmers generally repeat the module name as the first token in the command line.

If *lpApplicationName* is NULL, the first white space–delimited token of the command line specifies the module name. If you are using a long file name that contains a space, use quoted strings to indicate where the file name ends and the arguments begin (see the explanation for the *lpApplicationName* parameter). If the file name does not contain an extension, .exe is appended. Therefore, if the file name extension is .com, this parameter must include the .com extension. If the file name ends in a period (.) with no extension, or if the file name contains a

path, .exe is not appended. If the file name does not contain a directory path, the system searches for the executable file in the following sequence:

1. The directory from which the application loaded.
2. The current directory for the parent process.
3. The 32-bit Windows system directory. Use the GetSystemDirectory function to get the path of this directory.
4. The 16-bit Windows system directory. There is no function that obtains the path of this directory, but it is searched.
5. The Windows directory. Use the GetWindowsDirectory function to get the path of this directory.
6. The directories that are listed in the PATH environment variable. Note that this function does not search the per-application path specified by the **App Paths** registry key. To include this per-application path in the search sequence, use the ShellExecute function.

The system adds a null character to the command line string to separate the file name from the arguments. This divides the original string into two strings for internal processing.

`[in] dwCreationFlags`

The flags that control how the process is created. The CREATE_DEFAULT_ERROR_MODE, CREATE_NEW_CONSOLE, and CREATE_NEW_PROCESS_GROUP flags are enabled by default. For a list of values, see Process Creation Flags.

This parameter also controls the new process's priority class, which is used to determine the scheduling priorities of the process's threads. For a list of values, see GetPriorityClass. If none of the priority class flags is specified, the priority class defaults to NORMAL_PRIORITY_CLASS unless the priority class of the creating process is IDLE_PRIORITY_CLASS or BELOW_NORMAL_PRIORITY_CLASS. In this case, the child process receives the default priority class of the calling process.

If the dwCreationFlags parameter has a value of 0:

- The process gets the default error mode, creates a new console and creates a new process group.
- The environment block for the new process is assumed to contain ANSI characters (see *lpEnvironment* parameter for additional information).
- A 16-bit Windows-based application runs in a shared Virtual DOS machine (VDM).

`[in, optional] lpEnvironment`

A pointer to an environment block for the new process. If this parameter is NULL, the new process uses an environment created from the profile of the user specified by *lpUsername*.

An environment block consists of a null-terminated block of null-terminated strings. Each string is in the following form:

name=value

Because the equal sign (=) is used as a separator, it must not be used in the name of an environment variable.

An environment block can contain Unicode or ANSI characters. If the environment block pointed to by *lpEnvironment* contains Unicode characters, be sure that *dwCreationFlags* includes CREATE_UNICODE_ENVIRONMENT.

An ANSI environment block is terminated by two zero bytes: one for the last string, one more to terminate the block. A Unicode environment block is terminated by four zero bytes: two for the last string and two more to terminate the block.

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

`[in, optional] lpCurrentDirectory`

The full path to the current directory for the process. The string can also specify a UNC path.

If this parameter is NULL, the new process will have the same current drive and directory as the calling process. (This feature is provided primarily for shells that need to start an application and specify its initial drive and working directory.)

`[in] lpStartupInfo`

A pointer to a STARTUPINFO or STARTUPINFOEX structure.

If the **lpDesktop** member is NULL or an empty string, the new process inherits the desktop and window station of its parent process. The function adds permission for the specified user account to the inherited window station and desktop. Otherwise, if this member specifies a desktop, it is the responsibility of the application to add permission for the specified user account to the specified window station and desktop, even for WinSta0\Default.

Handles in STARTUPINFO or STARTUPINFOEX must be closed with CloseHandle when they are no longer needed.

> **Important** If the **dwFlags** member of the **STARTUPINFO** structure specifies **STARTF_USESTDHANDLES**, the standard handle fields are copied unchanged to the child process without validation. The caller is responsible for ensuring that these fields contain

valid handle values. Incorrect values can cause the child process to misbehave or crash. Use the **Application Verifier** runtime verification tool to detect invalid handles.

`[out] lpProcessInformation`

A pointer to a PROCESS_INFORMATION structure that receives identification information for the new process, including a handle to the process.

Handles in PROCESS_INFORMATION must be closed with the CloseHandle function when they are no longer needed.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Note that the function returns before the process has finished initialization. If a required DLL cannot be located or fails to initialize, the process is terminated. To get the termination status of a process, call GetExitCodeProcess.

Remarks

By default, **CreateProcessWithTokenW** does not load the specified user's profile into the **HKEY_USERS** registry key. This means that access to information in the **HKEY_CURRENT_USER** registry key may not produce results consistent with a normal interactive logon. It is your responsibility to load the user's registry hive into **HKEY_USERS** by either using LOGON_WITH_PROFILE, or by calling the LoadUserProfile function before calling this function.

If the *lpEnvironment* parameter is NULL, the new process uses an environment block created from the profile of the user specified by *lpUserName*. If the HOMEDRIVE and HOMEPATH variables are not set, **CreateProcessWithTokenW** modifies the environment block to use the drive and path of the user's working directory.

When created, the new process and thread handles receive full access rights (PROCESS_ALL_ACCESS and THREAD_ALL_ACCESS). For either handle, if a security descriptor is not provided, the handle can be used in any function that requires an object handle of that type. When a security descriptor is provided, an access check is performed on all subsequent uses of the handle before access is granted. If access is denied, the requesting process cannot use the handle to gain access to the process or thread.

To retrieve a security token, pass the process handle in the PROCESS_INFORMATION structure to the OpenProcessToken function.

The process is assigned a process identifier. The identifier is valid until the process terminates. It can be used to identify the process, or specified in the OpenProcess function to open a handle to the process. The initial thread in the process is also assigned a thread identifier. It can be specified in the OpenThread function to open a handle to the thread. The identifier is valid until the thread terminates and can be used to uniquely identify the thread within the system. These identifiers are returned in PROCESS_INFORMATION.

The calling thread can use the WaitForInputIdle function to wait until the new process has finished its initialization and is waiting for user input with no input pending. This can be useful for synchronization between parent and child processes, because **CreateProcessWithTokenW** returns without waiting for the new process to finish its initialization. For example, the creating process would use **WaitForInputIdle** before trying to find a window associated with the new process.

The preferred way to shut down a process is by using the ExitProcess function, because this function sends notification of approaching termination to all DLLs attached to the process. Other means of shutting down a process do not notify the attached DLLs. Note that when a thread calls **ExitProcess**, other threads of the process are terminated without an opportunity to execute any additional code (including the thread termination code of attached DLLs). For more information, see Terminating a Process.

To compile an application that uses this function, define _WIN32_WINNT as 0x0500 or later. For more information, see Using the Windows Headers.

Security Remarks

The *lpApplicationName* parameter can be NULL, in which case the executable name must be the first white space–delimited string in *lpCommandLine*. If the executable or path name has a space in it, there is a risk that a different executable could be run because of the way the function parses spaces. The following example is dangerous because the function will attempt to run "Program.exe", if it exists, instead of "MyApp.exe".

syntax

```
LPTSTR szCmdline = L"C:\\Program Files\\MyApp";
CreateProcessWithTokenW(/*...*/, szCmdline, /*...*/);
```

If a malicious user were to create an application called "Program.exe" on a system, any program that incorrectly calls **CreateProcessWithTokenW** using the Program Files directory will run this application instead of the intended application.

To avoid this problem, do not pass NULL for *lpApplicationName*. If you do pass NULL for *lpApplicationName*, use quotation marks around the executable path in *lpCommandLine*, as shown in the example below.

syntax

```
LPTSTR szCmdline = L"\"C:\\Program Files\\MyApp\"";
CreateProcessWithTokenW(/*...*/, szCmdline, /*...*/);
```

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Advapi32.lib
DLL	Advapi32.dll

See also

CloseHandle

CreateEnvironmentBlock

ExitProcess

GetEnvironmentStrings

GetExitCodeProcess

OpenProcess

PROCESS_INFORMATION

Process and Thread Functions

Processes

STARTUPINFO

SetErrorMode

WaitForInputIdle

CreateRemoteThread function (processthreadsapi.h)

11/01/2022

Creates a thread that runs in the virtual address space of another process.

Use the CreateRemoteThreadEx function to create a thread that runs in the virtual address space of another process and optionally specify extended attributes.

Syntax

C++

```
HANDLE CreateRemoteThread(
  [in]  HANDLE                 hProcess,
  [in]  LPSECURITY_ATTRIBUTES  lpThreadAttributes,
  [in]  SIZE_T                 dwStackSize,
  [in]  LPTHREAD_START_ROUTINE lpStartAddress,
  [in]  LPVOID                 lpParameter,
  [in]  DWORD                  dwCreationFlags,
  [out] LPDWORD                lpThreadId
);
```

Parameters

`[in] hProcess`

A handle to the process in which the thread is to be created. The handle must have the **PROCESS_CREATE_THREAD**, **PROCESS_QUERY_INFORMATION**, **PROCESS_VM_OPERATION**, **PROCESS_VM_WRITE**, and **PROCESS_VM_READ** access rights, and may fail without these rights on certain platforms. For more information, see Process Security and Access Rights.

`[in] lpThreadAttributes`

A pointer to a SECURITY_ATTRIBUTES structure that specifies a security descriptor for the new thread and determines whether child processes can inherit the returned handle. If *lpThreadAttributes* is NULL, the thread gets a default security descriptor and the handle cannot be inherited. The access control lists (ACL) in the default security descriptor for a thread come from the primary token of the creator.

`[in] dwStackSize`

The initial size of the stack, in bytes. The system rounds this value to the nearest page. If this parameter is 0 (zero), the new thread uses the default size for the executable. For more information, see Thread Stack Size.

`[in] lpStartAddress`

A pointer to the application-defined function of type **LPTHREAD_START_ROUTINE** to be executed by the thread and represents the starting address of the thread in the remote process. The function must exist in the remote process. For more information, see ThreadProc.

`[in] lpParameter`

A pointer to a variable to be passed to the thread function.

`[in] dwCreationFlags`

The flags that control the creation of the thread.

Expand table

Value	Meaning
0	The thread runs immediately after creation.
CREATE_SUSPENDED 0x00000004	The thread is created in a suspended state, and does not run until the ResumeThread function is called.
STACK_SIZE_PARAM_IS_A_RESERVATION 0x00010000	The *dwStackSize* parameter specifies the initial reserve size of the stack. If this flag is not specified, *dwStackSize* specifies the commit size.

`[out] lpThreadId`

A pointer to a variable that receives the thread identifier.

If this parameter is **NULL**, the thread identifier is not returned.

Return value

If the function succeeds, the return value is a handle to the new thread.

If the function fails, the return value is **NULL**. To get extended error information, call GetLastError.

Note that **CreateRemoteThread** may succeed even if *lpStartAddress* points to data, code, or is not accessible. If the start address is invalid when the thread runs, an exception occurs, and the

thread terminates. Thread termination due to a invalid start address is handled as an error exit for the thread's process. This behavior is similar to the asynchronous nature of CreateProcess, where the process is created even if it refers to invalid or missing dynamic-link libraries (DLL).

Remarks

The **CreateRemoteThread** function causes a new thread of execution to begin in the address space of the specified process. The thread has access to all objects that the process opens.

Prior to Windows 8, Terminal Services isolates each terminal session by design. Therefore, **CreateRemoteThread** fails if the target process is in a different session than the calling process.

The new thread handle is created with full access to the new thread. If a security descriptor is not provided, the handle may be used in any function that requires a thread object handle. When a security descriptor is provided, an access check is performed on all subsequent uses of the handle before access is granted. If the access check denies access, the requesting process cannot use the handle to gain access to the thread.

If the thread is created in a runnable state (that is, if the **CREATE_SUSPENDED** flag is not used), the thread can start running before CreateThread returns and, in particular, before the caller receives the handle and identifier of the created thread.

The thread is created with a thread priority of **THREAD_PRIORITY_NORMAL**. Use the GetThreadPriority and SetThreadPriority functions to get and set the priority value of a thread.

When a thread terminates, the thread object attains a signaled state, which satisfies the threads that are waiting for the object.

The thread object remains in the system until the thread has terminated and all handles to it are closed through a call to CloseHandle.

The ExitProcess, ExitThread, CreateThread, **CreateRemoteThread** functions, and a process that is starting (as the result of a CreateProcess call) are serialized between each other within a process. Only one of these events occurs in an address space at a time. This means the following restrictions hold:

- During process startup and DLL initialization routines, new threads can be created, but they do not begin execution until DLL initialization is done for the process.
- Only one thread in a process can be in a DLL initialization or detach routine at a time.
- ExitProcess returns after all threads have completed their DLL initialization or detach routines.

A common use of this function is to inject a thread into a process that is being debugged to issue a break. However, this use is not recommended, because the extra thread is confusing to

the person debugging the application and there are several side effects to using this technique:

- It converts single-threaded applications into multithreaded applications.
- It changes the timing and memory layout of the process.
- It results in a call to the entry point of each DLL in the process.

Another common use of this function is to inject a thread into a process to query heap or other process information. This can cause the same side effects mentioned in the previous paragraph. Also, the application can deadlock if the thread attempts to obtain ownership of locks that another thread is using.

Requirements

Expand table

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

See also

CloseHandle

CreateProcess

CreateRemoteThreadEx

CreateThread

ExitProcess

ExitThread

GetThreadPriority

Process and Thread Functions

ResumeThread

SECURITY_ATTRIBUTES

SetThreadPriority

ThreadProc

Threads

CreateRemoteThreadEx function (processthreadsapi.h)

11/01/2022

Creates a thread that runs in the virtual address space of another process and optionally specifies extended attributes such as processor group affinity.

Syntax

```C++
HANDLE CreateRemoteThreadEx(
  [in]            HANDLE                       hProcess,
  [in, optional]  LPSECURITY_ATTRIBUTES        lpThreadAttributes,
  [in]            SIZE_T                       dwStackSize,
  [in]            LPTHREAD_START_ROUTINE       lpStartAddress,
  [in, optional]  LPVOID                       lpParameter,
  [in]            DWORD                        dwCreationFlags,
  [in, optional]  LPPROC_THREAD_ATTRIBUTE_LIST lpAttributeList,
  [out, optional] LPDWORD                      lpThreadId
);
```

Parameters

`[in] hProcess`

A handle to the process in which the thread is to be created. The handle must have the PROCESS_CREATE_THREAD, PROCESS_QUERY_INFORMATION, PROCESS_VM_OPERATION, PROCESS_VM_WRITE, and PROCESS_VM_READ access rights. In Windows 10, version 1607, your code must obtain these access rights for the new handle. However, starting in Windows 10, version 1703, if the new handle is entitled to these access rights, the system obtains them for you. For more information, see Process Security and Access Rights.

`[in, optional] lpThreadAttributes`

A pointer to a SECURITY_ATTRIBUTES structure that specifies a security descriptor for the new thread and determines whether child processes can inherit the returned handle. If *lpThreadAttributes* is NULL, the thread gets a default security descriptor and the handle cannot be inherited. The access control lists (ACL) in the default security descriptor for a thread come from the primary token of the creator.

`[in] dwStackSize`

The initial size of the stack, in bytes. The system rounds this value to the nearest page. If this parameter is 0 (zero), the new thread uses the default size for the executable. For more information, see Thread Stack Size.

`[in] lpStartAddress`

A pointer to the application-defined function of type **LPTHREAD_START_ROUTINE** to be executed by the thread and represents the starting address of the thread in the remote process. The function must exist in the remote process. For more information, see ThreadProc.

`[in, optional] lpParameter`

A pointer to a variable to be passed to the thread function pointed to by *lpStartAddress*. This parameter can be NULL.

`[in] dwCreationFlags`

The flags that control the creation of the thread.

Expand table

Value	Meaning
0	The thread runs immediately after creation.
CREATE_SUSPENDED 0x00000004	The thread is created in a suspended state and does not run until the ResumeThread function is called.
STACK_SIZE_PARAM_IS_A_RESERVATION 0x00010000	The *dwStackSize* parameter specifies the initial reserve size of the stack. If this flag is not specified, *dwStackSize* specifies the commit size.

`[in, optional] lpAttributeList`

An attribute list that contains additional parameters for the new thread. This list is created by the InitializeProcThreadAttributeList function.

`[out, optional] lpThreadId`

A pointer to a variable that receives the thread identifier.

If this parameter is NULL, the thread identifier is not returned.

Return value

If the function succeeds, the return value is a handle to the new thread.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Remarks

The **CreateRemoteThreadEx** function causes a new thread of execution to begin in the address space of the specified process. The thread has access to all objects that the process opens. The *lpAttribute* parameter can be used to specify extended attributes such as processor group affinity for the new thread. If *lpAttribute* is NULL, the function's behavior is the same as CreateRemoteThread.

Prior to Windows 8, Terminal Services isolates each terminal session by design. Therefore, CreateRemoteThread fails if the target process is in a different session than the calling process.

The new thread handle is created with full access to the new thread. If a security descriptor is not provided, the handle may be used in any function that requires a thread object handle. When a security descriptor is provided, an access check is performed on all subsequent uses of the handle before access is granted. If the access check denies access, the requesting process cannot use the handle to gain access to the thread.

If the thread is created in a runnable state (that is, if the CREATE_SUSPENDED flag is not used), the thread can start running before CreateThread returns and, in particular, before the caller receives the handle and identifier of the created thread.

The thread is created with a thread priority of THREAD_PRIORITY_NORMAL. To get and set the priority value of a thread, use the GetThreadPriority and SetThreadPriority functions.

When a thread terminates, the thread object attains a signaled state, which satisfies the threads that are waiting for the object.

The thread object remains in the system until the thread has terminated and all handles to it are closed through a call to CloseHandle.

The ExitProcess, ExitThread, CreateThread, CreateRemoteThread functions, and a process that is starting (as the result of a CreateProcess call) are serialized between each other within a process. Only one of these events occurs in an address space at a time. This means the following restrictions hold:

- During process startup and DLL initialization routines, new threads can be created, but they do not begin execution until DLL initialization is done for the process.
- Only one thread in a process can be in a DLL initialization or detach routine at a time.
- ExitProcess returns after all threads have completed their DLL initialization or detach routines.

A common use of this function is to inject a thread into a process that is being debugged to issue a break. However, this use is not recommended, because the extra thread is confusing to the person debugging the application and there are several side effects to using this technique:

- It converts single-threaded applications into multithreaded applications.
- It changes the timing and memory layout of the process.
- It results in a call to the entry point of each DLL in the process.

Another common use of this function is to inject a thread into a process to query heap or other process information. This can cause the same side effects mentioned in the previous paragraph. Also, the application can deadlock if the thread attempts to obtain ownership of locks that another thread is using.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	processthreadsapi.h (include Windows.h on Windows Server 2008 Windows Server 2008 R2)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

CreateRemoteThread

CreateThread function (processthreadsapi.h)

07/31/2023

Creates a thread to execute within the virtual address space of the calling process.

To create a thread that runs in the virtual address space of another process, use the CreateRemoteThread function.

Syntax

```C++
HANDLE CreateThread(
  [in, optional]  LPSECURITY_ATTRIBUTES   lpThreadAttributes,
  [in]            SIZE_T                  dwStackSize,
  [in]            LPTHREAD_START_ROUTINE  lpStartAddress,
  [in, optional]  __drv_aliasesMem LPVOID lpParameter,
  [in]            DWORD                   dwCreationFlags,
  [out, optional] LPDWORD                 lpThreadId
);
```

Parameters

`[in, optional] lpThreadAttributes`

A pointer to a SECURITY_ATTRIBUTES structure that specifies a security descriptor for the new thread and determines whether child processes can inherit the returned handle. If *lpThreadAttributes* is NULL, the thread gets a default security descriptor and the handle cannot be inherited. The access control lists (ACL) in the default security descriptor for a thread come from the primary token of the creator.

`[in] dwStackSize`

The initial size of the stack, in bytes. The system rounds this value to the nearest page. If this parameter is zero, the new thread uses the default size for the executable. For more information, see Thread Stack Size.

`[in] lpStartAddress`

A pointer to the application-defined function to be executed by the thread. This pointer represents the starting address of the thread. For more information on the thread function, see

ThreadProc.

`[in, optional] lpParameter`

A pointer to a variable to be passed to the thread.

`[in] dwCreationFlags`

The flags that control the creation of the thread.

Expand table

Value	Meaning
0	The thread runs immediately after creation.
CREATE_SUSPENDED 0x00000004	The thread is created in a suspended state, and does not run until the ResumeThread function is called.
STACK_SIZE_PARAM_IS_A_RESERVATION 0x00010000	The *dwStackSize* parameter specifies the initial reserve size of the stack. If this flag is not specified, *dwStackSize* specifies the commit size.

`[out, optional] lpThreadId`

A pointer to a variable that receives the thread identifier. If this parameter is **NULL**, the thread identifier is not returned.

Return value

If the function succeeds, the return value is a handle to the new thread.

If the function fails, the return value is **NULL**. To get extended error information, call GetLastError.

Note that **CreateThread** may succeed even if *lpStartAddress* points to data, code, or is not accessible. If the start address is invalid when the thread runs, an exception occurs, and the thread terminates. Thread termination due to a invalid start address is handled as an error exit for the thread's process. This behavior is similar to the asynchronous nature of CreateProcess, where the process is created even if it refers to invalid or missing dynamic-link libraries (DLLs).

Remarks

The number of threads a process can create is limited by the available virtual memory. By default, every thread has one megabyte of stack space. Therefore, you cannot create 2,048 or

more threads on a 32-bit system without `/3GB` boot.ini option. If you reduce the default stack size, you can create more threads. However, your application will have better performance if 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.

The new thread handle is created with the **THREAD_ALL_ACCESS** access right. If a security descriptor is not provided when the thread is created, a default security descriptor is constructed for the new thread using the primary token of the process that is creating the thread. When a caller attempts to access the thread with the OpenThread function, the effective token of the caller is evaluated against this security descriptor to grant or deny access.

The newly created thread has full access rights to itself when calling the GetCurrentThread function.

Windows Server 2003: The thread's access rights to itself are computed by evaluating the primary token of the process in which the thread was created against the default security descriptor constructed for the thread. If the thread is created in a remote process, the primary token of the remote process is used. As a result, the newly created thread may have reduced access rights to itself when calling GetCurrentThread. Some access rights including **THREAD_SET_THREAD_TOKEN** and **THREAD_GET_CONTEXT** may not be present, leading to unexpected failures. For this reason, creating a thread while impersonating another user is not recommended.

If the thread is created in a runnable state (that is, if the **CREATE_SUSPENDED** flag is not used), the thread can start running before **CreateThread** returns and, in particular, before the caller receives the handle and identifier of the created thread.

The thread execution begins at the function specified by the *lpStartAddress* parameter. If this function returns, the **DWORD** return value is used to terminate the thread in an implicit call to the ExitThread function. Use the GetExitCodeThread function to get the thread's return value.

The thread is created with a thread priority of **THREAD_PRIORITY_NORMAL**. Use the GetThreadPriority and SetThreadPriority functions to get and set the priority value of a thread.

When a thread terminates, the thread object attains a signaled state, satisfying any threads that were waiting on the object.

The thread object remains in the system until the thread has terminated and all handles to it have been closed through a call to CloseHandle.

The ExitProcess, ExitThread, **CreateThread**, CreateRemoteThread functions, and a process that is starting (as the result of a call by CreateProcess) are serialized between each other within a

process. Only one of these events can happen in an address space at a time. This means that the following restrictions hold:

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

A thread in an executable that calls the C run-time library (CRT) should use the _beginthreadex and _endthreadex functions for thread management rather than **CreateThread** and ExitThread; this requires the use of the multithreaded version of the CRT. If a thread created using **CreateThread** calls the CRT, the CRT may terminate the process in low-memory conditions.

Windows Phone 8.1: This function is supported for Windows Phone Store apps on Windows Phone 8.1 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Examples

For an example, see Creating Threads.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	processthreadsapi.h (include Windows.h on Windows Server 2003, Windows Vista, Windows 7, Windows Server 2008 Windows Server 2008 R2)
Library	Kernel32.lib; WindowsPhoneCore.lib on Windows Phone 8.1
DLL	Kernel32.dll; KernelBase.dll on Windows Phone 8.1

See also

CloseHandle

CreateProcess

CreateRemoteThread

ExitProcess

ExitThread

GetExitCodeThread

GetThreadPriority

Process and Thread Functions

ResumeThread

SECURITY_ATTRIBUTES

SetThreadPriority

SuspendThread

ThreadProc

Threads

CreateThreadpool function (threadpoolapiset.h)

Article02/22/2024

Allocates a new pool of threads to execute callbacks.

Syntax

```
C++

PTP_POOL CreateThreadpool(
  PVOID reserved
);
```

Parameters

`reserved`

This parameter is reserved and must be NULL.

Return value

If the function succeeds, it returns a pointer to a **TP_POOL** structure representing the newly allocated thread pool. Applications do not modify the members of this structure.

If function fails, it returns NULL. To retrieve extended error information, call GetLastError.

Remarks

After creating the new thread pool, you should call SetThreadpoolThreadMaximum to specify the maximum number of threads that the pool can allocate and SetThreadpoolThreadMinimum to specify the minimum number of threads available in the pool.

To use the pool, you must associate the pool with a callback environment. To create the callback environment, call InitializeThreadpoolEnvironment. Then, call SetThreadpoolCallbackPool to associate the pool with the callback environment.

To release the thread pool, call CloseThreadpool.

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

Examples

For an example, see Using the Thread Pool Functions.

Requirements

Expand table

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

See also

CloseThreadpool

SetThreadpoolThreadMaximum

SetThreadpoolThreadMinimum

Thread Pools

CreateThreadpoolCleanupGroup function (threadpoolapiset.h)

Article02/22/2024

Creates a cleanup group that applications can use to track one or more thread pool callbacks.

Syntax

```
C++

PTP_CLEANUP_GROUP CreateThreadpoolCleanupGroup();
```

Return value

If the function succeeds, it returns a pointer to a **TP_CLEANUP_GROUP** structure of the newly allocated cleanup group. Applications do not modify the members of this structure.

If function fails, it returns NULL. To retrieve extended error information, call GetLastError.

Remarks

After creating the cleanup group, call SetThreadpoolCallbackCleanupGroup to associate the cleanup group with a callback environment.

A member is added to the group each time you call one of the following functions:

- CreateThreadpoolIo
- CreateThreadpoolTimer
- CreateThreadpoolWait
- CreateThreadpoolWork

You use one of the following corresponding close functions to remove a member from the group.

- CloseThreadpoolIo
- CloseThreadpoolTimer
- CloseThreadpoolWait
- CloseThreadpoolWork

To close all the callbacks, call CloseThreadpoolCleanupGroupMembers.

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

Examples

For an example, see Using the Thread Pool Functions.

Requirements

Expand table

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

See also

CloseThreadpoolCleanupGroup

CloseThreadpoolCleanupGroupMembers

SetThreadpoolCallbackCleanupGroup

Thread Pools

CreateThreadpoolIo function (threadpoolapiset.h)

Article02/22/2024

Creates a new I/O completion object.

Syntax

```
C++

PTP_IO CreateThreadpoolIo(
  [in]                HANDLE                fl,
  [in]                PTP_WIN32_IO_CALLBACK pfnio,
  [in, out, optional] PVOID                 pv,
  [in, optional]      PTP_CALLBACK_ENVIRON  pcbe
);
```

Parameters

`[in] fl`

The file handle to bind to this I/O completion object.

`[in] pfnio`

The callback function to be called each time an overlapped I/O operation completes on the file. For details, see IoCompletionCallback.

`[in, out, optional] pv`

Optional application-defined data to pass to the callback function.

`[in, optional] pcbe`

A pointer to a **TP_CALLBACK_ENVIRON** structure that defines the environment in which to execute the callback. Use the InitializeThreadpoolEnvironment function to initialize the structure before calling this function.

If this parameter is NULL, the callback executes in the default callback environment. For more information, see InitializeThreadpoolEnvironment.

Return value

If the function succeeds, it returns a pointer to a **TP_IO** structure that defines the I/O object. Applications do not modify the members of this structure.

If the function fails, it returns NULL. To retrieve extended error information, call GetLastError.

Remarks

To begin receiving overlapped I/O completion callbacks, call the StartThreadpoolIo function.

If the file handle bound to the I/O completion object has the notification mode FILE_SKIP_COMPLETION_PORT_ON_SUCCESS and an asychronous I/O operation returns immediately with success, the I/O completion callback function is not called and threadpool I/O notifications must be canceled. For more information, see CancelThreadpoolIo.

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

Requirements

Expand table

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

See also

CancelThreadpoolIo

CloseThreadpoolIo

StartThreadpoolIo

Thread Pools

WaitForThreadpoolIoCallbacks

CreateThreadpoolTimer function (threadpoolapiset.h)

Article02/22/2024

Creates a new timer object.

Syntax

```
C++

PTP_TIMER CreateThreadpoolTimer(
  [in]                PTP_TIMER_CALLBACK   pfnti,
  [in, out, optional] PVOID                pv,
  [in, optional]      PTP_CALLBACK_ENVIRON pcbe
);
```

Parameters

`[in] pfnti`

The callback function to call each time the timer object expires. For details, see TimerCallback.

`[in, out, optional] pv`

Optional application-defined data to pass to the callback function.

`[in, optional] pcbe`

A **TP_CALLBACK_ENVIRON** structure that defines the environment in which to execute the callback. The InitializeThreadpoolEnvironment function returns this structure.

If this parameter is NULL, the callback executes in the default callback environment. For more information, see InitializeThreadpoolEnvironment.

Return value

If the function succeeds, it returns a pointer to a **TP_TIMER** structure that defines the timer object. Applications do not modify the members of this structure.

If the function fails, it returns NULL. To retrieve extended error information, call GetLastError.

Remarks

To set the timer object, call the SetThreadpoolTimer function.

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

Examples

For an example, see Using the Thread Pool Functions.

Requirements

Expand table

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

See also

CloseThreadpoolTimer

IsThreadpoolTimerSet

SetThreadpoolTimer

Thread Pools

WaitForThreadpoolTimerCallbacks

CreateThreadpoolWait function (threadpoolapiset.h)

Article02/22/2024

Creates a new wait object.

Syntax

```
C++

PTP_WAIT CreateThreadpoolWait(
  [in]                PTP_WAIT_CALLBACK    pfnwa,
  [in, out, optional] PVOID                pv,
  [in, optional]      PTP_CALLBACK_ENVIRON pcbe
);
```

Parameters

`[in] pfnwa`

The callback function to call when the wait completes or times out. For details, see WaitCallback.

`[in, out, optional] pv`

Optional application-defined data to pass to the callback function.

`[in, optional] pcbe`

A **TP_CALLBACK_ENVIRON** structure that defines the environment in which to execute the callback. The InitializeThreadpoolEnvironment function returns this structure.

If this parameter is NULL, the callback executes in the default callback environment. For more information, see InitializeThreadpoolEnvironment.

Return value

If the function succeeds, it returns a pointer to a **TP_WAIT** structure that defines the wait object. Applications do not modify the members of this structure.

If the function fails, it returns NULL. To retrieve extended error information, call GetLastError.

Remarks

To set the wait object, call the SetThreadpoolWait or SetThreadpoolWaitEx function.

The work item and all functions it calls must not rely on the thread after they have returned. Therefore, you cannot call an asynchronous call that requires a persistent thread, such as the RegNotifyChangeKeyValue function without the **REG_NOTIFY_THREAD_AGNOSTIC** flag, from the default callback environment. Instead, use a thread whose lifetime you control.

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

Examples

For an example, see Using the Thread Pool Functions.

Requirements

Expand table

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

See also

CloseThreadpoolWait

SetThreadpoolWait

Thread Pools

WaitForThreadpoolWaitCallbacks

CreateThreadpoolWork function (threadpoolapiset.h)

Article02/22/2024

Creates a new work object.

Syntax

```
C++

PTP_WORK CreateThreadpoolWork(
  [in]                PTP_WORK_CALLBACK    pfnwk,
  [in, out, optional] PVOID                pv,
  [in, optional]      PTP_CALLBACK_ENVIRON pcbe
);
```

Parameters

`[in] pfnwk`

The callback function. A worker thread calls this callback each time you call SubmitThreadpoolWork to post the work object. For details, see WorkCallback.

`[in, out, optional] pv`

Optional application-defined data to pass to the callback function.

`[in, optional] pcbe`

A pointer to a **TP_CALLBACK_ENVIRON** structure that defines the environment in which to execute the callback. Use the InitializeThreadpoolEnvironment function to initialize the structure before calling this function.

If this parameter is NULL, the callback executes in the default callback environment. For more information, see InitializeThreadpoolEnvironment.

Return value

If the function succeeds, it returns a pointer to a **TP_WORK** structure that defines the work object. Applications do not modify the members of this structure.

If the function fails, it returns NULL. To retrieve extended error information, call GetLastError.

Remarks

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

Examples

For an example, see Using the Thread Pool Functions.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps | UWP apps]
Target Platform	Windows
Header	threadpoolapiset.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

CloseThreadpoolWork

SubmitThreadpoolWork

Thread Pools

WaitForThreadpoolWorkCallbacks

CreateUmsCompletionList function (winbase.h)

Article03/18/2022

Creates a user-mode scheduling (UMS) completion list.

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

Syntax

```C++
BOOL CreateUmsCompletionList(
  [out] PUMS_COMPLETION_LIST *UmsCompletionList
);
```

Parameters

`[out] UmsCompletionList`

A **PUMS_COMPLETION_LIST** variable. On output, this parameter receives a pointer to an empty UMS completion list.

Return value

If the function succeeds, it returns a nonzero value.

If the function fails, the return value is zero. To get extended error information, call GetLastError. Possible error values include the following.

Expand table

Return code	Description
ERROR_NOT_ENOUGH_MEMORY	Not enough memory is available to create the completion list.
ERROR_NOT_SUPPORTED	UMS is not supported.

Remarks

A completion list is associated with a UMS scheduler thread when the EnterUmsSchedulingMode function is called to create the scheduler thread. The system queues newly created UMS worker threads to the completion list. It also queues previously blocked UMS worker threads to the completion list when the threads are no longer blocked.

When an application's UmsSchedulerProc entry point function is called, the application's scheduler should retrieve items from the completion list by calling DequeueUmsCompletionListItems.

Each completion list has an associated completion list event which is signaled whenever the system queues items to an empty list. Use the GetUmsCompletionListEvent to obtain a handle to the event for a specified completion list.

When a completion list is no longer needed, use the DeleteUmsCompletionList to release the list. The list must be empty before it can be released.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 (64-bit only) [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
API set	api-ms-win-core-ums-l1-1-0 (introduced in Windows 7)

See also

DequeueUmsCompletionListItems

EnterUmsSchedulingMode

GetUmsCompletionListEvent

CreateUmsThreadContext function (winbase.h)

Article03/18/2022

Creates a user-mode scheduling (UMS) thread context to represent a UMS worker thread.

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

Syntax

```C++
BOOL CreateUmsThreadContext(
  [out] PUMS_CONTEXT *lpUmsThread
);
```

Parameters

`[out] lpUmsThread`

A PUMS_CONTEXT variable. On output, this parameter receives a pointer to a UMS thread context.

Return value

If the function succeeds, it returns a nonzero value.

If the function fails, the return value is zero. To get extended error information, call GetLastError. Possible error values include the following.

Expand table

Return code	Description
ERROR_NOT_ENOUGH_MEMORY	Not enough memory is available to create the UMS thread context.

Remarks

A UMS thread context represents the state of a UMS worker thread. Thread contexts are used to specify UMS worker threads in function calls.

A UMS worker thread is created by calling the CreateRemoteThreadEx function after using InitializeProcThreadAttributeList and UpdateProcThreadAttribute to prepare a list of UMS attributes for the thread.

The underlying structures for a UMS thread context are managed by the system and should not be modified directly. To get and set information about a UMS worker thread, use the QueryUmsThreadInformation and SetUmsThreadInformation functions.

After a UMS worker thread terminates, its thread context should be released by calling DeleteUmsThreadContext.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 (64-bit only) [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
API set	api-ms-win-core-ums-l1-1-0 (introduced in Windows 7)

See also

CreateRemoteThreadEx

DeleteUmsThreadContext

InitializeProcThreadAttributeList

QueryUmsThreadInformation

SetUmsThreadInformation

UpdateProcThreadAttribute

DeleteFiber function (winbase.h)

02/22/2024

Deletes an existing fiber.

Syntax

```
C++

VOID DeleteFiber(
  [in] LPVOID lpFiber
);
```

Parameters

`[in] lpFiber`

The address of the fiber to be deleted.

Return value

None

Remarks

The **DeleteFiber** function deletes all data associated with the fiber. 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 a currently running fiber is deleted by another fiber, the thread running the deleted fiber is likely to terminate abnormally because the fiber stack has been freed.

To compile an application that uses this function, define _WIN32_WINNT as 0x0400 or later. For more information, see Using the Windows Headers.

Examples

For an example, see Using Fibers.

Requirements

Expand table

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

See also

ExitThread

Fibers

Process and Thread Functions

DeleteProcThreadAttributeList function (processthreadsapi.h)

02/22/2024

Deletes the specified list of attributes for process and thread creation.

Syntax

```
C++

VOID DeleteProcThreadAttributeList(
  [in, out] LPPROC_THREAD_ATTRIBUTE_LIST lpAttributeList
);
```

Parameters

`[in, out] lpAttributeList`

The attribute list. This list is created by the InitializeProcThreadAttributeList function.

Return value

None

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	processthreadsapi.h (include Windows.h on Windows 7, Windows Server 2008 Windows Server 2008 R2)

Requirement	Value
Library	Kernel32.lib
DLL	Kernel32.dll

See also

InitializeProcThreadAttributeList

Process and Thread Functions

DeleteUmsCompletionList function (winbase.h)

Article03/18/2022

Deletes the specified user-mode scheduling (UMS) completion list. The list must be empty.

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

Syntax

```C++
BOOL DeleteUmsCompletionList(
  [in] PUMS_COMPLETION_LIST UmsCompletionList
);
```

Parameters

`[in] UmsCompletionList`

A pointer to the UMS completion list to be deleted. The CreateUmsCompletionList function provides this pointer.

Return value

If the function succeeds, it returns a nonzero value.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

If the completion list is shared, the caller is responsible for ensuring that no active UMS thread holds a reference to the list before deleting it.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 (64-bit only) [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
API set	api-ms-win-core-ums-l1-1-0 (introduced in Windows 7)

DeleteUmsThreadContext function (winbase.h)

Article02/22/2024

Deletes the specified user-mode scheduling (UMS) thread context. The thread must be terminated.

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

Syntax

```C++
BOOL DeleteUmsThreadContext(
  [in] PUMS_CONTEXT UmsThread
);
```

Parameters

`[in] UmsThread`

A pointer to the UMS thread context to be deleted. The CreateUmsThreadContext function provides this pointer.

Return value

If the function succeeds, it returns a nonzero value.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

A UMS thread context cannot be deleted until the associated thread has terminated.

When a UMS worker thread finishes running (for example, by returning from its thread entry point function), the system terminates the thread, sets the termination status in the thread's UMS thread context, and queues the UMS thread context to the associated completion list.

Any attempt to execute the UMS thread will fail because the thread is already terminated.

To check the termination status of a thread, the application's scheduler should call QueryUmsThreadInformation with the **UmsIsThreadTerminated** information class.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 (64-bit only) [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
API set	api-ms-win-core-ums-l1-1-0 (introduced in Windows 7)

See also

CreateUmsThreadContext

QueryUmsThreadInformation

DequeueUmsCompletionListItems function (winbase.h)

Article03/18/2022

Retrieves user-mode scheduling (UMS) worker threads from the specified UMS completion list.

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

Syntax

```C++
BOOL DequeueUmsCompletionListItems(
  [in]  PUMS_COMPLETION_LIST UmsCompletionList,
  [in]  DWORD                WaitTimeOut,
  [out] PUMS_CONTEXT         *UmsThreadList
);
```

Parameters

`[in] UmsCompletionList`

A pointer to the completion list from which to retrieve worker threads.

`[in] WaitTimeOut`

The time-out interval for the retrieval operation, in milliseconds. The function returns if the interval elapses, even if no worker threads are queued to the completion list.

If the *WaitTimeOut* parameter is zero, the completion list is checked for available worker threads without waiting for worker threads to become available. If the *WaitTimeOut* parameter is INFINITE, the function's time-out interval never elapses. This is not recommended, however, because it causes the function to block until one or more worker threads become available.

`[out] UmsThreadList`

A pointer to a UMS_CONTEXT variable. On output, this parameter receives a pointer to the first UMS thread context in a list of UMS thread contexts.

If no worker threads are available before the time-out specified by the *WaitTimeOut* parameter, this parameter is set to NULL.

Return value

If the function succeeds, it returns a nonzero value.

If the function fails, the return value is zero. To get extended error information, call GetLastError. Possible error values include the following.

Expand table

Return code	Description
ERROR_TIMEOUT	No threads became available before the specified time-out interval elapsed.
ERROR_NOT_SUPPORTED	UMS is not supported.

Remarks

The system queues a UMS worker thread to a completion list when the worker thread is created or when a previously blocked worker thread becomes unblocked. The **DequeueUmsCompletionListItems** function retrieves a pointer to a list of all thread contexts in the specified completion list. The GetNextUmsListItem function can be used to pop UMS thread contexts off the list into the scheduler's own ready thread queue. The scheduler is responsible for selecting threads to run based on priorities chosen by the application.

Do not run UMS threads directly from the list provided by **DequeueUmsCompletionListItems**, or run a thread transferred from the list to the ready thread queue before the list is completely empty. This can cause unpredictable behavior in the application.

If more than one caller attempts to retrieve threads from a shared completion list, only the first caller retrieves the threads. For subsequent callers, the **DequeueUmsCompletionListItems** function returns success but the *UmsThreadList* parameter is set to NULL.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 (64-bit only) [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
API set	api-ms-win-core-ums-l1-1-0 (introduced in Windows 7)

See also

GetNextUmsListItem

DestroyThreadpoolEnvironment function (winbase.h)

02/22/2024

Deletes the specified callback environment. Call this function when the callback environment is no longer needed for creating new thread pool objects.

Syntax

```C++
VOID DestroyThreadpoolEnvironment(
  [in, out] PTP_CALLBACK_ENVIRON pcbe
);
```

Parameters

`[in, out] pcbe`

A **TP_CALLBACK_ENVIRON** structure that defines the callback environment. The InitializeThreadpoolEnvironment function returns this structure.

Return value

None

Remarks

This function is implemented as an inline function.

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

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps | UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)

See also

InitializeThreadpoolEnvironment

SetThreadpoolCallbackCleanupGroup

SetThreadpoolCallbackLibrary

SetThreadpoolCallbackPool

SetThreadpoolCallbackPriority

SetThreadpoolCallbackRunsLong

Thread Pools

DisassociateCurrentThreadFromCallback function (threadpoolapiset.h)

02/22/2024

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.

Syntax

```C++
VOID DisassociateCurrentThreadFromCallback(
  [in, out] PTP_CALLBACK_INSTANCE pci
);
```

Parameters

`[in, out] pci`

A pointer to a **TP_CALLBACK_INSTANCE** structure that defines the callback instance. The pointer is passed to the callback function.

Return value

None

Remarks

If this is the last thread executing a callback on behalf of the object, any threads waiting for the object's callbacks to complete will be released.

The thread remains associated with the object's cleanup group until the thread returns to the thread pool. This lets DLL shutdown routines safely synchronize with outstanding callbacks and proceed with unloading the DLL's code when all callbacks have completed.

The callback-generating object remains valid for the duration of the callback. The callback object may be reused or released (although synchronization with cleanup group release is still required).

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

Requirements

Expand table

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

See also

CallbackMayRunLong

FreeLibraryWhenCallbackReturns

LeaveCriticalSectionWhenCallbackReturns

ReleaseMutexWhenCallbackReturns

ReleaseSemaphoreWhenCallbackReturns

SetEventWhenCallbackReturns

Thread Pools

TrySubmitThreadpoolCallback

EnterUmsSchedulingMode function (winbase.h)

Article03/18/2022

Converts the calling thread into a user-mode scheduling (UMS) scheduler thread.

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

Syntax

```C++
BOOL EnterUmsSchedulingMode(
  [in] PUMS_SCHEDULER_STARTUP_INFO SchedulerStartupInfo
);
```

Parameters

`[in] SchedulerStartupInfo`

A pointer to a UMS_SCHEDULER_STARTUP_INFO structure that specifies UMS attributes for the thread, including a completion list and a UmsSchedulerProc entry point function.

Return value

If the function succeeds, it returns a nonzero value.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

An application's UMS scheduler creates one UMS scheduler thread for each processor that will be used to run UMS threads. The scheduler typically sets the affinity of the scheduler thread for a single processor, effectively reserving the processor for the use of that scheduler thread. For more information about thread affinity, see Multiple Processors.

When a UMS scheduler thread is created, the system calls the UmsSchedulerProc entry point function specified with the **EnterUmsSchedulingMode** function call. The application's scheduler is responsible for finishing any application-specific initialization of the scheduler thread and selecting a UMS worker thread to run.

The application's scheduler selects a UMS worker thread to run by calling ExecuteUmsThread with the worker thread's UMS thread context. The worker thread runs until it yields control by calling UmsThreadYield, blocks, or terminates. The scheduler thread is then available to run another worker thread.

A scheduler thread should continue to run until all of its worker threads reach a natural stopping point: that is, all worker threads have yielded, blocked, or terminated.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 (64-bit only) [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
API set	api-ms-win-core-ums-l1-1-0 (introduced in Windows 7)

See also

ExecuteUmsThread

Multiple Processors

UMS_SCHEDULER_STARTUP_INFO

UmsSchedulerProc

User-Mode Scheduling

ExecuteUmsThread function (winbase.h)

Article03/18/2022

Runs the specified UMS worker thread.

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

Syntax

```C++
BOOL ExecuteUmsThread(
  [in, out] PUMS_CONTEXT UmsThread
);
```

Parameters

`[in, out] UmsThread`

A pointer to the UMS thread context of the worker thread to run.

Return value

If the function succeeds, it does not return a value.

If the function fails, the return value is zero. To get extended error information, call GetLastError. Possible error codes include the following.

Expand table

Return code	Description
ERROR_RETRY	The specified UMS worker thread is temporarily locked by the system. The caller can retry the operation.
ERROR_NOT_SUPPORTED	UMS is not supported.

Remarks

The **ExecuteUmsThread** function loads the state of the specified UMS worker thread over the state of the calling UMS scheduler thread so that the worker thread can run. The worker thread runs until it yields by calling the UmsThreadYield function, blocks, or terminates.

When a worker thread yields or blocks, the system calls the scheduler thread's UmsSchedulerProc entry point function. When a previously blocked worker thread becomes unblocked, the system queues the worker thread to the completion list specified with the UpdateProcThreadAttribute function when the worker thread was created.

The **ExecuteUmsThread** function does not return unless an error occurs. If the function returns ERROR_RETRY, the error is transitory and the operation can be retried.

If the function returns an error other than ERROR_RETRY, the application's scheduler should check whether the thread is suspended or terminated by calling QueryUmsThreadInformation with **UmsThreadIsSuspended** or **UmsThreadIsTerminated**, respectively. Other possible errors include calling the function on a thread that is not a UMS scheduler thread, passing an invalid UMS worker thread context, or specifying a worker thread that is already executing on another scheduler thread.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 (64-bit only) [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
API set	api-ms-win-core-ums-l1-1-0 (introduced in Windows 7)

See also

UmsSchedulerProc

UmsThreadYield

UpdateProcThreadAttribute

ExitProcess function (processthreadsapi.h)

11/01/2022

Ends the calling process and all its threads.

Syntax

```C++
VOID ExitProcess(
  [in] UINT uExitCode
);
```

Parameters

`[in] uExitCode`

The exit code for the process and all threads.

Return value

None

Remarks

Use the GetExitCodeProcess function to retrieve the process's exit value. Use the GetExitCodeThread function to retrieve a thread's exit value.

Exiting a process causes the following:

1. All of the threads in the process, except the calling thread, terminate their execution without receiving a DLL_THREAD_DETACH notification.
2. The states of all of the threads terminated in step 1 become signaled.
3. The entry-point functions of all loaded dynamic-link libraries (DLLs) are called with DLL_PROCESS_DETACH.
4. After all attached DLLs have executed any process termination code, the **ExitProcess** function terminates the current process, including the calling thread.
5. The state of the calling thread becomes signaled.
6. All of the object handles opened by the process are closed.

7. The termination status of the process changes from STILL_ACTIVE to the exit value of the process.
8. The state of the process object becomes signaled, satisfying any threads that had been waiting for the process to terminate.

If one of the terminated threads in the process holds a lock and the DLL detach code in one of the loaded DLLs attempts to acquire the same lock, then calling **ExitProcess** results in a deadlock. In contrast, if a process terminates by calling TerminateProcess, the DLLs that the process is attached to are not notified of the process termination. Therefore, if you do not know the state of all threads in your process, it is better to call **TerminateProcess** than **ExitProcess**. Note that returning from the **main** function of an application results in a call to **ExitProcess**.

Calling **ExitProcess** in a DLL can lead to unexpected application or system errors. Be sure to call **ExitProcess** from a DLL only if you know which applications or system components will load the DLL and that it is safe to call **ExitProcess** in this context.

Exiting a process does not cause child processes to be terminated.

Exiting a process does not necessarily remove the process object from the operating system. A process object is deleted when the last handle to the process is closed.

Examples

For an example, see Creating a Child Process with Redirected Input and Output.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	processthreadsapi.h (include Windows.h on Windows Server 2003, Windows Vista, Windows 7, Windows Server 2008 Windows Server 2008 R2)
Library	Kernel32.lib

Requirement	Value
DLL	Kernel32.dll

See also

CreateProcess

CreateRemoteThread

CreateThread

ExitThread

GetExitCodeProcess

GetExitCodeThread

OpenProcess

Process and Thread Functions

Processes

TerminateProcess

Terminating a Process

ExitThread function (processthreadsapi.h)

11/01/2022

Ends the calling thread.

Syntax

```
C++

VOID ExitThread(
  [in] DWORD dwExitCode
);
```

Parameters

`[in] dwExitCode`

The exit code for the thread.

Return value

None

Remarks

ExitThread is the preferred method of exiting a thread in C code. However, in C++ code, the thread is exited before any destructors can be called or any other automatic cleanup can be performed. Therefore, in C++ code, you should return from your thread function.

When this function is called (either explicitly or by returning from a thread procedure), the current thread's stack is deallocated, all pending I/O initiated by the thread that is not associated with a completion port is canceled, and the thread terminates. The entry-point function of all attached dynamic-link libraries (DLLs) is invoked with a value indicating that the thread is detaching from the DLL.

If the thread is the last thread in the process when this function is called, the thread's process is also terminated.

The state of the thread object becomes signaled, releasing any other threads that had been waiting for the thread to terminate. The thread's termination status changes from STILL_ACTIVE

to the value of the *dwExitCode* parameter.

Terminating a thread does not necessarily remove the thread object from the operating system. A thread object is deleted when the last handle to the thread is closed.

The ExitProcess, **ExitThread**, CreateThread, CreateRemoteThread functions, and a process that is starting (as the result of a CreateProcess call) are serialized between each other within a process. Only one of these events can happen in an address space at a time. This means the following restrictions hold:

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

A thread in an executable that is linked to the static C run-time library (CRT) should use **_beginthread** and **_endthread** for thread management rather than **CreateThread** and **ExitThread**. Failure to do so results in small memory leaks when the thread calls **ExitThread**. Another work around is to link the executable to the CRT in a DLL instead of the static CRT. Note that this memory leak only occurs from a DLL if the DLL is linked to the static CRT and a thread calls the DisableThreadLibraryCalls function. Otherwise, it is safe to call **CreateThread** and **ExitThread** from a thread in a DLL that links to the static CRT.

Use the GetExitCodeThread function to retrieve a thread's exit code.

Windows Phone 8.1: This function is supported for Windows Phone Store apps on Windows Phone 8.1 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Examples

For an example, see Using Event Objects.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	processthreadsapi.h (include Windows.h on Windows Server 2003, Windows Vista, Windows 7, Windows Server 2008 Windows Server 2008 R2)
Library	Kernel32.lib; WindowsPhoneCore.lib on Windows Phone 8.1
DLL	Kernel32.dll; KernelBase.dll on Windows Phone 8.1

See also

CreateProcess

CreateRemoteThread

CreateThread

ExitProcess

FreeLibraryAndExitThread

GetExitCodeThread

OpenThread

Process and Thread Functions

TerminateThread

Threads

PFIBER_START_ROUTINE callback function (winbase.h)

02/22/2024

An application-defined function used with the CreateFiber function. It serves as the starting address for a fiber. The **LPFIBER_START_ROUTINE** type defines a pointer to this callback function. **FiberProc** is a placeholder for the application-defined function name.

Syntax

```C++
PFIBER_START_ROUTINE PfiberStartRoutine;

VOID PfiberStartRoutine(
  LPVOID lpFiberParameter
)
{...}
```

Parameters

`lpFiberParameter`

Return value

None

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)

See also

CreateFiber

Fibers

Process and Thread Functions

FlsAlloc function (fibersapi.h)

02/22/2024

Allocates a fiber local storage (FLS) index. Any fiber in the process can subsequently use this index to store and retrieve values that are local to the fiber.

Syntax

```
C++

DWORD FlsAlloc(
  [in] PFLS_CALLBACK_FUNCTION lpCallback
);
```

Parameters

`[in] lpCallback`

A pointer to the application-defined callback function of type **PFLS_CALLBACK_FUNCTION**. This parameter is optional. For more information, see FlsCallback.

Return value

If the function succeeds, the return value is an FLS index initialized to zero.

If the function fails, the return value is FLS_OUT_OF_INDEXES. To get extended error information, call GetLastError.

Remarks

The fibers of the process can use the FLS index in subsequent calls to the FlsFree, FlsSetValue, or FlsGetValue functions.

FLS indexes are typically allocated during process or dynamic-link library (DLL) initialization. After an FLS index has been allocated, each fiber of the process can use it to access its own FLS storage slot. To store a value in its FLS slot, a fiber specifies the index in a call to FlsSetValue. The fiber specifies the same index in a subsequent call to FlsGetValue to retrieve the stored value.

FLS indexes are not valid across process boundaries. A DLL cannot assume that an index assigned in one process is valid in another process.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	fibersapi.h
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Fibers

FlsCallback

FlsFree

FlsGetValue

FlsSetValue

Process and Thread Functions

PFLS_CALLBACK_FUNCTION callback function (winnt.h)

02/22/2024

An application-defined function. If the FLS slot is in use, **FlsCallback** is called on fiber deletion, thread exit, and when an FLS index is freed. Specify this function when calling the FlsAlloc function. The *PFLS_CALLBACK_FUNCTION* type defines a pointer to this callback function. **FlsCallback** is a placeholder for the application-defined function name.

Syntax

```
C++

PFLS_CALLBACK_FUNCTION PflsCallbackFunction;

VOID PflsCallbackFunction(
  [in] PVOID lpFlsData
)
{...}
```

Parameters

`[in] lpFlsData`

The value stored in the FLS slot for the calling fiber.

Return value

None

Remarks

Each FLS index has an associated **FlsCallback** function. The callback function can be used for any purpose, but it is intended to be used primarily to free memory.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winnt.h

See also

Fibers

FlsAlloc

Process and Thread Functions

FlsFree function (fibersapi.h)

02/22/2024

Releases a fiber local storage (FLS) index, making it available for reuse.

Syntax

```C++
BOOL FlsFree(
  [in] DWORD dwFlsIndex
);
```

Parameters

`[in] dwFlsIndex`

The FLS index that was allocated by the FlsAlloc function.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

Freeing an FLS index frees the index for all instances of FLS in the current process. Freeing an FLS index also causes the associated callback routine to be called for each fiber, if the corresponding FLS slot contains a non-NULL value.

If the fibers of the process have allocated memory and stored a pointer to the memory in an FLS slot, they should free the memory before calling **FlsFree**. The **FlsFree** function does not free memory blocks whose addresses have been stored in the FLS slots associated with the FLS index. It is expected that DLLs call this function (if at all) only during DLL_PROCESS_DETACH.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	fibersapi.h
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Fibers

FlsAlloc

Process and Thread Functions

FlsGetValue function (fibersapi.h)

02/22/2024

Retrieves the value in the calling fiber's fiber local storage (FLS) slot for the specified FLS index. Each fiber has its own slot for each FLS index.

Syntax

```
C++

PVOID FlsGetValue(
  [in] DWORD dwFlsIndex
);
```

Parameters

`[in] dwFlsIndex`

The FLS index that was allocated by the FlsAlloc function.

Return value

If the function succeeds, the return value is the value stored in the calling fiber's FLS slot associated with the specified index.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Remarks

FLS indexes are typically allocated by the FlsAlloc function during process or DLL initialization. After an FLS index is allocated, each fiber of the process can use it to access its own FLS slot for that index. A fiber specifies an FLS index in a call to FlsSetValue to store a value in its slot. The thread specifies the same index in a subsequent call to **FlsSetValue** to retrieve the stored value.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	fibersapi.h
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Fibers

FlsAlloc

FlsSetValue

Process and Thread Functions

FlsSetValue function (fibersapi.h)

02/22/2024

Stores a value in the calling fiber's fiber local storage (FLS) slot for the specified FLS index. Each fiber has its own slot for each FLS index.

Syntax

```
C++

BOOL FlsSetValue(
  [in]           DWORD dwFlsIndex,
  [in, optional] PVOID lpFlsData
);
```

Parameters

`[in] dwFlsIndex`

The FLS index that was allocated by the FlsAlloc function.

`[in, optional] lpFlsData`

The value to be stored in the FLS slot for the calling fiber.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError. The following errors can be returned.

Expand table

Return code	Description
ERROR_INVALID_PARAMETER	The index is not in range.
ERROR_NO_MEMORY	The FLS array has not been allocated.

Remarks

FLS indexes are typically allocated by the FlsAlloc function during process or DLL initialization. After an FLS index is allocated, each fiber of the process can use it to access its own FLS slot for that index. A thread specifies an FLS index in a call to **FlsSetValue** to store a value in its slot. The thread specifies the same index in a subsequent call to FlsGetValue to retrieve the stored value.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	fibersapi.h
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Fibers

FlsAlloc

FlsGetValue

Process and Thread Functions

FlushProcessWriteBuffers function (processthreadsapi.h)

11/01/2022

Flushes the write queue of each processor that is running a thread of the current process.

Syntax

```C++
VOID FlushProcessWriteBuffers();
```

Return value

None

Remarks

The function generates an interprocessor interrupt (IPI) to all processors that are part of the current process affinity. It guarantees the visibility of write operations performed on one processor to the other processors.

Requirements

Expand table

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

Requirement	Value
DLL	Kernel32.dll

See also

Process and Thread Functions

FreeEnvironmentStringsA function (processenv.h)

Article11/20/2024

Frees a block of environment strings.

Syntax

```
C++

BOOL FreeEnvironmentStringsA(
  LPCH penv
);
```

Parameters

`penv`

A pointer to a block of environment strings. The pointer to the block must be obtained by calling the GetEnvironmentStrings function.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero To get extended error information, call GetLastError.

Remarks

If you used the ANSI version of GetEnvironmentStrings function, be sure to use the ANSI version of **FreeEnvironmentStrings**. Similarly, if you used the Unicode version of **GetEnvironmentStrings**, be sure to use the Unicode version of **FreeEnvironmentStrings**.

Examples

For an example, see Changing Environment Variables.

> ⓘ **Note**

The processenv.h header defines FreeEnvironmentStrings as an alias that automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Expand table

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

See also

Environment Variables

FreeLibraryWhenCallbackReturns function (threadpoolapiset.h)

02/22/2024

Specifies the DLL that the thread pool will unload when the current callback completes.

Syntax

```C++
VOID FreeLibraryWhenCallbackReturns(
  [in, out] PTP_CALLBACK_INSTANCE pci,
  [in]      HMODULE               mod
);
```

Parameters

`[in, out] pci`

A pointer to a **TP_CALLBACK_INSTANCE** structure that defines the callback instance. The pointer is passed to the callback function.

`[in] mod`

A handle to the DLL.

Return value

None

Remarks

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

Requirements

Expand table

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

See also

CallbackMayRunLong

DisassociateCurrentThreadFromCallback

LeaveCriticalSectionWhenCallbackReturns

ReleaseMutexWhenCallbackReturns

ReleaseSemaphoreWhenCallbackReturns

SetEventWhenCallbackReturns

Thread Pools

TrySubmitThreadpoolCallback

FreeMemoryJobObject function (jobapi2.h)

02/22/2024

Frees memory that a function related to job objects allocated. Functions related to job objects that allocate memory include QueryIoRateControlInformationJobObject.

Syntax

```
C++

VOID FreeMemoryJobObject(
  [in] _Frees_ptr_ VOID *Buffer
);
```

Parameters

`[in] Buffer`

A pointer to the buffer of allocated memory that you want to free.

Return value

None

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 10 [desktop apps only]
Minimum supported server	Windows Server 2016 [desktop apps only]
Target Platform	Windows
Header	jobapi2.h
Library	Kernel32.lib
DLL	Kernel32.dll

See also

QueryIoRateControlInformationJobObject

GetActiveProcessorCount function (winbase.h)

Article02/22/2024

Returns the number of active processors in a processor group or in the system.

Syntax

```
C++

DWORD GetActiveProcessorCount(
  [in] WORD GroupNumber
);
```

Parameters

`[in] GroupNumber`

The processor group number. If this parameter is ALL_PROCESSOR_GROUPS, the function returns the number of active processors in the system.

Return value

If the function succeeds, the return value is the number of active processors in the specified group.

If the function fails, the return value is zero. To get extended error information, use GetLastError.

Remarks

To compile an application that uses this function, set _WIN32_WINNT >= 0x0601. For more information, see Using the Windows Headers.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

GetActiveProcessorGroupCount function (winbase.h)

Article02/22/2024

Returns the number of active processor groups in the system.

Syntax

```
C++

WORD GetActiveProcessorGroupCount();
```

Return value

If the function succeeds, the return value is the number of active processor groups in the system.

If the function fails, the return value is zero.

Remarks

To compile an application that uses this function, set _WIN32_WINNT >= 0x0601. For more information, see Using the Windows Headers.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

GetAutoRotationState function (winuser.h)

02/22/2024

Retrieves an AR_STATE value containing the state of screen auto-rotation for the system, for example whether auto-rotation is supported, and whether it is enabled by the user. **GetAutoRotationState** provides a robust and diverse way of querying for auto-rotation state, and more. For example, if you want your app to behave differently when multiple monitors are attached then you can determine that from the **AR_STATE** returned.

Syntax

```C++
BOOL GetAutoRotationState(
  [out] PAR_STATE pState
);
```

Parameters

`[out] pState`

Pointer to a location in memory that will receive the current state of auto-rotation for the system.

Return value

TRUE if the method succeeds, otherwise FALSE.

See GetDisplayAutoRotationPreferences for an example of using this function.

Requirements

Expand table

Requirement	Value
Target Platform	Windows
Header	winuser.h

Requirement	Value
DLL	Kernel.dll

GetDisplayAutoRotationPreferencesByProcessId function (winuser.h)

Article02/22/2024

Retrieves the screen auto-rotation preferences for the process indicated by the *dwProcessId* parameter.

Syntax

```
C++

BOOL GetDisplayAutoRotationPreferencesByProcessId(
  [in]  DWORD                  dwProcessId,
  [out] ORIENTATION_PREFERENCE *pOrientation,
  [out] BOOL                   *fRotateScreen
);
```

Parameters

`[in] dwProcessId`

The process to get preference settings for.

`[out] pOrientation`

Pointer to a location in memory that will receive the current orientation preference setting for the indicated process.

`[out] fRotateScreen`

Pointer to a location in memory that will receive a TRUE or FALSE value indicating whether the screen was rotated to comply with the process orientation preferences.

Return value

TRUE if the method succeeds, otherwise FALSE.

Applications should use GetDisplayAutoRotationPreferences to retrieve their auto-rotation preferences instead of using this API. For an example, see GetDisplayAutoRotationPreferences.

Requirements

Expand table

Requirement	Value
Target Platform	Windows
Header	winuser.h
DLL	Kernel.dll

GetDisplayAutoRotationPreferences function (winuser.h)

02/22/2024

Retrieves the screen auto-rotation preferences for the current process.

Syntax

```C++
BOOL GetDisplayAutoRotationPreferences(
  [out] ORIENTATION_PREFERENCE *pOrientation
);
```

Parameters

`[out] pOrientation`

Pointer to a location in memory that will receive the current orientation preference setting for the calling process.

Return value

TRUE if the method succeeds, otherwise FALSE.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8 [desktop apps | UWP apps]
Minimum supported server	Windows Server 2012 [desktop apps | UWP apps]
Target Platform	Windows
Header	winuser.h
DLL	Kernel.dll

GetCommandLineA function (processenv.h)

Article10/23/2024

Retrieves the command-line string for the current process.

Syntax

```
C++

LPSTR GetCommandLineA();
```

Return value

The return value is a pointer to the command-line string for the current process.

Remarks

The lifetime of the returned value is managed by the system, applications should not free or modify this value.

Console processes can use the *argc* and *argv* arguments of the **main** or **wmain** functions by implementing those as the program entry point.

GUI processes can use the *lpCmdLine* argument of the WinMain or wWinMain functions by implementing those as the program entry point.

To convert the command line to an *argv* style array of strings, pass the result from GetCommandLineA to CommandLineToArgvW.

> **Note**
>
> The name of the executable in the command line that the operating system provides to a process is not necessarily identical to that in the command line that the calling process gives to the CreateProcessA function. The operating system may prepend a fully qualified path to an executable name that is provided without a fully qualified path.

Security Remarks

The command line returned by GetCommandLineA is a conversion of the Unicode command line to the 8-bit process code page.

For most code pages this conversion is lossy and the converted command line can differ from the Unicode command line, creating possible security issues like the following:

- The conversion may alter strings intended for use as file names. For example, if the ANSI code page is Windows-1252, the Unicode character U+0100 (Latin capital letter A with macron: Ā) converts to 0x41 (the Latin capital letter A). If a user passes a file name containing the character Ā, a program that uses GetCommandLineA will receive it with the character A and operate on the wrong file.
- The conversion may alter how the command line is parsed. For example, if the ANSI code page is Windows-1252, the Unicode character U+FF02 (Fullwidth quotation mark: ＂) converts to 0x22 (the ASCII quotation mark) and the Unicode character U+2010 (Hyphen: ‐) converts to 0x2D (the ASCII minus sign). Both of these can result in command line file arguments being misinterpreted as command line options.

To avoid this problem, use the GetCommandLineW function to receive the Unicode command line, or use an application manifest (on Windows Version 1903 or later) to set UTF-8 as the process code page.

> **Note**
>
> The processenv.h header defines GetCommandLine as an alias that automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Expand table

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

Requirement	Value
Library	Kernel32.lib
DLL	Kernel32.dll

See also

CreateProcessA function

Process and Thread Functions

GetCurrentProcess function (processthreadsapi.h)

02/22/2024

Retrieves a pseudo handle for the current process.

Syntax

```
C++

HANDLE GetCurrentProcess();
```

Return value

The return value is a pseudo handle to the current process.

Remarks

A pseudo handle is a special constant, currently (**HANDLE**)-1, that is interpreted as the current process handle. For compatibility with future operating systems, it is best to call **GetCurrentProcess** instead of hard-coding this constant value. The calling process can use a pseudo handle to specify its own process whenever a process handle is required. Pseudo handles are not inherited by child processes.

This handle has the **PROCESS_ALL_ACCESS** access right to the process object. For more information, see Process Security and Access Rights.

Windows Server 2003 and Windows XP: This handle has the maximum access allowed by the security descriptor of the process to the primary token of the process.

A process can create a "real" handle to itself that is valid in the context of other processes, or that can be inherited by other processes, by specifying the pseudo handle as the source handle in a call to the DuplicateHandle function. A process can also use the OpenProcess function to open a real handle to itself.

The pseudo handle need not be closed when it is no longer needed. Calling the CloseHandle function with a pseudo handle has no effect. If the pseudo handle is duplicated by DuplicateHandle, the duplicate handle must be closed.

Examples

For an example, see Creating a Child Process with Redirected Input and Output.

Requirements

Expand table

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

See also

CloseHandle

DuplicateHandle

GetCurrentProcessId

GetCurrentThread

OpenProcess

Process and Thread Functions

Processes

Vertdll APIs available in VBS enclaves

GetCurrentProcessId function (processthreadsapi.h)

02/22/2024

Retrieves the process identifier of the calling process.

Syntax

```
C++

DWORD GetCurrentProcessId();
```

Return value

The return value is the process identifier of the calling process.

Remarks

Until the process terminates, the process identifier uniquely identifies the process throughout the system.

Requirements

Expand table

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

See also

GetCurrentProcess

OpenProcess

Process and Thread Functions

Processes

GetCurrentProcessorNumber function (processthreadsapi.h)

02/22/2024

Retrieves the number of the processor the current thread was running on during the call to this function.

Syntax

```
C++

DWORD GetCurrentProcessorNumber();
```

Return value

The function returns the current processor number.

Remarks

This function is used to provide information for estimating process performance.

On systems with more than 64 logical processors, the **GetCurrentProcessorNumber** function returns the processor number within the processor group to which the logical processor is assigned. Use the GetCurrentProcessorNumberEx function to retrieve the processor group and number of the current processor.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows

Requirement	Value
Header	processthreadsapi.h (include Windows.h on Windows Vista, Windows 7, Windows Server 2008 Windows Server 2008 R2)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Multiple Processors

Process and Thread Functions

Processes

GetCurrentProcessorNumberEx function (processthreadsapi.h)

02/22/2024

Retrieves the processor group and number of the logical processor in which the calling thread is running.

Syntax

```C++
VOID GetCurrentProcessorNumberEx(
  [out] PPROCESSOR_NUMBER ProcNumber
);
```

Parameters

`[out] ProcNumber`

A pointer to a PROCESSOR_NUMBER structure that receives the processor group to which the logical processor is assigned and the number of the logical processor within its group.

Return value

None

Remarks

To compile an application that uses this function, set _WIN32_WINNT >= 0x0601. For more information, see Using the Windows Headers.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 [desktop apps | UWP apps]

Requirement	Value
Minimum supported server	Windows Server 2008 R2 [desktop apps | UWP apps]
Target Platform	Windows
Header	processthreadsapi.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

GetCurrentThread function (processthreadsapi.h)

02/06/2024

Retrieves a pseudo handle for the calling thread.

Syntax

```
C++

HANDLE GetCurrentThread();
```

Return value

The return value is a pseudo handle for the current thread.

Remarks

A pseudo handle is a special constant that is interpreted as the current thread handle. The calling thread can use this handle to specify itself whenever a thread handle is required. Pseudo handles are not inherited by child processes.

This handle has the **THREAD_ALL_ACCESS** access right to the thread object. For more information, see Thread Security and Access Rights.

Windows Server 2003 and Windows XP: This handle has the maximum access allowed by the security descriptor of the thread to the primary token of the process.

The function cannot be used by one thread to create a handle that can be used by other threads to refer to the first thread. The handle is always interpreted as referring to the thread that is using it. A thread can create a "real" handle to itself that can be used by other threads, or inherited by other processes, by specifying the pseudo handle as the source handle in a call to the DuplicateHandle function.

The pseudo handle need not be closed when it is no longer needed. Calling the CloseHandle function with this handle has no effect. If the pseudo handle is duplicated by DuplicateHandle, the duplicate handle must be closed.

Do not create a thread while impersonating a security context. The call will succeed, however the newly created thread will have reduced access rights to itself when calling

GetCurrentThread. The access rights granted this thread will be derived from the access rights the impersonated user has to the process. Some access rights including **THREAD_SET_THREAD_TOKEN** and **THREAD_GET_CONTEXT** may not be present, leading to unexpected failures.

Examples

For an example, see Checking Client Access.

Requirements

Expand table

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

See also

CloseHandle

DuplicateHandle

GetCurrentProcess

GetCurrentThreadId

OpenThread

Process and Thread Functions

Threads

Vertdll APIs available in VBS enclaves

GetCurrentThreadId function (processthreadsapi.h)

02/22/2024

Retrieves the thread identifier of the calling thread.

Syntax

```
C++

DWORD GetCurrentThreadId();
```

Return value

The return value is the thread identifier of the calling thread.

Remarks

Until the thread terminates, the thread identifier uniquely identifies the thread throughout the system.

Examples

For an example, see Using Thread Local Storage.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows

Requirement	Value
Header	processthreadsapi.h (include Windows.h on Windows Server 2003, Windows Vista, Windows 7, Windows Server 2008 Windows Server 2008 R2)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

GetCurrentThread

OpenThread

Process and Thread Functions

Threads

Vertdll APIs available in VBS enclaves

GetCurrentThreadStackLimits function (processthreadsapi.h)

02/22/2024

Retrieves the boundaries of the stack that was allocated by the system for the current thread.

Syntax

```C++
VOID GetCurrentThreadStackLimits(
  [out] PULONG_PTR LowLimit,
  [out] PULONG_PTR HighLimit
);
```

Parameters

`[out] LowLimit`

A pointer variable that receives the lower boundary of the current thread stack.

`[out] HighLimit`

A pointer variable that receives the upper boundary of the current thread stack.

Return value

None

Remarks

It is possible for user-mode code to execute in stack memory that is outside the region allocated by the system when the thread was created. Callers can use the **GetCurrentThreadStackLimits** function to verify that the current stack pointer is within the returned limits.

To compile an application that uses this function, set _WIN32_WINNT >= 0x0602. For more information, see Using the Windows Headers.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8 [desktop apps | UWP apps]
Minimum supported server	Windows Server 2012 [desktop apps | UWP apps]
Target Platform	Windows
Header	processthreadsapi.h
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Thread Stack Size

GetCurrentUmsThread function (winbase.h)

Article02/22/2024

Returns the user-mode scheduling (UMS) thread context of the calling UMS thread.

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

Syntax

```
C++

PUMS_CONTEXT GetCurrentUmsThread();
```

Return value

The function returns a pointer to the UMS thread context of the calling thread.

If calling thread is not a UMS thread, the function returns NULL. To get extended error information, call GetLastError.

Remarks

The **GetCurrentUmsThread** function can be called for a UMS scheduler thread or UMS worker thread.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 (64-bit only) [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows

Requirement	Value
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
API set	api-ms-win-core-ums-l1-1-0 (introduced in Windows 7)

IEAPProviderConfig::Initialize method (rrascfg.h)

Article02/22/2024

The system calls the **Initialize** method to initialize an EAP configuration session with the specified computer.

Syntax

```
C++

HRESULT Initialize(
  LPCOLESTR pszMachineName,
  DWORD     dwEapTypeId,
  ULONG_PTR *puConnectionParam
);
```

Parameters

`pszMachineName`

Pointer to a null-terminated string that contains the name of the computer on which to configure EAP. String length is not limited.

`dwEapTypeId`

Specifies the EAP for which to initialize a configuration session.

`puConnectionParam`

Pointer to an unsigned integer variable. On successful return, the value of this variable identifies this configuration session.

Return value

If the function succeeds, the return value should be **S_OK**.

If the function fails, the return value should be one of the following codes.

Expand table

Return code	Description
E_FAIL	Non-specific error.
E_INVALIDARG	One of the arguments is invalid.
E_OUTOFMEMORY	The method failed because it was unable to allocate required memory.
E_UNEXPECTED	An unexpected error occurred.

Remarks

The configuration UI should allow the user to configure the EAP provider on a remote computer. Establish the connection to the remote computer during the call to **Initialize**.

The DLL that implements IEAPProviderConfig can support more than one authentication protocol. The *dwEapTypeId* parameter specifies the authentication protocol to initialize a configuration session for.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps only]
Minimum supported server	Windows 2000 Server [desktop apps only]
Target Platform	Windows
Header	rrascfg.h

See also

EAP Interfaces

Extensible Authentication Protocol Reference

IEAPProviderConfig

IEAPProviderConfig::RouterInvokeConfigUI

IEAPProviderConfig::RouterInvokeCredentialsUI

IEAPProviderConfig::ServerInvokeConfigUI

IEAPProviderConfig::Uninitialize

GetEnvironmentVariable function (winbase.h)

Article02/22/2024

Retrieves the contents of the specified variable from the environment block of the calling process.

Syntax

```
C++

DWORD GetEnvironmentVariable(
  [in, optional]  LPCTSTR lpName,
  [out, optional] LPTSTR  lpBuffer,
  [in]            DWORD   nSize
);
```

Parameters

`[in, optional] lpName`

The name of the environment variable.

`[out, optional] lpBuffer`

A pointer to a buffer that receives the contents of the specified environment variable as a null-terminated string. An environment variable has a maximum size limit of 32,767 characters, including the null-terminating character.

`[in] nSize`

The size of the buffer pointed to by the *lpBuffer* parameter, including the null-terminating character, in characters.

Return value

If the function succeeds, the return value is the number of characters stored in the buffer pointed to by *lpBuffer*, not including the terminating null character.

If *lpBuffer* is not large enough to hold the data, the return value is the buffer size, in characters, required to hold the string and its terminating null character and the contents of *lpBuffer* are

undefined.

If the function fails, the return value is zero. If the specified environment variable was not found in the environment block, GetLastError returns ERROR_ENVVAR_NOT_FOUND.

Remarks

This function can retrieve either a system environment variable or a user environment variable.

Examples

For an example, see Changing Environment Variables.

Requirements

Expand table

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

See also

Environment Variables

GetEnvironmentStrings

SetEnvironmentVariable

GetExitCodeProcess function (processthreadsapi.h)

11/01/2022

Retrieves the termination status of the specified process.

Syntax

```
C++

BOOL GetExitCodeProcess(
  [in]  HANDLE  hProcess,
  [out] LPDWORD lpExitCode
);
```

Parameters

`[in] hProcess`

A handle to the process.

The handle must have the **PROCESS_QUERY_INFORMATION** or **PROCESS_QUERY_LIMITED_INFORMATION** access right. For more information, see Process Security and Access Rights.

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

`[out] lpExitCode`

A pointer to a variable to receive the process termination status. For more information, see Remarks.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

This function returns immediately. If the process has not terminated and the function succeeds, the status returned is **STILL_ACTIVE** (a macro for **STATUS_PENDING** (minwinbase.h)). If the process has terminated and the function succeeds, the status returned is one of the following values:

- The exit value specified in the ExitProcess or TerminateProcess function.
- The return value from the main or WinMain function of the process.
- The exception value for an unhandled exception that caused the process to terminate.

> **Important**
>
> The **GetExitCodeProcess** function returns a valid error code defined by the application only after the thread terminates. Therefore, an application should not use **STILL_ACTIVE** (259) as an error code (**STILL_ACTIVE** is a macro for **STATUS_PENDING** (minwinbase.h)). If a thread returns **STILL_ACTIVE** (259) as an error code, then applications that test for that value could interpret it to mean that the thread is still running, and continue to test for the completion of the thread after the thread has terminated, which could put the application into an infinite loop.

Requirements

Expand table

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

See also

ExitProcess

ExitThread

Process and Thread Functions

Processes

TerminateProcess

Terminating a Process

WinMain

GetExitCodeThread function (processthreadsapi.h)

02/22/2024

Retrieves the termination status of the specified thread.

Syntax

```
C++

BOOL GetExitCodeThread(
  [in]  HANDLE  hThread,
  [out] LPDWORD lpExitCode
);
```

Parameters

`[in] hThread`

A handle to the thread.

The handle must have the **THREAD_QUERY_INFORMATION** or **THREAD_QUERY_LIMITED_INFORMATION** access right. For more information, see Thread Security and Access Rights.

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

`[out] lpExitCode`

A pointer to a variable to receive the thread termination status. For more information, see Remarks.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

This function returns immediately. If the specified thread has not terminated and the function succeeds, the status returned is **STILL_ACTIVE**. If the thread has terminated and the function succeeds, the status returned is one of the following values:

- The exit value specified in the ExitThread or TerminateThread function.
- The return value from the thread function.
- The exit value of the thread's process.

> **Important** The **GetExitCodeThread** function returns a valid error code defined by the application only after the thread terminates. Therefore, an application should not use **STILL_ACTIVE** (259) as an error code. If a thread returns **STILL_ACTIVE** (259) as an error code, applications that test for this value could interpret it to mean that the thread is still running and continue to test for the completion of the thread after the thread has terminated, which could put the application into an infinite loop. To avoid this problem, callers should call the **GetExitCodeThread** function only after the thread has been confirmed to have exited. Use the **WaitForSingleObject** function with a wait duration of zero to determine whether a thread has exited.

Windows Phone 8.1: This function is supported for Windows Phone Store apps on Windows Phone 8.1 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Expand table

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

Requirement	Value
Library	Kernel32.lib; WindowsPhoneCore.lib on Windows Phone 8.1
DLL	Kernel32.dll; KernelBase.dll on Windows Phone 8.1

See also

ExitThread

GetExitCodeProcess

OpenThread

Process and Thread Functions

TerminateThread

Terminating a Thread

GetGuiResources function (winuser.h)

03/07/2023

Retrieves the count of handles to graphical user interface (GUI) objects in use by the specified process.

Syntax

```
C++

DWORD GetGuiResources(
  [in] HANDLE hProcess,
  [in] DWORD  uiFlags
);
```

Parameters

`[in] hProcess`

A handle to the process. The handle must refer to a process in the current session, and must have the **PROCESS_QUERY_LIMITED_INFORMATION** access right (see Process security and access rights).

If this parameter is the special value **GR_GLOBAL**, then the resource usage is reported across all processes in the current session.

Windows Server 2008, Windows Vista, Windows Server 2003, and Windows XP: The **GR_GLOBAL** value is not supported until Windows 7 and Windows Server 2008 R2.

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

`[in] uiFlags`

The GUI object type. This parameter can be one of the following values.

Expand table

Value	Meaning
GR_GDIOBJECTS 0	Return the count of GDI objects.

GR_GDIOBJECTS_PEAK 2	Return the peak count of GDI objects. **Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP:** This value is not supported until Windows 7 and Windows Server 2008 R2.
GR_USEROBJECTS 1	Return the count of USER objects.
GR_USEROBJECTS_PEAK 4	Return the peak count of USER objects. **Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP:** This value is not supported until Windows 7 and Windows Server 2008 R2.

Return value

If the function succeeds, the return value is the count of handles to GUI objects in use by the process. If no GUI objects are in use, the return value is zero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

A process without a graphical user interface does not use GUI resources, therefore, **GetGuiResources** will return zero.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	winuser.h (include windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-misc-l1-1-0 (introduced in Windows 8)

See also

CreateProcess

GetCurrentProcess

OpenProcess

Process and Thread Functions

GetLogicalProcessorInformation function (sysinfoapi.h)

10/13/2021

Retrieves information about logical processors and related hardware.

To retrieve information about logical processors and related hardware, including processor groups, use the GetLogicalProcessorInformationEx function.

Syntax

```
C++

BOOL GetLogicalProcessorInformation(
  [out]     PSYSTEM_LOGICAL_PROCESSOR_INFORMATION Buffer,
  [in, out] PDWORD                                ReturnedLength
);
```

Parameters

`[out] Buffer`

A pointer to a buffer that receives an array of SYSTEM_LOGICAL_PROCESSOR_INFORMATION structures. If the function fails, the contents of this buffer are undefined.

`[in, out] ReturnedLength`

On input, specifies the length of the buffer pointed to by *Buffer*, in bytes. If the buffer is large enough to contain all of the data, this function succeeds and *ReturnLength* is set to the number of bytes returned. If the buffer is not large enough to contain all of the data, the function fails, GetLastError returns ERROR_INSUFFICIENT_BUFFER, and *ReturnLength* is set to the buffer length required to contain all of the data. If the function fails with an error other than ERROR_INSUFFICIENT_BUFFER, the value of *ReturnLength* is undefined.

Return value

If the function succeeds, the return value is TRUE and at least one SYSTEM_LOGICAL_PROCESSOR_INFORMATION structure is written to the output buffer.

If the function fails, the return value is FALSE. To get extended error information, call GetLastError.

Remarks

GetLogicalProcessorInformation can be used to get information about the relationship between logical processors in the system, including:

- The logical processors that are part of a NUMA node.
- The logical processors that share resources. An example of this type of resource sharing would be hyperthreading scenarios.

Your application can use this information when affinitizing your threads and processes to take best advantage of the hardware properties of the platform, or to determine the number of logical and physical processors for licensing purposes.

Each of the SYSTEM_LOGICAL_PROCESSOR_INFORMATION structures returned in the buffer contains the following:

- A logical processor affinity mask, which indicates the logical processors that the information in the structure applies to.
- A logical processor mask of type LOGICAL_PROCESSOR_RELATIONSHIP, which indicates the relationship between the logical processors in the mask. Applications calling this function must be prepared to handle additional indicator values in the future.

Note that the order in which the structures are returned in the buffer may change between calls to this function.

The size of the SYSTEM_LOGICAL_PROCESSOR_INFORMATION structure varies between processor architectures and versions of Windows. For this reason, applications should first call this function to obtain the required buffer size, then dynamically allocate memory for the buffer.

On systems with more than 64 logical processors, the **GetLogicalProcessorInformation** function retrieves logical processor information about processors in the processor group to which the calling thread is currently assigned. Use the GetLogicalProcessorInformationEx function to retrieve information about processors in all processor groups on the system.

> **Note**
>
> Starting with *TBD Release Iron*, the behavior of this and other NUMA functions has been modified to better support systems with nodes containing more that 64 processors. For

more information about this change, including information about enabling the old behavior of this API, see NUMA Support.

Behavior starting with TBD Release Iron

The relationship structures for RelationNumaNode contain the affinity mask for the node's affinity within the calling thread's group.

Examples

The following C++ example uses the **GetLogicalProcessorInformation** function to display information about processors on the current system. Because **GetLogicalProcessorInformation** is not present on all systems, this example uses the GetProcAddress function instead of calling **GetLogicalProcessorInformation** directly.

This example reports the number of active processor cores. This example also reports the number of NUMA nodes, physical packages, and caches on systems that support this information. For more information, see the description of the **Relationship** member of the SYSTEM_LOGICAL_PROCESSOR_INFORMATION structure. **Windows Server 2003, Windows XP Professional x64 Edition and Windows XP with SP3:** This example reports the number of physical processors rather than the number of active processor cores.

C++

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

typedef BOOL (WINAPI *LPFN_GLPI)(
    PSYSTEM_LOGICAL_PROCESSOR_INFORMATION,
    PDWORD);


// Helper function to count set bits in the processor mask.
DWORD CountSetBits(ULONG_PTR bitMask)
{
    DWORD LSHIFT = sizeof(ULONG_PTR)*8 - 1;
    DWORD bitSetCount = 0;
    ULONG_PTR bitTest = (ULONG_PTR)1 << LSHIFT;
    DWORD i;

    for (i = 0; i <= LSHIFT; ++i)
    {
        bitSetCount += ((bitMask & bitTest)?1:0);
```

```
        bitTest/=2;
    }

    return bitSetCount;
}

int _cdecl _tmain ()
{
    LPFN_GLPI glpi;
    BOOL done = FALSE;
    PSYSTEM_LOGICAL_PROCESSOR_INFORMATION buffer = NULL;
    PSYSTEM_LOGICAL_PROCESSOR_INFORMATION ptr = NULL;
    DWORD returnLength = 0;
    DWORD logicalProcessorCount = 0;
    DWORD numaNodeCount = 0;
    DWORD processorCoreCount = 0;
    DWORD processorL1CacheCount = 0;
    DWORD processorL2CacheCount = 0;
    DWORD processorL3CacheCount = 0;
    DWORD processorPackageCount = 0;
    DWORD byteOffset = 0;
    PCACHE_DESCRIPTOR Cache;

    glpi = (LPFN_GLPI) GetProcAddress(
                            GetModuleHandle(TEXT("kernel32")),
                            "GetLogicalProcessorInformation");
    if (NULL == glpi)
    {
        _tprintf(TEXT("\nGetLogicalProcessorInformation is not supported.\n"));
        return (1);
    }

    while (!done)
    {
        DWORD rc = glpi(buffer, &returnLength);

        if (FALSE == rc)
        {
            if (GetLastError() == ERROR_INSUFFICIENT_BUFFER)
            {
                if (buffer)
                    free(buffer);

                buffer = (PSYSTEM_LOGICAL_PROCESSOR_INFORMATION)malloc(
                        returnLength);

                if (NULL == buffer)
                {
                    _tprintf(TEXT("\nError: Allocation failure\n"));
                    return (2);
                }
            }
            else
            {
                _tprintf(TEXT("\nError %d\n"), GetLastError());
```

```
                return (3);
            }
        }
        else
        {
            done = TRUE;
        }
    }

    ptr = buffer;

    while (byteOffset + sizeof(SYSTEM_LOGICAL_PROCESSOR_INFORMATION) <=
returnLength)
    {
        switch (ptr->Relationship)
        {
        case RelationNumaNode:
            // Non-NUMA systems report a single record of this type.
            numaNodeCount++;
            break;

        case RelationProcessorCore:
            processorCoreCount++;

            // A hyperthreaded core supplies more than one logical processor.
            logicalProcessorCount += CountSetBits(ptr->ProcessorMask);
            break;

        case RelationCache:
            // Cache data is in ptr->Cache, one CACHE_DESCRIPTOR structure for
each cache.
            Cache = &ptr->Cache;
            if (Cache->Level == 1)
            {
                processorL1CacheCount++;
            }
            else if (Cache->Level == 2)
            {
                processorL2CacheCount++;
            }
            else if (Cache->Level == 3)
            {
                processorL3CacheCount++;
            }
            break;

        case RelationProcessorPackage:
            // Logical processors share a physical package.
            processorPackageCount++;
            break;

        default:
            _tprintf(TEXT("\nError: Unsupported LOGICAL_PROCESSOR_RELATIONSHIP
value.\n"));
            break;
```

```
        }
        byteOffset += sizeof(SYSTEM_LOGICAL_PROCESSOR_INFORMATION);
        ptr++;
    }

    _tprintf(TEXT("\nGetLogicalProcessorInformation results:\n"));
    _tprintf(TEXT("Number of NUMA nodes: %d\n"),
             numaNodeCount);
    _tprintf(TEXT("Number of physical processor packages: %d\n"),
             processorPackageCount);
    _tprintf(TEXT("Number of processor cores: %d\n"),
             processorCoreCount);
    _tprintf(TEXT("Number of logical processors: %d\n"),
             logicalProcessorCount);
    _tprintf(TEXT("Number of processor L1/L2/L3 caches: %d/%d/%d\n"),
             processorL1CacheCount,
             processorL2CacheCount,
             processorL3CacheCount);

    free(buffer);

    return 0;
}
```

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista, Windows XP Professional x64 Edition, Windows XP with SP3 [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	sysinfoapi.h
Library	Kernel32.lib
DLL	Kernel32.dll

See also

GetLogicalProcessorInformationEx

LOGICAL_PROCESSOR_RELATIONSHIP

Process and Thread Functions

SYSTEM_LOGICAL_PROCESSOR_INFORMATION

GetLogicalProcessorInformationEx function (sysinfoapi.h)

03/06/2023

Retrieves information about the relationships of logical processors and related hardware.

Syntax

```C++
BOOL GetLogicalProcessorInformationEx(
  [in]            LOGICAL_PROCESSOR_RELATIONSHIP           RelationshipType,
  [out, optional] PSYSTEM_LOGICAL_PROCESSOR_INFORMATION_EX Buffer,
  [in, out]       PDWORD                                   ReturnedLength
);
```

Parameters

`[in] RelationshipType`

The type of relationship to retrieve. This parameter can be one of the following LOGICAL_PROCESSOR_RELATIONSHIP values.

Expand table

Value	Meaning
RelationProcessorCore 0	Retrieves information about logical processors that share a single processor core.
RelationNumaNode 1	Retrieves information about logical processors that are part of the same NUMA node.
RelationCache 2	Retrieves information about logical processors that share a cache.
RelationProcessorPackage 3	Retrieves information about logical processors that share a physical package.
RelationGroup 4	Retrieves information about logical processors that share a processor group.
RelationProcessorDie 5	Retrieves information about logical processors that share a processor die.

RelationNumaNodeEx 6	Retrieves information about logical processors that are part of the same NUMA node (with full affinity).
RelationProcessorModule 7	Retrieves information about logical processors that share a processor module.
RelationAll 0xffff	Retrieves information about logical processors for all relationship types (cache, NUMA node, processor core, physical package, processor group, processor die, and processor module).

`[out, optional] Buffer`

A pointer to a buffer that receives a sequence of variable-sized SYSTEM_LOGICAL_PROCESSOR_INFORMATION_EX structures. If the function fails, the contents of this buffer are undefined.

`[in, out] ReturnedLength`

On input, specifies the length of the buffer pointed to by *Buffer*, in bytes. If the buffer is large enough to contain all of the data, this function succeeds and *ReturnedLength* is set to the number of bytes returned. If the buffer is not large enough to contain all of the data, the function fails, GetLastError returns ERROR_INSUFFICIENT_BUFFER, and *ReturnedLength* is set to the buffer length required to contain all of the data. If the function fails with an error other than ERROR_INSUFFICIENT_BUFFER, the value of *ReturnedLength* is undefined.

Return value

If the function succeeds, the return value is TRUE and at least one SYSTEM_LOGICAL_PROCESSOR_INFORMATION_EX structure is written to the output buffer.

If the function fails, the return value is FALSE. To get extended error information, call GetLastError.

Remarks

If a 32-bit process running under WOW64 calls this function on a system with more than 64 processors, some of the processor affinity masks returned by the function may be incorrect. This is because the high-order **DWORD** of the 64-bit KAFFINITY structure that represents all 64 processors is "folded" into a 32-bit **KAFFINITY** structure in the caller's buffer. As a result, the affinity masks for processors 32 through 63 are incorrectly represented as duplicates of the masks for processors 0 through 31. In addition, the affinity masks for processors 32 through 63 are incorrectly represented as duplicates of the masks for processors 0 through 31. In addition,

the sum of all per-group **ActiveProcessorCount** and **MaximumProcessorCount** values reported in PROCESSOR_GROUP_INFO structures may exclude some active logical processors.

When this function is called with a relationship type of **RelationProcessorCore**, it returns a PROCESSOR_RELATIONSHIP structure for every active processor core in every processor group in the system. This is by design, because an unaffinitized 32-bit thread can run on any logical processor in a given group, including processors 32 through 63. A 32-bit caller can use the total count of **PROCESSOR_RELATIONSHIP** structures to determine the actual number of active processor cores on the system. However, the affinity of a 32-bit thread cannot be explicitly set to logical processor 32 through 63 of any processor group.

To compile an application that uses this function, set _WIN32_WINNT >= 0x0601. For more information, see Using the Windows Headers.

Behavior starting with Windows Server 2022 (21H2, build 20348)

The behavior of this and other NUMA functions has been modified to better support systems with nodes containing more that 64 processors. For more information about this change, including information about enabling the old behavior of this API, see NUMA Support.

Requests for RelationNumaNode will return NUMA_NODE_RELATIONSHIP structures that contain only the affinity of the node within it's primary group. The GroupCount value will be 1 and the structure size is fixed.

Requests for **RelationNumaNodeEx** or **RelationAll** will return **NUMA_NODE_RELATIONSHIP** structures that contain an array of affinities for the node over all groups. The **GroupCount** reports the number of affinities and the size of structure is variable.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 [desktop apps | UWP apps]
Minimum supported server	Windows Server 2008 R2 [desktop apps | UWP apps]
Target Platform	Windows
Header	sysinfoapi.h (include Windows.h)
Library	Kernel32.lib

Requirement	Value
DLL	Kernel32.dll

See also

SYSTEM_LOGICAL_PROCESSOR_INFORMATION_EX

GetMaximumProcessorCount function (winbase.h)

02/22/2024

Returns the maximum number of logical processors that a processor group or the system can have.

Syntax

```
C++

DWORD GetMaximumProcessorCount(
  [in] WORD GroupNumber
);
```

Parameters

`[in] GroupNumber`

The processor group number. If this parameter is ALL_PROCESSOR_GROUPS, the function returns the maximum number of processors that the system can have.

Return value

If the function succeeds, the return value is the maximum number of processors that the specified group can have.

If the function fails, the return value is zero. To get extended error information, use GetLastError.

Remarks

To compile an application that uses this function, set _WIN32_WINNT >= 0x0601. For more information, see Using the Windows Headers.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

GetMaximumProcessorGroupCount function (winbase.h)

02/22/2024

Returns the maximum number of processor groups that the system can have.

Syntax

```
C++

WORD GetMaximumProcessorGroupCount();
```

Return value

If the function succeeds, the return value is the maximum number of processor groups that the system can have.

If the function fails, the return value is zero.

Remarks

To compile an application that uses this function, set _WIN32_WINNT >= 0x0601. For more information, see Using the Windows Headers.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

GetNextUmsListItem function (winbase.h)

Article02/22/2024

Returns the next user-mode scheduling (UMS) thread context in a list of thread contexts.

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

Syntax

```C++
PUMS_CONTEXT GetNextUmsListItem(
  [in, out] PUMS_CONTEXT UmsContext
);
```

Parameters

`[in, out] UmsContext`

A pointer to a UMS context in a list of thread contexts. This list is retrieved by the DequeueUmsCompletionListItems function.

Return value

If the function succeeds, it returns a pointer to the next thread context in the list.

If there is no thread context after the context specified by the *UmsContext* parameter, the function returns NULL. To get extended error information, call GetLastError.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 (64-bit only) [desktop apps only]

Requirement	Value
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
API set	api-ms-win-core-ums-l1-1-0 (introduced in Windows 7)

See also

DequeueUmsCompletionListItems

GetNumaAvailableMemoryNode function (winbase.h)

Article10/13/2021

Retrieves the amount of memory available in the specified node.

Use the GetNumaAvailableMemoryNodeEx function to specify the node as a **USHORT** value.

Syntax

```
C++

BOOL GetNumaAvailableMemoryNode(
  [in]  UCHAR      Node,
  [out] PULONGLONG AvailableBytes
);
```

Parameters

`[in] Node`

The number of the node.

`[out] AvailableBytes`

The amount of available memory for the node, in bytes.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The **GetNumaAvailableMemoryNode** function returns the amount of memory consumed by free and zeroed pages on the specified node. On systems with more than one node, this memory does not include standby pages. Therefore, the sum of the available memory values for all nodes in the system is equal to the value of the Free & Zero Page List Bytes memory performance counter. On systems with only one node, the value returned by

GetNumaAvailableMemoryNode includes standby pages and is equal to the value of the Available Bytes memory performance counter. For more information about performance counters, see Memory Performance Information.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista, Windows XP Professional x64 Edition, Windows XP with SP2 [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

GetNumaAvailableMemoryNodeEx

NUMA Support

Process and Thread Functions

GetNumaAvailableMemoryNodeEx function (winbase.h)

02/22/2024

Retrieves the amount of memory that is available in a node specified as a **USHORT** value.

Syntax

```C++
BOOL GetNumaAvailableMemoryNodeEx(
  [in]  USHORT     Node,
  [out] PULONGLONG AvailableBytes
);
```

Parameters

`[in] Node`

The number of the node.

`[out] AvailableBytes`

The amount of available memory for the node, in bytes.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The **GetNumaAvailableMemoryNodeEx** function returns the amount of memory consumed by free and zeroed pages on the specified node. On systems with more than one node, this memory does not include standby pages. Therefore, the sum of the available memory values for all nodes in the system is equal to the value of the Free & Zero Page List Bytes memory performance counter. On systems with only one node, the value returned by GetNumaAvailableMemoryNode includes standby pages and is equal to the value of the

Available Bytes memory performance counter. For more information about performance counters, see Memory Performance Information.

The only difference between the **GetNumaAvailableMemoryNodeEx** function and the GetNumaAvailableMemoryNode function is the data type of the *Node* parameter.

To compile an application that uses this function, set _WIN32_WINNT >= 0x0601. For more information, see Using the Windows Headers.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

GetNumaAvailableMemoryNode

NUMA Support

GetNumaHighestNodeNumber function (systemtopologyapi.h)

09/23/2022

Retrieves the node that currently has the highest number.

Syntax

```C++
BOOL GetNumaHighestNodeNumber(
  [out] PULONG HighestNodeNumber
);
```

Parameters

`[out] HighestNodeNumber`

The number of the highest node.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The number of the highest node is not guaranteed to be the total number of nodes.

To retrieve a list of all processors in a node, use the GetNumaNodeProcessorMask function.

Examples

For an example, see Allocating Memory from a NUMA Node.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista, Windows XP Professional x64 Edition, Windows XP with SP2 [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	systemtopologyapi.h (include Windows.h on Windows Server 2003, Windows Vista, Windows 7, Windows Server 2008 Windows Server 2008 R2)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

GetNumaNodeProcessorMask

NUMA Support

Process and Thread Functions

GetNumaNodeNumberFromHandle function (winbase.h)

Article02/22/2024

Retrieves the NUMA node associated with the file or I/O device represented by the specified file handle.

Syntax

```C++
BOOL GetNumaNodeNumberFromHandle(
  [in]  HANDLE  hFile,
  [out] PUSHORT NodeNumber
);
```

Parameters

`[in] hFile`

A handle to a file or I/O device. Examples of I/O devices include files, file streams, volumes, physical disks, and sockets. For more information, see the CreateFile function.

`[out] NodeNumber`

A pointer to a variable to receive the number of the NUMA node associated with the specified file handle.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, use GetLastError.

Remarks

If the specified handle does not have a node associated with it, the function returns FALSE. The value of the *NodeNumber* parameter is undetermined and should not be used.

To compile an application that uses this function, set _WIN32_WINNT >= 0x0601. For more information, see Using the Windows Headers.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

CreateFile

NUMA Support

GetNumaNodeProcessorMask function (winbase.h)

10/13/2021

Retrieves the processor mask for the specified node.

Syntax

```C++
BOOL GetNumaNodeProcessorMask(
  [in]  UCHAR      Node,
  [out] PULONGLONG ProcessorMask
);
```

Parameters

`[in] Node`

The number of the node.

`[out] ProcessorMask`

The processor mask for the node. A processor mask is a bit vector in which each bit represents a processor and whether it is in the node.

If the node has no processors configured, the processor mask is zero.

On systems with more than 64 processors, this parameter is set to the processor mask for the node only if the node is in the same processor group as the calling thread. Otherwise, the parameter is set to zero.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

To 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.

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.

Use the GetNumaNodeProcessorMaskEx function to retrieve the processor mask for a node in any processor group.

> **Note**
>
> Starting with *TBD Release Iron*, the behavior of this and other NUMA functions has been modified to better support systems with nodes containing more that 64 processors. For more information about this change, including information about enabling the old behavior of this API, see NUMA Support.

Behavior starting with TBD Release Iron

Each node is assigned a primary group by the system. The mask returned by **GetNumaNodeProcessorMask** is for the node's primary group and is only returned if the calling thread belongs to that group.

Behavior in previous versions

The mask for the specified node is returned.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista, Windows XP Professional x64 Edition, Windows XP with SP2 [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows

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

See also

GetNumaNodeProcessorMaskEx

GetNumaProcessorNode

NUMA Support

Process and Thread Functions

SetProcessAffinityMask

GetNumaNodeProcessorMaskEx function (systemtopologyapi.h)

10/13/2021

Retrieves the processor mask for a node regardless of the processor group the node belongs to.

Syntax

```C++
BOOL GetNumaNodeProcessorMaskEx(
  [in]  USHORT          Node,
  [out] PGROUP_AFFINITY ProcessorMask
);
```

Parameters

`[in] Node`

The node number.

`[out] ProcessorMask`

A pointer to a GROUP_AFFINITY structure that receives the processor mask for the specified node. A processor mask is a bit vector in which each bit represents a processor and whether it is in the node.

If the specified node has no processors configured, the **Mask** member is zero and the **Group** member is undefined.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Remarks

The **GetNumaNodeProcessorMaskEx** function differs from GetNumaNodeProcessorMask in that it can retrieve the processor mask for a node regardless of the group the node belongs to. That is, the node does not have to be in the same group as the calling thread. The **GetNumaNodeProcessorMask** function can retrieve the processor mask only for nodes that are in the same group as the calling thread.

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.

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.

To compile an application that uses this function, set **_WIN32_WINNT** >= 0x0601. For more information, see Using the Windows Headers.

> **Note**
>
> Starting with *TBD Release Iron*, the behavior of this and other NUMA functions has been modified to better support systems with nodes containing more that 64 processors. For more information about this change, including information about enabling the old behavior of this API, see NUMA Support.

Behavior starting with TBD Release Iron

Each node is assigned a primary group by the system. The **GROUP_AFFINITY** returned by **GetNumaNodeProcessorMaskEx** is for the node's primary group.

Behavior in previous versions

The **GROUP_AFFINITY** for the specified node is returned.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 [desktop apps only]

Requirement	Value
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	systemtopologyapi.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

GROUP_AFFINITY

GetNumaNodeProcessorMask

NUMA Support

Processor Groups

GetNumaProcessorNode function (winbase.h)

Article10/13/2021

Retrieves the node number for the specified processor.

Use the GetNumaProcessorNodeEx function to specify a processor group and retrieve the node number as a **USHORT** value.

Syntax

```
C++

BOOL GetNumaProcessorNode(
  [in]  UCHAR  Processor,
  [out] PUCHAR NodeNumber
);
```

Parameters

`[in] Processor`

The processor number.

On a system with more than 64 logical processors, the processor number is relative to the processor group that contains the processor on which the calling thread is running.

`[out] NodeNumber`

The node number. If the processor does not exist, this parameter is 0xFF.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

To retrieve the list of processors on the system, use the GetProcessAffinityMask function.

Examples

For an example, see Allocating Memory from a NUMA Node.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista, Windows XP Professional x64 Edition, Windows XP with SP2 [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

GetNumaNodeProcessorMask

GetNumaProcessorNodeEx

GetNumaProximityNode

GetProcessAffinityMask

NUMA Support

GetNumaProcessorNodeEx function (winbase.h)

02/22/2024

Retrieves the node number as a **USHORT** value for the specified logical processor.

Syntax

```C++
BOOL GetNumaProcessorNodeEx(
  [in]  PPROCESSOR_NUMBER Processor,
  [out] PUSHORT           NodeNumber
);
```

Parameters

`[in] Processor`

A pointer to a PROCESSOR_NUMBER structure that represents the logical processor and the processor group to which it is assigned.

`[out] NodeNumber`

A pointer to a variable to receive the node number. If the specified processor does not exist, this parameter is set to MAXUSHORT.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

To compile an application that uses this function, set _WIN32_WINNT >= 0x0601. For more information, see Using the Windows Headers.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

GetNumaProcessorNode

PROCESSOR_NUMBER

GetNumaProximityNode function (winbase.h)

Article02/22/2024

Retrieves the NUMA node number that corresponds to the specified proximity domain identifier.

Use the GetNumaProximityNodeEx function to retrieve the node number as a **USHORT** value.

Syntax

```
C++

BOOL GetNumaProximityNode(
  [in]  ULONG  ProximityId,
  [out] PUCHAR NodeNumber
);
```

Parameters

`[in] ProximityId`

The proximity domain identifier of the node.

`[out] NodeNumber`

The node number. If the processor does not exist, this parameter is 0xFF.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

A proximity domain identifier is an index to a NUMA node on a NUMA system. Proximity domain identifiers are found in the ACPI System Resource Affinity Table (SRAT), where they are used to associate processors and memory regions with a particular NUMA node. Proximity domain identifiers are also found in the ACPI namespace, where they are used to associate a

device with a particular NUMA node. Proximity domain identifiers are typically used only by management applications provided by system manufacturers. Windows does not use proximity domain identifiers to identify NUMA nodes; instead, it assigns a unique number to each NUMA node in the system.

The relative distance between nodes on a system is stored in the ACPI System Locality Distance Information Table (SLIT), which is not exposed by any Windows functions. For more information about ACPI tables, see the ACPI specifications.

Requirements

Expand table

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

See also

GetNumaProcessorNode

GetNumaProximityNodeEx

NUMA Support

GetNumaProximityNodeEx function (systemtopologyapi.h)

02/22/2024

Retrieves the NUMA node number that corresponds to the specified proximity identifier as a **USHORT** value.

Syntax

```C++
BOOL GetNumaProximityNodeEx(
  [in]  ULONG   ProximityId,
  [out] PUSHORT NodeNumber
);
```

Parameters

`[in] ProximityId`

The proximity identifier of the node.

`[out] NodeNumber`

Points to a variable to receive the node number for the specified proximity identifier.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Remarks

A proximity domain identifier is an index to a NUMA node on a NUMA system. Proximity domain identifiers are found in the ACPI System Resource Affinity Table (SRAT), where they are used to associate processors and memory regions with a particular NUMA node. Proximity domain identifiers are also found in the ACPI namespace, where they are used to associate a device with a particular NUMA node. Proximity domain identifiers are typically used only by management applications provided by system manufacturers. Windows does not use proximity

domain identifiers to identify NUMA nodes; instead, it assigns a unique number to each NUMA node in the system.

The relative distance between nodes on a system is stored in the ACPI System Locality Distance Information Table (SLIT), which is not exposed by any Windows functions. For more information about ACPI tables, see the ACPI specifications.

The only difference between the **GetNumaProximityNodeEx** function and the GetNumaProximityNode function is the data type of the *NodeNumber* parameter.

To compile an application that uses this function, set _WIN32_WINNT >= 0x0601. For more information, see Using the Windows Headers.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	systemtopologyapi.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

GetNumaProximityNode

NUMA Support

GetPriorityClass function (processthreadsapi.h)

11/01/2022

Retrieves the priority class for the specified process. This value, together with the priority value of each thread of the process, determines each thread's base priority level.

Syntax

```C++
DWORD GetPriorityClass(
  [in] HANDLE hProcess
);
```

Parameters

`[in] hProcess`

A handle to the process.

The handle must have the **PROCESS_QUERY_INFORMATION** or **PROCESS_QUERY_LIMITED_INFORMATION** access right. For more information, see Process Security and Access Rights.

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

Return value

If the function succeeds, the return value is the priority class of the specified process.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

The process's priority class is one of the following values.

Expand table

Return code/value	Description

ABOVE_NORMAL_PRIORITY_CLASS 0x00008000	Process that has priority above **NORMAL_PRIORITY_CLASS** but below **HIGH_PRIORITY_CLASS**.
BELOW_NORMAL_PRIORITY_CLASS 0x00004000	Process that has priority above **IDLE_PRIORITY_CLASS** but below **NORMAL_PRIORITY_CLASS**.
HIGH_PRIORITY_CLASS 0x00000080	Process that performs time-critical tasks that must be executed immediately for it to run correctly. The threads of a high-priority class process preempt the threads of normal or idle priority class processes. An example is the Task List, which must respond quickly when called by the user, regardless of the load on the operating system. Use extreme care when using the high-priority class, because a high-priority class CPU-bound application can use nearly all available cycles.
IDLE_PRIORITY_CLASS 0x00000040	Process whose threads run only when the system is idle and are preempted by the threads of any process running in a higher priority class. An example is a screen saver. The idle priority class is inherited by child processes.
NORMAL_PRIORITY_CLASS 0x00000020	Process with no special scheduling needs.
REALTIME_PRIORITY_CLASS 0x00000100	Process that has the highest possible priority. The threads of a real-time priority class process preempt the threads of all other processes, including operating system processes performing important tasks. For example, a real-time process that executes for more than a very brief interval can cause disk caches not to flush or cause the mouse to be unresponsive.

Remarks

Every thread has a base priority level determined by the thread's priority value and the priority class of its process. The operating system uses the base priority level of all executable threads to determine which thread gets the next slice of CPU time. Threads are scheduled in a round-robin fashion at each priority level, and only when there are no executable threads at a higher level will scheduling of threads at a lower level take place.

For a table that shows the base priority levels for each combination of priority class and thread priority value, see Scheduling Priorities.

Priority class is maintained by the executive, so all processes have a priority class that can be queried.

Examples

For an example, see Taking a Snapshot and Viewing Processes.

Requirements

Expand table

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

See also

GetThreadPriority

Process and Thread Functions

Processes

Scheduling Priorities

SetPriorityClass

SetThreadPriority

GetProcessAffinityMask function (winbase.h)

Article01/27/2022

Retrieves the process affinity mask for the specified process and the system affinity mask for the system.

Syntax

```
C++

BOOL GetProcessAffinityMask(
  [in]  HANDLE     hProcess,
  [out] PDWORD_PTR lpProcessAffinityMask,
  [out] PDWORD_PTR lpSystemAffinityMask
);
```

Parameters

`[in] hProcess`

A handle to the process whose affinity mask is desired.

This handle must have the **PROCESS_QUERY_INFORMATION** or **PROCESS_QUERY_LIMITED_INFORMATION** access right. For more information, see Process Security and Access Rights.

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

`[out] lpProcessAffinityMask`

A pointer to a variable that receives the affinity mask for the specified process.

`[out] lpSystemAffinityMask`

A pointer to a variable that receives the affinity mask for the system.

Return value

If the function succeeds, the return value is nonzero and the function sets the variables pointed to by *lpProcessAffinityMask* and *lpSystemAffinityMask* to the appropriate affinity masks.

On a system with more than 64 processors, if the threads of the calling process are in a single processor group, the function sets the variables pointed to by *lpProcessAffinityMask* and *lpSystemAffinityMask* to the process affinity mask and the processor mask of active logical processors for that group. If the calling process contains threads in multiple groups, the function returns zero for both affinity masks.

If the function fails, the return value is zero, and the values of the variables pointed to by *lpProcessAffinityMask* and *lpSystemAffinityMask* are undefined. To get extended error information, call GetLastError.

Remarks

A process affinity mask is a bit vector in which each bit represents the processors that a process is allowed to run on. A system affinity mask is a bit vector in which each bit represents the processors that are configured into a system.

A process affinity mask is a subset of the system affinity mask. A process is only allowed to run on the processors configured into a system. Therefore, the process affinity mask cannot specify a 1 bit for a processor when the system affinity mask specifies a 0 bit for that processor.

Starting with Windows 11 and Windows Server 2022, on a system with more than 64 processors, process and thread affinities span all processors in the system, across all processor groups, by default. The **GetProcessAffinityMask** function sets the *lpProcessAffinityMask* and *lpSystemAffinityMask* to the process and system processor masks over the process' primary group. If the process had explicitly set the affinity of one or more of its threads outside of the process' primary group, the function returns zero for both affinity masks. If, however, *hHandle* specifies a handle to the current process, the function always uses the calling thread's primary group (which by default is the same as the process' primary group) in order to set the *lpProcessAffinityMask* and *lpSystemAffinityMask*.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)

Requirement	Value
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Multiple Processors

Process and Thread Functions

Processes

Processor Groups

SetProcessAffinityMask

SetThreadAffinityMask

SetProcessDefaultCpuSets function (processthreadsapi.h)

02/22/2024

Sets the default CPU Sets assignment for threads in the specified process. Threads that are created, which don't have CPU Sets explicitly set using SetThreadSelectedCpuSets, will inherit the sets specified by **SetProcessDefaultCpuSets** automatically.

Syntax

C++

```
BOOL SetProcessDefaultCpuSets(
  HANDLE      Process,
  const ULONG *CpuSetIds,
  ULONG       CpuSetIdCount
);
```

Parameters

`Process`

Specifies the process for which to set the default CPU Sets. This handle must have the PROCESS_SET_LIMITED_INFORMATION access right. The value returned by GetCurrentProcess can also be specified here.

`CpuSetIds`

Specifies the list of CPU Set IDs to set as the process default CPU set. If this is NULL, the **SetProcessDefaultCpuSets** clears out any assignment.

`CpuSetIdCount`

Specifies the number of IDs in the list passed in the **CpuSetIds** argument. If that value is NULL, this should be 0.

Return value

This function cannot fail when passed valid parameters

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	processthreadsapi.h
DLL	kernel32.dll

GetProcessGroupAffinity function (processtopologyapi.h)

Article09/23/2022

Retrieves the processor group affinity of the specified process.

Syntax

```C++
BOOL GetProcessGroupAffinity(
  [in]      HANDLE  hProcess,
  [in, out] PUSHORT GroupCount,
  [out]     PUSHORT GroupArray
);
```

Parameters

`[in] hProcess`

A handle to the process.

This handle must have the PROCESS_QUERY_INFORMATION or PROCESS_QUERY_LIMITED_INFORMATION access right. For more information, see Process Security and Access Rights.

`[in, out] GroupCount`

On input, specifies the number of elements in *GroupArray* array. On output, specifies the number of processor groups written to the array. If the array is too small, the function fails with ERROR_INSUFFICIENT_BUFFER and sets the *GroupCount* parameter to the number of elements required.

`[out] GroupArray`

An array of processor group numbers. A group number is included in the array if a thread in the process is assigned to a processor in the group.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, use GetLastError.

If the error value is ERROR_INSUFFICIENT_BUFFER, the *GroupCount* parameter contains the required buffer size in number of elements.

Remarks

Starting with Windows 11 and Windows Server 2022, on a system with more than 64 processors, process and thread affinities span all processors in the system, across all processor groups, by default.

To compile an application that uses this function, set _WIN32_WINNT >= 0x0601. For more information, see Using the Windows Headers.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	processtopologyapi.h (include Windows.h on Windows Server 2008 Windows Server 2008 R2)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

GetThreadGroupAffinity

Processor Groups

GetProcessHandleCount function (processthreadsapi.h)

02/22/2024

Retrieves the number of open handles that belong to the specified process.

Syntax

```
C++

BOOL GetProcessHandleCount(
  [in]      HANDLE hProcess,
  [in, out] PDWORD pdwHandleCount
);
```

Parameters

`[in] hProcess`

A handle to the process whose handle count is being requested. The handle must have the PROCESS_QUERY_INFORMATION or PROCESS_QUERY_LIMITED_INFORMATION access right. For more information, see Process Security and Access Rights.

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

`[in, out] pdwHandleCount`

A pointer to a variable that receives the number of open handles that belong to the specified process.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

This function retrieves information about the executive objects for the process. For more information, see Kernel Objects.

To compile an application that uses this function, define _WIN32_WINNT as 0x0501 or later. For more information, see Using the Windows Headers.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista, Windows XP with SP1 [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	processthreadsapi.h (include Windows.h on Windows Server 2003, Windows Vista, Windows 7, Windows Server 2008 Windows Server 2008 R2)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Process and Thread Functions

Processes

GetProcessId function (processthreadsapi.h)

02/22/2024

Retrieves the process identifier of the specified process.

Syntax

```C++
DWORD GetProcessId(
  [in] HANDLE Process
);
```

Parameters

`[in] Process`

A handle to the process. The handle must have the PROCESS_QUERY_INFORMATION or PROCESS_QUERY_LIMITED_INFORMATION access right. For more information, see Process Security and Access Rights.

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

Return value

If the function succeeds, the return value is the process identifier.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

Until a process terminates, its process identifier uniquely identifies it on the system. For more information about access rights, see Process Security and Access Rights.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista, Windows XP with SP1 [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	processthreadsapi.h (include Windows.h on Windows Server 2003, Windows Vista, Windows 7, Windows Server 2008 Windows Server 2008 R2)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

GetCurrentProcessId

GetProcessIdOfThread

GetThreadId

Processes

GetProcessIdOfThread function (processthreadsapi.h)

02/22/2024

Retrieves the process identifier of the process associated with the specified thread.

Syntax

```C++
DWORD GetProcessIdOfThread(
  [in] HANDLE Thread
);
```

Parameters

`[in] Thread`

A handle to the thread. The handle must have the THREAD_QUERY_INFORMATION or THREAD_QUERY_LIMITED_INFORMATION access right. For more information, see Thread Security and Access Rights.

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

Return value

If the function succeeds, the return value is the process identifier of the process associated with the specified thread.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

Until a process terminates, its process identifier uniquely identifies it on the system. For more information about access rights, see Thread Security and Access Rights.

Requirements

Expand table

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

See also

GetCurrentThreadId

GetProcessId

GetThreadId

Processes

GetProcessInformation function (processthreadsapi.h)

01/31/2024

Retrieves information about the specified process.

Syntax

```C++
BOOL GetProcessInformation(
  [in] HANDLE                    hProcess,
  [in] PROCESS_INFORMATION_CLASS ProcessInformationClass,
       LPVOID                    ProcessInformation,
  [in] DWORD                     ProcessInformationSize
);
```

Parameters

`[in] hProcess`

A handle to the process. This handle must have at least the **PROCESS_QUERY_LIMITED_INFORMATION** access right. For more information, see Process Security and Access Rights.

`[in] ProcessInformationClass`

A member of the PROCESS_INFORMATION_CLASS enumeration specifying the kind of information to retrieve.

`ProcessInformation`

Pointer to an object to receive the type of information specified by the *ProcessInformationClass* parameter.

If the *ProcessInformationClass* parameter is **ProcessMemoryPriority**, this parameter must point to a MEMORY_PRIORITY_INFORMATION structure.

If the *ProcessInformationClass* parameter is **ProcessPowerThrottling**, this parameter must point to a PROCESS_POWER_THROTTLING_STATE structure.

If the *ProcessInformationClass* parameter is **ProcessProtectionLevelInfo**, this parameter must point to a PROCESS_PROTECTION_LEVEL_INFORMATION structure.

If the *ProcessInformationClass* parameter is **ProcessLeapSecondInfo**, this parameter must point to a PROCESS_LEAP_SECOND_INFO structure.

If the *ProcessInformationClass* parameter is **ProcessAppMemoryInfo**, this parameter must point to a APP_MEMORY_INFORMATION structure.

If the *ProcessInformationClass* parameter is **ProcessMaxOverridePrefetchParameter**, this parameter must point to an OVERRIDE_PREFETCH_PARAMETER structure.

`[in] ProcessInformationSize`

The size in bytes of the structure specified by the *ProcessInformation* parameter.

If the *ProcessInformationClass* parameter is **ProcessMemoryPriority**, this parameter must be `sizeof(MEMORY_PRIORITY_INFORMATION)`.

If the *ProcessInformationClass* parameter is **ProcessPowerThrottling**, this parameter must be `sizeof(PROCESS_POWER_THROTTLING_STATE)`.

If the *ProcessInformationClass* parameter is **ProcessProtectionLevelInfo**, this parameter must be `sizeof(PROCESS_PROTECTION_LEVEL_INFORMATION)`.

If the *ProcessInformationClass* parameter is **ProcessLeapSecondInfo**, this parameter must be `sizeof(PROCESS_LEAP_SECOND_INFO)`.

If the *ProcessInformationClass* parameter is **ProcessAppMemoryInfo**, this parameter must be `sizeof(APP_MEMORY_INFORMATION)`.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError function.

Requirements

Expand table

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

See also

GetThreadInformation function, MEMORY_PRIORITY_INFORMATION structure, SetProcessInformation function, PROCESS_INFORMATION_CLASS enumeration, OVERRIDE_PREFETCH_PARAMETER structure

GetProcessIoCounters function (winbase.h)

Article02/22/2024

Retrieves accounting information for all I/O operations performed by the specified process.

Syntax

```
C++

BOOL GetProcessIoCounters(
  [in]  HANDLE       hProcess,
  [out] PIO_COUNTERS lpIoCounters
);
```

Parameters

`[in] hProcess`

A handle to the process. The handle must have the **PROCESS_QUERY_INFORMATION** or **PROCESS_QUERY_LIMITED_INFORMATION** access right. For more information, see Process Security and Access Rights.

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

`[out] lpIoCounters`

A pointer to an IO_COUNTERS structure that receives the I/O accounting information for the process.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Requirements

Expand table

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

See also

IO_COUNTERS

Process and Thread Functions

GetProcessMitigationPolicy function (processthreadsapi.h)

11/01/2022

Retrieves mitigation policy settings for the calling process.

Syntax

```C++
BOOL GetProcessMitigationPolicy(
  [in]  HANDLE                    hProcess,
  [in]  PROCESS_MITIGATION_POLICY MitigationPolicy,
  [out] PVOID                     lpBuffer,
  [in]  SIZE_T                    dwLength
);
```

Parameters

`[in] hProcess`

A handle to the process. This handle must have the PROCESS_QUERY_INFORMATION access right. For more information, see Process Security and Access Rights.

`[in] MitigationPolicy`

The mitigation policy to retrieve. This parameter can be one of the following values.

Expand table

Value	Meaning
ProcessDEPPolicy	The data execution prevention (DEP) policy of the process. The *lpBuffer* parameter points to a PROCESS_MITIGATION_DEP_POLICY structure that specifies the DEP policy flags.
ProcessASLRPolicy	The Address Space Layout Randomization (ASLR) policy of the process. The *lpBuffer* parameter points to a PROCESS_MITIGATION_ASLR_POLICY structure that specifies the ASLR policy flags.

Value	Meaning
ProcessDynamicCodePolicy	The dynamic code policy of the process. When turned on, the process cannot generate dynamic code or modify existing executable code. The *lpBuffer* parameter points to a PROCESS_MITIGATION_DYNAMIC_CODE_POLICY structure that specifies the dynamic code policy flags.
ProcessStrictHandleCheckPolicy	The process will receive a fatal error if it manipulates a handle that is not valid. The *lpBuffer* parameter points to a PROCESS_MITIGATION_STRICT_HANDLE_CHECK_POLICY structure that specifies the handle check policy flags.
ProcessSystemCallDisablePolicy	Disables the ability to use NTUser/GDI functions at the lowest layer. The *lpBuffer* parameter points to a PROCESS_MITIGATION_SYSTEM_CALL_DISABLE_POLICY structure that specifies the system call disable policy flags.
ProcessMitigationOptionsMask	Returns the mask of valid bits for all the mitigation options on the system. An application can set many mitigation options without querying the operating system for mitigation options by combining bitwise with the mask to exclude all non-supported bits at once. The *lpBuffer* parameter points to a **ULONG64** bit vector for the mask, or a two-element array of **ULONG64** bit vectors.
ProcessExtensionPointDisablePolicy	Prevents certain built-in third party extension points from being enabled, preventing legacy extension point DLLs from being loaded into the process. The *lpBuffer* parameter points to a PROCESS_MITIGATION_EXTENSION_POINT_DISABLE_POLICY structure that specifies the extension point disable policy flags.
ProcessControlFlowGuardPolicy	The Control Flow Guard (CFG) policy of the process. The *lpBuffer* parameter points to a PROCESS_MITIGATION_CONTROL_FLOW_GUARD_POLICY structure that specifies the CFG policy flags.
ProcessSignaturePolicy	The policy of a process that can restrict image loading to those images that are either signed by Microsoft, by the Windows Store, or by Microsoft, the Windows Store and the Windows Hardware Quality Labs (WHQL). he *lpBuffer* parameter points to a PROCESS_MITIGATION_BINARY_SIGNATURE_POLICY structure that specifies the signature policy flags.
ProcessFontDisablePolicy	The policy regarding font loading for the process. When turned on, the process cannot load non-system fonts.

Value	Meaning
	The *lpBuffer* parameter points to a PROCESS_MITIGATION_FONT_DISABLE_POLICY structure that specifies the policy flags for font loading.
ProcessImageLoadPolicy	The policy regarding image loading for the process, which determines the types of executable images that are allowed to be mapped into the process. When turned on, images cannot be loaded from some locations, such a remote devices or files that have the low mandatory label. The *lpBuffer* parameter points to a PROCESS_MITIGATION_IMAGE_LOAD_POLICY structure that specifies the policy flags for image loading.
ProcessRedirectionTrustPolicy	The RedirectionGuard policy of a process. The *lpBuffer* parameter points to a PROCESS_MITIGATION_REDIRECTION_TRUST_POLICY structure that specifies the mitigation mode.
ProcessSideChannelIsolationPolicy	Windows 10, version 1809 and above: The policy regarding isolation of side channels for the specified process. The *lpBuffer* parameter points to a PROCESS_MITIGATION_SIDE_CHANNEL_ISOLATION_POLICY structure that specifies the policy flags for side channel isolation.
ProcessUserShadowStackPolicy	Windows 10, version 2004 and above: The policy regarding user-mode Hardware-enforced Stack Protection for the specified process. The *lpBuffer* parameter points to a PROCESS_MITIGATION_USER_SHADOW_STACK_POLICY structure that specifies the policy flags for user-mode Hardware-enforced Stack Protection.

`[out] lpBuffer`

If the *MitigationPolicy* parameter is **ProcessDEPPolicy**, this parameter points to a PROCESS_MITIGATION_DEP_POLICY structure that receives the DEP policy flags.

If the *MitigationPolicy* parameter is **ProcessASLRPolicy**, this parameter points to a PROCESS_MITIGATION_ASLR_POLICY structure that receives the ASLR policy flags.

If the *MitigationPolicy* parameter is **ProcessDynamicCodePolicy**, this parameter points to a PROCESS_MITIGATION_DYNAMIC_CODE_POLICY structure that receives the dynamic code policy flags.

If the *MitigationPolicy* parameter is **ProcessStrictHandleCheckPolicy**, this parameter points to a PROCESS_MITIGATION_STRICT_HANDLE_CHECK_POLICY structure that specifies the handle check policy flags.

If the *MitigationPolicy* parameter is **ProcessSystemCallDisablePolicy**, this parameter points to a PROCESS_MITIGATION_SYSTEM_CALL_DISABLE_POLICY structure that specifies the system call disable policy flags.

If the *MitigationPolicy* parameter is **ProcessMitigationOptionsMask**, this parameter points to a **ULONG64** bit vector for the mask or a two-element array of **ULONG64** bit vectors.

If the *MitigationPolicy* parameter is **ProcessExtensionPointDisablePolicy**, this parameter points to a PROCESS_MITIGATION_EXTENSION_POINT_DISABLE_POLICY structure that specifies the extension point disable policy flags.

If the *MitigationPolicy* parameter is **ProcessControlFlowGuardPolicy**, this parameter points to a PROCESS_MITIGATION_CONTROL_FLOW_GUARD_POLICY structure that specifies the CFG policy flags.

If the *MitigationPolicy* parameter is **ProcessSignaturePolicy**, this parameter points to a PROCESS_MITIGATION_BINARY_SIGNATURE_POLICY structure that receives the signature policy flags.

If the *MitigationPolicy* parameter is **ProcessFontDisablePolicy**, this parameter points to a PROCESS_MITIGATION_FONT_DISABLE_POLICY structure that receives the policy flags for font loading.

If the *MitigationPolicy* parameter is **ProcessImageLoadPolicy**, this parameter points to a PROCESS_MITIGATION_IMAGE_LOAD_POLICY structure that receives the policy flags for image loading.

If the *MitigationPolicy* parameter is **ProcessRedirectionTrustPolicy**, this parameter points to a PROCESS_MITIGATION_REDIRECTION_TRUST_POLICY structure that specifies the mitigation mode.

If the *MitigationPolicy* parameter is **ProcessUserShadowStackPolicy**, this parameter points to a PROCESS_MITIGATION_USER_SHADOW_STACK_POLICY structure that receives the policy flags for user-mode Hardware-enforced Stack Protection.

`[in] dwLength`

The size of *lpBuffer*, in bytes.

Return value

If the function succeeds, it returns **TRUE**. If the function fails, it returns **FALSE**. To retrieve error values defined for this function, call GetLastError.

Remarks

To compile an application that uses this function, set _WIN32_WINNT >= 0x0602. For more information, see Using the Windows Headers.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8 [desktop apps | UWP apps]
Minimum supported server	Windows Server 2012 [desktop apps | UWP apps]
Target Platform	Windows
Header	processthreadsapi.h
Library	Kernel32.lib
DLL	Kernel32.dll

GetProcessPriorityBoost function (processthreadsapi.h)

02/22/2024

Retrieves the priority boost control state of the specified process.

Syntax

```C++
BOOL GetProcessPriorityBoost(
  [in]  HANDLE hProcess,
  [out] PBOOL  pDisablePriorityBoost
);
```

Parameters

`[in] hProcess`

A handle to the process. This handle must have the **PROCESS_QUERY_INFORMATION** or **PROCESS_QUERY_LIMITED_INFORMATION** access right. For more information, see Process Security and Access Rights.

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

`[out] pDisablePriorityBoost`

A pointer to a variable that receives the priority boost control state. A value of TRUE indicates that dynamic boosting is disabled. A value of FALSE indicates normal behavior.

Return value

If the function succeeds, the return value is nonzero. In that case, the variable pointed to by the *pDisablePriorityBoost* parameter receives the priority boost control state.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Requirements

Expand table

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

See also

Priority Boosts

Process and Thread Functions

Processes

Scheduling Priorities

SetProcessPriorityBoost

GetProcessShutdownParameters function (processthreadsapi.h)

02/22/2024

Retrieves the shutdown parameters for the currently calling process.

Syntax

```C++
BOOL GetProcessShutdownParameters(
  [out] LPDWORD lpdwLevel,
  [out] LPDWORD lpdwFlags
);
```

Parameters

`[out] lpdwLevel`

A pointer to a variable that receives the shutdown priority level. Higher levels shut down first. System level shutdown orders are reserved for system components. Higher numbers shut down first. Following are the level conventions.

Expand table

Value	Meaning
000-0FF	System reserved last shutdown range.
100-1FF	Application reserved last shutdown range.
200-2FF	Application reserved "in between" shutdown range.
300-3FF	Application reserved first shutdown range.
400-4FF	System reserved first shutdown range.

All processes start at shutdown level 0x280.

`[out] lpdwFlags`

A pointer to a variable that receives the shutdown flags. This parameter can be the following value.

Expand table

Value	Meaning
SHUTDOWN_NORETRY 0x00000001	If this process takes longer than the specified timeout to shut down, do not display a retry dialog box for the user. Instead, just cause the process to directly exit.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Requirements

Expand table

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

See also

Process and Thread Functions

Processes

SetProcessShutdownParameters

GetProcessTimes function (processthreadsapi.h)

11/01/2022

Retrieves timing information for the specified process.

Syntax

```C++
BOOL GetProcessTimes(
  [in]  HANDLE     hProcess,
  [out] LPFILETIME lpCreationTime,
  [out] LPFILETIME lpExitTime,
  [out] LPFILETIME lpKernelTime,
  [out] LPFILETIME lpUserTime
);
```

Parameters

`[in] hProcess`

A handle to the process whose timing information is sought. The handle must have the **PROCESS_QUERY_INFORMATION** or **PROCESS_QUERY_LIMITED_INFORMATION** access right. For more information, see Process Security and Access Rights.

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

`[out] lpCreationTime`

A pointer to a FILETIME structure that receives the creation time of the process.

`[out] lpExitTime`

A pointer to a FILETIME structure that receives the exit time of the process. If the process has not exited, the content of this structure is undefined.

`[out] lpKernelTime`

A pointer to a FILETIME structure that receives the amount of time that the process has executed in kernel mode. The time that each of the threads of the process has executed in

kernel mode is determined, and then all of those times are summed together to obtain this value.

`[out] lpUserTime`

A pointer to a FILETIME structure that receives the amount of time that the process has executed in user mode. The time that each of the threads of the process has executed in user mode is determined, and then all of those times are summed together to obtain this value. Note that this value can exceed the amount of real time elapsed (between *lpCreationTime* and *lpExitTime*) if the process executes across multiple CPU cores.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

All times are expressed using FILETIME data structures. Such a structure contains two 32-bit values that combine to form a 64-bit count of 100-nanosecond time units.

Process creation and exit times are points in time expressed as the amount of time that has elapsed since midnight on January 1, 1601 at Greenwich, England. There are several functions that an application can use to convert such values to more generally useful forms.

Process kernel mode and user mode times are amounts of time. For example, if a process has spent one second in kernel mode, this function will fill the FILETIME structure specified by *lpKernelTime* with a 64-bit value of ten million. That is the number of 100-nanosecond units in one second.

To retrieve the number of CPU clock cycles used by the threads of the process, use the QueryProcessCycleTime function.

Requirements

Expand table

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

See also

FILETIME

FileTimeToDosDateTime

FileTimeToLocalFileTime

FileTimeToSystemTime

Process and Thread Functions

Processes

GetProcessVersion function (processthreadsapi.h)

11/01/2022

Retrieves the major and minor version numbers of the system on which the specified process expects to run.

Syntax

```C++
DWORD GetProcessVersion(
  [in] DWORD ProcessId
);
```

Parameters

`[in] ProcessId`

The process identifier of the process of interest. A value of zero specifies the calling process.

Return value

If the function succeeds, the return value is the version of the system on which the process expects to run. The high word of the return value contains the major version number. The low word of the return value contains the minor version number.

If the function fails, the return value is zero. To get extended error information, call GetLastError. The function fails if *ProcessId* is an invalid value.

Remarks

The **GetProcessVersion** function performs less quickly when *ProcessId* is nonzero, specifying a process other than the calling process.

The version number returned by this function is the version number stamped in the image header of the .exe file the process is running. Linker programs set this value.

If this function is called from a 32-bit application running on WOW64, the specified process must be a 32-bit process or the function fails.

Requirements

Expand table

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

See also

Process and Thread Functions

Processes

GetProcessWorkingSetSize function (memoryapi.h)

02/22/2024

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

Syntax

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

Parameters

`[in] hProcess`

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

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

`[out] lpMinimumWorkingSetSize`

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

`[out] lpMaximumWorkingSetSize`

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

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

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

Examples

C++

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

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

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

    // Retrieve a handle to the process.

    hProcess = OpenProcess( PROCESS_QUERY_INFORMATION,
                            FALSE, atoi(argv[1]));
     if (!hProcess)
    {
        printf( "OpenProcess failed (%d)\n", GetLastError() );
        return 1;
    }

    // Retrieve the working set size of the process.

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

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

    CloseHandle(hProcess);

    return 0;
}
```

Requirements

Expand table

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

See also

Process Working Set

Processes

SetProcessWorkingSetSize function

SetProcessWorkingSetSizeEx function

GetProcessWorkingSetSizeEx function (memoryapi.h)

02/22/2024

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

Syntax

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

Parameters

`[in] hProcess`

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

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

`[out] lpMinimumWorkingSetSize`

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

`[out] lpMaximumWorkingSetSize`

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

`[out] Flags`

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

Expand table

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

Return value

None

Remarks

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

Requirements

Expand table

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

Requirement	Value
Library	onecore.lib
DLL	Kernel32.dll

See also

Process Working Set

Processes

SetProcessWorkingSetSizeEx

GetProcessorSystemCycleTime function (sysinfoapi.h)

02/22/2024

Retrieves the cycle time each processor in the specified processor group spent executing deferred procedure calls (DPCs) and interrupt service routines (ISRs) since the processor became active.

Syntax

```C++
BOOL GetProcessorSystemCycleTime(
  [in]      USHORT                                  Group,
  [out]     PSYSTEM_PROCESSOR_CYCLE_TIME_INFORMATION Buffer,
  [in, out] PDWORD                                  ReturnedLength
);
```

Parameters

`[in] Group`

The number of the processor group for which to retrieve the cycle time.

`[out] Buffer`

A pointer to a buffer to receive a SYSTEM_PROCESSOR_CYCLE_TIME_INFORMATION structure for each processor in the group. On output, the DWORD64 **CycleTime** member of this structure is set to the cycle time for one processor.

`[in, out] ReturnedLength`

The size of the buffer, in bytes. When the function returns, this parameter contains the number of bytes written to *Buffer*. If the buffer is too small for the data, the function fails with ERROR_INSUFFICIENT_BUFFER and sets the *ReturnedLength* parameter to the required buffer size.

Return value

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

If the function fails, the return value is zero. To get extended error information, use GetLastError.

If the error value is ERROR_INSUFFICIENT_BUFFER, the *ReturnedLength* parameter contains the required buffer size.

Remarks

To compile an application that uses this function, define _WIN32_WINNT as 0x0601 or later. For more information, see Using the Windows Headers.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	sysinfoapi.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Processor Groups

GetStartupInfoW function (processthreadsapi.h)

Article02/22/2024

Retrieves the contents of the STARTUPINFO structure that was specified when the calling process was created.

Syntax

```
C++

void GetStartupInfoW(
  [out] LPSTARTUPINFOW lpStartupInfo
);
```

Parameters

`[out] lpStartupInfo`

A pointer to a STARTUPINFO structure that receives the startup information.

Return value

None

Remarks

The STARTUPINFO structure was specified by the process that created the calling process. It can be used to specify properties associated with the main window of the calling process.

Requirements

Expand table

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

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

See also

CreateProcess

Process and Thread Functions

Processes

STARTUPINFO

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

GetSystemCpuSetInformation function

Article • 01/07/2021

Allows an application to query the available CPU Sets on the system, and their current state.

Syntax

C++

```
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

Feedback

Was this page helpful? Yes No

Get help at Microsoft Q&A

GetThreadDescription function (processthreadsapi.h)

11/01/2022

Retrieves the description that was assigned to a thread by calling SetThreadDescription.

Syntax

```
C++

HRESULT GetThreadDescription(
  [in]  HANDLE hThread,
  [out] PWSTR  *ppszThreadDescription
);
```

Parameters

`[in] hThread`

A handle to the thread for which to retrieve the description. The handle must have THREAD_QUERY_LIMITED_INFORMATION access.

`[out] ppszThreadDescription`

A Unicode string that contains the description of the thread.

Return value

If the function succeeds, the return value is the **HRESULT** that denotes a successful operation. If the function fails, the return value is an **HRESULT** that denotes the error.

Remarks

Windows Server 2016, **Windows 10 LTSB 2016** and **Windows 10 version 1607**: GetThreadDescription is only available by Run Time Dynamic Linking in KernelBase.dll.

The description for a thread can change at any time. For example, a different thread can change the description of a thread of interest while you try to retrieve that description.

Thread descriptions do not need to be unique.

To free the memory for the thread description, call the LocalFree method.

Examples

The following example gets the description for a thread, prints the description, and then frees the memory for the description.

```C++
HRESULT hr = GetThreadDescription(ThreadHandle, &data);
if (SUCCEEDED(hr))
{
    wprintf(L"%ls\n", data);
    LocalFree(data);
}
```

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 10, version 1607 [desktop apps | UWP apps]
Minimum supported server	Windows Server 2016 [desktop apps | UWP apps]
Target Platform	Windows
Header	processthreadsapi.h
Library	Kernel32.lib
DLL	Kernel32.dll

See also

LocalFree, SetThreadDescription

GetThreadGroupAffinity function (processtopologyapi.h)

Article02/22/2024

Retrieves the processor group affinity of the specified thread.

Syntax

```C++
BOOL GetThreadGroupAffinity(
  [in]  HANDLE          hThread,
  [out] PGROUP_AFFINITY GroupAffinity
);
```

Parameters

`[in] hThread`

A handle to the thread for which the processor group affinity is desired.

The handle must have the THREAD_QUERY_INFORMATION or THREAD_QUERY_LIMITED_INFORMATION access right. For more information, see Thread Security and Access Rights.

`[out] GroupAffinity`

A pointer to a GROUP_AFFINITY structure that receives the group affinity of the thread.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, use GetLastError.

Remarks

Starting with Windows 11 and Windows Server 2022, on a system with more than 64 processors, process and thread affinities span all processors in the system, across all processor

groups, by default. The **GetThreadGroupAffinity** function retrieves the group affinity over the thread's primary group.

To compile an application that uses this function, set _WIN32_WINNT >= 0x0601. For more information, see Using the Windows Headers.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	processtopologyapi.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

GetProcessGroupAffinity

Processor Groups

GetThreadId function (processthreadsapi.h)

02/22/2024

Retrieves the thread identifier of the specified thread.

Syntax

```
C++

DWORD GetThreadId(
  [in] HANDLE Thread
);
```

Parameters

`[in] Thread`

A handle to the thread. The handle must have the THREAD_QUERY_INFORMATION or THREAD_QUERY_LIMITED_INFORMATION access right. For more information about access rights, see Thread Security and Access Rights.

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

Return value

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

Until a thread terminates, its thread identifier uniquely identifies it on the system.

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

Requirements

Expand table

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

See also

GetCurrentThreadId

GetProcessId

GetProcessIdOfThread

Process and Thread Functions

Threads

GetThreadIdealProcessorEx function (processthreadsapi.h)

11/01/2022

Retrieves the processor number of the ideal processor for the specified thread.

Syntax

```
C++

BOOL GetThreadIdealProcessorEx(
  [in]  HANDLE            hThread,
  [out] PPROCESSOR_NUMBER lpIdealProcessor
);
```

Parameters

`[in] hThread`

A handle to the thread for which to retrieve the ideal processor. This handle must have been created with the THREAD_QUERY_LIMITED_INFORMATION access right. For more information, see Thread Security and Access Rights.

`[out] lpIdealProcessor`

Points to PROCESSOR_NUMBER structure to receive the number of the ideal processor.

Return value

If the function succeeds, it returns a nonzero value.

If the function fails, it returns zero. To get extended error information, use GetLastError.

Remarks

To compile an application that uses this function, set _WIN32_WINNT >= 0x0601. For more information, see Using the Windows Headers.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 [desktop apps | UWP apps]
Minimum supported server	Windows Server 2008 R2 [desktop apps | UWP apps]
Target Platform	Windows
Header	processthreadsapi.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

SetThreadIdealProcessorEx

GetThreadInformation function (processthreadsapi.h)

02/22/2024

Retrieves information about the specified thread.

Syntax

```C++
BOOL GetThreadInformation(
  [in] HANDLE                    hThread,
  [in] THREAD_INFORMATION_CLASS ThreadInformationClass,
       LPVOID                    ThreadInformation,
  [in] DWORD                     ThreadInformationSize
);
```

Parameters

`[in] hThread`

A handle to the thread. The handle must have THREAD_QUERY_INFORMATION access rights. For more information, see Thread Security and Access Rights.

`[in] ThreadInformationClass`

The class of information to retrieve. This value can be **ThreadMemoryPriority**, **ThreadAbsoluteCpuPriority** or **ThreadDynamicCodePolicy**.

> **Note**
>
> **ThreadDynamicCodePolicy** is supported in Windows Server 2016 and newer, Windows 10 LTSB 2016 and newer, and Windows 10 version 1607 and newer.

`ThreadInformation`

Pointer to a structure to receive the type of information specified by the *ThreadInformationClass* parameter.

If the *ThreadInformationClass* parameter is **ThreadMemoryPriority**, this parameter must point to a **MEMORY_PRIORITY_INFORMATION** structure.

If the *ThreadInformationClass* parameter is **ThreadAbsoluteCpuPriority**, this parameter must point to a **LONG**.

If the *ThreadInformationClass* parameter is **ThreadDynamicCodePolicy**, this parameter must point to a **DWORD**.

`[in] ThreadInformationSize`

The size in bytes of the structure specified by the *ThreadInformation* parameter.

If the *ThreadInformationClass* parameter is **ThreadMemoryPriority**, this parameter must be `sizeof(MEMORY_PRIORITY_INFORMATION)`.

If the *ThreadInformationClass* parameter is **ThreadAbsoluteCpuPriority**, this parameter must be `sizeof(LONG)`.

If the *ThreadInformationClass* parameter is **ThreadDynamicCodePolicy**, this parameter must be `sizeof(DWORD)`.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Requirements

Expand table

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

See also

GetProcessInformation, SetThreadInformation

GetThreadIOPendingFlag function (processthreadsapi.h)

11/01/2022

Determines whether a specified thread has any I/O requests pending.

Syntax

```C++
BOOL GetThreadIOPendingFlag(
  [in]      HANDLE hThread,
  [in, out] PBOOL  lpIOIsPending
);
```

Parameters

`[in] hThread`

A handle to the thread in question. This handle must have been created with the THREAD_QUERY_INFORMATION access right. For more information, see Thread Security and Access Rights.

`[in, out] lpIOIsPending`

A pointer to a variable which the function sets to TRUE if the specified thread has one or more I/O requests pending, or to FALSE otherwise.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

Keep in mind that the I/O status of the specified thread can change rapidly, and may already have changed by the time the function returns. For example, a pending I/O operation could complete between the time the function sets *lpIOIsPending* and the time it returns.

To compile an application that uses this function, define _WIN32_WINNT as 0x0501 or later. For more information, see Using the Windows Headers.

Requirements

Expand table

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

See also

Process and Thread Functions

Threads

GetThreadPriority function (processthreadsapi.h)

11/01/2022

Retrieves the priority value for the specified thread. This value, together with the priority class of the thread's process, determines the thread's base-priority level.

Syntax

```C++
int GetThreadPriority(
  [in] HANDLE hThread
);
```

Parameters

`[in] hThread`

A handle to the thread.

The handle must have the **THREAD_QUERY_INFORMATION** or **THREAD_QUERY_LIMITED_INFORMATION** access right. For more information, see Thread Security and Access Rights.

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

Return value

If the function succeeds, the return value is the thread's priority level.

If the function fails, the return value is **THREAD_PRIORITY_ERROR_RETURN**. To get extended error information, call GetLastError.

Windows Phone 8.1: This function will always return **THREAD_PRIORITY_NORMAL**.

The thread's priority level is one of the following values.

Expand table

Return code/value	Description
THREAD_PRIORITY_ABOVE_NORMAL 1	Priority 1 point above the priority class.
THREAD_PRIORITY_BELOW_NORMAL -1	Priority 1 point below the priority class.
THREAD_PRIORITY_HIGHEST 2	Priority 2 points above the priority class.
THREAD_PRIORITY_IDLE -15	Base priority of 1 for **IDLE_PRIORITY_CLASS**, **BELOW_NORMAL_PRIORITY_CLASS**, **NORMAL_PRIORITY_CLASS**, **ABOVE_NORMAL_PRIORITY_CLASS**, or **HIGH_PRIORITY_CLASS** processes, and a base priority of 16 for **REALTIME_PRIORITY_CLASS** processes.
THREAD_PRIORITY_LOWEST -2	Priority 2 points below the priority class.
THREAD_PRIORITY_NORMAL 0	Normal priority for the priority class.
THREAD_PRIORITY_TIME_CRITICAL 15	Base-priority level of 15 for **IDLE_PRIORITY_CLASS**, **BELOW_NORMAL_PRIORITY_CLASS**, **NORMAL_PRIORITY_CLASS**, **ABOVE_NORMAL_PRIORITY_CLASS**, or **HIGH_PRIORITY_CLASS** processes, and a base-priority level of 31 for **REALTIME_PRIORITY_CLASS** processes.

If the thread has the **REALTIME_PRIORITY_CLASS** base class, this function can also return one of the following values: -7, -6, -5, -4, -3, 3, 4, 5, or 6. For more information, see Scheduling Priorities.

Remarks

Every thread has a base-priority level determined by the thread's priority value and the priority class of its process. The operating system uses the base-priority level of all executable threads to determine which thread gets the next slice of CPU time. Threads are scheduled in a round-robin fashion at each priority level, and only when there are no executable threads at a higher level will scheduling of threads at a lower level take place.

For a table that shows the base-priority levels for each combination of priority class and thread priority value, refer to the SetPriorityClass function.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps.

Windows Phone 8.1: Windows Phone Store apps may call this function but it has no effect.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	processthreadsapi.h (include Windows.h on Windows Server 2003, Windows Vista, Windows 7, Windows Server 2008 Windows Server 2008 R2)
Library	Kernel32.lib; WindowsPhoneCore.lib on Windows Phone 8.1
DLL	Kernel32.dll; KernelBase.dll on Windows Phone 8.1

See also

GetPriorityClass

OpenThread

Process and Thread Functions

Scheduling Priorities

SetPriorityClass

SetThreadPriority

Threads

GetThreadPriorityBoost function (processthreadsapi.h)

02/22/2024

Retrieves the priority boost control state of the specified thread.

Syntax

```
C++

BOOL GetThreadPriorityBoost(
  [in]  HANDLE hThread,
  [out] PBOOL  pDisablePriorityBoost
);
```

Parameters

`[in] hThread`

A handle to the thread. The handle must have the **THREAD_QUERY_INFORMATION** or **THREAD_QUERY_LIMITED_INFORMATION** access right. For more information, see Thread Security and Access Rights.

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

`[out] pDisablePriorityBoost`

A pointer to a variable that receives the priority boost control state. A value of TRUE indicates that dynamic boosting is disabled. A value of FALSE indicates normal behavior.

Return value

If the function succeeds, the return value is nonzero. In that case, the variable pointed to by the *pDisablePriorityBoost* parameter receives the priority boost control state.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Requirements

Expand table

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

See also

OpenThread

Priority Boosts

Process and Thread Functions

Scheduling Priorities

SetThreadPriorityBoost

Threads

GetThreadSelectedCpuSets function (processthreadsapi.h)

Article08/23/2022

Returns the explicit CPU Set assignment of the specified thread, if any assignment was set using the SetThreadSelectedCpuSets API. If no explicit assignment is set, **RequiredIdCount** is set to 0 and the function returns TRUE.

Syntax

C++

```
BOOL GetThreadSelectedCpuSets(
  HANDLE Thread,
  PULONG CpuSetIds,
  ULONG  CpuSetIdCount,
  PULONG RequiredIdCount
);
```

Parameters

`Thread`

Specifies the thread for which to query the selected CPU Sets. This handle must have the THREAD_QUERY_LIMITED_INFORMATION access right. The value returned by GetCurrentThread can also be specified here.

`CpuSetIds`

Specifies an optional buffer to retrieve the list of CPU Set identifiers.

`CpuSetIdCount`

Specifies the capacity of the buffer specified in **CpuSetIds**. If the buffer is NULL, this must be 0.

`RequiredIdCount`

Specifies the required capacity of the buffer to hold the entire list of thread selected CPU Sets. On successful return, this specifies the number of IDs filled into the buffer.

Return value

This API returns TRUE on success. If the buffer is not large enough, the **GetLastError** value is ERROR_INSUFFICIENT_BUFFER. This API cannot fail when passed valid parameters and the return buffer is large enough.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	processthreadsapi.h
DLL	kernel32.dll

Feedback

Was this page helpful? Yes No

GetThreadTimes function (processthreadsapi.h)

11/01/2022

Retrieves timing information for the specified thread.

Syntax

```C++
BOOL GetThreadTimes(
  [in]  HANDLE     hThread,
  [out] LPFILETIME lpCreationTime,
  [out] LPFILETIME lpExitTime,
  [out] LPFILETIME lpKernelTime,
  [out] LPFILETIME lpUserTime
);
```

Parameters

`[in] hThread`

A handle to the thread whose timing information is sought. The handle must have the **THREAD_QUERY_INFORMATION** or **THREAD_QUERY_LIMITED_INFORMATION** access right. For more information, see Thread Security and Access Rights.

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

`[out] lpCreationTime`

A pointer to a FILETIME structure that receives the creation time of the thread.

`[out] lpExitTime`

A pointer to a FILETIME structure that receives the exit time of the thread. If the thread has not exited, the content of this structure is undefined.

`[out] lpKernelTime`

A pointer to a FILETIME structure that receives the amount of time that the thread has executed in kernel mode.

`[out] lpUserTime`

A pointer to a FILETIME structure that receives the amount of time that the thread has executed in user mode.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

All times are expressed using FILETIME data structures. Such a structure contains two 32-bit values that combine to form a 64-bit count of 100-nanosecond time units.

Thread creation and exit times are points in time expressed as the amount of time that has elapsed since midnight on January 1, 1601 at Greenwich, England. There are several functions that an application can use to convert such values to more generally useful forms; see Time Functions.

Thread kernel mode and user mode times are amounts of time. For example, if a thread has spent one second in kernel mode, this function will fill the FILETIME structure specified by *lpKernelTime* with a 64-bit value of ten million. That is the number of 100-nanosecond units in one second.

To retrieve the number of CPU clock cycles used by the threads, use the QueryThreadCycleTime function.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows

Requirement	Value
Header	processthreadsapi.h (include Windows.h on Windows Server 2003, Windows Vista, Windows 7, Windows Server 2008 Windows Server 2008 R2)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

FILETIME

FileTimeToDosDateTime

FileTimeToLocalFileTime

FileTimeToSystemTime

OpenThread

Process and Thread Functions

Threads

GetUmsCompletionListEvent function (winbase.h)

Article03/18/2022

Retrieves a handle to the event associated with the specified user-mode scheduling (UMS) completion list.

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

Syntax

```C++
BOOL GetUmsCompletionListEvent(
  [in]      PUMS_COMPLETION_LIST UmsCompletionList,
  [in, out] PHANDLE              UmsCompletionEvent
);
```

Parameters

`[in] UmsCompletionList`

A pointer to a UMS completion list. The CreateUmsCompletionList function provides this pointer.

`[in, out] UmsCompletionEvent`

A pointer to a HANDLE variable. On output, the *UmsCompletionEvent* parameter is set to a handle to the event associated with the specified completion list.

Return value

If the function succeeds, it returns a nonzero value.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The system signals a UMS completion list event when the system queues items to an empty completion list. A completion list event handle can be used with any wait function that takes a handle to an event. When the event is signaled, an application typically calls DequeueUmsCompletionListItems to retrieve the contents of the completion list.

The event handle remains valid until its completion list is deleted. Do not use the event handle to wait on a completion list that has been deleted or is in the process of being deleted.

When the handle is no longer needed, use the CloseHandle function to close the handle.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 (64-bit only) [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
API set	api-ms-win-core-ums-l1-1-0 (introduced in Windows 7)

See also

CreateUmsCompletionList

DequeueUmsCompletionListItems

Wait Functions

GetUmsSystemThreadInformation function (winbase.h)

Article01/24/2024

Queries whether the specified thread is a UMS scheduler thread, a UMS worker thread, or a non-UMS thread.

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

Syntax

```C++
BOOL GetUmsSystemThreadInformation(
  [in]      HANDLE                          ThreadHandle,
  [in, out] PUMS_SYSTEM_THREAD_INFORMATION SystemThreadInfo
);
```

Parameters

`[in] ThreadHandle`

A handle to a thread. The thread handle must have the THREAD_QUERY_INFORMATION access right. For more information, see Thread Security and Access Rights.

`[in, out] SystemThreadInfo`

A pointer to a UMS_SYSTEM_THREAD_INFORMATION structure that receives information about the specified thread.

The caller must initialize the `UmsVersion` member before calling.

Return value

If the function fails, the return value is zero.

To get extended error information, call GetLastError.

Remarks

The **GetUmsSystemThreadInformation** function is intended for use in debuggers, troubleshooting tools, and profiling applications. For example, thread-isolated tracing or single-stepping through instructions might involve suspending all other threads in the process. However, if the thread to be traced is a UMS worker thread, suspending UMS scheduler threads might cause a deadlock because a UMS worker thread requires the intervention of a UMS scheduler thread in order to run. A debugger can call **GetUmsSystemThreadInformation** for each thread that it might suspend to determine the kind of thread, and then suspend it or not as needed for the code being debugged.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 with SP1 [desktop apps only],Windows 7 (64-bit only) and Windows Server 2008 R2 with KB977165 installed
Minimum supported server	Windows Server 2008 R2 with SP1 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
API set	api-ms-win-core-ums-l1-1-0 (introduced in Windows 7)

InitializeProcThreadAttributeList function (processthreadsapi.h)

11/01/2022

Initializes the specified list of attributes for process and thread creation.

Syntax

```C++
BOOL InitializeProcThreadAttributeList(
  [out, optional] LPPROC_THREAD_ATTRIBUTE_LIST lpAttributeList,
  [in]            DWORD                        dwAttributeCount,
                  DWORD                        dwFlags,
  [in, out]       PSIZE_T                      lpSize
);
```

Parameters

`[out, optional] lpAttributeList`

The attribute list. This parameter can be NULL to determine the buffer size required to support the specified number of attributes.

`[in] dwAttributeCount`

The count of attributes to be added to the list.

`dwFlags`

This parameter is reserved and must be zero.

`[in, out] lpSize`

If *lpAttributeList* is not NULL, this parameter specifies the size in bytes of the *lpAttributeList* buffer on input. On output, this parameter receives the size in bytes of the initialized attribute list.

If *lpAttributeList* is NULL, this parameter receives the required buffer size in bytes.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

First, call this function with the *dwAttributeCount* parameter set to the maximum number of attributes you will be using and the *lpAttributeList* to NULL. The function returns the required buffer size in bytes in the *lpSize* parameter.

> **Note** This initial call will return an error by design. This is expected behavior.

Allocate enough space for the data in the *lpAttributeList* buffer and call the function again to initialize the buffer.

To add attributes to the list, call the UpdateProcThreadAttribute function. To specify these attributes when creating a process, specify EXTENDED_STARTUPINFO_PRESENT in the *dwCreationFlag* parameter and a STARTUPINFOEX structure in the *lpStartupInfo* parameter. Note that you can specify the same **STARTUPINFOEX** structure to multiple child processes.

When you have finished using the list, call the DeleteProcThreadAttributeList function.

Requirements

Expand table

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

See also

DeleteProcThreadAttributeList

Process and Thread Functions

UpdateProcThreadAttribute

InitializeThreadpoolEnvironment function (winbase.h)

02/22/2024

Initializes a callback environment.

Syntax

```C++
VOID InitializeThreadpoolEnvironment(
  [out] PTP_CALLBACK_ENVIRON pcbe
);
```

Parameters

`[out] pcbe`

A **TP_CALLBACK_ENVIRON** structure that defines a callback environment.

Return value

None

Remarks

By default, a callback executes in the default thread pool for the process. No cleanup group is associated with the callback environment, the caller is responsible for keeping the callback's DLL loaded while there are outstanding callbacks, and the callback is expected to run in a reasonable amount of time for the application.

Create a callback environment if you plan to call one of the following functions to modify the environment:

- SetThreadpoolCallbackCleanupGroup
- SetThreadpoolCallbackLibrary
- SetThreadpoolCallbackPool
- SetThreadpoolCallbackPriority
- SetThreadpoolCallbackRunsLong

To use the default callback environment, set the optional callback environment parameter to NULL when calling one of the following functions:

- CreateThreadpoolIo
- CreateThreadpoolTimer
- CreateThreadpoolWait
- CreateThreadpoolWork
- TrySubmitThreadpoolCallback

The **InitializeThreadpoolEnvironment** function is implemented as an inline function.

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

Examples

For an example, see Using the Thread Pool Functions.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps | UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)

See also

DestroyThreadpoolEnvironment

SetThreadpoolCallbackCleanupGroup

SetThreadpoolCallbackLibrary

SetThreadpoolCallbackPool

SetThreadpoolCallbackRunsLong

Thread Pools

IoCompletionCallback callback function

Article04/23/2018

Applications implement this callback if they call the StartThreadpoolIo function to start a worker thread for the I/O completion object.

The **PTP_WIN32_IO_CALLBACK** type defines a pointer to this callback function. *IoCompletionCallback* is a placeholder for the application-defined function name.

Syntax

```c++
VOID CALLBACK IoCompletionCallback(
  _Inout_     PTP_CALLBACK_INSTANCE Instance,
  _Inout_opt_ PVOID                 Context,
  _Inout_opt_ PVOID                 Overlapped,
  _In_        ULONG                 IoResult,
  _In_        ULONG_PTR             NumberOfBytesTransferred,
  _Inout_     PTP_IO                Io
);
```

Parameters

- *Instance* [in, out]
  A **TP_CALLBACK_INSTANCE** structure that defines the callback instance. Applications do not modify the members of this structure.

  This structure can be passed to one of the following functions:
  - CallbackMayRunLong
  - DisassociateCurrentThreadFromCallback
  - FreeLibraryWhenCallbackReturns
  - LeaveCriticalSectionWhenCallbackReturns
  - ReleaseMutexWhenCallbackReturns
  - ReleaseSemaphoreWhenCallbackReturns
  - SetEventWhenCallbackReturns

- *Context* [in, out, optional]
  The application-defined data.

- *Overlapped* [in, out, optional]
  A pointer to a variable that receives the address of the OVERLAPPED structure that was specified when the completed I/O operation was started.

- *IoResult* [in]
  The result of the I/O operation. If the I/O is successful, this parameter is NO_ERROR. Otherwise, this parameter is one of the system error codes.

- *NumberOfBytesTransferred* [in]
  The number of bytes transferred during the I/O operation that has completed.

- *Io* [in, out]
  A **TP_IO** structure that defines the I/O completion object that generated the callback.

Return value

This function does not return a value.

Remarks

If the file handle bound to the I/O completion object has the notification mode FILE_SKIP_COMPLETION_PORT_ON_SUCCESS and an asynchronous I/O operation returns immediately with success, the I/O completion callback function is not called and threadpool I/O notifications must be canceled. For more information, see **CancelThreadpoolIo**.

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

Requirements

Expand table

Minimum supported client	Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps | UWP apps]
Header	WinBase.h on Windows Vista, Windows 7, Windows Server 2008 and Windows Server 2008 R2 (include Windows.h); Threadpoolapiset.h on Windows 8 and Windows Server 2012

See also

StartThreadpoolIo

Thread Pools

IsImmersiveProcess function (winuser.h)

Article02/22/2024

Determines whether the process belongs to a Windows Store app.

Syntax

```
C++

BOOL IsImmersiveProcess(
  [in] HANDLE hProcess
);
```

Parameters

`[in] hProcess`

Target process handle.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]
Target Platform	Windows
Header	winuser.h
Library	User32.lib
DLL	User32.dll

IsProcessInJob function (jobapi.h)

Article02/22/2024

Determines whether the process is running in the specified job.

Syntax

```
C++

BOOL IsProcessInJob(
  [in]           HANDLE ProcessHandle,
  [in, optional] HANDLE JobHandle,
  [out]          PBOOL  Result
);
```

Parameters

`[in] ProcessHandle`

A handle to the process to be tested. The handle must have the PROCESS_QUERY_INFORMATION or PROCESS_QUERY_LIMITED_INFORMATION access right. For more information, see Process Security and Access Rights.

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

`[in, optional] JobHandle`

A handle to the job. If this parameter is NULL, the function tests if the process is running under any job.

If this parameter is not NULL, the handle must have the JOB_OBJECT_QUERY access right. For more information, see Job Object Security and Access Rights.

`[out] Result`

A pointer to a value that receives TRUE if the process is running in the job, and FALSE otherwise.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

An application cannot obtain a handle to the job object in which it is running unless it has the name of the job object. However, an application can call the QueryInformationJobObject function with NULL to obtain information about the job object.

To compile an application that uses this function, define _WIN32_WINNT as 0x0501 or later. For more information, see Using the Windows Headers.

Requirements

Expand table

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

See also

AssignProcessToJobObject

Job Objects

Process and Thread Functions

QueryInformationJobObject

IsProcessCritical function (processthreadsapi.h)

11/01/2022

Determines whether the specified process is considered critical.

Syntax

```C++
BOOL IsProcessCritical(
  [in]  HANDLE hProcess,
  [out] PBOOL  Critical
);
```

Parameters

`[in] hProcess`

A handle to the process to query. The process must have been opened with **PROCESS_QUERY_LIMITED_INFORMATION** access.

`[out] Critical`

A pointer to the **BOOL** value this function will use to indicate whether the process is considered critical.

Return value

This routine returns FALSE on failure. Any other value indicates success. Call GetLastError to query for the specific error reason on failure.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8.1 [desktop apps only]

Requirement	Value
Minimum supported server	Windows Server 2012 R2 [desktop apps only]
Target Platform	Windows
Header	processthreadsapi.h
Library	Kernel32.lib
DLL	kernel32.dll

See also

HRESULT_FROM_WIN32

IsThreadAFiber function (fibersapi.h)

02/22/2024

Determines whether the current thread is a fiber.

Syntax

```
C++

BOOL IsThreadAFiber();
```

Return value

The function returns **TRUE** if the thread is a fiber and **FALSE** otherwise.

Remarks

To compile an application that uses this function, define _WIN32_WINNT as 0x0600 or later. For more information, see Using the Windows Headers.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps | UWP apps]
Target Platform	Windows
Header	fibersapi.h
Library	Kernel32.lib
DLL	Kernel32.dll

See also

ConvertThreadToFiber

ConvertThreadToFiberEx

Fibers

Process and Thread Functions

IsThreadpoolTimerSet function (threadpoolapiset.h)

Article02/22/2024

Determines whether the specified timer object is currently set.

Syntax

```C++
BOOL IsThreadpoolTimerSet(
  [in, out] PTP_TIMER pti
);
```

Parameters

`[in, out] pti`

A pointer to a **TP_TIMER** structure that defines the timer object. The CreateThreadpoolTimer function returns this pointer.

Return value

The return value is TRUE if the timer is set; otherwise, the return value is FALSE.

Remarks

A timer is considered to be set if the most recent call to SetThreadpoolTimer or SetThreadpoolTimerEx function passed a non-null value for *pftDueTime*.

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

Requirements

Expand table

Requirement	Value
Minimum supported	Windows Vista [desktop apps | UWP apps]

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

See also

CloseThreadpoolTimer

CreateThreadpoolTimer

SetThreadpoolTimer

Thread Pools

WaitForThreadpoolTimerCallbacks

IsWow64Message function (winuser.h)

Article06/29/2021

Determines whether the last message read from the current thread's queue originated from a WOW64 process.

Syntax

```
C++

BOOL IsWow64Message();
```

Return value

The function returns TRUE if the last message read from the current thread's queue originated from a WOW64 process, and FALSE otherwise.

Remarks

This function is useful to helping you develop 64-bit native applications that can receive private messages sent from 32-bit client applications, if the messages are associated with data structures that contain pointer-dependent data. In these situations, you can call this function in your 64-bit native application to determine if the message originated from a WOW64 process and then thunk the message appropriately.

Examples

For compatibility with operating systems that do not support this function, call GetProcAddress to detect whether **IsWow64Message** is implemented in User32.dll. If **GetProcAddress** succeeds, it is safe to call this function. Otherwise, WOW64 is not present. Note that this technique is not a reliable way to detect whether the operating system is a 64-bit version of Windows because the User32.dll in current versions of 32-bit Windows also contains this function.

```
C++

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

typedef BOOL (WINAPI *LPFN_ISWOW64MESSAGE) (void);
```

```
LPFN_ISWOW64MESSAGE fnIsWow64Message;

BOOL IsWow64Msg()
{
    // IsWow64Message is not available on all supported versions of Windows
    // Use LoadLibrary to ensure that the DLL containing the function is loaded
    // and GetProcAddress to get a pointer to the function if available.

    fnIsWow64Message = (LPFN_ISWOW64MESSAGE) GetProcAddress(
        LoadLibrary(TEXT("user32")), "IsWow64Message");

    if (NULL != fnIsWow64Message)
    {
        return (fnIsWow64Message());
    }
    else return FALSE;
}

int main( void )
{
    if(IsWow64Msg())
    {
        _tprintf(TEXT("The last message was from a 32-bit process.\n"));
    }
    else if (NULL == fnIsWow64Message )
    {
        _tprintf(TEXT("The IsWow64Message function is not available (%d).\n"),
GetLastError());
    }
    else
    {
        _tprintf(TEXT("The last message was from a 64-bit process.\n"));
    }
    return 0;
}
```

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista, Windows XP with SP2 [desktop apps only]
Minimum supported server	Windows Server 2008, Windows Server 2003 with SP1 [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)

Requirement	Value
Library	User32.lib
DLL	User32.dll

See also

GetNativeSystemInfo

IsWow64Process

WOW64

IsWow64Process function (wow64apiset.h)

01/27/2025

Determines whether the specified process is running under WOW64 or an Intel64 of x64 processor.

Syntax

```C++
BOOL IsWow64Process(
  [in]  HANDLE hProcess,
  [out] PBOOL  Wow64Process
);
```

Parameters

`[in] hProcess`

A handle to the process. The handle must have the PROCESS_QUERY_INFORMATION or PROCESS_QUERY_LIMITED_INFORMATION access right. For more information, see Process Security and Access Rights.

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

`[out] Wow64Process`

A pointer to a value that is set to TRUE if the process is running under WOW64 on an Intel64, x64, or ARM64 processor. If the process is running under 32-bit Windows, the value is set to FALSE. If the process is a 32-bit application running under 64-bit Windows 10 on ARM, the value is set to FALSE. If the process is a 64-bit application running under 64-bit Windows, the value is also set to FALSE.

Return value

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

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

Applications should use IsWow64Process2 instead of **IsWow64Process** to determine if a process is running under WOW. **IsWow64Process2** removes the ambiguity inherent to multiple WOW environments by explicitly returning both the architecture of the host and guest for a given process. Applications can use this information to reliably identify situations such as running under emulation on ARM64. To compile an application that uses this function, define _WIN32_WINNT as 0x0501 or later. For more information, see Using the Windows Headers.

Examples

For compatibility with operating systems that do not support this function, call GetProcAddress to detect whether **IsWow64Process** is implemented in Kernel32.dll. If **GetProcAddress** succeeds, it is safe to call this function. Otherwise, WOW64 is not present. Note that this technique is not a reliable way to detect whether the operating system is a 64-bit version of Windows because the Kernel32.dll in current versions of 32-bit Windows also contains this function.

C++

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

typedef BOOL (WINAPI *LPFN_ISWOW64PROCESS) (HANDLE, PBOOL);

LPFN_ISWOW64PROCESS fnIsWow64Process;

BOOL IsWow64()
{
    BOOL bIsWow64 = FALSE;

    //IsWow64Process is not available on all supported versions of Windows.
    //Use GetModuleHandle to get a handle to the DLL that contains the function
    //and GetProcAddress to get a pointer to the function if available.

    fnIsWow64Process = (LPFN_ISWOW64PROCESS) GetProcAddress(
        GetModuleHandle(TEXT("kernel32")),"IsWow64Process");

    if(NULL != fnIsWow64Process)
    {
        if (!fnIsWow64Process(GetCurrentProcess(),&bIsWow64))
        {
            //handle error
        }
    }
    return bIsWow64;
}
```

```
int main( void )
{
    if(IsWow64())
        _tprintf(TEXT("The process is running under WOW64.\n"));
    else
        _tprintf(TEXT("The process is not running under WOW64.\n"));

    return 0;
}
```

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista, Windows XP with SP2 [desktop apps | UWP apps]
Minimum supported server	Windows Server 2008, Windows Server 2003 with SP1 [desktop apps | UWP apps]
Target Platform	Windows
Header	wow64apiset.h (include Windows.h on Windows Server 2003, Windows Vista, Windows 7, Windows Server 2008 Windows Server 2008 R2)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

GetNativeSystemInfo

IsWow64Message

WOW64

IsWow64Process2 function (wow64apiset.h)

02/22/2024

Determines whether the specified process is running under WOW64; also returns additional machine process and architecture information.

Syntax

```C++
BOOL IsWow64Process2(
  [in]            HANDLE hProcess,
  [out]           USHORT *pProcessMachine,
  [out, optional] USHORT *pNativeMachine
);
```

Parameters

`[in] hProcess`

A handle to the process. The handle must have the **PROCESS_QUERY_INFORMATION** or **PROCESS_QUERY_LIMITED_INFORMATION** access right. For more information, see Process Security and Access Rights.

`[out] pProcessMachine`

On success, returns a pointer to an IMAGE_FILE_MACHINE_* value. The value will be **IMAGE_FILE_MACHINE_UNKNOWN** if the target process is not a WOW64 process; otherwise, it will identify the type of WoW process.

`[out, optional] pNativeMachine`

On success, returns a pointer to a possible IMAGE_FILE_MACHINE_* value identifying the native architecture of host system.

Return value

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

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

IsWow64Process2 provides an improved direct replacement for IsWow64Process. In addition to determining if the specified process is running under WOW64, **IsWow64Process2** returns the following information:

- Whether the target process, specified by *hProcess*, is running under Wow or not.
- The architecture of the target process.
- Optionally, the architecture of the host system.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 10, version 1709 [desktop apps | UWP apps]
Minimum supported server	Windows Server 2016, version 1709 [desktop apps | UWP apps]
Target Platform	Windows
Header	wow64apiset.h
Library	Kernel32.lib
DLL	Kernel32.dll

LeaveCriticalSectionWhenCallbackReturns function (threadpoolapiset.h)

09/23/2022

Specifies the critical section that the thread pool will release when the current callback completes.

Syntax

```C++
VOID LeaveCriticalSectionWhenCallbackReturns(
  [in, out] PTP_CALLBACK_INSTANCE pci,
  [in, out] PCRITICAL_SECTION     pcs
);
```

Parameters

`[in, out] pci`

A pointer to a **TP_CALLBACK_INSTANCE** structure that defines the callback instance. The pointer is passed to the callback function.

`[in, out] pcs`

The critical section.

Return value

None

Remarks

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

Requirements

Expand table

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

See also

CallbackMayRunLong

DisassociateCurrentThreadFromCallback

FreeLibraryWhenCallbackReturns

ReleaseMutexWhenCallbackReturns

ReleaseSemaphoreWhenCallbackReturns

SetEventWhenCallbackReturns

Thread Pools

TrySubmitThreadpoolCallback

NeedCurrentDirectoryForExePathA function (processenv.h)

Article11/20/2024

Determines whether the current directory should be included in the search path for the specified executable.

Syntax

```
C++

BOOL NeedCurrentDirectoryForExePathA(
  [in] LPCSTR ExeName
);
```

Parameters

`[in] ExeName`

The name of the executable file.

Return value

If the current directory should be part of the search path, the return value is TRUE. Otherwise, the return value is FALSE.

Remarks

This function should only be called in instances where the caller must explicitly resolve a relative executable name to an absolute name. If CreateProcess is called with a relative executable name, it will automatically search for the executable, calling this function to determine the search path.

Most system functions perform their own path resolution, therefore, this function should only be called if you are attempting to resolve a search path for the specified executable based on the current directory.

The value of the NoDefaultCurrentDirectoryInExePath environment variable determines the value this function returns. However, you should call this function rather than checking the environment variable directly, as the registry location of this environment variable can change.

If the value of the *ExeName* parameter contains a backslash (\), this function will always return TRUE. If it does not contain a backslash, the existence of the NoDefaultCurrentDirectoryInExePath environment variable is checked, and not its value.

An example of an instance when this function should be called instead of relying on the default search path resolution algorithm in CreateProcess is the "cmd.exe" executable. It calls this function to determine the command search path because it does its own path resolution before calling **CreateProcess**. If this function returns TRUE, cmd.exe uses the path ".;%PATH%" for the executable search. If it returns FALSE, cmd.exe uses the path "%PATH%" for the search.

> **Note**
>
> The processenv.h header defines NeedCurrentDirectoryForExePath as an alias that automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Expand table

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

See also

CreateProcess

NtGetCurrentProcessorNumber function

Article • 03/22/2021

[**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

```
C++

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

Feedback

Was this page helpful? Yes No

Get help at Microsoft Q&A

NtQueryInformationProcess function (winternl.h)

Article02/22/2024

[**NtQueryInformationProcess** 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

```C++
__kernel_entry NTSTATUS NtQueryInformationProcess(
  [in]            HANDLE           ProcessHandle,
  [in]            PROCESSINFOCLASS ProcessInformationClass,
  [out]           PVOID            ProcessInformation,
  [in]            ULONG            ProcessInformationLength,
  [out, optional] PULONG           ReturnLength
);
```

Parameters

`[in] ProcessHandle`

A handle to the process for which information is to be retrieved.

`[in] ProcessInformationClass`

The type of process information to be retrieved. This parameter can be one of the following values from the **PROCESSINFOCLASS** enumeration.

Expand table

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. 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. 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). Use the IsWow64Process2 function to obtain this information.
ProcessImageFileName 27	Retrieves a **UNICODE_STRING** value containing the name of the image file for the process. Use the QueryFullProcessImageName or GetProcessImageFileName function to obtain this information.
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.
ProcessTelemetryIdInformation 64	Retrieves a **PROCESS_TELEMETRY_ID_INFORMATION_TYPE** value that contains metadata about a process.
ProcessSubsystemInformation 75	Retrieves a **SUBSYSTEM_INFORMATION_TYPE** value indicating the subsystem type of the process. The buffer pointed to by the *ProcessInformation* parameter should be large enough to hold a single SUBSYSTEM_INFORMATION_TYPE enumeration.

`[out] ProcessInformation`

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 data type 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:

syntax

```
typedef struct _PROCESS_BASIC_INFORMATION {
    NTSTATUS ExitStatus;
    PPEB PebBaseAddress;
    ULONG_PTR AffinityMask;
    KPRIORITY BasePriority;
    ULONG_PTR UniqueProcessId;
    ULONG_PTR InheritedFromUniqueProcessId;
} PROCESS_BASIC_INFORMATION;
```

Expand table

Field	Meaning
ExitStatus	Contains the same value that **GetExitCodeProcess** returns. 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** returns for the `lpProcessAffinityMask` parameter.
BasePriority	Contains the process priority as described in Scheduling Priorities.
UniqueProcessId	Can be cast to a **DWORD** and contains a unique identifier for this process. We recommend using 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, the process is not running in a WOW64 environment.

Use the IsWow64Process2 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.

`[in] ProcessInformationLength`

The size of the buffer pointed to by the *ProcessInformation* parameter, in bytes.

`[out, optional] ReturnLength`

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 (if the buffer was too small, this is the minimum size of buffer needed to receive the information successfully).

Return value

The function 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. See Logging Errors for more details.

Remarks

The **NtQueryInformationProcess** 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 **NtQueryInformationProcess**, 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

Expand table

Requirement	Value
Target Platform	Windows
Header	winternl.h
Library	ntdll.lib
DLL	ntdll.dll

See also

CheckRemoteDebuggerPresent

GetProcessId

IsDebuggerPresent

IsWow64Process

IsWow64Process2

NtQueryInformationThread function (winternl.h)

Article07/27/2022

[**NtQueryInformationThread** 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 thread.

Syntax

```C++
__kernel_entry NTSTATUS NtQueryInformationThread(
  [in]            HANDLE          ThreadHandle,
  [in]            THREADINFOCLASS ThreadInformationClass,
  [in, out]       PVOID           ThreadInformation,
  [in]            ULONG           ThreadInformationLength,
  [out, optional] PULONG          ReturnLength
);
```

Parameters

`[in] ThreadHandle`

A handle to the thread about which information is being requested.

`[in] ThreadInformationClass`

If this parameter is the **ThreadIsIoPending** value of the **THREADINFOCLASS** enumeration, the function determines whether the thread has any I/O operations pending.

Use the public function GetThreadIOPendingFlag instead to obtain this information.

If this parameter is the **ThreadQuerySetWin32StartAddress** value of the **THREADINFOCLASS** enumeration, the function returns the start address of the thread. Note that on versions of Windows prior to Windows Vista, the returned start address is only reliable before the thread starts running.

If this parameter is the **ThreadSubsystemInformation** value of the **THREADINFOCLASS** enumeration, the function retrieves a SUBSYSTEM_INFORMATION_TYPE value indicating the subsystem type of the thread. The buffer pointed to by the *ThreadInformation* parameter should be large enough to hold a single **SUBSYSTEM_INFORMATION_TYPE** enumeration.

`[in, out] ThreadInformation`

A pointer to a buffer in which the function writes the requested information. If **ThreadIsIoPending** is specified for the *ThreadInformationClass* parameter, this buffer must be large enough to hold a **ULONG** value, which indicates whether the specified thread has I/O requests pending. If this value is equal to zero, then there are no I/O operations pending; otherwise, if the value is nonzero, then the thread does have I/O operations pending.

Use the public function GetThreadIOPendingFlag instead to obtain this information.

If **ThreadQuerySetWin32StartAddress** is specified for the *ThreadInformationClass* parameter, this buffer must be large enough to hold a PVOID value, which is the start address of the thread.

`[in] ThreadInformationLength`

The size of the buffer pointed to by the *ThreadInformation* parameter, in bytes.

`[out, optional] ReturnLength`

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 *ThreadInformation* parameter, but if the buffer was too small, this is the minimum size of buffer required 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 **NtQueryInformationThread** function is 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 the public function previously mentioned instead.

If you do use **NtQueryInformationThread**, 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

Expand table

Requirement	Value
Target Platform	Windows
Header	winternl.h
Library	ntdll.lib
DLL	ntdll.dll

See also

GetThreadIOPendingFlag

OpenJobObjectA function (winbase.h)

Article04/26/2023

Opens an existing job object.

Syntax

```
C++

HANDLE OpenJobObjectA(
  [in] DWORD  dwDesiredAccess,
  [in] BOOL   bInheritHandle,
  [in] LPCSTR lpName
);
```

Parameters

`[in] dwDesiredAccess`

The access to the job object. This parameter can be one or more of the job object access rights. This access right is checked against any security descriptor for the object.

`[in] bInheritHandle`

If this value is TRUE, processes created by this process will inherit the handle. Otherwise, the processes do not inherit this handle.

`[in] lpName`

The name of the job to be opened. Name comparisons are case sensitive.

This function can open objects in a private namespace. For more information, see Object Namespaces.

Terminal Services: The name can have a "Global" or "Local" prefix to explicitly open the object in the global or session namespace. The remainder of the name can contain any character except the backslash character (\). For more information, see Kernel Object Namespaces.

Return value

If the function succeeds, the return value is a handle to the job. The handle provides the requested access to the job.

If the function fails, the return value is **NULL**. To get extended error information, call GetLastError.

An error of ERROR_FILE_NOT_FOUND indicates that the job specified in *lpName* does not exist.

Remarks

To associate a process with a job, use the AssignProcessToJobObject function.

To compile an application that uses this function, define **_WIN32_WINNT** as 0x0500 or later. For more information, see Using the Windows Headers.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h, Jobapi2.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

AssignProcessToJobObject

Job Objects

Process and Thread Functions

OpenProcess function (processthreadsapi.h)

11/01/2022

Opens an existing local process object.

Syntax

```C++
HANDLE OpenProcess(
  [in] DWORD dwDesiredAccess,
  [in] BOOL  bInheritHandle,
  [in] DWORD dwProcessId
);
```

Parameters

`[in] dwDesiredAccess`

The access to the process object. This access right is checked against the security descriptor for the process. This parameter can be one or more of the process access rights.

If the caller has enabled the SeDebugPrivilege privilege, the requested access is granted regardless of the contents of the security descriptor.

`[in] bInheritHandle`

If this value is TRUE, processes created by this process will inherit the handle. Otherwise, the processes do not inherit this handle.

`[in] dwProcessId`

The identifier of the local process to be opened.

If the specified process is the System Idle Process (0x00000000), the function fails and the last error code is `ERROR_INVALID_PARAMETER`. If the specified process is the System process or one of the Client Server Run-Time Subsystem (CSRSS) processes, this function fails and the last error code is `ERROR_ACCESS_DENIED` because their access restrictions prevent user-level code from opening them.

If you are using GetCurrentProcessId as an argument to this function, consider using GetCurrentProcess instead of OpenProcess, for improved performance.

Return value

If the function succeeds, the return value is an open handle to the specified process.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Remarks

To open a handle to another local 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 **OpenProcess** function can be used in any function that requires a handle to a process, such as the wait functions, provided the appropriate access rights were requested.

When you are finished with the handle, be sure to close it using the CloseHandle function.

Examples

For an example, see Taking a Snapshot and Viewing Processes.

Requirements

Expand table

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

Requirement	Value
DLL	Kernel32.dll

See also

AssignProcessToJobObject

CloseHandle

CreateProcess

CreateRemoteThread

DuplicateHandle

GetCurrentProcess

GetCurrentProcessId

GetExitCodeProcess

GetModuleFileNameEx

GetPriorityClass

Process and Thread Functions

Processes

ReadProcessMemory

SetPriorityClass

SetProcessWorkingSetSize

TerminateProcess

VirtualProtectEx

WriteProcessMemory

OpenThread function (processthreadsapi.h)

02/22/2024

Opens an existing thread object.

Syntax

```C++
HANDLE OpenThread(
  [in] DWORD dwDesiredAccess,
  [in] BOOL  bInheritHandle,
  [in] DWORD dwThreadId
);
```

Parameters

`[in] dwDesiredAccess`

The access to the thread object. This access right is checked against the security descriptor for the thread. This parameter can be one or more of the thread access rights.

If the caller has enabled the SeDebugPrivilege privilege, the requested access is granted regardless of the contents of the security descriptor.

`[in] bInheritHandle`

If this value is TRUE, processes created by this process will inherit the handle. Otherwise, the processes do not inherit this handle.

`[in] dwThreadId`

The identifier of the thread to be opened.

Return value

If the function succeeds, the return value is an open handle to the specified thread.

If the function fails, the return value is NULL. To get extended error information, call GetLastError.

Remarks

The handle returned by **OpenThread** can be used in any function that requires a handle to a thread, such as the wait functions, provided you requested the appropriate access rights. The handle is granted access to the thread object only to the extent it was specified in the *dwDesiredAccess* parameter.

When you are finished with the handle, be sure to close it by using the CloseHandle function.

Requirements

Expand table

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

See also

CloseHandle

GetExitCodeThread

GetThreadContext

Process and Thread Functions

ResumeThread

SetThreadContext

SetTokenInformation

SuspendThread

TerminateThread

Threads

QueryFullProcessImageNameA function (winbase.h)

Article02/09/2023

Retrieves the full name of the executable image for the specified process.

Syntax

```
C++

BOOL QueryFullProcessImageNameA(
  [in]      HANDLE hProcess,
  [in]      DWORD  dwFlags,
  [out]     LPSTR  lpExeName,
  [in, out] PDWORD lpdwSize
);
```

Parameters

`[in] hProcess`

A handle to the process. This handle must be created with the PROCESS_QUERY_INFORMATION or PROCESS_QUERY_LIMITED_INFORMATION access right. For more information, see Process Security and Access Rights.

`[in] dwFlags`

This parameter can be one of the following values.

Expand table

Value	Meaning
0	The name should use the Win32 path format.
PROCESS_NAME_NATIVE 0x00000001	The name should use the native system path format.

`[out] lpExeName`

The path to the executable image. If the function succeeds, this string is null-terminated.

`[in, out] lpdwSize`

On input, specifies the size of the *lpExeName* buffer, in characters. On success, receives the number of characters written to the buffer, not including the null-terminating character.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

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

> **Note**
>
> The winbase.h header defines QueryFullProcessImageName as an alias which automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Expand table

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

See also

GetModuleFileNameEx

GetProcessImageFileName

Process and Thread Functions

QueryIdleProcessorCycleTime function (realtimeapiset.h)

09/26/2024

Retrieves the cycle time for the idle thread of each processor in the system.

On a system with more than 64 processors, this function retrieves the cycle time for the idle thread of each processor in the processor group to which the calling thread is assigned. Use the QueryIdleProcessorCycleTimeEx function to retrieve the cycle time for the idle thread on each logical processor for a specific processor group.

Syntax

```
C++

BOOL QueryIdleProcessorCycleTime(
  [in, out] PULONG   BufferLength,
  [out]     PULONG64 ProcessorIdleCycleTime
);
```

Parameters

`[in, out] BufferLength`

On input, specifies the size of the *ProcessorIdleCycleTime* buffer, in bytes. This buffer is expected to be 8 times the number of processors in the group.

On output, specifies the number of elements written to the buffer. If the buffer size is not sufficient, the function fails and this parameter receives the required length of the buffer.

`[out] ProcessorIdleCycleTime`

The number of CPU clock cycles used by each idle thread. This buffer must be 8 times the number of processors in the system in size.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

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

Requirements

Expand table

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

See also

Processor Groups

QueryIdleProcessorCycleTimeEx

QueryProcessCycleTime

QueryThreadCycleTime

QueryIdleProcessorCycleTimeEx function (realtimeapiset.h)

09/26/2024

Retrieves the accumulated cycle time for the idle thread on each logical processor in the specified processor group.

Syntax

```C++
BOOL QueryIdleProcessorCycleTimeEx(
  [in]      USHORT   Group,
  [in, out] PULONG   BufferLength,
  [out]     PULONG64 ProcessorIdleCycleTime
);
```

Parameters

`[in] Group`

The number of the processor group for which to retrieve the cycle time.

`[in, out] BufferLength`

On input, specifies the size of the *ProcessorIdleCycleTime* buffer, in bytes. This buffer is expected to be 8 times the number of processors in the group.

On output, specifies the number of elements written to the buffer. If the buffer size is not sufficient, the function fails and this parameter receives the required length of the buffer.

`[out] ProcessorIdleCycleTime`

The number of CPU clock cycles used by each idle thread. If this parameter is NULL, the function updates the *BufferLength* parameter with the required length.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, use GetLastError.

Remarks

To compile an application that uses this function, set _WIN32_WINNT >= 0x0601. For more information, see Using the Windows Headers.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	realtimeapiset.h (include Windows.h)
Library	Mincore.lib
DLL	Kernel32.dll

See also

Processor Groups

QueryIdleProcessorCycleTime

QueryInformationJobObject function (jobapi2.h)

Article10/13/2021

Retrieves limit and job state information from the job object.

Syntax

```
C++

BOOL QueryInformationJobObject(
  [in, optional]  HANDLE             hJob,
  [in]            JOBOBJECTINFOCLASS JobObjectInformationClass,
  [out]           LPVOID             lpJobObjectInformation,
  [in]            DWORD              cbJobObjectInformationLength,
  [out, optional] LPDWORD            lpReturnLength
);
```

Parameters

`[in, optional] hJob`

A handle to the job whose information is being queried. The CreateJobObject or OpenJobObject function returns this handle. The handle must have the **JOB_OBJECT_QUERY** access right. For more information, see Job Object Security and Access Rights.

If this value is NULL and the calling process is associated with a job, the job associated with the calling process is used. If the job is nested, the immediate job of the calling process is used.

`[in] JobObjectInformationClass`

The information class for the limits to be queried. This parameter can be one of the following values.

Expand table

Value	Meaning
JobObjectBasicAccountingInformation 1	The *lpJobObjectInfo* parameter is a pointer to a JOBOBJECT_BASIC_ACCOUNTING_INFORMATION structure.

JobObjectBasicAndIoAccountingInformation 8	The *lpJobObjectInfo* parameter is a pointer to a JOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION structure.
JobObjectBasicLimitInformation 2	The *lpJobObjectInfo* parameter is a pointer to a JOBOBJECT_BASIC_LIMIT_INFORMATION structure.
JobObjectBasicProcessIdList 3	The *lpJobObjectInfo* parameter is a pointer to a JOBOBJECT_BASIC_PROCESS_ID_LIST structure.
JobObjectBasicUIRestrictions 4	The *lpJobObjectInfo* parameter is a pointer to a JOBOBJECT_BASIC_UI_RESTRICTIONS structure.
JobObjectCpuRateControlInformation 15	The *lpJobObjectInfo* parameter is a pointer to a JOBOBJECT_CPU_RATE_CONTROL_INFORMATION structure. **Windows 7, Windows Server 2008 R2, Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP:** This flag is not supported.
JobObjectEndOfJobTimeInformation 6	The *lpJobObjectInfo* parameter is a pointer to a JOBOBJECT_END_OF_JOB_TIME_INFORMATION structure.
JobObjectExtendedLimitInformation 9	The *lpJobObjectInfo* parameter is a pointer to a JOBOBJECT_EXTENDED_LIMIT_INFORMATION structure.
JobObjectGroupInformation 11	The *lpJobObjectInfo* parameter is a pointer to a buffer that receives the list of processor groups to which the job is currently assigned. The variable pointed to by the *lpReturnLength* parameter is set to the size of the group data. Divide this value by `sizeof(USHORT)` to determine the number of groups. **Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP:** This flag is not supported.
JobObjectGroupInformationEx 14	The *lpJobObjectInfo* parameter is a pointer to a buffer that receives an array of GROUP_AFFINITY structures that indicate the affinity of the job in the processor groups to which the job is currently assigned. The variable pointed to by the *lpReturnLength* parameter is set to the size of the group affinity data. Divide this value by `sizeof(GROUP_AFFINITY)` to determine the number of groups. **Windows 7, Windows Server 2008 R2, Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP:** This flag is not supported.

JobObjectLimitViolationInformation 13	The *lpJobObjectInfo* parameter is a pointer to a JOBOBJECT_LIMIT_VIOLATION_INFORMATION structure. **Windows 7, Windows Server 2008 R2, Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP:** This flag is not supported.
JobObjectLimitViolationInformation2 34	The *lpJobObjectInfo* parameter is a pointer to a JOBOBJECT_LIMIT_VIOLATION_INFORMATION_2 structure. **Windows 8.1, Windows Server 2012 R2, Windows 8, Windows Server 2012, Windows 7, Windows Server 2008 R2, Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP:** This flag is not supported.
JobObjectNetRateControlInformation 32	The *lpJobObjectInfo* parameter is a pointer to a JOBOBJECT_NET_RATE_CONTROL_INFORMATION structure. **Windows 8.1, Windows Server 2012 R2, Windows 8, Windows Server 2012, Windows 7, Windows Server 2008 R2, Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP:** This flag is not supported.
JobObjectNotificationLimitInformation 12	The *lpJobObjectInfo* parameter is a pointer to a JOBOBJECT_NOTIFICATION_LIMIT_INFORMATION structure. **Windows 7, Windows Server 2008 R2, Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP:** This flag is not supported.
JobObjectNotificationLimitInformation2 33	The *lpJobObjectInfo* parameter is a pointer to a JOBOBJECT_NOTIFICATION_LIMIT_INFORMATION_2 structure. **Windows 8.1, Windows Server 2012 R2, Windows 8, Windows Server 2012, Windows 7, Windows Server 2008 R2, Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP:** This flag is not supported.
JobObjectSecurityLimitInformation 5	This flag is not supported. Applications must set security limits individually for each process. **Windows Server 2003 and Windows XP:** The *lpJobObjectInfo* parameter is a pointer to a JOBOBJECT_SECURITY_LIMIT_INFORMATION structure.

`[out] lpJobObjectInformation`

The limit or job state information. The format of this data depends on the value of the *JobObjectInfoClass* parameter.

`[in] cbJobObjectInformationLength`

The count of the job information being queried, in bytes. This value depends on the value of the *JobObjectInfoClass* parameter.

`[out, optional] lpReturnLength`

A pointer to a variable that receives the length of data written to the structure pointed to by the *lpJobObjectInfo* parameter. Specify **NULL** to not receive this information.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

Use **QueryInformationJobObject** to obtain the current limits and modify them. Use the SetInformationJobObject function to set new limits.

To compile an application that uses this function, define **_WIN32_WINNT** as 0x0500 or later. For more information, see Using the Windows Headers.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	jobapi2.h (include Windows.h)
Library	Kernel32.lib

Requirement	Value
DLL	Kernel32.dll

See also

JOBOBJECT_BASIC_ACCOUNTING_INFORMATION

JOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION

JOBOBJECT_BASIC_LIMIT_INFORMATION

JOBOBJECT_BASIC_PROCESS_ID_LIST

JOBOBJECT_BASIC_UI_RESTRICTIONS

JOBOBJECT_CPU_RATE_CONTROL_INFORMATION

JOBOBJECT_END_OF_JOB_TIME_INFORMATION

JOBOBJECT_EXTENDED_LIMIT_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

Job Objects

Process and Thread Functions

SetInformationJobObject

QueryIoRateControlInformationJobObject function (jobapi2.h)

Article01/16/2025

Windows 10, version 1607, and newer: This function is not supported.

Gets information about the control of the I/O rate for a job object.

Syntax

```C++
DWORD QueryIoRateControlInformationJobObject(
  [in, optional] HANDLE                                hJob,
  [in, optional] PCWSTR                                VolumeName,
  [out]          JOBOBJECT_IO_RATE_CONTROL_INFORMATION **InfoBlocks,
  [out]          ULONG                                 *InfoBlockCount
);
```

Parameters

`[in, optional] hJob`

A handle to the job to query for information. Get this handle from the CreateJobObject or OpenJobObject function. The handle must have the **JOB_OBJECT_QUERY** access right. For more information about access rights, see Job Object Security and Access Rights.

If this value is NULL and the process that calls **QueryIoRateControlInformationJobObject** is associated with a job, the function uses job that is associated with the process. If the job is nested within another job, the function uses the immediate job for the process.

`[in, optional] VolumeName`

The name of the volume to query. If this value is NULL, the function gets the information about I/O rate control for the job for all of the volumes for the system.

`[out] InfoBlocks`

A pointer to array of JOBOBJECT_IO_RATE_CONTROL_INFORMATION structures that contain the information about I/O rate control for the job. Your code must free the memory for this array by calling the FreeMemoryJobObject function with the address of the array.

`[out] InfoBlockCount`

The number of JOBOBJECT_IO_RATE_CONTROL_INFORMATION structures that the function allocated in the array to which the *InfoBlocks* parameter points.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 10 [desktop apps only]
Minimum supported server	Windows Server 2016 [desktop apps only]
Target Platform	Windows
Header	jobapi2.h
Library	Kernel32.lib
DLL	Kernel32.dll

See also

FreeMemoryJobObject

JOBOBJECT_IO_RATE_CONTROL_INFORMATION

SetIoRateControlInformationJobObject

QueryProcessAffinityUpdateMode function (processthreadsapi.h)

02/22/2024

Retrieves the affinity update mode of the specified process.

Syntax

```C++
BOOL QueryProcessAffinityUpdateMode(
  [in]            HANDLE  hProcess,
  [out, optional] LPDWORD lpdwFlags
);
```

Parameters

`[in] hProcess`

A handle to the process. The handle must have the PROCESS_QUERY_INFORMATION or PROCESS_QUERY_LIMITED_INFORMATION access right. For more information, see Process Security and Access Rights.

`[out, optional] lpdwFlags`

The affinity update mode. This parameter can be one of the following values.

Expand table

Value	Meaning
0	Dynamic update of the process affinity by the system is disabled.
PROCESS_AFFINITY_ENABLE_AUTO_UPDATE 0x00000001UL	Dynamic update of the process affinity by the system is enabled.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

To compile an application that calls this function, define _WIN32_WINNT as 0x0600 or later. For more information, see Using the Windows Headers.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista with SP1 [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	processthreadsapi.h (include Windows.h on Windows 7, Windows Server 2008 Windows Server 2008 R2)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

SetProcessAffinityUpdateMode

QueryProcessCycleTime function (realtimeapiset.h)

09/30/2024

Retrieves the sum of the cycle time of all threads of the specified process.

Syntax

```C++
BOOL QueryProcessCycleTime(
  [in]  HANDLE   ProcessHandle,
  [out] PULONG64 CycleTime
);
```

Parameters

`[in] ProcessHandle`

A handle to the process. The handle must have the PROCESS_QUERY_INFORMATION or PROCESS_QUERY_LIMITED_INFORMATION access right. For more information, see Process Security and Access Rights.

`[out] CycleTime`

The number of CPU clock cycles used by the threads of the process. This value includes cycles spent in both user mode and kernel mode.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

To enumerate the processes in the system, use the EnumProcesses function.

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

Requirements

Expand table

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

See also

QueryIdleProcessorCycleTime

QueryThreadCycleTime

QueryProtectedPolicy function (processthreadsapi.h)

02/22/2024

Queries the value associated with a protected policy.

Syntax

```C++
BOOL QueryProtectedPolicy(
  [in]  LPCGUID    PolicyGuid,
  [out] PULONG_PTR PolicyValue
);
```

Parameters

`[in] PolicyGuid`

The globally-unique identifier of the policy to query.

`[out] PolicyValue`

Receives the value that the supplied policy is set to.

Return value

True if the function succeeds; otherwise, false.

Remarks

Protected policies are process-wide configuration settings that are stored in read-only memory. This is intended to help protect the policy from being corrupted or altered in an unintended way while an application is executing.

To compile an application that calls this function, define _WIN32_WINNT as 0x0603 or later. For more information, see Using the Windows Headers.

This function became available in Windows 8.1 and Windows Server 2012 R2 update 3 (the November 2014 update).

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8.1 [desktop apps only]
Minimum supported server	Windows Server 2012 R2 [desktop apps only]
Target Platform	Windows
Header	processthreadsapi.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

QueryThreadCycleTime function (realtimeapiset.h)

05/21/2025

Retrieves the cycle time for the specified thread.

Syntax

```C++
BOOL QueryThreadCycleTime(
  [in]  HANDLE   ThreadHandle,
  [out] PULONG64 CycleTime
);
```

Parameters

`[in] ThreadHandle`

A handle to the thread. The handle must have the THREAD_QUERY_INFORMATION or THREAD_QUERY_LIMITED_INFORMATION access right. For more information, see Thread Security and Access Rights.

`[out] CycleTime`

The number of CPU clock cycles used by the thread. This value includes cycles spent in both user mode and kernel mode.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

To enumerate the threads of the process, use the Thread32First and Thread32Next functions. To get the thread handle for a thread identifier, use the OpenThread function.

Do not attempt to convert the CPU clock cycles returned by **QueryThreadCycleTime** to elapsed time. This function uses timer services provided by the CPU, which can vary in implementation. For example, some CPUs will vary the frequency of the timer when changing the frequency at which the CPU runs and others will leave it at a fixed rate. The behavior of each CPU is described in the documentation provided by the CPU vendor.

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

Requirements

Expand table

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

See also

QueryIdleProcessorCycleTime

QueryProcessCycleTime

QueryThreadpoolStackInformation function (threadpoolapiset.h)

Article02/22/2024

Retrieves the stack reserve and commit sizes for threads in the specified thread pool.

Syntax

```C++
BOOL QueryThreadpoolStackInformation(
  [in]  PTP_POOL                   ptpp,
  [out] PTP_POOL_STACK_INFORMATION ptpsi
);
```

Parameters

`[in] ptpp`

A pointer to a **TP_POOL** structure that specifies the thread pool. The CreateThreadpool function returns this pointer.

`[out] ptpsi`

A pointer to a **TP_POOL_STACK_INFORMATION** structure that receives the stack reserve and commit size.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

To compile an application that uses this function, set _WIN32_WINNT to _WIN32_WINNT_WIN7. For more information, see Using the Windows Headers.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 [desktop apps | UWP apps]
Minimum supported server	Windows Server 2008 R2 [desktop apps | UWP apps]
Target Platform	Windows
Header	threadpoolapiset.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

SetThreadpoolStackInformation

QueryUmsThreadInformation function (winbase.h)

Article08/23/2022

Retrieves information about the specified user-mode scheduling (UMS) worker thread.

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

Syntax

```C++
BOOL QueryUmsThreadInformation(
  [in]            PUMS_CONTEXT          UmsThread,
  [in]            UMS_THREAD_INFO_CLASS UmsThreadInfoClass,
  [out]           PVOID                 UmsThreadInformation,
  [in]            ULONG                 UmsThreadInformationLength,
  [out, optional] PULONG                ReturnLength
);
```

Parameters

`[in] UmsThread`

A pointer to a UMS thread context.

`[in] UmsThreadInfoClass`

A UMS_THREAD_INFO_CLASS value that specifies the kind of information to retrieve.

`[out] UmsThreadInformation`

A pointer to a buffer to receive the specified information. The required size of this buffer depends on the specified information class.

If the information class is **UmsThreadContext** or **UmsThreadTeb**, the buffer must be `sizeof(PVOID)`.

If the information class is **UmsThreadIsSuspended** or **UmsThreadIsTerminated**, the buffer must be `sizeof(BOOLEAN)`.

`[in] UmsThreadInformationLength`

The size of the *UmsThreadInformation* buffer, in bytes.

`[out, optional] ReturnLength`

A pointer to a ULONG variable. On output, this parameter receives the number of bytes written to the *UmsThreadInformation* buffer.

Return value

If the function succeeds, it returns a nonzero value.

If the function fails, the return value is zero. To get extended error information, call GetLastError. Possible error values include the following.

Expand table

Return code	Description
ERROR_INFO_LENGTH_MISMATCH	The buffer is too small for the requested information.
ERROR_INVALID_INFO_CLASS	The specified information class is not supported.
ERROR_NOT_SUPPORTED	UMS is not supported.

Remarks

The **QueryUmsThreadInformation** function retrieves information about the specified UMS worker thread such as its application-defined context, its thread execution block (TEB), and whether the thread is suspended or terminated.

The underlying structures for UMS worker threads are managed by the system. Information that is not exposed through **QueryUmsThreadInformation** should be considered reserved.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 (64-bit only) [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
API set	api-ms-win-core-ums-l1-1-0 (introduced in Windows 7)

See also

SetUmsThreadInformation

UMS_THREAD_INFO_CLASS

QueueUserWorkItem function (threadpoollegacyapiset.h)

Article09/23/2022

Queues a work item to a worker thread in the thread pool.

Syntax

```
C++

BOOL QueueUserWorkItem(
  [in]           LPTHREAD_START_ROUTINE Function,
  [in, optional] PVOID                  Context,
  [in]           ULONG                  Flags
);
```

Parameters

`[in] Function`

A pointer to the application-defined callback function of type **LPTHREAD_START_ROUTINE** to be executed by the thread in the thread pool. This value represents the starting address of the thread. This callback function must not call the TerminateThread function.

The return value of the callback function is not used.

For more information, see ThreadProc.

`[in, optional] Context`

A single parameter value to be passed to the thread function.

`[in] Flags`

The flags that control execution. This parameter can be one or more of the following values.

Expand table

Value	Meaning
WT_EXECUTEDEFAULT 0x00000000	By default, the callback function is queued to a non-I/O worker thread. The callback function is queued to a thread that uses I/O completion ports, which means they cannot perform an

	alertable wait. Therefore, if I/O completes and generates an APC, the APC might wait indefinitely because there is no guarantee that the thread will enter an alertable wait state after the callback completes.
WT_EXECUTEINIOTHREAD 0x00000001	This flag is not used. **Windows Server 2003 and Windows XP:** The callback function is queued to an I/O worker thread. This flag should be used if the function should be executed in a thread that waits in an alertable state. I/O worker threads were removed starting with Windows Vista and Windows Server 2008.
WT_EXECUTEINPERSISTENTTHREAD 0x00000080	The callback function is queued to a thread that never terminates. It does not guarantee that the same thread is used each time. This flag should be used only for short tasks or it could affect other timer operations. This flag must be set if the thread calls functions that use APCs. For more information, see Asynchronous Procedure Calls. Note that currently no worker thread is truly persistent, although worker threads do not terminate if there are any pending I/O requests.
WT_EXECUTELONGFUNCTION 0x00000010	The callback function can perform a long wait. This flag helps the system to decide if it should create a new thread.
WT_TRANSFER_IMPERSONATION 0x00000100	Callback functions will use the current access token, whether it is a process or impersonation token. If this flag is not specified, callback functions execute only with the process token. **Windows XP:** This flag is not supported until Windows XP SP2 and Windows Server 2003.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

If a function in a DLL is queued to a worker thread, be sure that the function has completed execution before the DLL is unloaded.

By default, the thread pool has a maximum of 512 threads per process. To raise the queue limit, use the **WT_SET_MAX_THREADPOOL_THREAD** macro defined in WinNT.h.

syntax

```
#define WT_SET_MAX_THREADPOOL_THREADS(Flags,Limit) \
    ((Flags)|=(Limit)<<16)
```

Use this macro in the call to **QueueUserWorkItem** to specify the *Flags* parameter. The macro parameters are the desired flags and the new limit, up to (2<<16)-1 threads. However, the size of the queue is limited by the size of the kernel nonpaged pool. Note that your application can improve its performance by keeping the number of worker threads low.

To compile an application that uses this function, define **_WIN32_WINNT** as 0x0500 or later. For more information, see Using the Windows Headers.

Requirements

Expand table

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

See also

Process and Thread Functions

Thread Pooling

ThreadProc

ReleaseMutexWhenCallbackReturns function (threadpoolapiset.h)

02/22/2024

Specifies the mutex that the thread pool will release when the current callback completes.

Syntax

```C++
VOID ReleaseMutexWhenCallbackReturns(
  [in, out] PTP_CALLBACK_INSTANCE pci,
  [in]      HANDLE                mut
);
```

Parameters

`[in, out] pci`

A pointer to a **TP_CALLBACK_INSTANCE** structure that defines the callback instance. The pointer is passed to the callback function.

`[in] mut`

A handle to the mutex.

Return value

None

Remarks

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

Requirements

Expand table

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

See also

CallbackMayRunLong

DisassociateCurrentThreadFromCallback

FreeLibraryWhenCallbackReturns

LeaveCriticalSectionWhenCallbackReturns

ReleaseSemaphoreWhenCallbackReturns

SetEventWhenCallbackReturns

Thread Pools

TrySubmitThreadpoolCallback

ReleaseSemaphoreWhenCallbackReturns function (threadpoolapiset.h)

02/22/2024

Specifies the semaphore that the thread pool will release when the current callback completes.

Syntax

```C++
VOID ReleaseSemaphoreWhenCallbackReturns(
  [in, out] PTP_CALLBACK_INSTANCE pci,
  [in]      HANDLE                sem,
  [in]      DWORD                 crel
);
```

Parameters

`[in, out] pci`

A pointer to a **TP_CALLBACK_INSTANCE** structure that defines the callback instance. The pointer is passed to the callback function.

`[in] sem`

A handle to the semaphore.

`[in] crel`

The amount by which to increment the semaphore object's count.

Return value

None

Remarks

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

Requirements

Expand table

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

See also

CallbackMayRunLong

DisassociateCurrentThreadFromCallback

FreeLibraryWhenCallbackReturns

LeaveCriticalSectionWhenCallbackReturns

ReleaseMutexWhenCallbackReturns

SetEventWhenCallbackReturns

Thread Pools

TrySubmitThreadpoolCallback

ResumeThread function (processthreadsapi.h)

02/22/2024

Decrements a thread's suspend count. When the suspend count is decremented to zero, the execution of the thread is resumed.

Syntax

```C++
DWORD ResumeThread(
  [in] HANDLE hThread
);
```

Parameters

`[in] hThread`

A handle to the thread to be restarted.

This handle must have the THREAD_SUSPEND_RESUME access right. For more information, see Thread Security and Access Rights.

Return value

If the function succeeds, the return value is the thread's previous suspend count.

If the function fails, the return value is (DWORD) -1. To get extended error information, call GetLastError.

Remarks

The **ResumeThread** function checks the suspend count of the subject thread. If the suspend count is zero, the thread is not currently suspended. Otherwise, the subject thread's suspend count is decremented. If the resulting value is zero, then the execution of the subject thread is resumed.

If the return value is zero, the specified thread was not suspended. If the return value is 1, the specified thread was suspended but was restarted. If the return value is greater than 1, the specified thread is still suspended.

Note that while reporting debug events, all threads within the reporting process are frozen. Debuggers are expected to use the SuspendThread and **ResumeThread** functions to limit the set of threads that can execute within a process. By suspending all threads in a process except for the one reporting a debug event, it is possible to "single step" a single thread. The other threads are not released by a continue operation if they are suspended.

Windows Phone 8.1: This function is supported for Windows Phone Store apps on Windows Phone 8.1 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	processthreadsapi.h (include Windows.h on Windows Server 2003, Windows Vista, Windows 7, Windows Server 2008 Windows Server 2008 R2)
Library	Kernel32.lib; WindowsPhoneCore.lib on Windows Phone 8.1
DLL	Kernel32.dll; KernelBase.dll on Windows Phone 8.1

See also

OpenThread

Process and Thread Functions

SuspendThread

Suspending Thread Execution

Threads

SetDisplayAutoRotationPreferences function (winuser.h)

02/22/2024

Sets the screen auto-rotation preferences for the current process.

Syntax

```C++
BOOL SetDisplayAutoRotationPreferences(
  [in] ORIENTATION_PREFERENCE orientation
);
```

Parameters

`[in] orientation`

Pointer to a location in memory with the screen orientation preferences to set for the calling process.

Return value

TRUE if the method succeeds, otherwise FALSE.

See GetDisplayAutoRotationPreferences for an example of using this function.

Requirements

Expand table

Requirement	Value
Target Platform	Windows
Header	winuser.h
DLL	Kernel.dll

SetEnvironmentVariable function (winbase.h)

Article09/23/2022

Sets the contents of the specified environment variable for the current process.

Syntax

```
C++

BOOL SetEnvironmentVariable(
  [in]           LPCTSTR lpName,
  [in, optional] LPCTSTR lpValue
);
```

Parameters

`[in] lpName`

The name of the environment variable. The operating system creates the environment variable if it does not exist and *lpValue* is not NULL.

`[in, optional] lpValue`

The contents of the environment variable. The maximum size of a user-defined environment variable is 32,767 characters. For more information, see Environment Variables.

Windows Server 2003 and Windows XP: The total size of the environment block for a process may not exceed 32,767 characters.

If this parameter is NULL, the variable is deleted from the current process's environment.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

This function has no effect on the system environment variables or the environment variables of other processes.

Examples

For an example, see Changing Environment Variables.

Requirements

Expand table

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

See also

Environment Variables

GetEnvironmentVariable

SetEventWhenCallbackReturns function (threadpoolapiset.h)

02/22/2024

Specifies the event that the thread pool will set when the current callback completes.

Syntax

```C++
VOID SetEventWhenCallbackReturns(
  [in, out] PTP_CALLBACK_INSTANCE pci,
  [in]      HANDLE                evt
);
```

Parameters

`[in, out] pci`

A pointer to a **TP_CALLBACK_INSTANCE** structure that defines the callback instance. The pointer is passed to the callback function.

`[in] evt`

A handle to the event to be set.

Return value

None

Remarks

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

Requirements

Expand table

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

See also

CallbackMayRunLong

DisassociateCurrentThreadFromCallback

FreeLibraryWhenCallbackReturns

LeaveCriticalSectionWhenCallbackReturns

ReleaseMutexWhenCallbackReturns

ReleaseSemaphoreWhenCallbackReturns

Thread Pools

TrySubmitThreadpoolCallback

SetInformationJobObject function (jobapi2.h)

Article11/23/2021

Sets limits for a job object.

Syntax

```
C++

BOOL SetInformationJobObject(
  [in] HANDLE             hJob,
  [in] JOBOBJECTINFOCLASS JobObjectInformationClass,
  [in] LPVOID             lpJobObjectInformation,
  [in] DWORD              cbJobObjectInformationLength
);
```

Parameters

`[in] hJob`

A handle to the job whose limits are being set. The CreateJobObject or OpenJobObject function returns this handle. The handle must have the **JOB_OBJECT_SET_ATTRIBUTES** access right. For more information, see Job Object Security and Access Rights.

`[in] JobObjectInformationClass`

The information class for the limits to be set. This parameter can be one of the following values.

Expand table

Value	Meaning
JobObjectAssociateCompletionPortInformation 7	The *lpJobObjectInfo* parameter is a pointer to a JOBOBJECT_ASSOCIATE_COMPLETION_PORT structure.
JobObjectBasicLimitInformation 2	The *lpJobObjectInfo* parameter is a pointer to a JOBOBJECT_BASIC_LIMIT_INFORMATION structure.
JobObjectBasicUIRestrictions 4	The *lpJobObjectInfo* parameter is a pointer to a JOBOBJECT_BASIC_UI_RESTRICTIONS structure.

JobObjectCpuRateControlInformation 15	The *lpJobObjectInfo* parameter is a pointer to a JOBOBJECT_CPU_RATE_CONTROL_INFORMATION structure. **Windows 7, Windows Server 2008 R2, Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP:** This flag is not supported. If Dynamic Fair Share Scheduling (DFSS) is enabled, the CPU rate cannot be set and SetInformationJobObject will fail with error code 50 ("The request is not supported").
JobObjectEndOfJobTimeInformation 6	The *lpJobObjectInfo* parameter is a pointer to a JOBOBJECT_END_OF_JOB_TIME_INFORMATION structure.
JobObjectExtendedLimitInformation 9	The *lpJobObjectInfo* parameter is a pointer to a JOBOBJECT_EXTENDED_LIMIT_INFORMATION structure.
JobObjectGroupInformation 11	The *lpJobObjectInfo* parameter is a pointer to a **USHORT** value that specifies the list of processor groups to assign the job to. The *cbJobObjectInfoLength* parameter is set to the size of the group data. Divide this value by `sizeof(USHORT)` to determine the number of groups. **Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP:** This flag is not supported.
JobObjectGroupInformationEx 14	The *lpJobObjectInfo* parameter is a pointer to a buffer that contains an array of GROUP_AFFINITY structures that specify the affinity of the job for the processor groups to which the job is currently assigned. The *cbJobObjectInfoLength* parameter is set to the size of the group affinity data. Divide this value by `sizeof(GROUP_AFFINITY)` to determine the number of groups. **Windows 7, Windows Server 2008 R2, Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP:** This flag is not supported.
JobObjectLimitViolationInformation2 34	The *lpJobObjectInfo* parameter is a pointer to a JOBOBJECT_LIMIT_VIOLATION_INFORMATION_2 structure. **Windows 8.1, Windows Server 2012 R2, Windows 8, Windows Server 2012, Windows 7, Windows**

	Server 2008 R2, Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP: This flag is not supported.
JobObjectNetRateControlInformation 32	The *lpJobObjectInfo* parameter is a pointer to a JOBOBJECT_NET_RATE_CONTROL_INFORMATION structure. **Windows 8.1, Windows Server 2012 R2, Windows 8, Windows Server 2012, Windows 7, Windows Server 2008 R2, Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP:** This flag is not supported.
JobObjectNotificationLimitInformation 12	The *lpJobObjectInfo* parameter is a pointer to a JOBOBJECT_NOTIFICATION_LIMIT_INFORMATION structure. **Windows 7, Windows Server 2008 R2, Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP:** This flag is not supported.
JobObjectNotificationLimitInformation2 33	The *lpJobObjectInfo* parameter is a pointer to a JOBOBJECT_NOTIFICATION_LIMIT_INFORMATION_2 structure. **Windows 8.1, Windows Server 2012 R2, Windows 8, Windows Server 2012, Windows 7, Windows Server 2008 R2, Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP:** This flag is not supported.
JobObjectSecurityLimitInformation 5	This flag is not supported. Applications must set security limitations individually for each process. **Windows Server 2003 and Windows XP:** The *lpJobObjectInfo* parameter is a pointer to a JOBOBJECT_SECURITY_LIMIT_INFORMATION structure. The *hJob* handle must have the **JOB_OBJECT_SET_SECURITY_ATTRIBUTES** access right associated with it.

`[in] lpJobObjectInformation`

The limits or job state to be set for the job. The format of this data depends on the value of *JobObjectInfoClass*.

`[in] cbJobObjectInformationLength`

The size of the job information being set, in bytes.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

Use the **SetInformationJobObject** function to set several limits in a single call. To establish the limits one at a time or change a subset of the limits, call the QueryInformationJobObject function to obtain the current limits, modify these limits, and then call **SetInformationJobObject**.

You must set security limits 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: Use the **SetInformationJobObject** function to set security limits for the job object.

To compile an application that uses this function, define _WIN32_WINNT as 0x0500 or later. For more information, see Using the Windows Headers.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	jobapi2.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

JOBOBJECT_ASSOCIATE_COMPLETION_PORT

JOBOBJECT_BASIC_LIMIT_INFORMATION

JOBOBJECT_BASIC_UI_RESTRICTIONS

JOBOBJECT_CPU_RATE_CONTROL_INFORMATION

JOBOBJECT_END_OF_JOB_TIME_INFORMATION

JOBOBJECT_EXTENDED_LIMIT_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

Job Objects

Process Security and Access Rights

Process and Thread Functions

QueryInformationJobObject

SetIoRateControlInformationJobObject function (jobapi2.h)

Article01/16/2025

Windows 10, version 1607, and newer: This function is not supported.

Sets I/O limits on a job object.

Syntax

```
C++

DWORD SetIoRateControlInformationJobObject(
  [in] HANDLE                                hJob,
  [in] JOBOBJECT_IO_RATE_CONTROL_INFORMATION *IoRateControlInfo
);
```

Parameters

`[in] hJob`

A handle to the job on which to set I/O limits. Get this handle from the CreateJobObject or OpenJobObject function. The handle must have the **JOB_OBJECT_SET_ATTRIBUTES** access right. For more information about access rights, see Job Object Security and Access Rights.

`[in] IoRateControlInfo`

A pointer to a JOBOBJECT_IO_RATE_CONTROL_INFORMATION structure that specifies the I/O limits to set for the job.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 10 [desktop apps only]
Minimum supported server	Windows Server 2016 [desktop apps only]
Target Platform	Windows
Header	jobapi2.h
Library	Kernel32.lib
DLL	Kernel32.dll

See also

JOBOBJECT_IO_RATE_CONTROL_INFORMATION

QueryIoRateControlInformationJobObject

SetPriorityClass function (processthreadsapi.h)

11/01/2022

Sets the priority class for the specified process. This value together with the priority value of each thread of the process determines each thread's base priority level.

Syntax

```
C++

BOOL SetPriorityClass(
  [in] HANDLE hProcess,
  [in] DWORD  dwPriorityClass
);
```

Parameters

`[in] hProcess`

A handle to the process.

The handle must have the **PROCESS_SET_INFORMATION** access right. For more information, see Process Security and Access Rights.

`[in] dwPriorityClass`

The priority class for the process. This parameter can be one of the following values.

Expand table

Priority	Meaning
ABOVE_NORMAL_PRIORITY_CLASS 0x00008000	Process that has priority above **NORMAL_PRIORITY_CLASS** but below **HIGH_PRIORITY_CLASS**.
BELOW_NORMAL_PRIORITY_CLASS 0x00004000	Process that has priority above **IDLE_PRIORITY_CLASS** but below **NORMAL_PRIORITY_CLASS**.
HIGH_PRIORITY_CLASS 0x00000080	Process that performs time-critical tasks that must be executed immediately. The threads of the process preempt the threads of normal or idle priority class processes. An example is the Task List, which must respond quickly when

	called by the user, regardless of the load on the operating system. Use extreme care when using the high-priority class, because a high-priority class application can use nearly all available CPU time.
IDLE_PRIORITY_CLASS 0x00000040	Process whose threads run only when the system is idle. The threads of the process are preempted by the threads of any process running in a higher priority class. An example is a screen saver. The idle-priority class is inherited by child processes.
NORMAL_PRIORITY_CLASS 0x00000020	Process with no special scheduling needs.
PROCESS_MODE_BACKGROUND_BEGIN 0x00100000	Begin background processing mode. The system lowers the resource scheduling priorities of the process (and its threads) so that it can perform background work without significantly affecting activity in the foreground. This value can be specified only if *hProcess* is a handle to the current process. The function fails if the process is already in background processing mode. **Windows Server 2003 and Windows XP:** This value is not supported.
PROCESS_MODE_BACKGROUND_END 0x00200000	End background processing mode. The system restores the resource scheduling priorities of the process (and its threads) as they were before the process entered background processing mode. This value can be specified only if *hProcess* is a handle to the current process. The function fails if the process is not in background processing mode. **Windows Server 2003 and Windows XP:** This value is not supported.
REALTIME_PRIORITY_CLASS 0x00000100	Process that has the highest possible priority. The threads of the process preempt the threads of all other processes, including operating system processes performing important tasks. For example, a real-time process that executes for more than a very brief interval can cause disk caches not to flush or cause the mouse to be unresponsive.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

Every thread has a base priority level determined by the thread's priority value and the priority class of its process. The system uses the base priority level of all executable threads to determine which thread gets the next slice of CPU time. The SetThreadPriority function enables setting the base priority level of a thread relative to the priority class of its process. For more information, see Scheduling Priorities.

The ***_PRIORITY_CLASS** values affect the CPU scheduling priority of the process. For processes that perform background work such as file I/O, network I/O, or data processing, it is not sufficient to adjust the CPU scheduling priority; even an idle CPU priority process can easily interfere with system responsiveness when it uses the disk and memory. Processes that perform background work should use the **PROCESS_MODE_BACKGROUND_BEGIN** and **PROCESS_MODE_BACKGROUND_END** values to adjust their resource scheduling priorities; processes that interact with the user should not use **PROCESS_MODE_BACKGROUND_BEGIN**.

If a process is in background processing mode, the new threads it creates will also be in background processing mode. When a thread is in background processing mode, it should minimize sharing resources such as critical sections, heaps, and handles with other threads in the process, otherwise priority inversions can occur. If there are threads executing at high priority, a thread in background processing mode may not be scheduled promptly, but it will never be starved.

Each thread can enter background processing mode independently using SetThreadPriority. Do not call **SetPriorityClass** to enter background processing mode after a thread in the process has called **SetThreadPriority** to enter background processing mode. After a process ends background processing mode, it resets all threads in the process; however, it is not possible for the process to know which threads were already in background processing mode.

Examples

The following example demonstrates the use of process background mode.

C++

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

int main( void )
{
   DWORD dwError, dwPriClass;

   if(!SetPriorityClass(GetCurrentProcess(), PROCESS_MODE_BACKGROUND_BEGIN))
   {
      dwError = GetLastError();
```

```
        if( ERROR_PROCESS_MODE_ALREADY_BACKGROUND == dwError)
           _tprintf(TEXT("Already in background mode\n"));
        else _tprintf(TEXT("Failed to enter background mode (%d)\n"), dwError);
        goto Cleanup;
    }

    // Display priority class

    dwPriClass = GetPriorityClass(GetCurrentProcess());

    _tprintf(TEXT("Current priority class is 0x%x\n"), dwPriClass);

    //
    // Perform background work
    //
    ;

    if(!SetPriorityClass(GetCurrentProcess(), PROCESS_MODE_BACKGROUND_END))
    {
        _tprintf(TEXT("Failed to end background mode (%d)\n"), GetLastError());
    }

Cleanup:
    // Clean up
    ;
return 0;
}
```

Requirements

Expand table

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

See also

CreateProcess

CreateThread

GetPriorityClass

GetThreadPriority

Process and Thread Functions

Processes

Scheduling Priorities

SetThreadPriority

SetProcessAffinityMask function (winbase.h)

Article01/27/2022

Sets a processor affinity mask for the threads of the specified process.

Syntax

```
C++

BOOL SetProcessAffinityMask(
  [in] HANDLE    hProcess,
  [in] DWORD_PTR dwProcessAffinityMask
);
```

Parameters

`[in] hProcess`

A handle to the process whose affinity mask is to be set. This handle must have the **PROCESS_SET_INFORMATION** access right. For more information, see Process Security and Access Rights.

`[in] dwProcessAffinityMask`

The affinity mask for the threads of the process.

On a system with more than 64 processors, the affinity mask must specify processors in a single processor group.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

If the process affinity mask requests a processor that is not configured in the system, the last error code is **ERROR_INVALID_PARAMETER**.

On a system with more than 64 processors, if the calling process contains threads in more than one processor group, the last error code is **ERROR_INVALID_PARAMETER**.

Remarks

A process affinity mask is a bit vector in which each bit represents a logical processor on which the threads of the process are allowed to run. The value of the process affinity mask must be a subset of the system affinity mask values obtained by the GetProcessAffinityMask function. A process is only allowed to run on the processors configured into a system. Therefore, the process affinity mask cannot specify a 1 bit for a processor when the system affinity mask specifies a 0 bit for that processor.

Process affinity is inherited by any child process or newly instantiated local process.

Do not call **SetProcessAffinityMask** in a DLL that may be called by processes other than your own.

On a system with more than 64 processors, the **SetProcessAffinityMask** function can be used to set the process affinity mask only for processes with threads in a single processor group. Use the SetThreadAffinityMask function to set the affinity mask for individual threads in multiple groups. This effectively changes the group assignment of the process.

Starting with Windows 11 and Windows Server 2022, on a system with more than 64 processors, process and thread affinities span all processors in the system, across all processor groups, by default. Instead of always failing in case the calling process contains threads in more than one processor group, the **SetProcessAffinityMask** function fails (returning zero with **ERROR_INVALID_PARAMETER** last error code) if the process had explicitly set the affinity of one or more of its threads outside of the process' primary group.

Requirements

Expand table

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

See also

CreateProcess

GetProcessAffinityMask

Multiple Processors

Process and Thread Functions

Processes

Processor Groups

SetThreadAffinityMask

SetProcessAffinityUpdateMode function (processthreadsapi.h)

11/01/2022

Sets the affinity update mode of the specified process.

Syntax

```C++
BOOL SetProcessAffinityUpdateMode(
  [in] HANDLE hProcess,
  [in] DWORD  dwFlags
);
```

Parameters

`[in] hProcess`

A handle to the process. This handle must be returned by the GetCurrentProcess function.

`[in] dwFlags`

The affinity update mode. This parameter can be one of the following values.

Expand table

Value	Meaning
0	Disables dynamic update of the process affinity by the system.
PROCESS_AFFINITY_ENABLE_AUTO_UPDATE 0x00000001UL	Enables dynamic update of the process affinity by the system.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The system can adjust process affinity under various conditions, such as when a processor is added dynamically. By default, dynamic updates to the process affinity are disabled for each process.

Processes should use this function to indicate whether they can handle dynamic adjustment of process affinity by the system. After a process enables affinity update mode, it can call this function to disable it. However, a process cannot enable affinity update mode after it has used this function to disable it.

Child processes do not inherit the affinity update mode of the parent process. The affinity update mode must be explicitly set for each child process.

To compile an application that calls this function, define _WIN32_WINNT as 0x0600 or later. For more information, see Using the Windows Headers.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista with SP1 [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Target Platform	Windows
Header	processthreadsapi.h (include Windows.h on Windows 7, Windows Server 2008 Windows Server 2008 R2)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

QueryProcessAffinityUpdateMode

SetProcessDefaultCpuSets function (processthreadsapi.h)

02/22/2024

Sets the default CPU Sets assignment for threads in the specified process. Threads that are created, which don't have CPU Sets explicitly set using SetThreadSelectedCpuSets, will inherit the sets specified by **SetProcessDefaultCpuSets** automatically.

Syntax

C++

```
BOOL SetProcessDefaultCpuSets(
  HANDLE      Process,
  const ULONG *CpuSetIds,
  ULONG       CpuSetIdCount
);
```

Parameters

`Process`

Specifies the process for which to set the default CPU Sets. This handle must have the PROCESS_SET_LIMITED_INFORMATION access right. The value returned by GetCurrentProcess can also be specified here.

`CpuSetIds`

Specifies the list of CPU Set IDs to set as the process default CPU set. If this is NULL, the **SetProcessDefaultCpuSets** clears out any assignment.

`CpuSetIdCount`

Specifies the number of IDs in the list passed in the **CpuSetIds** argument. If that value is NULL, this should be 0.

Return value

This function cannot fail when passed valid parameters

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	processthreadsapi.h
DLL	kernel32.dll

SetProcessInformation function (processthreadsapi.h)

10/31/2024

Sets information for the specified process.

Syntax

```C++
BOOL SetProcessInformation(
  [in] HANDLE                    hProcess,
  [in] PROCESS_INFORMATION_CLASS ProcessInformationClass,
       LPVOID                    ProcessInformation,
  [in] DWORD                     ProcessInformationSize
);
```

Parameters

`[in] hProcess`

A handle to the process. This handle must have the **PROCESS_SET_INFORMATION** access right. For more information, see Process Security and Access Rights.

`[in] ProcessInformationClass`

A member of the PROCESS_INFORMATION_CLASS enumeration specifying the kind of information to set.

`ProcessInformation`

Pointer to an object that contains the type of information specified by the *ProcessInformationClass* parameter.

If the *ProcessInformationClass* parameter is **ProcessMemoryPriority**, this parameter must point to a MEMORY_PRIORITY_INFORMATION structure.

If the *ProcessInformationClass* parameter is **ProcessPowerThrottling**, this parameter must point to a PROCESS_POWER_THROTTLING_STATE structure.

If the *ProcessInformationClass* parameter is **ProcessLeapSecondInfo**, this parameter must point to a PROCESS_LEAP_SECOND_INFO structure.

If the *ProcessInformationClass* parameter is **ProcessOverrideSubsequentPrefetchParameter**, this parameter must point to an OVERRIDE_PREFETCH_PARAMETER structure.

`[in] ProcessInformationSize`

The size in bytes of the structure specified by the *ProcessInformation* parameter.

If the *ProcessInformationClass* parameter is **ProcessMemoryPriority**, this parameter must be `sizeof(MEMORY_PRIORITY_INFORMATION)`.

If the *ProcessInformationClass* parameter is **ProcessPowerThrottling**, this parameter must be `sizeof(PROCESS_POWER_THROTTLING_STATE)`.

If the *ProcessInformationClass* parameter is **ProcessLeapSecondInfo**, this parameter must be `sizeof(PROCESS_LEAP_SECOND_INFO)`.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

To help improve system performance, applications should use the **SetProcessInformation** function with **ProcessMemoryPriority** to lower the default memory priority of threads that perform background operations or access files and data that are not expected to be accessed again soon. For example, a file indexing application might set a lower default priority for the process that performs the indexing task.

Memory priority helps to determine how long pages remain in the working set of a process before they are trimmed. A process's memory priority determines the default priority of the physical pages that are added to the process working set by the threads of that process. When the memory manager trims the working set, it trims lower priority pages before higher priority pages. This improves overall system performance because higher priority pages are less likely to be trimmed from the working set and then trigger a page fault when they are accessed again.

ProcessPowerThrottling enables throttling policies on a process, which can be used to balance out performance and power efficiency in cases where optimal performance is not required.

When a process opts into enabling `PROCESS_POWER_THROTTLING_EXECUTION_SPEED`, the process will be classified as EcoQoS. The system will try to increase power efficiency through strategies such as reducing CPU frequency or using more power efficient cores. EcoQoS should be used when the work is not contributing to the foreground user experience, which provides longer battery life, and reduced heat and fan noise. EcoQoS should not be used for performance critical or foreground user experiences. (Prior to Windows 11, the EcoQoS level did not exist and the process was labeled as LowQoS). If an application does not explicitly enable `PROCESS_POWER_THROTTLING_EXECUTION_SPEED`, the system will use its own heuristics to automatically infer a Quality of Service level. For more information, see Quality of Service.

When a process opts into enabling `PROCESS_POWER_THROTTLING_IGNORE_TIMER_RESOLUTION`, any current timer resolution requests made by the process will be ignored. Timers belonging to the process are no longer guaranteed to expire with higher timer resolution, which can improve power efficiency. After explicitly disabling `PROCESS_POWER_THROTTLING_IGNORE_TIMER_RESOLUTION`, the system remembers and honors any previous timer resolution request by the process. By default in Windows 11 if a window owning process becomes fully occluded, minimized, or otherwise non-visible to the end user, and non-audible, Windows may automatically ignore the timer resolution request and thus does not guarantee a higher resolution than the default system resolution.

Examples

The following example shows how to call **SetProcessInformation** with **ProcessMemoryPriority** to set low memory priority as the default for the calling process.

syntax

```
DWORD ErrorCode;
BOOL Success;
MEMORY_PRIORITY_INFORMATION MemPrio;

//
// Set low memory priority on the current process.
//

ZeroMemory(&MemPrio, sizeof(MemPrio));
MemPrio.MemoryPriority = MEMORY_PRIORITY_LOW;

Success = SetProcessInformation(GetCurrentProcess(),
                                ProcessMemoryPriority,
                                &MemPrio,
                                sizeof(MemPrio));

if (!Success) {
    ErrorCode = GetLastError();
    fprintf(stderr, "Set process memory priority failed: %d\n", ErrorCode);
```

```
        goto cleanup;
    }
```

The following example shows how to call **SetProcessInformation** with **ProcessPowerThrottling** to control the Quality of Service of a process.

syntax

```
PROCESS_POWER_THROTTLING_STATE PowerThrottling;
RtlZeroMemory(&PowerThrottling, sizeof(PowerThrottling));
PowerThrottling.Version = PROCESS_POWER_THROTTLING_CURRENT_VERSION;

//
// EcoQoS
// Turn EXECUTION_SPEED throttling on.
// ControlMask selects the mechanism and StateMask declares which mechanism should
be on or off.
//

PowerThrottling.ControlMask = PROCESS_POWER_THROTTLING_EXECUTION_SPEED;
PowerThrottling.StateMask = PROCESS_POWER_THROTTLING_EXECUTION_SPEED;

SetProcessInformation(GetCurrentProcess(),
                      ProcessPowerThrottling,
                      &PowerThrottling,
                      sizeof(PowerThrottling));

//
// HighQoS
// Turn EXECUTION_SPEED throttling off.
// ControlMask selects the mechanism and StateMask is set to zero as mechanisms
should be turned off.
//

PowerThrottling.ControlMask = PROCESS_POWER_THROTTLING_EXECUTION_SPEED;
PowerThrottling.StateMask = 0;

SetProcessInformation(GetCurrentProcess(),
                      ProcessPowerThrottling,
                      &PowerThrottling,
                      sizeof(PowerThrottling));
```

The following example shows how to call **SetProcessInformation** with **ProcessPowerThrottling** to control the Timer Resolution of a process.

syntax

```
PROCESS_POWER_THROTTLING_STATE PowerThrottling;
RtlZeroMemory(&PowerThrottling, sizeof(PowerThrottling));
PowerThrottling.Version = PROCESS_POWER_THROTTLING_CURRENT_VERSION;
```

```
//
// Ignore Timer Resolution Requests.
// Turn IGNORE_TIMER_RESOLUTION throttling on.
// ControlMask selects the mechanism and StateMask declares which mechanism should
be on or off.
//

PowerThrottling.ControlMask = PROCESS_POWER_THROTTLING_IGNORE_TIMER_RESOLUTION;
PowerThrottling.StateMask = PROCESS_POWER_THROTTLING_IGNORE_TIMER_RESOLUTION;

SetProcessInformation(GetCurrentProcess(),
                      ProcessPowerThrottling,
                      &PowerThrottling,
                      sizeof(PowerThrottling));

//
// Always honor Timer Resolution Requests.
// Turn IGNORE_TIMER_RESOLUTION throttling off.
// ControlMask selects the mechanism and StateMask is set to zero as mechanisms
should be turned off.
//

PowerThrottling.ControlMask = PROCESS_POWER_THROTTLING_IGNORE_TIMER_RESOLUTION;
PowerThrottling.StateMask = 0;

SetProcessInformation(GetCurrentProcess(),
                      ProcessPowerThrottling,
                      &PowerThrottling,
                      sizeof(PowerThrottling));
```

The following example shows how to call **SetProcessInformation** with **ProcessPowerThrottling** to reset to the default system managed behavior.

syntax

```
PROCESS_POWER_THROTTLING_STATE PowerThrottling;
RtlZeroMemory(&PowerThrottling, sizeof(PowerThrottling));
PowerThrottling.Version = PROCESS_POWER_THROTTLING_CURRENT_VERSION;

//
// Let system manage all power throttling. ControlMask is set to 0 as we don't
want
// to control any mechanisms.
//

PowerThrottling.ControlMask = 0;
PowerThrottling.StateMask = 0;

SetProcessInformation(GetCurrentProcess(),
                      ProcessPowerThrottling,
                      &PowerThrottling,
```

```
                        sizeof(PowerThrottling));
```

Requirements

Expand table

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

See also

GetProcessInformation function, SetThreadInformation function, MEMORY_PRIORITY_INFORMATION structure, SetProcessInformation function, PROCESS_INFORMATION_CLASS enumeration, OVERRIDE_PREFETCH_PARAMETER structure

SetProcessMitigationPolicy function (processthreadsapi.h)

11/01/2022

Sets a mitigation policy for the calling process. Mitigation policies enable a process to harden itself against various types of attacks.

Syntax

```C++
BOOL SetProcessMitigationPolicy(
  [in] PROCESS_MITIGATION_POLICY MitigationPolicy,
  [in] PVOID                     lpBuffer,
  [in] SIZE_T                    dwLength
);
```

Parameters

`[in] MitigationPolicy`

The mitigation policy to apply. This parameter can be one of the following values.

Expand table

Value	Meaning
ProcessDEPPolicy	The data execution prevention (DEP) policy of the process. The *lpBuffer* parameter points to a PROCESS_MITIGATION_DEP_POLICY structure that specifies the DEP policy flags.
ProcessASLRPolicy	The Address Space Layout Randomization (ASLR) policy of the process. The *lpBuffer* parameter points to a PROCESS_MITIGATION_ASLR_POLICY structure that specifies the ASLR policy flags.
ProcessDynamicCodePolicy	The dynamic code policy of the process. When turned on, the process cannot generate dynamic code or modify existing executable code. The *lpBuffer* parameter points to a

Value	Meaning
	PROCESS_MITIGATION_DYNAMIC_CODE_POLICY structure that specifies the dynamic code policy flags.
ProcessStrictHandleCheckPolicy	The process will receive a fatal error if it manipulates a handle that is not valid. The *lpBuffer* parameter points to a PROCESS_MITIGATION_STRICT_HANDLE_CHECK_POLICY structure that specifies the handle check policy flags.
ProcessSystemCallDisablePolicy	Disables the ability to use NTUser/GDI functions at the lowest layer. The *lpBuffer* parameter points to a PROCESS_MITIGATION_SYSTEM_CALL_DISABLE_POLICY structure that specifies the system call disable policy flags.
ProcessMitigationOptionsMask	Returns the mask of valid bits for all the mitigation options on the system. An application can set many mitigation options without querying the operating system for mitigation options by combining bitwise with the mask to exclude all non-supported bits at once. The *lpBuffer* parameter points to a **ULONG64** bit vector for the mask, or to accommodate more than 64 bits, a two-element array of **ULONG64** bit vectors.
ProcessExtensionPointDisablePolicy	The *lpBuffer* parameter points to a PROCESS_MITIGATION_EXTENSION_POINT_DISABLE_POLICY structure that specifies the extension point disable policy flags.
ProcessControlFlowGuardPolicy	The Control Flow Guard (CFG) policy of the process. The *lpBuffer* parameter points to a PROCESS_MITIGATION_CONTROL_FLOW_GUARD_POLICY structure that specifies the CFG policy flags.
ProcessSignaturePolicy	The policy of a process that can restrict image loading to those images that are either signed by Microsoft, by the Windows Store, or by Microsoft, the Windows Store and the Windows Hardware Quality Labs (WHQL). he *lpBuffer* parameter points to a PROCESS_MITIGATION_BINARY_SIGNATURE_POLICY structure that specifies the signature policy flags.
ProcessFontDisablePolicy	The policy regarding font loading for the process. When turned on, the process cannot load non-system fonts. The *lpBuffer* parameter points to a PROCESS_MITIGATION_FONT_DISABLE_POLICY structure that specifies the policy flags for font loading.
ProcessImageLoadPolicy	The policy regarding image loading for the process, which determines the types of executable images that are allowed to be

Value	Meaning
	mapped into the process. When turned on, images cannot be loaded from some locations, such a remote devices or files that have the low mandatory label. The *lpBuffer* parameter points to a PROCESS_MITIGATION_IMAGE_LOAD_POLICY structure that specifies the policy flags for image loading.
ProcessRedirectionTrustPolicy	The RedirectionGuard policy of a process. The *lpBuffer* parameter points to a PROCESS_MITIGATION_REDIRECTION_TRUST_POLICY structure that specifies the mitigation mode.
ProcessSideChannelIsolationPolicy	Windows 10, version 1809 and above: The policy regarding isolation of side channels for the specified process. The *lpBuffer* parameter points to a PROCESS_MITIGATION_SIDE_CHANNEL_ISOLATION_POLICY structure that specifies the policy flags for side channel isolation.
ProcessUserShadowStackPolicy	Windows 10, version 2004 and above: The policy regarding user-mode Hardware-enforced Stack Protection for the process. The *lpBuffer* parameter points to a PROCESS_MITIGATION_USER_SHADOW_STACK_POLICY structure that specifies the policy flags for user-mode Hardware-enforced Stack Protection.

`[in] lpBuffer`

If the *MitigationPolicy* parameter is **ProcessDEPPolicy**, this parameter points to a PROCESS_MITIGATION_DEP_POLICY structure that specifies the DEP policy flags.

If the *MitigationPolicy* parameter is **ProcessASLRPolicy**, this parameter points to a PROCESS_MITIGATION_ASLR_POLICY structure that specifies the ASLR policy flags.

If the *MitigationPolicy* parameter is **ProcessImageLoadPolicy**, this parameter points to a PROCESS_MITIGATION_IMAGE_LOAD_POLICY structure that receives the policy flags for image loading.

If the *MitigationPolicy* parameter is **ProcessStrictHandleCheckPolicy**, this parameter points to a PROCESS_MITIGATION_STRICT_HANDLE_CHECK_POLICY structure that specifies the handle check policy flags.

If the *MitigationPolicy* parameter is **ProcessSystemCallDisablePolicy**, this parameter points to a PROCESS_MITIGATION_SYSTEM_CALL_DISABLE_POLICY structure that specifies the system call disable policy flags.

If the *MitigationPolicy* parameter is **ProcessMitigationOptionsMask**, this parameter points to a **ULONG64** bit vector for the mask, or to accommodate more than 64 bits, a two-element array

of **ULONG64** bit vectors.

If the *MitigationPolicy* parameter is **ProcessExtensionPointDisablePolicy**, this parameter points to a PROCESS_MITIGATION_EXTENSION_POINT_DISABLE_POLICY structure that specifies the extension point disable policy flags.

If the *MitigationPolicy* parameter is **ProcessControlFlowGuardPolicy**, this parameter points to a PROCESS_MITIGATION_CONTROL_FLOW_GUARD_POLICY structure that specifies the CFG policy flags.

If the *MitigationPolicy* parameter is **ProcessSignaturePolicy**, this parameter points to a PROCESS_MITIGATION_BINARY_SIGNATURE_POLICY structure that specifies the signature policy flags.

If the *MitigationPolicy* parameter is **ProcessFontDisablePolicy**, this parameter points to a PROCESS_MITIGATION_FONT_DISABLE_POLICY structure that specifies the policy flags for font loading.

If the *MitigationPolicy* parameter is **ProcessImageLoadPolicy**, this parameter points to a PROCESS_MITIGATION_IMAGE_LOAD_POLICY structure that specifies the policy flags for image loading.

If the *MitigationPolicy* parameter is **ProcessRedirectionTrustPolicy**, this parameter points to a PROCESS_MITIGATION_REDIRECTION_TRUST_POLICY structure that specifies the mitigation mode.

If the *MitigationPolicy* parameter is **ProcessUserShadowStackPolicy**, this parameter points to a PROCESS_MITIGATION_USER_SHADOW_STACK_POLICY structure that specifies the policy flags for user-mode Hardware-enforced Stack Protection.

`[in] dwLength`

The size of *lpBuffer*, in bytes.

Return value

If the function succeeds, it returns **TRUE**. If the function fails, it returns **FALSE**. To retrieve error values defined for this function, call GetLastError.

Remarks

Setting mitigation policy for a process helps prevent an attacker from exploiting security vulnerabilities. Use the **SetProcessMitigationPolicy** function to enable or disable security

mitigation programmatically.

For maximum effectiveness, mitigation policies should be applied before or during process initialization. For example, setting the ASLR policy that enables forced relocation of images is effective only if it is applied before all of the images in a process have been loaded.

ASLR mitigation policies cannot be made less restrictive after they have been applied.

To compile an application that uses this function, set _WIN32_WINNT >= 0x0602. For more information, see Using the Windows Headers.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]
Target Platform	Windows
Header	processthreadsapi.h
Library	Kernel32.lib
DLL	Kernel32.dll

SetProcessPriorityBoost function (processthreadsapi.h)

02/22/2024

Disables or enables the ability of the system to temporarily boost the priority of the threads of the specified process.

Syntax

```C++
BOOL SetProcessPriorityBoost(
  [in] HANDLE hProcess,
  [in] BOOL   bDisablePriorityBoost
);
```

Parameters

`[in] hProcess`

A handle to the process. This handle must have the PROCESS_SET_INFORMATION access right. For more information, see Process Security and Access Rights.

`[in] bDisablePriorityBoost`

If this parameter is TRUE, dynamic boosting is disabled. If the parameter is FALSE, dynamic boosting is enabled.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

When a thread is running in one of the dynamic priority classes, the system temporarily boosts the thread's priority when it is taken out of a wait state. If **SetProcessPriorityBoost** is called with the *DisablePriorityBoost* parameter set to TRUE, its threads' priorities are not boosted. This

setting affects all existing threads and any threads subsequently created by the process. To restore normal behavior, call **SetProcessPriorityBoost** with *DisablePriorityBoost* set to FALSE.

Requirements

Expand table

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

See also

GetProcessPriorityBoost

Priority Boosts

Process and Thread Functions

Processes

Scheduling Priorities

SetProcessRestrictionExemption function (winuser.h)

04/02/2021

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.

This function only succeeds if a developer license is present on the system. Once successful the calling process will be able to perform the following actions, subject to User Interface Privilege Isolation (UIPI) restrictions:

- Attach global hooks (and event hooks) to Windows Store app processes.
- Attach input queues between Windows Store app processes, Windows Store app browsers, system processes, and desktop application processes.
- Change foreground arbitrarily between the Windows Store app and desktop environments.

Syntax

```C++
BOOL SetProcessRestrictionExemption(
  BOOL fEnableExemption
);
```

Parameters

`fEnableExemption`

When set to TRUE, indicates a request to disable exemption for the calling process.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

Any process can call this function, including desktop and Windows Store app processes and processes that use IL code.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8 [desktop apps | UWP apps]
Minimum supported server	Windows Server 2012 [desktop apps | UWP apps]
Target Platform	Windows
Header	winuser.h
Library	User32.lib
DLL	User32.dll

SetProcessShutdownParameters function (processthreadsapi.h)

11/01/2022

Sets shutdown parameters for the currently calling process. This function sets a shutdown order for a process relative to the other processes in the system.

Syntax

```C++
BOOL SetProcessShutdownParameters(
  [in] DWORD dwLevel,
  [in] DWORD dwFlags
);
```

Parameters

`[in] dwLevel`

The shutdown priority for a process relative to other processes in the system. The system shuts down processes from high *dwLevel* values to low. The highest and lowest shutdown priorities are reserved for system components. This parameter must be in the following range of values.

Expand table

Value	Meaning
000-0FF	System reserved last shutdown range.
100-1FF	Application reserved last shutdown range.
200-2FF	Application reserved "in between" shutdown range.
300-3FF	Application reserved first shutdown range.
400-4FF	System reserved first shutdown range.

All processes start at shutdown level 0x280.

`[in] dwFlags`

This parameter can be the following value.

Expand table

Value	Meaning
SHUTDOWN_NORETRY 0x00000001	The system terminates the process without displaying a retry dialog box for the user.

Return value

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

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

Applications running in the system security context do not get shut down by the operating system. They get notified of shutdown or logoff through the callback function installable via SetConsoleCtrlHandler. They also get notified in the order specified by the *dwLevel* parameter.

Requirements

Expand table

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

See also

GetProcessShutdownParameters

Process and Thread Functions

Processes

SetConsoleCtrlHandler

SetProcessWorkingSetSize function (memoryapi.h)

12/09/2022

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

Syntax

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

Parameters

`[in] hProcess`

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

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

`[in] dwMinimumWorkingSetSize`

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

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

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

`[in] dwMaximumWorkingSetSize`

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

and available memory is low.

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

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

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. Call GetLastError to obtain extended error information.

Remarks

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

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

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

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

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

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

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

Requirements

Expand table

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

See also

GetProcessWorkingSetSize

Process Working Set

Process and Thread Functions

Processes

SetProcessWorkingSetSizeEx

VirtualLock

SetProcessWorkingSetSizeEx function (memoryapi.h)

09/23/2022

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

Syntax

```
C++

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

Parameters

`[in] hProcess`

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

The handle must have **PROCESS_SET_QUOTA** access rights. For more information, see Process Security and Access Rights.

`[in] dwMinimumWorkingSetSize`

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

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

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

`[in] dwMaximumWorkingSetSize`

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

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

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

`[in] Flags`

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

Expand table

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

Return value

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

If the function fails, the return value is zero. To get extended error information, call GetLastError. If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

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

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

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

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

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

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

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

must use the SetProcessWorkingSetSize function carefully. You must always consider the performance of the whole system when you are designing an application.

An application can use the VirtualLock function to lock ranges of the application's virtual address space in memory; however, that can potentially degrade the performance of the system.

Requirements

Expand table

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

See also

GetProcessWorkingSetSizeEx

Process Working Set

Process and Thread Functions

Processes

VirtualLock

Working Set

SetProtectedPolicy function (processthreadsapi.h)

02/22/2024

Sets a protected policy. This function is for use primarily by Windows, and not designed for external use.

Syntax

```C++
BOOL SetProtectedPolicy(
  [in]  LPCGUID    PolicyGuid,
  [in]  ULONG_PTR  PolicyValue,
  [out] PULONG_PTR OldPolicyValue
);
```

Parameters

`[in] PolicyGuid`

The globally-unique identifier of the policy to set.

`[in] PolicyValue`

The value to set the policy to.

`[out] OldPolicyValue`

Optionally receives the original value that was associated with the supplied policy.

Return value

True if the function succeeds; otherwise, false. To retrieve error values for this function, call GetLastError.

Remarks

Protected policies are process-wide configuration settings that are stored in read-only memory. This is intended to help protect the policy from being corrupted or altered in an

unintended way while an application is executing. Protected policies are primarily a construct internal to Windows.

To compile an application that calls this function, define _WIN32_WINNT as 0x0603 or later. For more information, see Using the Windows Headers.

This function became available in update 3 (the November 2014 update) for Windows 8.1 and Windows Server 2012 R2.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8.1 [desktop apps only]
Minimum supported server	Windows Server 2012 R2 [desktop apps only]
Target Platform	Windows
Header	processthreadsapi.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

SetThreadAffinityMask function (winbase.h)

Article01/27/2022

Sets a processor affinity mask for the specified thread.

Syntax

```
C++

DWORD_PTR SetThreadAffinityMask(
  [in] HANDLE    hThread,
  [in] DWORD_PTR dwThreadAffinityMask
);
```

Parameters

`[in] hThread`

A handle to the thread whose affinity mask is to be set.

This handle must have the **THREAD_SET_INFORMATION** or **THREAD_SET_LIMITED_INFORMATION** access right and the **THREAD_QUERY_INFORMATION** or **THREAD_QUERY_LIMITED_INFORMATION** access right. For more information, see Thread Security and Access Rights.

Windows Server 2003 and Windows XP: The handle must have the **THREAD_SET_INFORMATION** and **THREAD_QUERY_INFORMATION** access rights.

`[in] dwThreadAffinityMask`

The affinity mask for the thread.

On a system with more than 64 processors, the affinity mask must specify processors in the thread's current processor group.

Return value

If the function succeeds, the return value is the thread's previous affinity mask.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

If the thread affinity mask requests a processor that is not selected for the process affinity mask, the last error code is **ERROR_INVALID_PARAMETER**.

Remarks

A thread affinity mask is a bit vector in which each bit represents a logical processor that a thread is allowed to run on. A thread affinity mask must be a subset of the process affinity mask for the containing process of a thread. A thread can only run on the processors its process can run on. Therefore, the thread affinity mask cannot specify a 1 bit for a processor when the process affinity mask specifies a 0 bit for that processor.

Setting an affinity mask for a process or thread can result in threads receiving less processor time, as the system is restricted from running the threads on certain processors. In most cases, it is better to let the system select an available processor.

If the new thread affinity mask does not specify the processor that is currently running the thread, the thread is rescheduled on one of the allowable processors.

Starting with Windows 11 and Windows Server 2022, on a system with more than 64 processors, process and thread affinities span all processors in the system, across all processor groups, by default. The *dwThreadAffinityMask* must specify processors in the thread's current primary group.

Requirements

Expand table

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

See also

GetProcessAffinityMask

Multiple Processors

OpenThread

Process and Thread Functions

Processor Groups

SetProcessAffinityMask

SetThreadIdealProcessor

Threads

SetThreadDescription function (processthreadsapi.h)

02/22/2024

Assigns a description to a thread.

Syntax

```C++
HRESULT SetThreadDescription(
  [in] HANDLE hThread,
  [in] PCWSTR lpThreadDescription
);
```

Parameters

`[in] hThread`

A handle for the thread for which you want to set the description. The handle must have THREAD_SET_LIMITED_INFORMATION access.

`[in] lpThreadDescription`

A Unicode string that specifies the description of the thread.

Return value

If the function succeeds, the return value is the **HRESULT** that denotes a successful operation. If the function fails, the return value is an **HRESULT** that denotes the error.

Remarks

The description of a thread can be set more than once; the most recently set value is used. You can retrieve the description of a thread by calling GetThreadDescription.

Windows Server 2016, **Windows 10 LTSB 2016** and **Windows 10 version 1607**: SetThreadDescription is only available by Run Time Dynamic Linking in KernelBase.dll.

Examples

The following example sets the description for the current thread to `simulation_thread`.

```C++
HRESULT hr = SetThreadDescription(GetCurrentThread(), L"simulation_thread");
if (FAILED(hr))
{
    // Call failed.
}
```

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 10, version 1607 [desktop apps | UWP apps]
Minimum supported server	Windows Server 2016 [desktop apps | UWP apps]
Target Platform	Windows
Header	processthreadsapi.h
Library	Kernel32.lib
DLL	Kernel32.dll

See also

GetThreadDescription, How to: Set a Thread Name in Native Code

SetThreadGroupAffinity function (processtopologyapi.h)

Article01/27/2022

Sets the processor group affinity for the specified thread.

Syntax

```C++
BOOL SetThreadGroupAffinity(
  [in]            HANDLE               hThread,
  [in]            const GROUP_AFFINITY *GroupAffinity,
  [out, optional] PGROUP_AFFINITY      PreviousGroupAffinity
);
```

Parameters

`[in] hThread`

A handle to the thread.

The handle must have the THREAD_SET_INFORMATION access right. For more information, see Thread Security and Access Rights.

`[in] GroupAffinity`

A GROUP_AFFINITY structure that specifies the processor group affinity to be used for the specified thread.

`[out, optional] PreviousGroupAffinity`

A pointer to a GROUP_AFFINITY structure to receive the thread's previous group affinity. This parameter can be NULL.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, use GetLastError.

Remarks

Starting with Windows 11 and Windows Server 2022, on a system with more than 64 processors, process and thread affinities span all processors in the system, across all processor groups, by default. The **SetThreadGroupAffinity** function restricts a thread's affinity to the processors over the single processor group specified by the given *GroupAffinity*. This group will also become the thread's primary group.

To compile an application that uses this function, set _WIN32_WINNT >= 0x0601. For more information, see Using the Windows Headers.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	processtopologyapi.h
Library	Kernel32.lib
DLL	Kernel32.dll

See also

GROUP_AFFINITY

SetThreadIdealProcessor function (processthreadsapi.h)

02/22/2024

Sets a preferred processor for a thread. The system schedules threads on their preferred processors whenever possible.

On a system with more than 64 processors, this function sets the preferred processor to a logical processor in the processor group to which the calling thread is assigned. Use the SetThreadIdealProcessorEx function to specify a processor group and preferred processor.

Syntax

```
C++

DWORD SetThreadIdealProcessor(
  [in] HANDLE hThread,
  [in] DWORD  dwIdealProcessor
);
```

Parameters

`[in] hThread`

A handle to the thread whose preferred processor is to be set. The handle must have the THREAD_SET_INFORMATION access right. For more information, see Thread Security and Access Rights.

`[in] dwIdealProcessor`

The number of the preferred processor for the thread. This value is zero-based. If this parameter is MAXIMUM_PROCESSORS, the function returns the current ideal processor without changing it.

Return value

If the function succeeds, the return value is the previous preferred processor.

If the function fails, the return value is (DWORD) – 1. To get extended error information, call GetLastError.

Remarks

You can use the GetSystemInfo function to determine the number of processors on the computer. You can also use the GetProcessAffinityMask function to check the processors on which the thread is allowed to run. Note that **GetProcessAffinityMask** returns a bitmask whereas **SetThreadIdealProcessor** uses an integer value to represent the processor.

Starting with Windows 11 and Windows Server 2022, on a system with more than 64 processors, process and thread affinities span all processors in the system, across all processor groups, by default. The **SetThreadIdealProcessor** function sets the preferred processor to a logical processor in the thread's primary group.

To compile an application that uses this function, define _WIN32_WINNT as 0x0400 or later. For more information, see Using the Windows Headers.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Expand table

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

See also

GetProcessAffinityMask

GetSystemInfo

Multiple Processors

OpenThread

Process and Thread Functions

SetThreadAffinityMask

SetThreadIdealProcessorEx

Threads

SetThreadIdealProcessorEx function (processthreadsapi.h)

11/01/2022

Sets the ideal processor for the specified thread and optionally retrieves the previous ideal processor.

Syntax

```
C++

BOOL SetThreadIdealProcessorEx(
  [in]            HANDLE            hThread,
  [in]            PPROCESSOR_NUMBER lpIdealProcessor,
  [out, optional] PPROCESSOR_NUMBER lpPreviousIdealProcessor
);
```

Parameters

`[in] hThread`

A handle to the thread for which to set the ideal processor. This handle must have been created with the THREAD_SET_INFORMATION access right. For more information, see Thread Security and Access Rights.

`[in] lpIdealProcessor`

A pointer to a PROCESSOR_NUMBER structure that specifies the processor number of the desired ideal processor.

`[out, optional] lpPreviousIdealProcessor`

A pointer to a PROCESSOR_NUMBER structure to receive the previous ideal processor. This parameter can point to the same memory location as the *lpIdealProcessor* parameter. This parameter can be NULL if the previous ideal processor is not required.

Return value

If the function succeeds, it returns a nonzero value.

If the function fails, it returns zero. To get extended error information, use GetLastError.

Remarks

Specifying a thread ideal processor provides a hint to the scheduler about the preferred processor for a thread. The scheduler runs the thread on the thread's ideal processor when possible.

Starting with Windows 11 and Windows Server 2022, on a system with more than 64 processors, process and thread affinities span all processors in the system, across all processor groups, by default. The **SetThreadIdealProcessorEx**, in setting the preferred processor, also sets the thread's primary group to the group of the preferred processor.

To compile an application that uses this function, set _WIN32_WINNT >= 0x0601. For more information, see Using the Windows Headers.

Windows Phone 8.1: This function is supported for Windows Phone Store apps on Windows Phone 8.1 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 [desktop apps | UWP apps]
Minimum supported server	Windows Server 2008 R2 [desktop apps | UWP apps]
Target Platform	Windows
Header	processthreadsapi.h (include Windows.h)
Library	Kernel32.lib; WindowsPhoneCore.lib on Windows Phone 8.1
DLL	Kernel32.dll; KernelBase.dll on Windows Phone 8.1

See also

GetThreadIdealProcessorEx

SetThreadIdealProcessor

SetThreadInformation function (processthreadsapi.h)

11/01/2022

Sets information for the specified thread.

Syntax

```C++
BOOL SetThreadInformation(
  [in] HANDLE                    hThread,
  [in] THREAD_INFORMATION_CLASS ThreadInformationClass,
       LPVOID                    ThreadInformation,
  [in] DWORD                     ThreadInformationSize
);
```

Parameters

`[in] hThread`

A handle to the thread. The handle must have THREAD_SET_INFORMATION access right. For more information, see Thread Security and Access Rights.

`[in] ThreadInformationClass`

The class of information to set. The only supported values are **ThreadMemoryPriority** and **ThreadPowerThrottling**.

`ThreadInformation`

Pointer to a structure that contains the type of information specified by the *ThreadInformationClass* parameter.

If the *ThreadInformationClass* parameter is **ThreadMemoryPriority**, this parameter must point to a **MEMORY_PRIORITY_INFORMATION** structure.

If the *ThreadInformationClass* parameter is **ThreadPowerThrottling**, this parameter must point to a **THREAD_POWER_THROTTLING_STATE** structure.

`[in] ThreadInformationSize`

The size in bytes of the structure specified by the *ThreadInformation* parameter.

If the *ThreadInformationClass* parameter is **ThreadMemoryPriority**, this parameter must be `sizeof(MEMORY_PRIORITY_INFORMATION)`.

If the *ThreadInformationClass* parameter is **ThreadPowerThrottling**, this parameter must be `sizeof(THREAD_POWER_THROTTLING_STATE)`.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

To help improve system performance, applications should use the **SetThreadInformation** function with **ThreadMemoryPriority** to lower the memory priority of threads that perform background operations or access files and data that are not expected to be accessed again soon. For example, an anti-malware application might lower the priority of threads involved in scanning files.

Memory priority helps to determine how long pages remain in the working set of a process before they are trimmed. A thread's memory priority determines the minimum priority of the physical pages that are added to the process working set by that thread. When the memory manager trims the working set, it trims lower priority pages before higher priority pages. This improves overall system performance because higher priority pages are less likely to be trimmed from the working set and then trigger a page fault when they are accessed again.

ThreadPowerThrottling enables throttling policies on a thread, which can be used to balance out performance and power efficiency in cases where optimal performance is not required. When a thread opts into enabling `THREAD_POWER_THROTTLING_EXECUTION_SPEED`, the thread will be classified as EcoQoS. The system will try to increase power efficiency through strategies such as reducing CPU frequency or using more power efficient cores. EcoQoS should be used when the work is not contributing to the foreground user experience, which provides longer battery life, and reduced heat and fan noise. EcoQoS should not be used for performance critical or foreground user experiences. (Prior to Windows 11, the EcoQoS level did not exist and the process was instead labeled as LowQoS). If an application does not explicitly enable `THREAD_POWER_THROTTLING_EXECUTION_SPEED`, the system will use its own heuristics to automatically infer a Quality of Service level. For more information, see Quality of Service.

Examples

The following example shows how to call **SetThreadInformation** with **ThreadMemoryPriority** to set low memory priority on the current thread.

C

```c
DWORD ErrorCode;
BOOL Success;
MEMORY_PRIORITY_INFORMATION MemPrio;

//
// Set low memory priority on the current thread.
//

ZeroMemory(&MemPrio, sizeof(MemPrio));
MemPrio.MemoryPriority = MEMORY_PRIORITY_LOW;

Success = SetThreadInformation(GetCurrentThread(),
                               ThreadMemoryPriority,
                               &MemPrio,
                               sizeof(MemPrio));

if (!Success) {
    ErrorCode = GetLastError();
    fprintf(stderr, "Set thread memory priority failed: %d\n", ErrorCode);
}
```

The following example shows how to call **SetThreadInformation** with **ThreadPowerThrottling** to control the Quality of Service of a thread.

C

```c
THREAD_POWER_THROTTLING_STATE PowerThrottling;
ZeroMemory(&PowerThrottling, sizeof(PowerThrottling));
PowerThrottling.Version = THREAD_POWER_THROTTLING_CURRENT_VERSION;

//
// EcoQoS
// Turn EXECUTION_SPEED throttling on.
// ControlMask selects the mechanism and StateMask declares which mechanism should
be on or off.
//

PowerThrottling.ControlMask = THREAD_POWER_THROTTLING_EXECUTION_SPEED;
PowerThrottling.StateMask = THREAD_POWER_THROTTLING_EXECUTION_SPEED;

SetThreadInformation(GetCurrentThread(),
                     ThreadPowerThrottling,
                     &PowerThrottling,
                     sizeof(PowerThrottling));
```

```
//
// HighQoS
// Turn EXECUTION_SPEED throttling off.
// ControlMask selects the mechanism and StateMask is set to zero as mechanisms
should be turned off.
//

PowerThrottling.ControlMask = THREAD_POWER_THROTTLING_EXECUTION_SPEED;
PowerThrottling.StateMask = 0;

SetThreadInformation(GetCurrentThread(),
                     ThreadPowerThrottling,
                     &PowerThrottling,
                     sizeof(PowerThrottling));

//
// Let system manage all power throttling. ControlMask is set to 0 as we don't
want
// to control any mechanisms.
//

PowerThrottling.ControlMask = 0;
PowerThrottling.StateMask = 0;

SetThreadInformation(GetCurrentThread(),
                     ThreadPowerThrottling,
                     &PowerThrottling,
                     sizeof(PowerThrottling));
```

Requirements

Expand table

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

See also

GetThreadInformation

SetProcessInformation

SetThreadpoolCallbackCleanupGroup function (winbase.h)

07/27/2022

Associates the specified cleanup group with the specified callback environment.

Syntax

```C++
VOID SetThreadpoolCallbackCleanupGroup(
  [in, out]      PTP_CALLBACK_ENVIRON              pcbe,
  [in]           PTP_CLEANUP_GROUP                 ptpcg,
  [in, optional] PTP_CLEANUP_GROUP_CANCEL_CALLBACK pfng
);
```

Parameters

`[in, out] pcbe`

A **TP_CALLBACK_ENVIRON** structure that defines the callback environment. The InitializeThreadpoolEnvironment function returns this structure.

`[in] ptpcg`

A **TP_CLEANUP_GROUP** structure that defines the cleanup group. The CreateThreadpoolCleanupGroup function returns this structure.

`[in, optional] pfng`

The cleanup callback to be called if the cleanup group is canceled before the associated object is released. The function is called when you call CloseThreadpoolCleanupGroupMembers.

Return value

None

Remarks

This function is implemented as an inline function.

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

Examples

For an example, see Using the Thread Pool Functions.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps | UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)

See also

DestroyThreadpoolEnvironment

InitializeThreadpoolEnvironment

SetThreadpoolCallbackLibrary

SetThreadpoolCallbackPool

SetThreadpoolCallbackPriority

SetThreadpoolCallbackRunsLong

Thread Pools

SetThreadpoolCallbackLibrary function (winbase.h)

02/22/2024

Ensures that the specified DLL remains loaded as long as there are outstanding callbacks.

Syntax

```C++
VOID SetThreadpoolCallbackLibrary(
  [in, out] PTP_CALLBACK_ENVIRON pcbe,
  [in]      PVOID                mod
);
```

Parameters

`[in, out] pcbe`

A **TP_CALLBACK_ENVIRON** structure that defines the callback environment. The InitializeThreadpoolEnvironment function returns this structure.

`[in] mod`

A handle to the DLL.

Return value

None

Remarks

You should call this function if a callback might acquire the loader lock. This prevents a deadlock from occurring when one thread in DllMain is waiting for the callback to end, and another thread that is executing the callback attempts to acquire the loader lock.

If the DLL containing the callback might be unloaded, the cleanup code in DllMain must cancel outstanding callbacks before releasing the object.

Managing callbacks created with a TP_CALLBACK_ENVIRON that specifies a callback library is somewhat processing-intensive. You should consider other options for ensuring that the library is not unloaded while callbacks are executing, or to guarantee that callbacks which may be executing do not acquire the loader lock.

The thread pool assumes ownership of the library reference supplied to this function. The caller should not call FreeLibrary on a module handle after passing it to this function.

This function is implemented as an inline function.

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

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps | UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)

See also

DestroyThreadpoolEnvironment

FreeLibraryWhenCallbackReturns

InitializeThreadpoolEnvironment

SetThreadpoolCallbackCleanupGroup

SetThreadpoolCallbackPool

SetThreadpoolCallbackPriority

SetThreadpoolCallbackRunsLong

Thread Pools

SetThreadpoolCallbackPersistent function (winbase.h)

02/22/2024

Specifies that the callback should run on a persistent thread.

Syntax

```
C++

VOID SetThreadpoolCallbackPersistent(
  [in, out] PTP_CALLBACK_ENVIRON pcbe
);
```

Parameters

`[in, out] pcbe`

A **TP_CALLBACK_ENVIRON** structure that defines the callback environment. The InitializeThreadpoolEnvironment function returns this structure.

Return value

None

Remarks

This function is implemented as an inline function.

To compile an application that uses this function, set _WIN32_WINNT to _WIN32_WINNT_WIN7. For more information, see Using the Windows Headers.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)

SetThreadpoolCallbackPool function (winbase.h)

10/13/2021

Sets the thread pool to be used when generating callbacks.

Syntax

```C++
VOID SetThreadpoolCallbackPool(
  [in, out] PTP_CALLBACK_ENVIRON pcbe,
  [in]      PTP_POOL             ptpp
);
```

Parameters

`[in, out] pcbe`

A **TP_CALLBACK_ENVIRON** structure that defines the callback environment. The InitializeThreadpoolEnvironment function returns this structure.

`[in] ptpp`

A **TP_POOL** structure that defines the thread pool. The CreateThreadpool function returns this structure.

Return value

None

Remarks

If you do not specify a thread pool, the global thread pool is used.

This function is implemented as an inline function.

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

Examples

For an example, see Using the Thread Pool Functions.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps | UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)

See also

DestroyThreadpoolEnvironment

InitializeThreadpoolEnvironment

SetThreadpoolCallbackCleanupGroup

SetThreadpoolCallbackLibrary

SetThreadpoolCallbackPriority

SetThreadpoolCallbackRunsLong

Thread Pools

SetThreadpoolCallbackPriority function (winbase.h)

02/22/2024

Specifies the priority of a callback function relative to other work items in the same thread pool.

Syntax

```C++
VOID SetThreadpoolCallbackPriority(
  [in, out] PTP_CALLBACK_ENVIRON pcbe,
  [in]      TP_CALLBACK_PRIORITY Priority
);
```

Parameters

`[in, out] pcbe`

A **TP_CALLBACK_ENVIRON** structure that defines the callback environment. The InitializeThreadpoolEnvironment function returns this structure.

`[in] Priority`

The priority for the callback relative to other callbacks in the same thread pool. This parameter can be one of the following **TP_CALLBACK_PRIORITY** enumeration values:

Expand table

Value	Meaning
TP_CALLBACK_PRIORITY_HIGH	The callback should run at high priority.
TP_CALLBACK_PRIORITY_LOW	The callback should run at low priority.
TP_CALLBACK_PRIORITY_NORMAL	The callback should run at normal priority.

Return value

None

Remarks

Higher priority callbacks are guaranteed to be run first by the first available worker thread, but they are not guaranteed to finish before lower priority callbacks.

This function is implemented as an inline function.

To compile an application that uses this function, set _WIN32_WINNT >= _WIN32_WINNT_WIN7. For more information, see Using the Windows Headers.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 [desktop apps | UWP apps]
Minimum supported server	Windows Server 2008 R2 [desktop apps | UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)

SetThreadpoolCallbackRunsLong function (winbase.h)

02/22/2024

Indicates that callbacks associated with this callback environment may not return quickly.

Syntax

```C++
VOID SetThreadpoolCallbackRunsLong(
  [in, out] PTP_CALLBACK_ENVIRON pcbe
);
```

Parameters

`[in, out] pcbe`

A **TP_CALLBACK_ENVIRON** structure that defines the callback environment. The InitializeThreadpoolEnvironment function returns this structure.

Return value

None

Remarks

The thread pool may use this information to better determine when a new thread should be created.

This function is implemented as an inline function.

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

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps | UWP apps]
Target Platform	Windows
Header	winbase.h (include Windows.h)

See also

CallbackMayRunLong

DestroyThreadpoolEnvironment

InitializeThreadpoolEnvironment

SetThreadpoolCallbackCleanupGroup

SetThreadpoolCallbackLibrary

SetThreadpoolCallbackPool

SetThreadpoolCallbackPriority

Thread Pools

SetThreadpoolStackInformation function (threadpoolapiset.h)

Article10/13/2021

Sets the stack reserve and commit sizes for new threads in the specified thread pool. Stack reserve and commit sizes for existing threads are not changed.

Syntax

```
C++

BOOL SetThreadpoolStackInformation(
  [in, out] PTP_POOL                   ptpp,
  [in]      PTP_POOL_STACK_INFORMATION ptpsi
);
```

Parameters

`[in, out] ptpp`

A pointer to a **TP_POOL** structure that specifies the thread pool. The CreateThreadpool function returns this pointer.

`[in] ptpsi`

A pointer to a **TP_POOL_STACK_INFORMATION** structure that specifies the stack reserve and commit size for threads in the pool.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

To compile an application that uses this function, set _WIN32_WINNT to _WIN32_WINNT_WIN7. For more information, see Using the Windows Headers.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 [desktop apps | UWP apps]
Minimum supported server	Windows Server 2008 R2 [desktop apps | UWP apps]
Target Platform	Windows
Header	threadpoolapiset.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

QueryThreadpoolStackInformation

SetThreadpoolThreadMaximum function (threadpoolapiset.h)

02/22/2024

Sets the maximum number of threads that the specified thread pool can allocate to process callbacks.

Syntax

```C++
VOID SetThreadpoolThreadMaximum(
  [in, out] PTP_POOL ptpp,
  [in]      DWORD    cthrdMost
);
```

Parameters

`[in, out] ptpp`

A pointer to a **TP_POOL** structure that defines the thread pool. The CreateThreadpool function returns this pointer.

`[in] cthrdMost`

The maximum number of threads.

Return value

None

Remarks

To specify the minimum number of threads available in the pool, call SetThreadpoolThreadMinimum.

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

Examples

For an example, see Using the Thread Pool Functions.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps | UWP apps]
Target Platform	Windows
Header	threadpoolapiset.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

CloseThreadpool

CreateThreadpool

SetThreadpoolThreadMinimum

Thread Pools

SetThreadpoolThreadMinimum function (threadpoolapiset.h)

Article02/22/2024

Sets the minimum number of threads that the specified thread pool must make available to process callbacks.

Syntax

```
C++

BOOL SetThreadpoolThreadMinimum(
  [in, out] PTP_POOL ptpp,
  [in]      DWORD    cthrdMic
);
```

Parameters

`[in, out] ptpp`

A pointer to a **TP_POOL** structure that defines the thread pool. The CreateThreadpool function returns this pointer.

`[in] cthrdMic`

The minimum number of threads.

Return value

If the function succeeds, it returns TRUE.

If the function fails, it returns FALSE. To retrieve extended error information, call GetLastError.

Remarks

To specify the maximum number of threads that the pool may allocate, call SetThreadpoolThreadMaximum.

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

Examples

For an example, see Using the Thread Pool Functions.

Requirements

Expand table

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

See also

CloseThreadpool

CreateThreadpool

SetThreadpoolThreadMaximum

Thread Pools

SetThreadpoolTimer function (threadpoolapiset.h)

09/23/2022

Sets the timer object, replacing the previous timer, if any. A worker thread calls the timer object's callback after the specified timeout expires.

Syntax

```C++
VOID SetThreadpoolTimer(
  [in, out]      PTP_TIMER pti,
  [in, optional] PFILETIME pftDueTime,
  [in]           DWORD     msPeriod,
  [in]           DWORD     msWindowLength
);
```

Parameters

`[in, out] pti`

A pointer to a **TP_TIMER** structure that defines the timer object to set. The CreateThreadpoolTimer function returns this pointer.

`[in, optional] pftDueTime`

A pointer to a FILETIME structure that specifies the absolute or relative time at which the timer should expire. If positive or zero, it indicates the absolute time since January 1, 1601 (UTC), measured in 100 nanosecond units. If negative, it indicates the amount of time to wait relative to the current time. For more information about time values, see File Times.

If this parameter is NULL, the timer object will cease to queue new callbacks (but callbacks already queued will still occur).

The timer is set if the *pftDueTime* parameter is non-NULL.

`[in] msPeriod`

The timer period, in milliseconds. If this parameter is zero, the timer is signaled once. If this parameter is greater than zero, the timer is periodic. A periodic timer automatically reactivates each time the period elapses, until the timer is canceled.

`[in] msWindowLength`

The maximum amount of time the system can delay before calling the timer callback. If this parameter is not set to zero, the system can batch calls to conserve power.

Return value

None

Remarks

Setting the timer cancels the previous timer, if any.

In some cases, callback functions might run after an application closes the threadpool timer. To prevent this behavior, an application should follow the steps described in CloseThreadpoolTimer.

If the due time specified by *pftDueTime* is relative, the time that the system spends in sleep or hibernation does not count toward the expiration of the timer. The timer is signaled when the cumulative amount of elapsed time the system spends in the waking state equals the timer's relative due time or period. If the due time specified by *pftDueTime* is absolute, the time that the system spends in sleep or hibernation does count toward the expiration of the timer. If the timer expires while the system is sleeping, the timer is signaled immediately when the system wakes.

If the due time specified by *pftDueTime* is zero, then the timer expires immediately.

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

Examples

For an example, see Using the Thread Pool Functions.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista [desktop apps | UWP apps]

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

See also

CloseThreadpoolTimer

CreateThreadpoolTimer

IsThreadpoolTimerSet

Thread Pools

WaitForThreadpoolTimerCallbacks

SetThreadpoolTimerEx function (threadpoolapiset.h)

Article02/22/2024

Sets the timer object, replacing the previous timer, if any. A worker thread calls the timer object's callback after the specified timeout expires.

Syntax

```C++
BOOL SetThreadpoolTimerEx(
  [in, out]      PTP_TIMER pti,
  [in, optional] PFILETIME pftDueTime,
  [in]           DWORD     msPeriod,
  [in, optional] DWORD     msWindowLength
);
```

Parameters

`[in, out] pti`

A pointer to a **TP_TIMER** structure that defines the timer object to set. The CreateThreadpoolTimer function returns this pointer.

`[in, optional] pftDueTime`

A pointer to a FILETIME structure that specifies the absolute or relative time at which the timer should expire. If this parameter points to a positive value, it indicates the absolute time since January 1, 1601 (UTC), measured in 100 nanosecond units. If this parameter points to a negative value, it indicates the amount of time to wait relative to the current time. If this parameter points to zero, then the timer expires immediately. For more information about time values, see File Times.

If this parameter is NULL, the timer object will cease to queue new callbacks (but callbacks already queued will still occur).

The timer is set if the *pftDueTime* parameter is non-NULL.

`[in] msPeriod`

The timer period, in milliseconds. If this parameter is zero, the timer is signaled once. If this parameter is greater than zero, the timer is periodic. A periodic timer automatically reactivates each time the period elapses, until the timer is canceled.

`[in, optional] msWindowLength`

The maximum amount of time the system can delay before calling the timer callback. If this parameter is set, the system can batch calls to conserve power.

Return value

Returns TRUE if the timer was previously set and was canceled. Otherwise returns FALSE.

If the timer's previous state was "set", and the function returns FALSE, then a callback is in progress or about to commence. See the remarks for further discussion.

Remarks

Setting the timer cancels the previous timer, if any.

In some cases, callback functions might run after an application closes the threadpool timer. To prevent this behavior, an application should follow the steps described in CloseThreadpoolTimer.

If the due time specified by *pftDueTime* is relative, the time that the system spends in sleep or hibernation does not count toward the expiration of the timer. The timer is signaled when the cumulative amount of elapsed time the system spends in the waking state equals the timer's relative due time or period. If the due time specified by *pftDueTime* is absolute, the time that the system spends in sleep or hibernation does count toward the expiration of the timer. If the timer expires while the system is sleeping, the timer is signaled immediately when the system wakes.

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

Examples

For an example, see Using the Thread Pool Functions.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8 [desktop apps | UWP apps]
Minimum supported server	Windows Server 2012 [desktop apps | UWP apps]
Target Platform	Windows
Header	threadpoolapiset.h
Library	Kernel32.lib
DLL	Kernel32.dll

See also

CloseThreadpoolTimer

CreateThreadpoolTimer

IsThreadpoolTimerSet

Thread Pools

WaitForThreadpoolTimerCallbacks

SetThreadpoolWait function (threadpoolapiset.h)

06/26/2024

Sets the wait object, replacing the previous wait object, if any. A worker thread calls the wait object's callback function after the handle becomes signaled or after the specified timeout expires.

Syntax

```C++
VOID SetThreadpoolWait(
  [in, out]      PTP_WAIT  pwa,
  [in, optional] HANDLE    h,
  [in, optional] PFILETIME pftTimeout
);
```

Parameters

`[in, out] pwa`

A pointer to a **TP_WAIT** structure that defines the wait object. The CreateThreadpoolWait function returns this pointer.

`[in, optional] h`

A handle.

If this parameter is NULL, the wait object will cease to queue new callbacks (but callbacks already queued will still occur).

If this parameter is not NULL, it must refer to a valid waitable object.

Mutex is not supported. If a handle to a mutex is passed in, the thread pool raises a STATUS_THREADPOOL_HANDLE_EXCEPTION exception and ExceptionRecord.ExceptionInformation[0] will be equal to STATUS_INVALID_PARAMETER_3.

If this handle is closed while the wait is still pending, the function's behavior is undefined. If the wait is still pending and the handle must be closed, use CloseThreadpoolWait to cancel the wait and then close the handle.

The wait is considered set if this parameter is non-NULL.

`[in, optional] pftTimeout`

A pointer to a FILETIME structure that specifies the absolute or relative time at which the wait operation should time out. If this parameter points to a positive value, it indicates the absolute time since January 1, 1601 (UTC), in 100-nanosecond intervals. If this parameter points to a negative value, it indicates the amount of time to wait relative to the current time. For more information about time values, see File Times.

If this parameter points to 0, the wait times out immediately. If this parameter is NULL, the wait will not time out.

Return value

None

Remarks

A wait object can wait for only one handle. Setting the handle for a wait object replaces the previous handle, if any.

You must re-register the event with the wait object before signaling it each time to trigger the wait callback.

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

Examples

For an example, see Using the Thread Pool Functions.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps | UWP apps]

Requirement	Value
Target Platform	Windows
Header	threadpoolapiset.h (include Windows.h on Windows 7, Windows Server 2008 Windows Server 2008 R2)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

CloseThreadpoolWait

CreateThreadpoolWait

SetThreadpoolWaitEx

Thread Pools

WaitForThreadpoolWaitCallbacks

SetThreadpoolWaitEx function (threadpoolapiset.h)

Article07/27/2022

Sets the wait object, replacing the previous wait object, if any. A worker thread calls the wait object's callback function after the handle becomes signaled or after the specified timeout expires.

Syntax

C++

```
BOOL SetThreadpoolWaitEx(
  [in, out]      PTP_WAIT  pwa,
  [in, optional] HANDLE    h,
  [in, optional] PFILETIME pftTimeout,
                 PVOID     Reserved
);
```

Parameters

`[in, out] pwa`

A pointer to a **TP_WAIT** structure that defines the wait object. The CreateThreadpoolWait function returns this pointer.

`[in, optional] h`

A handle.

If this parameter is NULL, the wait object will cease to queue new callbacks (but callbacks already queued will still occur).

If this parameter is not NULL, it must refer to a valid waitable object.

If this handle is closed while the wait is still pending, the function's behavior is undefined. If the wait is still pending and the handle must be closed, use CloseThreadpoolWait to cancel the wait and then close the handle.

The wait is considered set if this parameter is non-NULL.

`[in, optional] pftTimeout`

A pointer to a FILETIME structure that specifies the absolute or relative time at which the wait operation should time out. If this parameter points to a positive value, it indicates the absolute time since January 1, 1601 (UTC), in 100-nanosecond intervals. If this parameter points to a negative value, it indicates the amount of time to wait relative to the current time. If this parameter points to zero, then the wait times out immediately. For more information about time values, see File Times.

If this parameter is NULL, then the wait does not time out.

`Reserved`

Reserved. Must be NULL.

Return value

Returns TRUE if the wait was previously set and was canceled. Otherwise returns FALSE.

If the wait's previous state was "set", and the function returns FALSE, then a callback is in progress or about to commence. See the remarks for further discussion.

Remarks

A wait object can wait for only one handle. Setting the handle for a wait object replaces the previous wait handle, if any.

In some cases, callback functions might run after an application closes the threadpool timer. To prevent this behavior, an application should follow the steps described in CloseThreadpoolWait.

If the timeout specified by *pftTimeout* is relative, the time that the system spends in sleep or hibernation does not count toward the expiration of the wait. The wait is signaled when the cumulative amount of elapsed time the system spends in the waking state equals the wait's relative timeout. If the timeout specified by *pftTimeout* is absolute, the time that the system spends in sleep or hibernation does count toward the expiration of the wait. If the wait expires while the system is sleeping, the wait is signaled immediately when the system wakes.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8 [desktop apps | UWP apps]
Minimum supported server	Windows Server 2012 [desktop apps | UWP apps]
Target Platform	Windows
Header	threadpoolapiset.h
Library	Kernel32.lib
DLL	Kernel32.dll

See also

CloseThreadpoolWait

CreateThreadpoolWait

Thread Pools

WaitForThreadpoolWaitCallbacks

SetThreadPriority function (processthreadsapi.h)

09/23/2022

Sets the priority value for the specified thread. This value, together with the priority class of the thread's process, determines the thread's base priority level.

Syntax

```C++
BOOL SetThreadPriority(
  [in] HANDLE hThread,
  [in] int    nPriority
);
```

Parameters

`[in] hThread`

A handle to the thread whose priority value is to be set.

The handle must have the **THREAD_SET_INFORMATION** or **THREAD_SET_LIMITED_INFORMATION** access right. For more information, see Thread Security and Access Rights.**Windows Server 2003:** The handle must have the **THREAD_SET_INFORMATION** access right.

`[in] nPriority`

The priority value for the thread. This parameter can be one of the following values.

Expand table

Priority	Meaning
THREAD_MODE_BACKGROUND_BEGIN 0x00010000	Begin background processing mode. The system lowers the resource scheduling priorities of the thread so that it can perform background work without significantly affecting activity in the foreground. This value can be specified only if *hThread* is a handle to the current thread. The function fails if the thread is already in background processing mode.

	Windows Server 2003: This value is not supported.
THREAD_MODE_BACKGROUND_END 0x00020000	End background processing mode. The system restores the resource scheduling priorities of the thread as they were before the thread entered background processing mode. This value can be specified only if *hThread* is a handle to the current thread. The function fails if the thread is not in background processing mode. **Windows Server 2003:** This value is not supported.
THREAD_PRIORITY_ABOVE_NORMAL 1	Priority 1 point above the priority class.
THREAD_PRIORITY_BELOW_NORMAL -1	Priority 1 point below the priority class.
THREAD_PRIORITY_HIGHEST 2	Priority 2 points above the priority class.
THREAD_PRIORITY_IDLE -15	Base priority of 1 for **IDLE_PRIORITY_CLASS**, **BELOW_NORMAL_PRIORITY_CLASS**, **NORMAL_PRIORITY_CLASS**, **ABOVE_NORMAL_PRIORITY_CLASS**, or **HIGH_PRIORITY_CLASS** processes, and a base priority of 16 for **REALTIME_PRIORITY_CLASS** processes.
THREAD_PRIORITY_LOWEST -2	Priority 2 points below the priority class.
THREAD_PRIORITY_NORMAL 0	Normal priority for the priority class.
THREAD_PRIORITY_TIME_CRITICAL 15	Base priority of 15 for **IDLE_PRIORITY_CLASS**, **BELOW_NORMAL_PRIORITY_CLASS**, **NORMAL_PRIORITY_CLASS**, **ABOVE_NORMAL_PRIORITY_CLASS**, or **HIGH_PRIORITY_CLASS** processes, and a base priority of 31 for **REALTIME_PRIORITY_CLASS** processes.

If the thread has the **REALTIME_PRIORITY_CLASS** base class, this parameter can also be -7, -6, -5, -4, -3, 3, 4, 5, or 6. For more information, see Scheduling Priorities.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Windows Phone 8.1: Windows Phone Store apps may call this function but it has no effect. The function will return a nonzero value indicating success.

Remarks

Every thread has a base priority level determined by the thread's priority value and the priority class of its process. The system uses the base priority level of all executable threads to determine which thread gets the next slice of CPU time. Threads are scheduled in a round-robin fashion at each priority level, and only when there are no executable threads at a higher level does scheduling of threads at a lower level take place.

The **SetThreadPriority** function enables setting the base priority level of a thread relative to the priority class of its process. For example, specifying **THREAD_PRIORITY_HIGHEST** in a call to **SetThreadPriority** for a thread of an **IDLE_PRIORITY_CLASS** process sets the thread's base priority level to 6. For a table that shows the base priority levels for each combination of priority class and thread priority value, see Scheduling Priorities.

For **IDLE_PRIORITY_CLASS**, **BELOW_NORMAL_PRIORITY_CLASS**, **NORMAL_PRIORITY_CLASS**, **ABOVE_NORMAL_PRIORITY_CLASS**, and **HIGH_PRIORITY_CLASS** processes, the system dynamically boosts a thread's base priority level when events occur that are important to the thread. **REALTIME_PRIORITY_CLASS** processes do not receive dynamic boosts.

All threads initially start at **THREAD_PRIORITY_NORMAL**. Use the GetPriorityClass and SetPriorityClass functions to get and set the priority class of a process. Use the GetThreadPriority function to get the priority value of a thread.

Use the priority class of a process to differentiate between applications that are time critical and those that have normal or below normal scheduling requirements. Use thread priority values to differentiate the relative priorities of the tasks of a process. For example, a thread that handles input for a window could have a higher priority level than a thread that performs intensive calculations for the CPU.

When manipulating priorities, be very careful to ensure that a high-priority thread does not consume all of the available CPU time. A thread with a base priority level above 11 interferes with the normal operation of the operating system. Using **REALTIME_PRIORITY_CLASS** may cause disk caches to not flush, cause the mouse to stop responding, and so on.

The **THREAD_PRIORITY_*** values affect the CPU scheduling priority of the thread. For threads that perform background work such as file I/O, network I/O, or data processing, it is not sufficient to adjust the CPU scheduling priority; even an idle CPU priority thread can easily

interfere with system responsiveness when it uses the disk and memory. Threads that perform background work should use the **THREAD_MODE_BACKGROUND_BEGIN** and **THREAD_MODE_BACKGROUND_END** values to adjust their resource scheduling priorities; threads that interact with the user should not use **THREAD_MODE_BACKGROUND_BEGIN**.

When a thread is in background processing mode, it should minimize sharing resources such as critical sections, heaps, and handles with other threads in the process, otherwise priority inversions can occur. If there are threads executing at high priority, a thread in background processing mode may not be scheduled promptly, but it will never be starved.

Windows Server 2008 and Windows Vista: While the system is starting, the **SetThreadPriority** function returns a success return value but does not change thread priority for applications that are started from the system Startup folder or listed in the **HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run** registry key. These applications run at reduced priority for a short time (approximately 60 seconds) to make the system more responsive to user actions during startup.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps.

Windows Phone 8.1: Windows Phone Store apps may call this function but it has no effect.

Examples

The following example demonstrates the use of thread background mode.

C++

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

int main( void )
{
   DWORD dwError, dwThreadPri;

   if(!SetThreadPriority(GetCurrentThread(), THREAD_MODE_BACKGROUND_BEGIN))
   {
      dwError = GetLastError();
      if( ERROR_THREAD_MODE_ALREADY_BACKGROUND == dwError)
         _tprintf(TEXT("Already in background mode\n"));
      else _tprintf(TEXT("Failed to enter background mode (%d)\n"), dwError);
      goto Cleanup;
   }

   // Display thread priority

   dwThreadPri = GetThreadPriority(GetCurrentThread());

   _tprintf(TEXT("Current thread priority is 0x%x\n"), dwThreadPri);
```

```
    //
    // Perform background work
    //
    ;

    if(!SetThreadPriority(GetCurrentThread(), THREAD_MODE_BACKGROUND_END))
    {
       _tprintf(TEXT("Failed to end background mode (%d)\n"), GetLastError());
    }

 Cleanup:
    // Clean up
    ;
 return 0;
 }
```

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	processthreadsapi.h (include Windows.h on Windows Vista, Windows 7, Windows Server 2008 Windows Server 2008 R2)
Library	Kernel32.lib; WindowsPhoneCore.lib on Windows Phone 8.1
DLL	Kernel32.dll; KernelBase.dll on Windows Phone 8.1

See also

GetPriorityClass

GetThreadPriority

Process and Thread Functions

Scheduling Priorities

SetPriorityClass

Threads

SetThreadPriorityBoost function (processthreadsapi.h)

11/01/2022

Disables or enables the ability of the system to temporarily boost the priority of a thread.

Syntax

```
C++

BOOL SetThreadPriorityBoost(
  [in] HANDLE hThread,
  [in] BOOL   bDisablePriorityBoost
);
```

Parameters

`[in] hThread`

A handle to the thread whose priority is to be boosted. The handle must have the **THREAD_SET_INFORMATION** or **THREAD_SET_LIMITED_INFORMATION** access right. For more information, see Thread Security and Access Rights.

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

`[in] bDisablePriorityBoost`

If this parameter is **TRUE**, dynamic boosting is disabled. If the parameter is **FALSE**, dynamic boosting is enabled.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

When a thread is running in one of the dynamic priority classes, the system temporarily boosts the thread's priority when it is taken out of a wait state. If **SetThreadPriorityBoost** is called with the *DisablePriorityBoost* parameter set to **TRUE**, the thread's priority is not boosted. To restore normal behavior, call **SetThreadPriorityBoost** with *DisablePriorityBoost* set to **FALSE**.

Requirements

Expand table

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

See also

GetThreadPriorityBoost

OpenThread

Priority Boosts

Process and Thread Functions

Scheduling Priorities

Threads

SetThreadSelectedCpuSets function (processthreadsapi.h)

02/22/2024

Sets the selected CPU Sets assignment for the specified thread. This assignment overrides the process default assignment, if one is set.

Syntax

C++

```
BOOL SetThreadSelectedCpuSets(
  HANDLE      Thread,
  const ULONG *CpuSetIds,
  ULONG       CpuSetIdCount
);
```

Parameters

`Thread`

Specifies the thread on which to set the CPU Set assignment. This handle must have the THREAD_SET_LIMITED_INFORMATION access right. The value returned by **GetCurrentThread** can also be used.

`CpuSetIds`

Specifies the list of CPU Set IDs to set as the thread selected CPU set. If this is NULL, the API clears out any assignment, reverting to process default assignment if one is set.

`CpuSetIdCount`

Specifies the number of IDs in the list passed in the **CpuSetIds** argument. If that value is NULL, this should be 0.

Return value

If the function succeeds, the return value is nonzero.

This function cannot fail when passed valid parameters.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	processthreadsapi.h
DLL	kernel32.dll

SetThreadStackGuarantee function (processthreadsapi.h)

02/06/2024

Sets the minimum size of the stack associated with the calling thread or fiber that will be available during any stack overflow exceptions. This is useful for handling stack overflow exceptions; the application can safely use the specified number of bytes during exception handling.

Syntax

```C++
BOOL SetThreadStackGuarantee(
  [in, out] PULONG StackSizeInBytes
);
```

Parameters

`[in, out] StackSizeInBytes`

The size of the stack, in bytes. On return, this value is set to the size of the previous stack, in bytes.

If this parameter is 0 (zero), the function succeeds and the parameter contains the size of the current stack.

If the specified size is less than the current size, the function succeeds but ignores this request. Therefore, you cannot use this function to reduce the size of the stack.

This value cannot be larger than the reserved stack size.

Return value

If the function succeeds, the return value is nonzero.

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

Remarks

If the function is successful, the application can handle possible EXCEPTION_STACK_OVERFLOW exceptions using structured exception handling. To resume execution after handling a stack overflow, you must perform certain recovery steps. If you are using the Microsoft C/C++ compiler, call the **_resetstkoflw** function. If you are using another compiler, see the documentation for the compiler for information on recovering from stack overflows.

To set the stack guarantee for a fiber, you must first call the SwitchToFiber function to execute the fiber. After you set the guarantee for this fiber, it is used by the fiber no matter which thread executes the fiber.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista, Windows XP Professional x64 Edition [desktop apps only]
Minimum supported server	Windows Server 2008, Windows Server 2003 with SP1 [desktop apps only]
Target Platform	Windows
Header	processthreadsapi.h (include Windows.h on Windows Server 2003, Windows Vista, Windows 7, Windows Server 2008 Windows Server 2008 R2)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

Process and Thread Functions

Thread Stack Size

Threads

Vertdll APIs available in VBS enclaves

SetUmsThreadInformation function (winbase.h)

Article03/03/2023

Sets application-specific context information for the specified user-mode scheduling (UMS) worker thread.

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

Syntax

```C++
BOOL SetUmsThreadInformation(
  [in] PUMS_CONTEXT          UmsThread,
  [in] UMS_THREAD_INFO_CLASS UmsThreadInfoClass,
  [in] PVOID                 UmsThreadInformation,
  [in] ULONG                 UmsThreadInformationLength
);
```

Parameters

`[in] UmsThread`

A pointer to a UMS thread context.

`[in] UmsThreadInfoClass`

A UMS_THREAD_INFO_CLASS value that specifies the kind of information to set. This parameter must be **UmsThreadUserContext**.

`[in] UmsThreadInformation`

A pointer to a buffer that contains the information to set.

`[in] UmsThreadInformationLength`

The size of the *UmsThreadInformation* buffer, in bytes.

Return value

If the function succeeds, it returns a nonzero value.

If the function fails, the return value is zero. To get extended error information, call GetLastError. Possible error values include the following.

Expand table

Return code	Description
ERROR_INFO_LENGTH_MISMATCH	The buffer size does not match the required size for the specified information class.
ERROR_INVALID_INFO_CLASS	The *UmsThreadInfoClass* parameter specifies an information class that is not supported.
ERROR_NOT_SUPPORTED	UMS is not supported.

Remarks

The **SetUmsThreadInformation** function can be used to set an application-defined context for the specified UMS worker thread. The context information can consist of anything the application might find useful to track, such as per-scheduler or per-worker thread state. The underlying structures for UMS worker threads are managed by the system and should not be modified directly.

The QueryUmsThreadInformation function can be used to retrieve other exposed information about the specified thread, such as its thread execution block (TEB) and whether the thread is suspended or terminated. Information that is not exposed through **QueryUmsThreadInformation** should be considered reserved.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 (64-bit only) [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows

Requirement	Value
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
API set	api-ms-win-core-ums-l1-1-0 (introduced in Windows 7)

See also

QueryUmsThreadInformation

SimpleCallback callback function

Article04/23/2018

Applications implement this callback if they call the TrySubmitThreadpoolCallback function to start a worker thread.

The **PTP_SIMPLE_CALLBACK** type defines a pointer to this callback function. *SimpleCallback* is a placeholder for the application-defined function name.

Syntax

```c++
VOID CALLBACK SimpleCallback(
  _Inout_     PTP_CALLBACK_INSTANCE Instance,
  _Inout_opt_ PVOID                 Context
);
```

Parameters

- *Instance* [in, out]
  A **TP_CALLBACK_INSTANCE** structure that defines the callback instance. Applications do not modify the members of this structure.

  This structure can be passed to one of the following functions:
  - CallbackMayRunLong
  - DisassociateCurrentThreadFromCallback
  - FreeLibraryWhenCallbackReturns
  - LeaveCriticalSectionWhenCallbackReturns
  - ReleaseMutexWhenCallbackReturns
  - ReleaseSemaphoreWhenCallbackReturns
  - SetEventWhenCallbackReturns

- *Context* [in, out, optional]
  The application-defined data.

Return value

This function does not return a value.

Requirements

Expand table

Minimum supported client	Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps | UWP apps]
Header	WinBase.h on Windows Vista, Windows 7, Windows Server 2008 and Windows Server 2008 R2 (include Windows.h); Threadpoolapiset.h on Windows 8 and Windows Server 2012

See also

TrySubmitThreadpoolCallback

Sleep function (synchapi.h)

09/23/2022

Suspends the execution of the current thread until the time-out interval elapses.

To enter an alertable wait state, use the SleepEx function.

Syntax

```
C++

VOID Sleep(
  [in] DWORD dwMilliseconds
);
```

Parameters

`[in] dwMilliseconds`

The time interval for which execution is to be suspended, in milliseconds.

A value of zero causes the thread to relinquish the remainder of its time slice to any other thread that is ready to run. If there are no other threads ready to run, the function returns immediately, and the thread continues execution. **Windows XP:** A value of zero causes the thread to relinquish the remainder of its time slice to any other thread of equal priority that is ready to run. If there are no other threads of equal priority ready to run, the function returns immediately, and the thread continues execution. This behavior changed starting with Windows Server 2003.

A value of INFINITE indicates that the suspension should not time out.

Return value

None

Remarks

This function causes a thread to relinquish the remainder of its time slice and become unrunnable for an interval based on the value of *dwMilliseconds*. The system clock "ticks" at a constant rate. If *dwMilliseconds* is less than the resolution of the system clock, the thread may

sleep for less than the specified length of time. If *dwMilliseconds* is greater than one tick but less than two, the wait can be anywhere between one and two ticks, and so on. To increase the accuracy of the sleep interval, call the **timeGetDevCaps** function to determine the supported minimum timer resolution and the **timeBeginPeriod** function to set the timer resolution to its minimum. Use caution when calling **timeBeginPeriod**, as frequent calls can significantly affect the system clock, system power usage, and the scheduler. If you call **timeBeginPeriod**, call it one time early in the application and be sure to call the **timeEndPeriod** function at the very end of the application.

After the sleep interval has passed, the thread is ready to run. If you specify 0 milliseconds, the thread will relinquish the remainder of its time slice but remain ready. Note that a ready thread is not guaranteed to run immediately. Consequently, the thread may not run until some time after the sleep interval elapses. For more information, see Scheduling Priorities.

Be careful when using **Sleep** in the following scenarios:

- Code that directly or indirectly creates windows (for example, DDE and COM **CoInitialize**). If a thread creates any windows, it must process messages. Message broadcasts are sent to all windows in the system. If you have a thread that uses **Sleep** with infinite delay, the system will deadlock.
- Threads that are under concurrency control. For example, an I/O completion port or thread pool limits the number of associated threads that can run. If the maximum number of threads is already running, no additional associated thread can run until a running thread finishes. If a thread uses **Sleep** with an interval of zero to wait for one of the additional associated threads to accomplish some work, the process might deadlock.

For these scenarios, use MsgWaitForMultipleObjects or MsgWaitForMultipleObjectsEx, rather than **Sleep**.

Windows Phone 8.1: This function is supported for Windows Phone Store apps on Windows Phone 8.1 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Examples

For an example, see Using Thread Local Storage.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	synchapi.h (include Windows.h on Windows Vista, Windows 7, Windows Server 2008 Windows Server 2008 R2)
Library	Kernel32.lib; WindowsPhoneCore.lib on Windows Phone 8.1
DLL	Kernel32.dll; KernelBase.dll on Windows Phone 8.1

See also

MsgWaitForMultipleObjects

MsgWaitForMultipleObjectsEx

Process and Thread Functions

SleepEx

Suspending Thread Execution

Threads

WaitOnAddress

SleepEx function (synchapi.h)

Article09/23/2022

Suspends the current thread until the specified condition is met. Execution resumes when one of the following occurs:

- An I/O completion callback function is called.
- An asynchronous procedure call (APC) is queued to the thread.
- The time-out interval elapses.

Syntax

```C++
DWORD SleepEx(
  [in] DWORD dwMilliseconds,
  [in] BOOL  bAlertable
);
```

Parameters

`[in] dwMilliseconds`

The time interval for which execution is to be suspended, in milliseconds.

A value of zero causes the thread to relinquish the remainder of its time slice to any other thread that is ready to run. If there are no other threads ready to run, the function returns immediately, and the thread continues execution.**Windows XP:** A value of zero causes the thread to relinquish the remainder of its time slice to any other thread of equal priority that is ready to run. If there are no other threads of equal priority ready to run, the function returns immediately, and the thread continues execution. This behavior changed starting with Windows Server 2003.

A value of INFINITE indicates that the suspension should not time out.

`[in] bAlertable`

If this parameter is FALSE, the function does not return until the time-out period has elapsed. If an I/O completion callback occurs, the function does not immediately return and the I/O completion function is not executed. If an APC is queued to the thread, the function does not immediately return and the APC function is not executed.

If the parameter is TRUE and the thread that called this function is the same thread that called the extended I/O function (ReadFileEx or WriteFileEx), the function returns when either the time-out period has elapsed or when an I/O completion callback function occurs. If an I/O completion callback occurs, the I/O completion function is called. If an APC is queued to the thread (QueueUserAPC), the function returns when either the time-out period has elapsed or when the APC function is called.

Return value

The return value is zero if the specified time interval expired.

The return value is **WAIT_IO_COMPLETION** if the function returned due to one or more I/O completion callback functions. This can happen only if *bAlertable* is TRUE, and if the thread that called the **SleepEx** function is the same thread that called the extended I/O function.

Remarks

This function causes a thread to relinquish the remainder of its time slice and become unrunnable for an interval based on the value of *dwMilliseconds*. After the sleep interval has passed, the thread is ready to run. Note that a ready thread is not guaranteed to run immediately. Consequently, the thread will not run until some arbitrary time after the sleep interval elapses, based upon the system "tick" frequency and the load factor from other processes. The system clock "ticks" at a constant rate. To increase the accuracy of the sleep interval, call the **timeGetDevCaps** function to determine the supported minimum timer resolution and the **timeBeginPeriod** function to set the timer resolution to its minimum. Use caution when calling **timeBeginPeriod**, as frequent calls can significantly affect the system clock, system power usage, and the scheduler. If you call **timeBeginPeriod**, call it one time early in the application and be sure to call the **timeEndPeriod** function at the very end of the application. If you specify 0 milliseconds, the thread will relinquish the remainder of its time slice but remain ready. For more information, see Scheduling Priorities.

This function can be used with the ReadFileEx or WriteFileEx functions to suspend a thread until an I/O operation has been completed. These functions specify a completion routine that is to be executed when the I/O operation has been completed. For the completion routine to be executed, the thread that called the I/O function must be in an alertable wait state when the completion callback function occurs. A thread goes into an alertable wait state by calling either **SleepEx**, MsgWaitForMultipleObjectsEx, WaitForSingleObjectEx, or WaitForMultipleObjectsEx, with the function's *bAlertable* parameter set to TRUE.

Be careful when using **SleepEx** in the following scenarios:

- Code that directly or indirectly creates windows (for example, DDE and COM **CoInitialize**). If a thread creates any windows, it must process messages. Message broadcasts are sent to all windows in the system. If you have a thread that uses **SleepEx** with infinite delay, the system will deadlock.
- Threads that are under concurrency control. For example, an I/O completion port or thread pool limits the number of associated threads that can run. If the maximum number of threads is already running, no additional associated thread can run until a running thread finishes. If a thread uses **SleepEx** with an interval of zero to wait for one of the additional associated threads to accomplish some work, the process might deadlock.

For these scenarios, use MsgWaitForMultipleObjects or MsgWaitForMultipleObjectsEx, rather than **SleepEx**.

Windows Phone 8.1: This function is supported for Windows Phone Store apps on Windows Phone 8.1 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	synchapi.h (include Windows.h on Windows Vista, Windows 7, Windows Server 2008 Windows Server 2008 R2)
Library	Kernel32.lib; WindowsPhoneCore.lib on Windows Phone 8.1
DLL	KernelBase.dll on Windows Phone 8.1; Kernel32.dll

See also

MsgWaitForMultipleObjects

MsgWaitForMultipleObjectsEx

Process and Thread Functions

QueueUserAPC

ReadFileEx

Sleep

Suspending Thread Execution

Threads

WaitForMultipleObjectsEx

WaitForSingleObjectEx

WaitOnAddress

WriteFileEx

StartThreadpoolIo function (threadpoolapiset.h)

09/23/2022

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.

Syntax

```
C++

VOID StartThreadpoolIo(
  [in, out] PTP_IO pio
);
```

Parameters

`[in, out] pio`

A pointer to a **TP_IO** structure that defines the I/O completion object. The CreateThreadpoolIo function returns this pointer.

Return value

None

Remarks

You must call this function before initiating each asynchronous I/O operation on the file handle bound to the I/O completion object. Failure to do so will cause the thread pool to ignore an I/O operation when it completes and will cause memory corruption.

If the I/O operation fails, call the CancelThreadpoolIo function to cancel this notification.

If the file handle bound to the I/O completion object has the notification mode FILE_SKIP_COMPLETION_PORT_ON_SUCCESS and an asynchronous I/O operation returns immediately with success, the object's I/O completion callback function is not called and threadpool I/O notifications must be canceled. For more information, see CancelThreadpoolIo.

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

Requirements

Expand table

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

See also

CancelThreadpoolIo

CloseThreadpoolIo

CreateThreadpoolIo

Thread Pools

WaitForThreadpoolIoCallbacks

SubmitThreadpoolWork function (threadpoolapiset.h)

02/22/2024

Posts a work object to the thread pool. A worker thread calls the work object's callback function.

Syntax

```C++
VOID SubmitThreadpoolWork(
  [in, out] PTP_WORK pwk
);
```

Parameters

`[in, out] pwk`

A pointer to a **TP_WORK** structure that defines the work object. The CreateThreadpoolWork function returns this pointer.

Return value

None

Remarks

You can post a work object one or more times (up to MAXULONG) without waiting for prior callbacks to complete. The callbacks will execute in parallel. To improve efficiency, the thread pool may throttle the threads.

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

Examples

For an example, see Using the Thread Pool Functions.

Requirements

Expand table

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

See also

CloseThreadpoolWork

CreateThreadpoolWork

Thread Pools

WaitForThreadpoolWorkCallbacks

SuspendThread function (processthreadsapi.h)

02/22/2024

Suspends the specified thread.

> ⓘ **Note**
>
> A 64-bit application can suspend a WOW64 thread thread using the Wow64SuspendThread function.

Syntax

```C++
DWORD SuspendThread(
  [in] HANDLE hThread
);
```

Parameters

`[in] hThread`

A handle to the thread that is to be suspended.

The handle must have the **THREAD_SUSPEND_RESUME** access right. For more information, see Thread Security and Access Rights.

Return value

If the function succeeds, the return value is the thread's previous suspend count; otherwise, it is `(DWORD) -1`. To get extended error information, use the GetLastError function.

Remarks

If the function succeeds, execution of the specified thread is suspended and the thread's suspend count is incremented. Suspending a thread causes the thread to stop executing user-mode (application) code.

This function is primarily designed for use by debuggers. It is not intended to be used for thread synchronization. Calling **SuspendThread** on a thread that owns a synchronization object, such as a mutex or critical section, can lead to a deadlock if the calling thread tries to obtain a synchronization object owned by a suspended thread. To avoid this situation, a thread within an application that is not a debugger should signal the other thread to suspend itself. The target thread must be designed to watch for this signal and respond appropriately.

Each thread has a suspend count (with a maximum value of **MAXIMUM_SUSPEND_COUNT**). If the suspend count is greater than zero, the thread is suspended; otherwise, the thread is not suspended and is eligible for execution. Calling **SuspendThread** causes the target thread's suspend count to be incremented. Attempting to increment past the maximum suspend count causes an error without incrementing the count.

The ResumeThread function decrements the suspend count of a suspended thread.

Windows Phone 8.1: This function is supported for Windows Phone Store apps on Windows Phone 8.1 and later.

Windows 8.1 and **Windows Server 2012 R2**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and later.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	processthreadsapi.h (include Windows.h on Windows Vista, Windows 7, Windows Server 2008 Windows Server 2008 R2)
Library	Kernel32.lib; WindowsPhoneCore.lib on Windows Phone 8.1
DLL	Kernel32.dll; KernelBase.dll on Windows Phone 8.1

See also

OpenThread

Process and Thread Functions

ResumeThread

Suspending Thread Execution

Threads

SwitchToFiber function (winbase.h)

02/22/2024

Schedules a fiber. The function must be called on a fiber.

Syntax

```
C++

VOID SwitchToFiber(
  [in] LPVOID lpFiber
);
```

Parameters

`[in] lpFiber`

The address of the fiber to be scheduled.

Return value

None

Remarks

You create fibers with the CreateFiber function. Before you can schedule fibers associated with a thread, you must call ConvertThreadToFiber to set up an area in which to save the fiber state information. The thread is now the currently executing fiber.

The **SwitchToFiber** function saves the state information of the current fiber and restores the state of the specified fiber. 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.

Avoid making the following call:

```
syntax

SwitchToFiber( GetCurrentFiber() );
```

This call can cause unpredictable problems.

To compile an application that uses this function, define _WIN32_WINNT as 0x0400 or later. For more information, see Using the Windows Headers.

Requirements

Expand table

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

See also

ConvertThreadToFiber

CreateFiber

Fibers

Process and Thread Functions

SwitchToThread function (processthreadsapi.h)

02/22/2024

Causes the calling thread to yield execution to another thread that is ready to run on the current processor. The operating system selects the next thread to be executed.

Syntax

```C++
BOOL SwitchToThread();
```

Return value

If calling the **SwitchToThread** function caused the operating system to switch execution to another thread, the return value is nonzero.

If there are no other threads ready to execute, the operating system does not switch execution to another thread, and the return value is zero.

Remarks

The yield of execution is in effect for up to one thread-scheduling time slice on the processor of the calling thread. The operating system will not switch execution to another processor, even if that processor is idle or is running a thread of lower priority.

After the yielding thread's time slice elapses, the operating system reschedules execution for the yielding thread. The rescheduling is determined by the priority of the yielding thread and the status of other threads that are available to run.

Note that the operating system will not switch to a thread that is being prevented from running only by concurrency control. For example, an I/O completion port or thread pool limits the number of associated threads that can run. If the maximum number of threads is already running, no additional associated thread can run until a running thread finishes. If a thread uses **SwitchToThread** to wait for one of the additional associated threads to accomplish some work, the process might deadlock.

To compile an application that uses this function, define _WIN32_WINNT as 0x0400 or later. For more information, see Using the Windows Headers.

Requirements

Expand table

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

See also

Process and Thread Functions

SuspendThread

Suspending Thread Execution

Threads

TerminateJobObject function (jobapi2.h)

Article10/13/2021

Terminates all processes currently associated with the job. If the job is nested, this function terminates all processes currently associated with the job and all of its child jobs in the hierarchy.

Syntax

```
C++

BOOL TerminateJobObject(
  [in] HANDLE hJob,
  [in] UINT   uExitCode
);
```

Parameters

`[in] hJob`

A handle to the job whose processes will be terminated. The CreateJobObject or OpenJobObject function returns this handle. This handle must have the JOB_OBJECT_TERMINATE access right. For more information, see Job Object Security and Access Rights.

The handle for each process in the job object must have the PROCESS_TERMINATE access right. For more information, see Process Security and Access Rights.

`[in] uExitCode`

The exit code to be used by all processes and threads in the job object. Use the GetExitCodeProcess function to retrieve each process's exit value. Use the GetExitCodeThread function to retrieve each thread's exit value.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

It is not possible for any of the processes associated with the job to postpone or handle the termination. It is as if TerminateProcess were called for each process associated with the job.

Terminating a nested job additionally terminates all child job objects. Resources used by the terminated jobs are charged up the parent job chain in the hierarchy.

To compile an application that uses this function, define _WIN32_WINNT as 0x0500 or later. For more information, see Using the Windows Headers.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	jobapi2.h (include Windows.h, Jobapi2.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

CreateJobObject

Job Objects

OpenJobObject

Process and Thread Functions

TerminateProcess

TerminateProcess function (processthreadsapi.h)

02/06/2024

Terminates the specified process and all of its threads.

Syntax

```
C++

BOOL TerminateProcess(
  [in] HANDLE hProcess,
  [in] UINT   uExitCode
);
```

Parameters

`[in] hProcess`

A handle to the process to be terminated.

The handle must have the **PROCESS_TERMINATE** access right. For more information, see Process Security and Access Rights.

`[in] uExitCode`

The exit code to be used by the process and threads terminated as a result of this call. Use the GetExitCodeProcess function to retrieve a process's exit value. Use the GetExitCodeThread function to retrieve a thread's exit value.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The **TerminateProcess** function is used to unconditionally cause a process to exit. The state of global data maintained by dynamic-link libraries (DLLs) may be compromised if **TerminateProcess** is used rather than ExitProcess.

This function stops execution of all threads within the process and requests cancellation of all pending I/O. The terminated process cannot exit until all pending I/O has been completed or canceled. When a process terminates, its kernel object is not destroyed until all processes that have open handles to the process have released those handles.

When a process terminates itself, **TerminateProcess** stops execution of the calling thread and does not return. Otherwise, **TerminateProcess** is asynchronous; it initiates termination and returns immediately. If you need to be sure the process has terminated, call the WaitForSingleObject function with a handle to the process.

A process cannot prevent itself from being terminated.

After a process has terminated, call to **TerminateProcess** with open handles to the process fails with **ERROR_ACCESS_DENIED** (5) error code.

Requirements

Expand table

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

See also

ExitProcess

GetExitCodeProcess

GetExitCodeThread

OpenProcess

Process and Thread Functions

Processes

Terminating a Process

Vertdll APIs available in VBS enclaves

TerminateThread function (processthreadsapi.h)

02/22/2024

Terminates a thread.

Syntax

```
C++

BOOL TerminateThread(
  [in, out] HANDLE hThread,
  [in]      DWORD  dwExitCode
);
```

Parameters

`[in, out] hThread`

A handle to the thread to be terminated.

The handle must have the **THREAD_TERMINATE** access right. For more information, see Thread Security and Access Rights.

`[in] dwExitCode`

The exit code for the thread. Use the GetExitCodeThread function to retrieve a thread's exit value.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

TerminateThread is used to cause a thread to exit. When this occurs, the target thread has no chance to execute any user-mode code. DLLs attached to the thread are not notified that the

thread is terminating. The system frees the thread's initial stack.

Windows Server 2003 and Windows XP: The target thread's initial stack is not freed, causing a resource leak.

TerminateThread is a dangerous function that should only be used in the most extreme cases. You should call **TerminateThread** only if you know exactly what the target thread is doing, and you control all of the code that the target thread could possibly be running at the time of the termination. For example, **TerminateThread** can result in the following problems:

- If the target thread owns a critical section, the critical section will not be released.
- If the target thread is allocating memory from the heap, the heap lock will not be released.
- If the target thread is executing certain kernel32 calls when it is terminated, the kernel32 state for the thread's process could be inconsistent.
- If the target thread is manipulating the global state of a shared DLL, the state of the DLL could be destroyed, affecting other users of the DLL.

A thread cannot protect itself against **TerminateThread**, other than by controlling access to its handles. The thread handle returned by the CreateThread and CreateProcess functions has **THREAD_TERMINATE** access, so any caller holding one of these handles can terminate your thread.

If the target thread is the last thread of a process when this function is called, the thread's process is also terminated.

The state of the thread object becomes signaled, releasing any other threads that had been waiting for the thread to terminate. The thread's termination status changes from **STILL_ACTIVE** to the value of the *dwExitCode* parameter.

Terminating a thread does not necessarily remove the thread object from the system. A thread object is deleted when the last thread handle is closed.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]

Requirement	Value
Target Platform	Windows
Header	processthreadsapi.h (include Windows.h on Windows Server 2003, Windows Vista, Windows 7, Windows Server 2008 Windows Server 2008 R2)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

CreateProcess

CreateThread

ExitThread

GetExitCodeThread

OpenThread

Process and Thread Functions

Terminating a Thread

Threads

ThreadProc callback function

Article04/23/2018

An application-defined function that serves as the starting address for a thread. Specify this address when calling the CreateThread, CreateRemoteThread, or CreateRemoteThreadEx function.

The **LPTHREAD_START_ROUTINE** type defines a pointer to this callback function. **ThreadProc** is a placeholder for the application-defined function name.

Syntax

```c++
DWORD WINAPI ThreadProc(
  _In_ LPVOID lpParameter
);
```

Parameters

- *lpParameter* [in]
  The thread data passed to the function using the *lpParameter* parameter of the CreateThread, CreateRemoteThread, or CreateRemoteThreadEx function.

Return value

The return value indicates the success or failure of this function. The return value should never be set to STILL_ACTIVE (259), as noted in GetExitCodeThread.

Do not declare this callback function with a **void** return type and cast the function pointer to **LPTHREAD_START_ROUTINE** when creating the thread. Code that does this is common, but it can crash on 64-bit Windows.

Remarks

A process can determine when a thread it created has completed by using one of the wait functions. It can also obtain the return value of its **ThreadProc** by calling the GetExitCodeThread function.

Each thread receives a unique copy of the local variables of this function. Any static or global variables are shared by all threads in the process. To provide unique data to each thread using a global index, use thread local storage.

Examples

For an example, see Creating Threads.

Requirements

Expand table

Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	WinBase.h on Windows XP, Windows Server 2003, Windows Vista, Windows 7, Windows Server 2008 and Windows Server 2008 R2 (include Windows.h); Processthreadsapi.h on Windows 8 and Windows Server 2012

See also

CreateThread

CreateRemoteThread

CreateRemoteThreadEx

GetExitCodeThread

TimerCallback callback function

Article04/23/2018

Applications implement this callback if they call the SetThreadpoolTimer function to start a worker thread for the timer object.

The **PTP_TIMER_CALLBACK** type defines a pointer to this callback function. *TimerCallback* is a placeholder for the application-defined function name.

Syntax

```c++
VOID CALLBACK TimerCallback(
  _Inout_     PTP_CALLBACK_INSTANCE Instance,
  _Inout_opt_ PVOID                 Context,
  _Inout_     PTP_TIMER             Timer
);
```

Parameters

- *Instance* [in, out]
  A **TP_CALLBACK_INSTANCE** structure that defines the callback instance. Applications do not modify the members of this structure.

  This structure can be passed to one of the following functions:
  - CallbackMayRunLong
  - DisassociateCurrentThreadFromCallback
  - FreeLibraryWhenCallbackReturns
  - LeaveCriticalSectionWhenCallbackReturns
  - ReleaseMutexWhenCallbackReturns
  - ReleaseSemaphoreWhenCallbackReturns
  - SetEventWhenCallbackReturns

- *Context* [in, out, optional]
  The application-defined data.

- *Timer* [in, out]
  A **TP_TIMER** structure that defines the timer object that generated the callback.

Return value

This function does not return a value.

Requirements

Expand table

Minimum supported client	Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps | UWP apps]
Header	WinBase.h on Windows Vista, Windows 7, Windows Server 2008 and Windows Server 2008 R2 (include Windows.h); Threadpoolapiset.h on Windows 8 and Windows Server 2012

See also

SetThreadpoolTimer

TlsAlloc function (processthreadsapi.h)

12/05/2024

Allocates a thread local storage (TLS) index. Any thread of the process can subsequently use this index to store and retrieve values that are local to the thread, because each thread receives its own slot for the index.

Syntax

```
C++

DWORD TlsAlloc();
```

Return value

If the function succeeds, the return value is a TLS index. The slots for the index are initialized to zero.

If the function fails, the return value is **TLS_OUT_OF_INDEXES**. To get extended error information, call GetLastError.

Remarks

Windows Phone 8.1: This function is supported for Windows Phone Store apps on Windows Phone 8.1 and later. When a Windows Phone Store app calls this function, it is replaced with an inline call to **FlsAlloc**. Refer to FlsAlloc for function documentation.

Windows 8.1, **Windows Server 2012 R2**, and **Windows 10, version 1507**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and Windows 10, version 1507. When a Windows Store app calls this function, it is replaced with an inline call to **FlsAlloc**. Refer to FlsAlloc for function documentation.

Windows 10, version 1511 and **Windows 10, version 1607**: This function is fully supported for Universal Windows Platform (UWP) apps, and is no longer replaced with an inline call to **FlsAlloc**.

The threads of the process can use the TLS index in subsequent calls to the TlsFree, TlsSetValue, or TlsGetValue functions. The value of the TLS index should be treated as an opaque value; do not assume that it is an index into a zero-based array.

TLS indexes are typically allocated during process or dynamic-link library (DLL) initialization. When a TLS index is allocated, its storage slots are initialized to **NULL**. After a TLS index has been allocated, each thread of the process can use it to access its own TLS storage slot. To store a value in its TLS slot, a thread specifies the index in a call to TlsSetValue. The thread specifies the same index in a subsequent call to TlsGetValue, to retrieve the stored value.

TLS indexes are not valid across process boundaries. A DLL cannot assume that an index assigned in one process is valid in another process.

Examples

For an example, see Using Thread Local Storage or Using Thread Local Storage in a Dynamic-Link Library.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	processthreadsapi.h (include Windows.h on Windows Vista, Windows 7, Windows Server 2008 Windows Server 2008 R2)
Library	Kernel32.lib; WindowsPhoneCore.lib on Windows Phone 8.1
DLL	KernelBase.dll on Windows Phone 8.1; Kernel32.dll

See also

Process and Thread Functions

Thread Local Storage

TlsFree

TlsGetValue

TlsSetValue

Vertdll APIs available in VBS enclaves

TlsFree function (processthreadsapi.h)

02/06/2024

Releases a thread local storage (TLS) index, making it available for reuse.

Syntax

```
C++

BOOL TlsFree(
  [in] DWORD dwTlsIndex
);
```

Parameters

`[in] dwTlsIndex`

The TLS index that was allocated by the TlsAlloc function.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

Windows Phone 8.1: This function is supported for Windows Phone Store apps on Windows Phone 8.1 and later. When a Windows Phone Store app calls this function, it is replaced with an inline call to **FlsFree**. Refer to FlsFree for function documentation.

Windows 8.1, **Windows Server 2012 R2**, and **Windows 10, version 1507**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and Windows 10, version 1507. When a Windows Store app calls this function, it is replaced with an inline call to **FlsFree**. Refer to FlsFree for function documentation.

Windows 10, version 1511 and **Windows 10, version 1607**: This function is fully supported for Universal Windows Platform (UWP) apps, and is no longer replaced with an inline call to **FlsFree**.

If the threads of the process have allocated memory and stored a pointer to the memory in a TLS slot, they should free the memory before calling **TlsFree**. The **TlsFree** function does not free memory blocks whose addresses have been stored in the TLS slots associated with the TLS index. It is expected that DLLs call this function (if at all) only during **DLL_PROCESS_DETACH**.

For more information, see Thread Local Storage.

Examples

For an example, see Using Thread Local Storage or Using Thread Local Storage in a Dynamic-Link Library.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	processthreadsapi.h (include Windows.h on Windows Vista, Windows 7, Windows Server 2008 Windows Server 2008 R2)
Library	Kernel32.lib; WindowsPhoneCore.lib on Windows Phone 8.1
DLL	KernelBase.dll on Windows Phone 8.1; Kernel32.dll

See also

Processes and Threads Overview

Thread Local Storage

TlsAlloc

TlsGetValue

TlsSetValue

Vertdll APIs available in VBS enclaves

TlsGetValue function (processthreadsapi.h)

11/18/2024

Retrieves the value in the calling thread's thread local storage (TLS) slot for the specified TLS index. Each thread of a process has its own slot for each TLS index.

Syntax

```
C++

LPVOID TlsGetValue(
  [in] DWORD dwTlsIndex
);
```

Parameters

`[in] dwTlsIndex`

The TLS index that was allocated by the TlsAlloc function.

Return value

If the function succeeds, the return value is the value stored in the calling thread's TLS slot associated with the specified index. If *dwTlsIndex* is a valid index allocated by a successful call to TlsAlloc, this function always succeeds.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

The data stored in a TLS slot can have a value of 0 because it still has its initial value or because the thread called the TlsSetValue function with 0. Therefore, if the return value is 0, you must check whether GetLastError returns **ERROR_SUCCESS** before determining that the function has failed. If **GetLastError** returns **ERROR_SUCCESS**, then the function has succeeded and the data stored in the TLS slot is 0. Otherwise, the function has failed.

Functions that return indications of failure call SetLastError when they fail. They generally do not call **SetLastError** when they succeed. The **TlsGetValue** function is an exception to this general rule. The **TlsGetValue** function calls **SetLastError** to clear a thread's last error when it succeeds. That allows checking for the error-free retrieval of zero values.

Remarks

Windows 8.1, **Windows Server 2012 R2**, and **Windows 10, version 1507**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and Windows 10, version 1507. When a Windows Store app calls this function, it is replaced with an inline call to **FlsGetValue**. Refer to FlsGetValue for function documentation.

Windows 10, version 1511 and **Windows 10, version 1607**: This function is fully supported for Universal Windows Platform (UWP) apps, and is no longer replaced with an inline call to **FlsGetValue**.

TLS indexes are typically allocated by the TlsAlloc function during process or DLL initialization. After a TLS index is allocated, each thread of the process can use it to access its own TLS slot for that index. A thread specifies a TLS index in a call to TlsSetValue to store a value in its slot. The thread specifies the same index in a subsequent call to **TlsGetValue** to retrieve the stored value.

TlsGetValue was implemented with speed as the primary goal. The function performs minimal parameter validation and error checking. In particular, it succeeds if *dwTlsIndex* is in the range 0 through (**TLS_MINIMUM_AVAILABLE**– 1). It is up to the programmer to ensure that the index is valid and that the thread calls TlsSetValue before calling **TlsGetValue**.

TlsGetValue always sets a thread's last error. In some cases, an application (such as those with custom heaps that support malloc) may need to call **GetLastError** before calling **TlsGetValue** to save the thread's last error (followed by **SetLastError** to restore the saved error). Unfortunately, this can incur a non-trivial performance cost on certain CPUs.

Windows 11 24H2 and later: Use the **TlsGetValue2** function, which is identical to **TlsGetValue** except that it doesn't set the thread's last error. Applications calling **TlsGetValue2** should avoid using 0 as a valid value because **GetLastError** cannot be called to check if **TlsGetValue2** failed.

Examples

For an example, see Using Thread Local Storage or Using Thread Local Storage in a Dynamic-Link Library.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	processthreadsapi.h (include Windows.h on Windows Vista, Windows 7, Windows Server 2008 Windows Server 2008 R2)
Library	Kernel32.lib; WindowsPhoneCore.lib on Windows Phone 8.1
DLL	KernelBase.dll on Windows Phone 8.1; Kernel32.dll

See also

TlsGetValue2

Process and Thread Functions

Thread Local Storage

TlsAlloc

TlsFree

TlsSetValue

Vertdll APIs available in VBS enclaves

TlsSetValue function (processthreadsapi.h)

02/22/2024

Stores a value in the calling thread's thread local storage (TLS) slot for the specified TLS index. Each thread of a process has its own slot for each TLS index.

Syntax

```
C++

BOOL TlsSetValue(
  [in]           DWORD  dwTlsIndex,
  [in, optional] LPVOID lpTlsValue
);
```

Parameters

`[in] dwTlsIndex`

The TLS index that was allocated by the TlsAlloc function.

`[in, optional] lpTlsValue`

The value to be stored in the calling thread's TLS slot for the index.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

Windows Phone 8.1: This function is supported for Windows Phone Store apps on Windows Phone 8.1 and later. When a Windows Phone Store app calls this function, it is replaced with an inline call to **FlsSetValue**. Refer to FlsSetValue for function documentation.

Windows 8.1, **Windows Server 2012 R2**, and **Windows 10, version 1507**: This function is supported for Windows Store apps on Windows 8.1, Windows Server 2012 R2, and

Windows 10, version 1507. When a Windows Store app calls this function, it is replaced with an inline call to **FlsSetValue**. Refer to FlsSetValue for function documentation.

Windows 10, version 1511 and **Windows 10, version 1607**: This function is fully supported for Universal Windows Platform (UWP) apps, and is no longer replaced with an inline call to **FlsSetValue**.

TLS indexes are typically allocated by the TlsAlloc function during process or DLL initialization. When a TLS index is allocated, its storage slots are initialized to NULL. After a TLS index is allocated, each thread of the process can use it to access its own TLS slot for that index. A thread specifies a TLS index in a call to **TlsSetValue**, to store a value in its slot. The thread specifies the same index in a subsequent call to TlsGetValue, to retrieve the stored value.

TlsSetValue was implemented with speed as the primary goal. The function performs minimal parameter validation and error checking. In particular, it succeeds if *dwTlsIndex* is in the range 0 through (**TLS_MINIMUM_AVAILABLE** – 1). It is up to the programmer to ensure that the index is valid before calling TlsGetValue.

Examples

For an example, see Using Thread Local Storage or Using Thread Local Storage in a Dynamic-Link Library.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	processthreadsapi.h (include Windows.h on Windows Vista, Windows 7, Windows Server 2008 Windows Server 2008 R2)
Library	Kernel32.lib; WindowsPhoneCore.lib on Windows Phone 8.1
DLL	KernelBase.dll on Windows Phone 8.1; Kernel32.dll

See also

Process and Thread Functions

Thread Local Storage

TlsAlloc

TlsFree

TlsGetValue

Vertdll APIs available in VBS enclaves

TpInitializeCallbackEnviron function (winnt.h)

02/22/2024

Initializes a callback environment for the thread pool.

Syntax

```
C++

VOID TpInitializeCallbackEnviron(
  [out] PTP_CALLBACK_ENVIRON CallbackEnviron
);
```

Parameters

`[out] CallbackEnviron`

A **TP_CALLBACK_ENVIRON** structure that defines the callback environment. Allocate space for this structure and initialize it using this function.

Return value

None

Remarks

The thread pool callback environment is subject to default behaviors that can be changed. For example, callbacks execute in the global pool by default, but a different thread pool can be specified using TpSetCallbackThreadpool. Thread pool callback environment behavior can be changed with the following functions:

- TpSetCallbackActivationContext
- TpSetCallbackCleanupGroup
- TpSetCallbackFinalizationCallback
- TpSetCallbackLongFunction
- TpSetCallbackNoActivationContext
- TpSetCallbackPersistent
- TpSetCallbackPriority

- TpSetCallbackRaceWithDll
- TpSetCallbackThreadpool

Call **TpInitializeCallbackEnviron** to create a callback environment that can be modified. Call TpDestroyCallbackEnviron to destroy the callback environment.

This function is implemented as an inline function.

Requirements

Expand table

Requirement	Value
Target Platform	Windows
Header	winnt.h

TpDestroyCallbackEnviron function (winnt.h)

02/22/2024

Deletes the specified callback environment. Call this function when the callback environment is no longer needed for creating new thread pool objects.

Syntax

```C++
VOID TpDestroyCallbackEnviron(
  [in] PTP_CALLBACK_ENVIRON CallbackEnviron
);
```

Parameters

`[in] CallbackEnviron`

A **TP_CALLBACK_ENVIRON** structure that defines the callback environment. The TpInitializeCallbackEnviron function returns this structure.

Return value

None

Requirements

Expand table

Requirement	Value
Target Platform	Windows
Header	winnt.h

See also

TpInitializeCallbackEnviron

TpSetCallbackActivationContext

TpSetCallbackCleanupGroup

TpSetCallbackFinalizationCallback

TpSetCallbackLongFunction

TpSetCallbackNoActivationContext

TpSetCallbackPersistent

TpSetCallbackPriority

TpSetCallbackRaceWithDll

TpSetCallbackThreadpool

TpSetCallbackActivationContext function (winnt.h)

02/22/2024

Assigns an activation context to the callback environment.

Syntax

```
C++

VOID TpSetCallbackActivationContext(
  [in, out]      PTP_CALLBACK_ENVIRON CallbackEnviron,
  [in, optional] _ACTIVATION_CONTEXT  *ActivationContext
);
```

Parameters

`[in, out] CallbackEnviron`

A **TP_CALLBACK_ENVIRON** structure that defines the callback environment. The TpInitializeCallbackEnviron function returns this structure.

`[in, optional] ActivationContext`

Pointer to an **_ACTIVATION_CONTEXT** structure.

Return value

None

Remarks

This function is implemented as an inline function.

Requirements

Expand table

Requirement	Value
Target Platform	Windows
Header	winnt.h

See also

TpDestroyCallbackEnviron

TpInitializeCallbackEnviron

TpSetCallbackCleanupGroup

TpSetCallbackFinalizationCallback

TpSetCallbackLongFunction

TpSetCallbackNoActivationContext

TpSetCallbackPersistent

TpSetCallbackPriority

TpSetCallbackRaceWithDll

TpSetCallbackThreadpool

TpSetCallbackCleanupGroup function (winnt.h)

02/22/2024

Associates the specified cleanup group with the specified callback environment.

Syntax

```C++
VOID TpSetCallbackCleanupGroup(
  [in, out]      PTP_CALLBACK_ENVIRON              CallbackEnviron,
  [in]           PTP_CLEANUP_GROUP                 CleanupGroup,
  [in, optional] PTP_CLEANUP_GROUP_CANCEL_CALLBACK CleanupGroupCancelCallback
);
```

Parameters

`[in, out] CallbackEnviron`

A **TP_CALLBACK_ENVIRON** structure that defines the callback environment. The TpInitializeCallbackEnviron function returns this structure.

`[in] CleanupGroup`

A **TP_CLEANUP_GROUP** structure that defines the cleanup group. The CreateThreadpoolCleanupGroup function returns this structure.

`[in, optional] CleanupGroupCancelCallback`

The cleanup callback to be called if the cleanup group is canceled before the associated object is released. The function is called when you call CloseThreadpoolCleanupGroupMembers.

Return value

None

Remarks

This function is implemented as an inline function.

Requirements

Expand table

Requirement	Value
Target Platform	Windows
Header	winnt.h

See also

TpDestroyCallbackEnviron

TpInitializeCallbackEnviron

TpSetCallbackActivationContext

TpSetCallbackFinalizationCallback

TpSetCallbackLongFunction

TpSetCallbackNoActivationContext

TpSetCallbackPersistent

TpSetCallbackPriority

TpSetCallbackRaceWithDll

TpSetCallbackThreadpool

TpSetCallbackFinalizationCallback function (winnt.h)

02/22/2024

Indicates a function to call when the callback environment is finalized.

Syntax

```
C++

VOID TpSetCallbackFinalizationCallback(
  [in, out] PTP_CALLBACK_ENVIRON CallbackEnviron,
  [in]      PTP_SIMPLE_CALLBACK  FinalizationCallback
);
```

Parameters

`[in, out] CallbackEnviron`

A **TP_CALLBACK_ENVIRON** structure that defines the callback environment. The TpInitializeCallbackEnviron function returns this structure.

`[in] FinalizationCallback`

Pointer to a **TP_SIMPLE_CALLBACK** structure indicating a function to call when the callback environment is finalized.

Return value

None

Remarks

This function is implemented as an inline function.

Requirements

Expand table

Requirement	Value
Target Platform	Windows
Header	winnt.h

See also

TpDestroyCallbackEnviron

TpInitializeCallbackEnviron

TpSetCallbackActivationContext

TpSetCallbackCleanupGroup

TpSetCallbackLongFunction

TpSetCallbackNoActivationContext

TpSetCallbackPersistent

TpSetCallbackPriority

TpSetCallbackRaceWithDll

TpSetCallbackThreadpool

TpSetCallbackLongFunction function (winnt.h)

02/22/2024

Indicates that callbacks associated with this callback environment may not return quickly.

Syntax

```
C++

VOID TpSetCallbackLongFunction(
  [in, out] PTP_CALLBACK_ENVIRON CallbackEnviron
);
```

Parameters

`[in, out] CallbackEnviron`

A **TP_CALLBACK_ENVIRON** structure that defines the callback environment. The TpInitializeCallbackEnviron function returns this structure.

Return value

None

Remarks

The thread pool may use this information to better determine when a new thread should be created.

This function is implemented as an inline function.

Requirements

Expand table

Requirement	Value
Target Platform	Windows
Header	winnt.h

See also

TpDestroyCallbackEnviron

TpInitializeCallbackEnviron

TpSetCallbackActivationContext

TpSetCallbackCleanupGroup

TpSetCallbackFinalizationCallback

TpSetCallbackNoActivationContext

TpSetCallbackPersistent

TpSetCallbackPriority

TpSetCallbackRaceWithDll

TpSetCallbackThreadpool

TpSetCallbackNoActivationContext function (winnt.h)

02/22/2024

Indicates that the callback environment has no activation context.

Syntax

```
C++

VOID TpSetCallbackNoActivationContext(
  [in, out] PTP_CALLBACK_ENVIRON CallbackEnviron
);
```

Parameters

`[in, out] CallbackEnviron`

A **TP_CALLBACK_ENVIRON** structure that defines the callback environment. The TpInitializeCallbackEnviron function returns this structure.

Return value

None

Remarks

This function is implemented as an inline function.

Requirements

Expand table

Requirement	Value
Target Platform	Windows
Header	winnt.h

See also

TpDestroyCallbackEnviron

TpInitializeCallbackEnviron

TpSetCallbackActivationContext

TpSetCallbackCleanupGroup

TpSetCallbackFinalizationCallback

TpSetCallbackLongFunction

TpSetCallbackPersistent

TpSetCallbackPriority

TpSetCallbackRaceWithDll

TpSetCallbackThreadpool

TpSetCallbackPersistent function (winnt.h)

02/22/2024

Specifies that the callback should run on a persistent thread.

Syntax

```
C++

VOID TpSetCallbackPersistent(
  [in, out] PTP_CALLBACK_ENVIRON CallbackEnviron
);
```

Parameters

`[in, out] CallbackEnviron`

A **TP_CALLBACK_ENVIRON** structure that defines the callback environment. The TpInitializeCallbackEnviron function returns this structure.

Return value

None

Remarks

This function is implemented as an inline function.

Requirements

Expand table

Requirement	Value
Target Platform	Windows
Header	winnt.h

See also

TpDestroyCallbackEnviron

TpInitializeCallbackEnviron

TpSetCallbackActivationContext

TpSetCallbackCleanupGroup

TpSetCallbackFinalizationCallback

TpSetCallbackLongFunction

TpSetCallbackNoActivationContext

TpSetCallbackPriority

TpSetCallbackRaceWithDll

TpSetCallbackThreadpool

TpSetCallbackPriority function (winnt.h)

02/22/2024

Specifies the priority of a callback function relative to other work items in the same thread pool.

Syntax

```
C++

VOID TpSetCallbackPriority(
  [in, out] PTP_CALLBACK_ENVIRON CallbackEnviron,
  [in]      TP_CALLBACK_PRIORITY Priority
);
```

Parameters

`[in, out] CallbackEnviron`

A **TP_CALLBACK_ENVIRON** structure that defines the callback environment. The TpInitializeCallbackEnviron function returns this structure.

`[in] Priority`

The priority for the callback relative to other callbacks in the same thread pool. This parameter can be one of the following **TP_CALLBACK_PRIORITY** enumeration values:

Expand table

Value	Meaning
TP_CALLBACK_PRIORITY_HIGH	The callback should run at high priority.
TP_CALLBACK_PRIORITY_LOW	The callback should run at low priority.
TP_CALLBACK_PRIORITY_NORMAL	The callback should run at normal priority.

Return value

None

Remarks

Higher priority callbacks are guaranteed to be run first by the first available worker thread, but they are not guaranteed to finish before lower priority callbacks.

This function is implemented as an inline function.

Requirements

Expand table

Requirement	Value
Target Platform	Windows
Header	winnt.h

See also

TpDestroyCallbackEnviron

TpInitializeCallbackEnviron

TpSetCallbackActivationContext

TpSetCallbackCleanupGroup

TpSetCallbackFinalizationCallback

TpSetCallbackLongFunction

TpSetCallbackNoActivationContext

TpSetCallbackPersistent

TpSetCallbackRaceWithDll

TpSetCallbackThreadpool

TpSetCallbackRaceWithDll function (winnt.h)

02/22/2024

Ensures that the specified DLL remains loaded as long as there are outstanding callbacks.

Syntax

```C++
VOID TpSetCallbackRaceWithDll(
  [in, out] PTP_CALLBACK_ENVIRON CallbackEnviron,
  [in]      PVOID                DllHandle
);
```

Parameters

`[in, out] CallbackEnviron`

A **TP_CALLBACK_ENVIRON** structure that defines the callback environment. The InitializeThreadpoolEnvironment function returns this structure.

`[in] DllHandle`

A handle to the DLL.

Return value

None

Remarks

You should call this function if a callback might acquire the loader lock. This prevents a deadlock from occurring when one thread in DllMain is waiting for the callback to end, and another thread that is executing the callback attempts to acquire the loader lock.

If the DLL containing the callback might be unloaded, the cleanup code in DllMain must cancel outstanding callbacks before releasing the object.

Managing callbacks created with a TP_CALLBACK_ENVIRON that specifies a callback library is somewhat processing-intensive. You should consider other options for ensuring that the library is not unloaded while callbacks are executing, or to guarantee that callbacks which may be executing do not acquire the loader lock.

This function is implemented as an inline function.

Requirements

Expand table

Requirement	Value
Target Platform	Windows
Header	winnt.h

See also

TpDestroyCallbackEnviron

TpInitializeCallbackEnviron

TpSetCallbackActivationContext

TpSetCallbackCleanupGroup

TpSetCallbackFinalizationCallback

TpSetCallbackLongFunction

TpSetCallbackNoActivationContext

TpSetCallbackPersistent

TpSetCallbackPriority

TpSetCallbackThreadpool

TpSetCallbackThreadpool function (winnt.h)

02/22/2024

Assigns a thread pool to a callback environment.

Syntax

```C++
VOID TpSetCallbackThreadpool(
  [in, out] PTP_CALLBACK_ENVIRON CallbackEnviron,
  [in]      PTP_POOL             Pool
);
```

Parameters

`[in, out] CallbackEnviron`

A **TP_CALLBACK_ENVIRON** structure that defines the callback environment. The TpInitializeCallbackEnviron function returns this structure.

`[in] Pool`

A **TP_POOL** structure that defines a thread pool. The CreateThreadpool function returns this structure.

Return value

None

Remarks

If you do not specify a thread pool, the global thread pool is used.

This function is implemented as an inline function.

Requirements

Expand table

Requirement	Value
Target Platform	Windows
Header	winnt.h

See also

TpDestroyCallbackEnviron

TpInitializeCallbackEnviron

TpSetCallbackActivationContext

TpSetCallbackCleanupGroup

TpSetCallbackFinalizationCallback

TpSetCallbackLongFunction

TpSetCallbackNoActivationContext

TpSetCallbackPersistent

TpSetCallbackPriority

TpSetCallbackRaceWithDll

TrySubmitThreadpoolCallback function (threadpoolapiset.h)

Article09/23/2022

Requests that a thread pool worker thread call the specified callback function.

Syntax

```C++
BOOL TrySubmitThreadpoolCallback(
  [in]                PTP_SIMPLE_CALLBACK  pfns,
  [in, out, optional] PVOID                pv,
  [in, optional]      PTP_CALLBACK_ENVIRON pcbe
);
```

Parameters

`[in] pfns`

The callback function. For details, see SimpleCallback.

`[in, out, optional] pv`

Optional application-defined data to pass to the callback function.

`[in, optional] pcbe`

A pointer to a **TP_CALLBACK_ENVIRON** structure that defines the environment in which to execute the callback function. Use the InitializeThreadpoolEnvironment function to initialize the structure before calling this function.

If this parameter is NULL, the callback executes in the default callback environment. For more information, see InitializeThreadpoolEnvironment.

Return value

If the function succeeds, it returns TRUE.

If the function fails, it returns FALSE. To retrieve extended error information, call GetLastError.

Remarks

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

Requirements

Expand table

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

See also

CallbackMayRunLong

DisassociateCurrentThreadFromCallback

FreeLibraryWhenCallbackReturns

InitializeThreadpoolEnvironment

LeaveCriticalSectionWhenCallbackReturns

ReleaseMutexWhenCallbackReturns

ReleaseSemaphoreWhenCallbackReturns

SetEventWhenCallbackReturns

Thread Pools

RTL_UMS_SCHEDULER_ENTRY_POINT callback function (winnt.h)

10/13/2021

The application-defined user-mode scheduling (UMS) scheduler entry point function associated with a UMS completion list.

The **PUMS_SCHEDULER_ENTRY_POINT** type defines a pointer to this function. *UmsSchedulerProc* is a placeholder for the application-defined function name.

Syntax

```C++
RTL_UMS_SCHEDULER_ENTRY_POINT RtlUmsSchedulerEntryPoint;

VOID RtlUmsSchedulerEntryPoint(
  [in] RTL_UMS_SCHEDULER_REASON Reason,
  [in] ULONG_PTR ActivationPayload,
  [in] PVOID SchedulerParam
)
{...}
```

Parameters

`[in] Reason`

The reason the scheduler entry point is being called. This parameter can be one of the following values.

Expand table

Value	Meaning
UmsSchedulerStartup 0	A UMS scheduler thread was created. The entry point is called with this reason once each time EnterUmsSchedulingMode is called.
UmsSchedulerThreadBlocked 1	A UMS worker thread blocked.
UmsSchedulerThreadYield 2	An executing UMS worker thread yielded control by calling the UmsThreadYield function.

`[in] ActivationPayload`

If the *Reason* parameter is **UmsSchedulerStartup**, this parameter is NULL.

If the *Reason* parameter is **UmsSchedulerThreadBlocked**, bit 0 of this parameter indicates the type of activity that was being serviced when the UMS worker thread blocked.

Expand table

Value	Meaning
0	The thread blocked on a trap (for example, a hard page fault) or an interrupt (for example, an asynchronous procedure call).
1	The thread blocked on a system call.

If the *Reason* parameter is **UmsSchedulerThreadYield**, this parameter is a pointer to the UMS thread context of the UMS worker thread that yielded.

`[in] SchedulerParam`

If the *Reason* parameter is **UmsSchedulerStartup**, this parameter is the **SchedulerParam** member of the UMS_SCHEDULER_STARTUP_INFO structure passed to the EnterUmsSchedulingMode function that triggered the entry point call.

If the *Reason* parameter is **UmsSchedulerThreadYield** this parameter is the SchedulerParam parameter passed to the UmsThreadYield function that triggered the entry point call.

If the *Reason* parameter is **UmsSchedulerThreadBlocked**, this parameter is NULL.

Return value

None

Remarks

The *UmsSchedulerProc* function pointer type is defined as **PUMS_SCHEDULER_ENTRY_POINT** in WinBase.h. The underlying function type is defined as **RTL_UMS_SCHEDULER_ENTRY_POINT** in WinNT.h

Each UMS scheduler thread has an associated *UmsSchedulerProc* entry point function that is specified when the thread calls the EnterUmsSchedulingMode function. The system calls the

scheduler entry point function with a reason of **UmsSchedulerStartup** when the scheduler thread is converted for UMS.

Subsequently, when a UMS worker thread that is running on the scheduler thread yields or blocks, the system calls the scheduler thread's entry point function with a pointer to the UMS thread context of the worker thread.

The application's scheduler is responsible for selecting the next UMS worker thread to run. The scheduler implements all policies that influence execution of its UMS threads, including processor affinity and thread priority. For example, a scheduler might give priority to I/O-intensive threads, or it might run threads on a first-come, first-served basis. This logic can be implemented in the scheduler entry point function or elsewhere in the application.

When a blocked UMS worker thread becomes unblocked, the system queues the unblocked thread to the associated completion list and signals the completion list event. To retrieve UMS worker threads from the completion list, use the DequeueUmsCompletionListItems function.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 (64-bit only) [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winnt.h (include WinBase.h, Windows.h)

See also

DequeueUmsCompletionListItems

EnterUmsSchedulingMode

UmsThreadYield

UmsThreadYield function (winbase.h)

Article03/18/2022

Yields control to the user-mode scheduling (UMS) scheduler thread on which the calling UMS worker thread is running.

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

Syntax

```C++
BOOL UmsThreadYield(
  [in] PVOID SchedulerParam
);
```

Parameters

`[in] SchedulerParam`

A parameter to pass to the scheduler thread's UmsSchedulerProc function.

Return value

If the function succeeds, it returns a nonzero value.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

A UMS worker thread calls the **UmsThreadYield** function to cooperatively yield control to the UMS scheduler thread on which the worker thread is running. If a UMS worker thread never calls **UmsThreadYield**, the worker thread runs until it either blocks or is terminated.

When control switches to the UMS scheduler thread, the system calls the associated scheduler entry point function with the reason **UmsSchedulerThreadYield** and the *ScheduleParam*

parameter specified by the worker thread in the **UmsThreadYield** call.

The application's scheduler is responsible for rescheduling the worker thread.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 (64-bit only) [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
API set	api-ms-win-core-ums-l1-1-0 (introduced in Windows 7)

See also

UmsSchedulerProc

UpdateProcThreadAttribute function (processthreadsapi.h)

11/01/2022

Updates the specified attribute in a list of attributes for process and thread creation.

Syntax

```
C++

BOOL UpdateProcThreadAttribute(
  [in, out]       LPPROC_THREAD_ATTRIBUTE_LIST lpAttributeList,
  [in]            DWORD                        dwFlags,
  [in]            DWORD_PTR                    Attribute,
  [in]            PVOID                        lpValue,
  [in]            SIZE_T                       cbSize,
  [out, optional] PVOID                        lpPreviousValue,
  [in, optional]  PSIZE_T                      lpReturnSize
);
```

Parameters

`[in, out] lpAttributeList`

A pointer to an attribute list created by the InitializeProcThreadAttributeList function.

`[in] dwFlags`

This parameter is reserved and must be zero.

`[in] Attribute`

The attribute key to update in the attribute list. This parameter can be one of the following values.

Expand table

Value	Meaning
PROC_THREAD_ATTRIBUTE_GROUP_AFFINITY	The *lpValue* parameter is a pointer to a GROUP_AFFINITY structure that specifies the processor group affinity for the new thread. Supported in Windows 7 and newer and Windows Server 2008 R2 and newer.
PROC_THREAD_ATTRIBUTE_HANDLE_LIST	The *lpValue* parameter is a pointer to a list of handles to be inherited by the child

	process. These handles must be created as inheritable handles and must not include pseudo handles such as those returned by the GetCurrentProcess or GetCurrentThread function. **Note** if you use this attribute, pass in a value of TRUE for the *bInheritHandles* parameter of the **CreateProcess** function.
PROC_THREAD_ATTRIBUTE_IDEAL_PROCESSOR	The *lpValue* parameter is a pointer to a PROCESSOR_NUMBER structure that specifies the ideal processor for the new thread. Supported in Windows 7 and newer and Windows Server 2008 R2 and newer.
PROC_THREAD_ATTRIBUTE_MACHINE_TYPE	The *lpValue* parameter is a pointer to a **WORD** that specifies the machine architecture of the child process. Supported in Windows 11 and newer. The **WORD** pointed to by *lpValue* can be a value listed on IMAGE FILE MACHINE CONSTANTS.
PROC_THREAD_ATTRIBUTE_MITIGATION_POLICY	The *lpValue* parameter is a pointer to a **DWORD** or **DWORD64** that specifies the exploit mitigation policy for the child process. Starting in Windows 10, version 1703, this parameter can also be a pointer to a two-element **DWORD64** array. The specified policy overrides the policies set for the application and the system and cannot be changed after the child process starts running. The **DWORD** or **DWORD64** pointed to by *lpValue* can be one or more of the values listed in the remarks. Supported in Windows 7 and newer and Windows Server 2008 R2 and newer.
PROC_THREAD_ATTRIBUTE_PARENT_PROCESS	The *lpValue* parameter is a pointer to the handle of a process to use (instead of the calling process) as the parent for the

	process being created. The handle for the process used must have the **PROCESS_CREATE_PROCESS** access right. Attributes inherited from the specified process include handles, the device map, processor affinity, priority, quotas, the process token, and job object. (Note that some attributes such as the debug port will come from the creating process, not the process specified by this handle.)
PROC_THREAD_ATTRIBUTE_PREFERRED_NODE	The *lpValue* parameter is a pointer to the node number of the preferred NUMA node for the new process. Supported in Windows 7 and newer and Windows Server 2008 R2 and newer.
PROC_THREAD_ATTRIBUTE_UMS_THREAD	The *lpValue* parameter is a pointer to a UMS_CREATE_THREAD_ATTRIBUTES structure that specifies a user-mode scheduling (UMS) thread context and a UMS completion list to associate with the thread. After the UMS thread is created, the system queues it to the specified completion list. The UMS thread runs only when an application's UMS scheduler retrieves the UMS thread from the completion list and selects it to run. For more information, see User-Mode Scheduling. Supported in Windows 7 and newer and Windows Server 2008 R2 and newer. Not supported in Windows 11 and newer (see User-Mode Scheduling).
PROC_THREAD_ATTRIBUTE_SECURITY_CAPABILITIES	The *lpValue* parameter is a pointer to a SECURITY_CAPABILITIES structure that defines the security capabilities of an app container. If this attribute is set the new process will be created as an AppContainer process. Supported in Windows 8 and newer and Windows Server 2012 and newer.
PROC_THREAD_ATTRIBUTE_PROTECTION_LEVEL	The *lpValue* parameter is a pointer to a **DWORD** value of **PROTECTION_LEVEL_SAME**. This specifies the protection level of the child process to

	be the same as the protection level of its parent process. Supported in Windows 8.1 and newer and Windows Server 2012 R2 and newer.
PROC_THREAD_ATTRIBUTE_CHILD_PROCESS_POLICY	The *lpValue* parameter is a pointer to a **DWORD** value that specifies the child process policy. The policy specifies whether to allow a child process to be created. For information on the possible values for the **DWORD** to which *lpValue* points, see Remarks. Supported in Windows 10 and newer and Windows Server 2016 and newer.
PROC_THREAD_ATTRIBUTE_DESKTOP_APP_POLICY	This attribute is relevant only to win32 applications that have been converted to UWP packages by using the Desktop Bridge. The *lpValue* parameter is a pointer to a **DWORD** value that specifies the desktop app policy. The policy specifies whether descendant processes should continue to run in the desktop environment. For information about the possible values for the **DWORD** to which *lpValue* points, see Remarks. Supported in Windows 10 Version 1703 and newer and Windows Server Version 1709 and newer.
PROC_THREAD_ATTRIBUTE_JOB_LIST	The *lpValue* parameter is a pointer to a list of job handles to be assigned to the child process, in the order specified. Supported in Windows 10 and newer and Windows Server 2016 and newer.
PROC_THREAD_ATTRIBUTE_ENABLE_OPTIONAL_XSTATE_FEATURES	The *lpValue* parameter is a pointer to a **DWORD64** value that specifies the set of optional XState features to enable for the new thread. Supported in Windows 11 and newer and Windows Server 2022 and newer.

`[in] lpValue`

A pointer to the attribute value. **This value must persist until the attribute list is destroyed using the DeleteProcThreadAttributeList function**.

`[in] cbSize`

The size of the attribute value specified by the *lpValue* parameter.

`[out, optional] lpPreviousValue`

This parameter is reserved and must be NULL.

`[in, optional] lpReturnSize`

This parameter is reserved and must be NULL.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

An attribute list is an opaque structure that consists of a series of key/value pairs, one for each attribute. A process can update only the attribute keys described in this topic.

The **DWORD** or **DWORD64** pointed to by *lpValue* can be one or more of the following values when you specify **PROC_THREAD_ATTRIBUTE_MITIGATION_POLICY** for the *Attribute* parameter:

PROCESS_CREATION_MITIGATION_POLICY_DEP_ENABLE (0x00000001)Enables data execution prevention (DEP) for the child process. For more information, see Data Execution Prevention.
PROCESS_CREATION_MITIGATION_POLICY_DEP_ATL_THUNK_ENABLE (0x00000002)Enables DEP-ATL thunk emulation for the child process. DEP-ATL thunk emulation causes the system to intercept NX faults that originate from the Active Template Library (ATL) thunk layer. This value can be specified only with PROCESS_CREATION_MITIGATION_POLICY_DEP_ENABLE.
PROCESS_CREATION_MITIGATION_POLICY_SEHOP_ENABLE (0x00000004)Enables structured exception handler overwrite protection (SEHOP) for the child process. SEHOP blocks exploits that use the structured exception handler (SEH) overwrite technique.
Windows 7, Windows Server 2008 R2, Windows Server 2008 and Windows Vista: The following values are not supported until Windows 8 and Windows Server 2012.
The force Address Space Layout Randomization (ASLR) policy, if enabled, forcibly rebases images that are not dynamic base compatible by acting as though an image base collision happened at load time. If relocations are required, images that do not have a base relocation section will not be loaded.

The following mitigation options are available for mandatory ASLR policy:

PROCESS_CREATION_MITIGATION_POLICY_FORCE_RELOCATE_IMAGES_ALWAYS_ON (0x00000001 << 8)
PROCESS_CREATION_MITIGATION_POLICY_FORCE_RELOCATE_IMAGES_ALWAYS_OFF (0x00000002 << 8)
PROCESS_CREATION_MITIGATION_POLICY_FORCE_RELOCATE_IMAGES_ALWAYS_ON_REQ_RELOCS (0x00000003 << 8)
The heap terminate on corruption policy, if enabled, causes the heap to terminate if it becomes corrupt. Note that 'always off' does not override the default opt-in for binaries with current subsystem versions set in the image header. Heap terminate on corruption is user mode enforced.

The following mitigation options are available for heap terminate on corruption policy:
PROCESS_CREATION_MITIGATION_POLICY_HEAP_TERMINATE_ALWAYS_ON (0x00000001 << 12)
PROCESS_CREATION_MITIGATION_POLICY_HEAP_TERMINATE_ALWAYS_OFF (0x00000002 << 12)
The bottom-up randomization policy, which includes stack randomization options, causes a random location to be used as the lowest user address.

The following mitigation options are available for the bottom-up randomization policy:
PROCESS_CREATION_MITIGATION_POLICY_BOTTOM_UP_ASLR_ALWAYS_ON (0x00000001 << 16)
PROCESS_CREATION_MITIGATION_POLICY_BOTTOM_UP_ASLR_ALWAYS_OFF (0x00000002 << 16)
The high-entropy bottom-up randomization policy, if enabled, causes up to 1TB of bottom-up variance to be used. Note that high-entropy bottom-up randomization is effective if and only if bottom-up ASLR is also enabled; high-entropy bottom-up randomization is only meaningful for native 64-bit processes.

The following mitigation options are available for the high-entropy bottom-up randomization policy:
PROCESS_CREATION_MITIGATION_POLICY_HIGH_ENTROPY_ASLR_ALWAYS_ON (0x00000001 << 20)
PROCESS_CREATION_MITIGATION_POLICY_HIGH_ENTROPY_ASLR_ALWAYS_OFF (0x00000002 << 20)
The strict handle checking enforcement policy, if enabled, causes an exception to be raised immediately on a bad handle reference. If this policy is not enabled, a failure status will be returned from the handle reference instead.

The following mitigation options are available for the strict handle checking enforcement policy:
PROCESS_CREATION_MITIGATION_POLICY_STRICT_HANDLE_CHECKS_ALWAYS_ON (0x00000001 << 24)
PROCESS_CREATION_MITIGATION_POLICY_STRICT_HANDLE_CHECKS_ALWAYS_OFF (0x00000002 << 24)
The Win32k system call disable policy, if enabled, prevents a process from making Win32k calls.

The following mitigation options are available for the Win32k system call disable policy:
PROCESS_CREATION_MITIGATION_POLICY_WIN32K_SYSTEM_CALL_DISABLE_ALWAYS_ON (0x00000001 << 28)
PROCESS_CREATION_MITIGATION_POLICY_WIN32K_SYSTEM_CALL_DISABLE_ALWAYS_OFF (0x00000002 << 28)
The Extension Point Disable policy, if enabled, prevents certain built-in third party extension points from being used. This policy blocks the following extension points:

- AppInit DLLs
- Winsock Layered Service Providers (LSPs)
- Global Windows Hooks
- Legacy Input Method Editors (IMEs)

Local hooks still work with the Extension Point Disable policy enabled. This behavior is used to prevent legacy extension points from being loaded into a process that does not use them.

The following mitigation options are available for the extension point disable policy:
PROCESS_CREATION_MITIGATION_POLICY_EXTENSION_POINT_DISABLE_ALWAYS_ON (0x00000001 << 32)
PROCESS_CREATION_MITIGATION_POLICY_EXTENSION_POINT_DISABLE_ALWAYS_OFF (0x00000002 << 32)
The Control Flow Guard (CFG) policy, if turned on, places additional restrictions on indirect calls in code that has been built with CFG enabled.

The following mitigation options are available for controlling the CFG policy:

- **PROCESS_CREATION_MITIGATION_POLICY_CONTROL_FLOW_GUARD_MASK** (0x00000003ui64 << 40)
- **PROCESS_CREATION_MITIGATION_POLICY_CONTROL_FLOW_GUARD_DEFER** (0x00000000ui64 << 40)
- **PROCESS_CREATION_MITIGATION_POLICY_CONTROL_FLOW_GUARD_ALWAYS_ON** (0x00000001ui64 << 40)
- **PROCESS_CREATION_MITIGATION_POLICY_CONTROL_FLOW_GUARD_ALWAYS_OFF** (0x00000002ui64 << 40)
- **PROCESS_CREATION_MITIGATION_POLICY_CONTROL_FLOW_GUARD_EXPORT_SUPPRESSION** (0x00000003ui64 << 40)

In addition, the following policy can be specified to enforce that EXEs/DLLs must enable CFG. If an attempt is made to load an EXE/DLL that does not enable CFG, the load will fail:

- **PROCESS_CREATION_MITIGATION_POLICY2_STRICT_CONTROL_FLOW_GUARD_MASK** (0x00000003ui64 << 8)
- **PROCESS_CREATION_MITIGATION_POLICY2_STRICT_CONTROL_FLOW_GUARD_DEFER** (0x00000000ui64 << 8)
- **PROCESS_CREATION_MITIGATION_POLICY2_STRICT_CONTROL_FLOW_GUARD_ALWAYS_ON** (0x00000001ui64 << 8)
- **PROCESS_CREATION_MITIGATION_POLICY2_STRICT_CONTROL_FLOW_GUARD_ALWAYS_OFF** (0x00000002ui64 << 8)
- **PROCESS_CREATION_MITIGATION_POLICY2_STRICT_CONTROL_FLOW_GUARD_RESERVED** (0x00000003ui64 << 8)

The dynamic code policy, if turned on, prevents a process from generating dynamic code or modifying executable code.

The following mitigation options are available for the dynamic code policy:

PROCESS_CREATION_MITIGATION_POLICY_PROHIBIT_DYNAMIC_CODE_MASK (0x00000003ui64 << 36)
PROCESS_CREATION_MITIGATION_POLICY_PROHIBIT_DYNAMIC_CODE_DEFER (0x00000000ui64 << 36)
PROCESS_CREATION_MITIGATION_POLICY_PROHIBIT_DYNAMIC_CODE_ALWAYS_ON (0x00000001ui64 << 36)
PROCESS_CREATION_MITIGATION_POLICY_PROHIBIT_DYNAMIC_CODE_ALWAYS_OFF (0x00000002ui64 << 36)
PROCESS_CREATION_MITIGATION_POLICY_PROHIBIT_DYNAMIC_CODE_ALWAYS_ON_ALLOW_OPT_OUT (0x00000003ui64 << 36)
The binary signature policy requires EXEs/DLLs to be properly signed.

The following mitigation options are available for the binary signature policy:

- **PROCESS_CREATION_MITIGATION_POLICY_BLOCK_NON_MICROSOFT_BINARIES_MASK** (0x00000003ui64 << 44)
- **PROCESS_CREATION_MITIGATION_POLICY_BLOCK_NON_MICROSOFT_BINARIES_DEFER** (0x00000000ui64 << 44)
- **PROCESS_CREATION_MITIGATION_POLICY_BLOCK_NON_MICROSOFT_BINARIES_ALWAYS_ON** (0x00000001ui64 << 44)
- **PROCESS_CREATION_MITIGATION_POLICY_BLOCK_NON_MICROSOFT_BINARIES_ALWAYS_OFF** (0x00000002ui64 << 44)
- **PROCESS_CREATION_MITIGATION_POLICY_BLOCK_NON_MICROSOFT_BINARIES_ALLOW_STORE** (0x00000003ui64 << 44)

The font loading prevention policy for the process determines whether non-system fonts can be loaded for a process. When the policy is turned on, the process is prevented from loading non-system fonts.

The following mitigation options are available for the font loading prevention policy:
PROCESS_CREATION_MITIGATION_POLICY_FONT_DISABLE_MASK (0x00000003ui64 << 48)
PROCESS_CREATION_MITIGATION_POLICY_FONT_DISABLE_DEFER (0x00000000ui64 << 48)
PROCESS_CREATION_MITIGATION_POLICY_FONT_DISABLE_ALWAYS_ON (0x00000001ui64 << 48)
PROCESS_CREATION_MITIGATION_POLICY_FONT_DISABLE_ALWAYS_OFF (0x00000002ui64 << 48)
PROCESS_CREATION_MITIGATION_POLICY_AUDIT_NONSYSTEM_FONTS (0x00000003ui64 << 48)
The image loading policy of the process determines the types of executable images that can be mapped into the process. When the policy is turned on, images cannot be loaded from some locations, such as remove devices or files that have the Low mandatory label.

The following mitigation options are available for the image loading policy:
PROCESS_CREATION_MITIGATION_POLICY_IMAGE_LOAD_NO_REMOTE_MASK (0x00000003ui64 << 52)
PROCESS_CREATION_MITIGATION_POLICY_IMAGE_LOAD_NO_REMOTE_DEFER (0x00000000ui64 << 52)

PROCESS_CREATION_MITIGATION_POLICY_IMAGE_LOAD_NO_REMOTE_ALWAYS_ON (0x00000001ui64 << 52)
PROCESS_CREATION_MITIGATION_POLICY_IMAGE_LOAD_NO_REMOTE_ALWAYS_OFF (0x00000002ui64 << 52)
PROCESS_CREATION_MITIGATION_POLICY_IMAGE_LOAD_NO_REMOTE_RESERVED (0x00000003ui64 << 52)
PROCESS_CREATION_MITIGATION_POLICY_IMAGE_LOAD_NO_LOW_LABEL_MASK (0x00000003ui64 << 56)
PROCESS_CREATION_MITIGATION_POLICY_IMAGE_LOAD_NO_LOW_LABEL_DEFER (0x00000000ui64 << 56)
PROCESS_CREATION_MITIGATION_POLICY_IMAGE_LOAD_NO_LOW_LABEL_ALWAYS_ON (0x00000001ui64 << 56)
PROCESS_CREATION_MITIGATION_POLICY_IMAGE_LOAD_NO_LOW_LABEL_ALWAYS_OFF (0x00000002ui64 << 56)
PROCESS_CREATION_MITIGATION_POLICY_IMAGE_LOAD_NO_LOW_LABEL_RESERVED (0x00000003ui64 << 56)
PROCESS_CREATION_MITIGATION_POLICY_IMAGE_LOAD_PREFER_SYSTEM32_MASK (0x00000003ui64 << 60)
PROCESS_CREATION_MITIGATION_POLICY_IMAGE_LOAD_PREFER_SYSTEM32_DEFER (0x00000000ui64 << 60)
PROCESS_CREATION_MITIGATION_POLICY_IMAGE_LOAD_PREFER_SYSTEM32_ALWAYS_ON (0x00000001ui64 << 60)
PROCESS_CREATION_MITIGATION_POLICY_IMAGE_LOAD_PREFER_SYSTEM32_ALWAYS_OFF (0x00000002ui64 << 60)
PROCESS_CREATION_MITIGATION_POLICY_IMAGE_LOAD_PREFER_SYSTEM32_RESERVED (0x00000003ui64 << 60)
Windows 10, version 1709: The following value is available only in Windows 10, version 1709 or later and only with the January 2018 Windows security updates and any applicable firmware updates from the OEM device manufacturer. See Windows Client Guidance for IT Pros to protect against speculative execution side-channel vulnerabilities.
PROCESS_CREATION_MITIGATION_POLICY2_RESTRICT_INDIRECT_BRANCH_PREDICTION_ALWAYS_ON (0x00000001ui64 << 16)This flag can be used by processes to protect against sibling hardware threads (hyperthreads) from interfering with indirect branch predictions. Processes that have sensitive information in their address space should consider enabling this flag to protect against attacks involving indirect branch prediction (such as CVE-2017-5715).
Windows 10, version 1809: The following value is available only in Windows 10, version 1809 or later.
PROCESS_CREATION_MITIGATION_POLICY2_SPECULATIVE_STORE_BYPASS_DISABLE_ALWAYS_ON (0x00000001ui64 << 24)This flag can be used by processes to disable the Speculative Store Bypass (SSB) feature of CPUs that may be vulnerable to speculative execution side channel attacks involving SSB (CVE-2018-3639). This flag is only supported by certain Intel CPUs that have the requisite hardware features. On CPUs that do not support this feature, the flag has no effect.

Windows 10, version 2004: The following values are available only in Windows 10, version 2004 or later.

Hardware-enforced Stack Protection (HSP) is a hardware-based security feature where the CPU verifies function return addresses at runtime by employing a shadow stack mechanism. For user-mode HSP, the default mode is compatibility mode, where only shadow stack violations occurring in modules that are considered compatible with shadow stacks (CETCOMPAT) are fatal. In strict mode, all shadow stack violations are fatal.

The following mitigation options are available for user-mode Hardware-enforced Stack Protection and related features:
PROCESS_CREATION_MITIGATION_POLICY2_CET_USER_SHADOW_STACKS_ALWAYS_ON
(0x00000001ui64 << 28)
PROCESS_CREATION_MITIGATION_POLICY2_CET_USER_SHADOW_STACKS_ALWAYS_OFF
(0x00000002ui64 << 28)
PROCESS_CREATION_MITIGATION_POLICY2_CET_USER_SHADOW_STACKS_STRICT_MODE
(0x00000003ui64 << 28)

Instruction Pointer validation:
PROCESS_CREATION_MITIGATION_POLICY2_USER_CET_SET_CONTEXT_IP_VALIDATION_ALWAYS_ON
(0x00000001ui64 << 32)
PROCESS_CREATION_MITIGATION_POLICY2_USER_CET_SET_CONTEXT_IP_VALIDATION_ALWAYS_OFF
(0x00000002ui64 << 32)
PROCESS_CREATION_MITIGATION_POLICY2_USER_CET_SET_CONTEXT_IP_VALIDATION_RELAXED_MODE
(0x00000003ui64 << 32)

Blocking the load of non-CETCOMPAT/non-EHCONT binaries:
PROCESS_CREATION_MITIGATION_POLICY2_BLOCK_NON_CET_BINARIES_ALWAYS_ON
(0x00000001ui64 << 36)
PROCESS_CREATION_MITIGATION_POLICY2_BLOCK_NON_CET_BINARIES_ALWAYS_OFF
(0x00000002ui64 << 36)
PROCESS_CREATION_MITIGATION_POLICY2_BLOCK_NON_CET_BINARIES_NON_EHCONT
(0x00000003ui64 << 36)

Restricting certain HSP APIs used to specify security properties of dynamic code to only be callable from outside of the process:
PROCESS_CREATION_MITIGATION_POLICY2_CET_DYNAMIC_APIS_OUT_OF_PROC_ONLY_ALWAYS_ON
(0x00000001ui64 << 48)
PROCESS_CREATION_MITIGATION_POLICY2_CET_DYNAMIC_APIS_OUT_OF_PROC_ONLY_ALWAYS_OFF
(0x00000002ui64 << 48)

The FSCTL system call disable policy, if enabled, prevents a process from making NtFsControlFile calls. The following mitigation options are available for the FSCTL system call disable policy:
PROCESS_CREATION_MITIGATION_POLICY2_FSCTL_SYSTEM_CALL_DISABLE_ALWAYS_ON
(0x00000001ui64 << 56)
PROCESS_CREATION_MITIGATION_POLICY2_FSCTL_SYSTEM_CALL_DISABLE_ALWAYS_OFF
(0x00000002ui64 << 56)

The **DWORD** pointed to by *lpValue* can be one or more of the following values when you specify **PROC_THREAD_ATTRIBUTE_CHILD_PROCESS_POLICY** for the *Attribute* parameter:

PROCESS_CREATION_CHILD_PROCESS_RESTRICTED 0x01

The process being created is not allowed to create child processes. This restriction becomes a property of the token as which the process runs. It should be noted that this restriction is only effective in sandboxed applications (such as AppContainer) which ensure privileged process handles are not accessible to the process. For example, if a process restricting child process creation is able to access another process handle with PROCESS_CREATE_PROCESS or PROCESS_VM_WRITE access rights, then it may be possible to bypass the child process restriction.

PROCESS_CREATION_CHILD_PROCESS_OVERRIDE 0x02

The process being created is allowed to create a child process, if it would otherwise be restricted. You can only specify this value if the process that is creating the new process is not restricted.

The **DWORD** pointed to by *lpValue* can be one or more of the following values when you specify **PROC_THREAD_ATTRIBUTE_DESKTOP_APP_POLICY** for the *Attribute* parameter:

PROCESS_CREATION_DESKTOP_APP_BREAKAWAY_ENABLE_PROCESS_TREE 0x01

The process being created will create any child processes outside of the desktop app runtime environment. This behavior is the default for processes for which no policy has been set.

PROCESS_CREATION_DESKTOP_APP_BREAKAWAY_DISABLE_PROCESS_TREE 0x02

The process being created will create any child processes inside of the desktop app runtime environment. This policy is inherited by the descendant processes until it is overridden by creating a process with **PROCESS_CREATION_DESKTOP_APP_BREAKAWAY_ENABLE_PROCESS_TREE**.

PROCESS_CREATION_DESKTOP_APP_BREAKAWAY_OVERRIDE 0x04

The process being created will run inside the desktop app runtime environment. This policy applies only to the process being created, not its descendants..

In order to launch the child process with the same protection level as the parent, the parent process must specify the **PROC_THREAD_ATTRIBUTE_PROTECTION_LEVEL** attribute for the child process. This can be used for both protected and unprotected processes. For example, when this flag is used by an unprotected process, the system will launch a child process at unprotected level. The **CREATE_PROTECTED_PROCESS** flag must be specified in both cases.

The following example launches a child process with the same protection level as the parent process:

```
C++

DWORD ProtectionLevel = PROTECTION_LEVEL_SAME;
SIZE_T AttributeListSize;

STARTUPINFOEXW StartupInfoEx = { 0 };
```

```
StartupInfoEx.StartupInfo.cb = sizeof(StartupInfoEx);

InitializeProcThreadAttributeList(NULL, 1, 0, &AttributeListSize)


StartupInfoEx.lpAttributeList = (LPPROC_THREAD_ATTRIBUTE_LIST) HeapAlloc(
    GetProcessHeap(),
    0,
    AttributeListSize
    );

if (InitializeProcThreadAttributeList(StartupInfoEx.lpAttributeList,
                                      1,
                                      0,
                                      &AttributeListSize) == FALSE)
{
    Result = GetLastError();
    goto exitFunc;
}

if (UpdateProcThreadAttribute(StartupInfoEx.lpAttributeList,
                              0,
                              PROC_THREAD_ATTRIBUTE_PROTECTION_LEVEL,
                              &ProtectionLevel,
                              sizeof(ProtectionLevel),
                              NULL,
                              NULL) == FALSE)
{
    Result = GetLastError();
    goto exitFunc;
}

PROCESS_INFORMATION ProcessInformation = { 0 };

if (CreateProcessW(ApplicationName,
                   CommandLine,
                   ProcessAttributes,
                   ThreadAttributes,
                   InheritHandles,
                   EXTENDED_STARTUPINFO_PRESENT | CREATE_PROTECTED_PROCESS,
                   Environment,
                   CurrentDirectory,
                   (LPSTARTUPINFOW)&StartupInfoEx,
                   &ProcessInformation) == FALSE)
{
    Result = GetLastError();
    goto exitFunc;
}
```

Requirements

Expand table

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

See also

DeleteProcThreadAttributeList

InitializeProcThreadAttributeList

Process and Thread Functions

UserHandleGrantAccess function (winuser.h)

Article02/22/2024

Grants or denies access to a handle to a User object to a job that has a user-interface restriction. When access is granted, all processes associated with the job can subsequently recognize and use the handle. When access is denied, the processes can no longer use the handle. For more information see User Objects.

Syntax

```C++
BOOL UserHandleGrantAccess(
  [in] HANDLE hUserHandle,
  [in] HANDLE hJob,
  [in] BOOL   bGrant
);
```

Parameters

`[in] hUserHandle`

A handle to the User object.

`[in] hJob`

A handle to the job to be granted access to the User handle. The CreateJobObject or OpenJobObject function returns this handle.

`[in] bGrant`

If this parameter is TRUE, all processes associated with the job can recognize and use the handle. If the parameter is FALSE, the processes cannot use the handle.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call GetLastError.

Remarks

The **UserHandleGrantAccess** function can be called only from a process not associated with the job specified by the *hJob* parameter. The User handle must not be owned by a process or thread associated with the job.

To create user-interface restrictions, call the SetInformationJobObject function with the JobObjectBasicUIRestrictions job information class.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll

See also

CreateJobObject

Job Objects

OpenJobObject

Process and Thread Functions

SetInformationJobObject

WaitCallback callback function

Article04/23/2018

Applications implement this callback if they call the SetThreadpoolWait function to start a worker thread for the wait object.

The **PTP_WAIT_CALLBACK** type defines a pointer to this callback function. *WaitCallback* is a placeholder for the application-defined function name.

Syntax

```c++
VOID CALLBACK WaitCallback(
  _Inout_     PTP_CALLBACK_INSTANCE Instance,
  _Inout_opt_ PVOID                 Context,
  _Inout_     PTP_WAIT              Wait,
  _In_        TP_WAIT_RESULT        WaitResult
);
```

Parameters

Instance [in, out]

A **TP_CALLBACK_INSTANCE** structure that defines the callback instance. Applications do not modify the members of this structure.

This structure can be passed to one of the following functions:

- CallbackMayRunLong
- DisassociateCurrentThreadFromCallback
- FreeLibraryWhenCallbackReturns
- LeaveCriticalSectionWhenCallbackReturns
- ReleaseMutexWhenCallbackReturns
- ReleaseSemaphoreWhenCallbackReturns
- SetEventWhenCallbackReturns

Context [in, out, optional]

The application-defined data.

Wait [in, out]

A **TP_WAIT** structure that defines the wait object that generated the callback.

WaitResult [in]

The result of the wait operation. This parameter can be one of the following values from WaitForMultipleObjects:

WAIT_OBJECT_0

WAIT_TIMEOUT

Return value

This function does not return a value.

Requirements

Expand table

Minimum supported client	Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps | UWP apps]
Header	WinBase.h on Windows Vista, Windows 7, Windows Server 2008 and Windows Server 2008 R2 (include Windows.h); Threadpoolapiset.h on Windows 8 and Windows Server 2012

See also

SetThreadpoolWait

WaitForInputIdle function (winuser.h)

02/22/2024

Waits until the specified process has finished processing its initial input and is waiting for user input with no input pending, or until the time-out interval has elapsed.

Syntax

```
C++

DWORD WaitForInputIdle(
  [in] HANDLE hProcess,
  [in] DWORD  dwMilliseconds
);
```

Parameters

`[in] hProcess`

A handle to the process. If this process is a console application or does not have a message queue, **WaitForInputIdle** returns immediately.

`[in] dwMilliseconds`

The time-out interval, in milliseconds. If *dwMilliseconds* is INFINITE, the function does not return until the process is idle.

Return value

The following table shows the possible return values for this function.

Expand table

Return code/value	Description
0	The wait was satisfied successfully.
WAIT_TIMEOUT	The wait was terminated because the time-out interval elapsed.
WAIT_FAILED	An error occurred.

Remarks

The **WaitForInputIdle** function enables a thread to suspend its execution until the specified process has finished its initialization and is waiting for user input with no input pending. If the process has multiple threads, the **WaitForInputIdle** function returns as soon as any thread becomes idle.

WaitForInputIdle can be used at any time, not just during application startup. However, **WaitForInputIdle** waits only once for a process to become idle; subsequent **WaitForInputIdle** calls return immediately, whether the process is idle or busy.

WaitForInputIdle can be useful for synchronizing a parent process and a newly created child process. When a parent process creates a child process, the CreateProcess function returns without waiting for the child process to finish its initialization. Before trying to communicate with the child process, the parent process can use the **WaitForInputIdle** function to determine when the child's initialization has been completed. For example, the parent process should use the **WaitForInputIdle** function before trying to find a window associated with the child process.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	winuser.h (include Windows.h)
Library	User32.lib
DLL	User32.dll
API set	ext-ms-win-ntuser-misc-l1-1-0 (introduced in Windows 8)

See also

CreateProcess

Process and Thread Functions

WaitForThreadpoolIoCallbacks function (threadpoolapiset.h)

02/22/2024

Waits for outstanding I/O completion callbacks to complete and optionally cancels pending callbacks that have not yet started to execute.

Syntax

```C++
VOID WaitForThreadpoolIoCallbacks(
  [in, out] PTP_IO pio,
  [in]      BOOL   fCancelPendingCallbacks
);
```

Parameters

`[in, out] pio`

A pointer to a **TP_IO** structure that defines the I/O completion object. The CreateThreadpoolIo function returns this pointer.

`[in] fCancelPendingCallbacks`

Indicates whether to cancel queued callbacks that have not yet started to execute.

Return value

None

Remarks

When *fCancelPendingCallbacks* is set to TRUE, only queued callbacks are canceled. Pending I/O requests are not canceled. Therefore, the caller should call GetOverlappedResult for the OVERLAPPED structure to check whether the I/O operation has completed before freeing the structure. As an alternative, set *fCancelPendingCallbacks* to FALSE and have the associated I/O completion callback free the **OVERLAPPED** structure. Be careful not to free the **OVERLAPPED** structure while I/O requests are still pending; use **GetOverlappedResult** to determine the

status of the I/O operation and wait for the operation to complete. The CancelIoEx function can optionally be used first to cancel outstanding I/O requests, potentially shortening the wait. For more information, see Canceling Pending I/O Operations.

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

Requirements

Expand table

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

See also

CancelThreadpoolIo

CloseThreadpoolIo

CreateThreadpoolIo

StartThreadpoolIo

Thread Pools

WaitForThreadpoolTimerCallbacks function (threadpoolapiset.h)

02/22/2024

Waits for outstanding timer callbacks to complete and optionally cancels pending callbacks that have not yet started to execute.

Syntax

```
C++

VOID WaitForThreadpoolTimerCallbacks(
  [in, out] PTP_TIMER pti,
  [in]      BOOL      fCancelPendingCallbacks
);
```

Parameters

`[in, out] pti`

A pointer to a **TP_TIMER** structure that defines the timer object. The CreateThreadpoolTimer function returns this pointer.

`[in] fCancelPendingCallbacks`

Indicates whether to cancel queued callbacks that have not yet started to execute.

Return value

None

Remarks

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

Requirements

Expand table

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

See also

CloseThreadpoolTimer

CreateThreadpoolTimer

IsThreadpoolTimerSet

SetThreadpoolTimer

Thread Pools

WaitForThreadpoolWaitCallbacks function (threadpoolapiset.h)

02/22/2024

Waits for outstanding wait callbacks to complete and optionally cancels pending callbacks that have not yet started to execute.

Syntax

```C++
VOID WaitForThreadpoolWaitCallbacks(
  [in, out] PTP_WAIT pwa,
  [in]      BOOL     fCancelPendingCallbacks
);
```

Parameters

`[in, out] pwa`

A pointer to a **TP_WAIT** structure that defines the wait object. The CreateThreadpoolWait function returns this pointer.

`[in] fCancelPendingCallbacks`

Indicates whether to cancel queued callbacks that have not yet started to execute.

Return value

None

Remarks

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

Examples

For an example, see Using the Thread Pool Functions.

Requirements

Expand table

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

See also

CloseThreadpoolWait

CreateThreadpoolWait

SetThreadpoolWait

Thread Pools

WaitForThreadpoolWorkCallbacks function (threadpoolapiset.h)

02/22/2024

Waits for outstanding work callbacks to complete and optionally cancels pending callbacks that have not yet started to execute.

Syntax

```C++
VOID WaitForThreadpoolWorkCallbacks(
  [in, out] PTP_WORK pwk,
  [in]      BOOL     fCancelPendingCallbacks
);
```

Parameters

`[in, out] pwk`

A pointer to a **TP_WORK** structure that defines the work object. The CreateThreadpoolWork function returns this pointer.

`[in] fCancelPendingCallbacks`

Indicates whether to cancel queued callbacks that have not yet started to execute.

Return value

None

Remarks

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

Requirements

Expand table

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

See also

CloseThreadpoolWork

CreateThreadpoolWork

SubmitThreadpoolWork

Thread Pools

WinExec function (winbase.h)

Article10/13/2021

Runs the specified application.

> **Note** This function is provided only for compatibility with 16-bit Windows. Applications should use the **CreateProcess** function.

Syntax

```
C++

UINT WinExec(
  [in] LPCSTR lpCmdLine,
  [in] UINT   uCmdShow
);
```

Parameters

`[in] lpCmdLine`

The command line (file name plus optional parameters) for the application to be executed. If the name of the executable file in the *lpCmdLine* parameter does not contain a directory path, the system searches for the executable file in this sequence:

1. The directory from which the application loaded.
2. The current directory.
3. The Windows system directory. The GetSystemDirectory function retrieves the path of this directory.
4. The Windows directory. The GetWindowsDirectory function retrieves the path of this directory.
5. The directories listed in the PATH environment variable.

`[in] uCmdShow`

The display options. For a list of the acceptable values, see the description of the *nCmdShow* parameter of the ShowWindow function.

Return value

If the function succeeds, the return value is greater than 31.

If the function fails, the return value is one of the following error values.

Expand table

Return code/value	Description
0	The system is out of memory or resources.
ERROR_BAD_FORMAT	The .exe file is invalid.
ERROR_FILE_NOT_FOUND	The specified file was not found.
ERROR_PATH_NOT_FOUND	The specified path was not found.

Remarks

The **WinExec** function returns when the started process calls the GetMessage function or a time-out limit is reached. To avoid waiting for the time out delay, call the **GetMessage** function as soon as possible in any process started by a call to **WinExec**.

Security Remarks

The executable name is treated as the first white space-delimited string in *lpCmdLine*. If the executable or path name has a space in it, there is a risk that a different executable could be run because of the way the function parses spaces. The following example is dangerous because the function will attempt to run "Program.exe", if it exists, instead of "MyApp.exe".

syntax

```
WinExec("C:\\Program Files\\MyApp", ...)
```

If a malicious user were to create an application called "Program.exe" on a system, any program that incorrectly calls **WinExec** using the Program Files directory will run this application instead of the intended application.

To avoid this problem, use CreateProcess rather than **WinExec**. However, if you must use **WinExec** for legacy reasons, make sure the application name is enclosed in quotation marks as shown in the example below.

syntax

```
WinExec("\"C:\\Program Files\\MyApp.exe\" -L -S", ...)
```

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Target Platform	Windows
Header	winbase.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll
API set	ext-ms-win-kernel32-process-l1-1-0 (introduced in Windows 10, version 10.0.14393)

See also

CreateProcess

WorkCallback callback function

Article04/23/2018

Applications implement this callback if they call the SubmitThreadpoolWork function to start a worker thread for the work object.

The **PTP_WORK_CALLBACK** type defines a pointer to this callback function. *WorkCallback* is a placeholder for the application-defined function name.

Syntax

```c++
VOID CALLBACK WorkCallback(
  _Inout_     PTP_CALLBACK_INSTANCE Instance,
  _Inout_opt_ PVOID                 Context,
  _Inout_     PTP_WORK              Work
);
```

Parameters

- *Instance* [in, out]
  A **TP_CALLBACK_INSTANCE** structure that defines the callback instance. Applications do not modify the members of this structure.

  This structure can be passed to one of the following functions:
  - SetEventWhenCallbackReturns
  - ReleaseSemaphoreWhenCallbackReturns
  - LeaveCriticalSectionWhenCallbackReturns
  - ReleaseMutexWhenCallbackReturns
  - FreeLibraryWhenCallbackReturns
  - CallbackMayRunLong
  - DisassociateCurrentThreadFromCallback

- *Context* [in, out, optional]
  The application-defined data.

- *Work* [in, out]
  A **TP_WORK** structure that defines the work object that generated the callback.

Return value

This function does not return a value.

Requirements

Expand table

Minimum supported client	Windows Vista [desktop apps | UWP apps]
Minimum supported server	Windows Server 2008 [desktop apps | UWP apps]
Header	WinBase.h on Windows Vista, Windows 7, Windows Server 2008 and Windows Server 2008 R2 (include Windows.h); Threadpoolapiset.h on Windows 8 and Windows Server 2012

See also

SubmitThreadpoolWork

Wow64SuspendThread function (wow64apiset.h)

Article11/18/2022

Suspends the specified WOW64 thread.

Syntax

```
C++

DWORD Wow64SuspendThread(
  HANDLE hThread
);
```

Parameters

`hThread`

A handle to the thread that is to be suspended. The handle must have the THREAD_SUSPEND_RESUME access right. For more information, see Thread Security and Access Rights.

Return value

If the function succeeds, the return value is the thread's previous suspend count; otherwise, it is (DWORD) -1. To get extended error information, use the GetLastError function.

Remarks

If the function succeeds, execution of the specified thread is suspended and the thread's suspend count is incremented. Suspending a thread causes the thread to stop executing user-mode (application) code.

This function is primarily designed for use by debuggers. It is not intended to be used for thread synchronization. Calling **Wow64SuspendThread** on a thread that owns a synchronization object, such as a mutex or critical section, can lead to a deadlock if the calling thread tries to obtain a synchronization object owned by a suspended thread. To

avoid this situation, a thread within an application that is not a debugger should signal the other thread to suspend itself. The target thread must be designed to watch for this signal and respond appropriately.

Each thread has a suspend count (with a maximum value of MAXIMUM_SUSPEND_COUNT). If the suspend count is greater than zero, the thread is suspended; otherwise, the thread is not suspended and is eligible for execution. Calling **Wow64SuspendThread** causes the target thread's suspend count to be incremented. Attempting to increment past the maximum suspend count causes an error without incrementing the count.

The ResumeThread function decrements the suspend count of a suspended thread.

This function is intended for 64-bit applications. It is not supported on 32-bit Windows; such calls fail and set the last error code to ERROR_INVALID_FUNCTION. A 32-bit application can call this function on a WOW64 thread; the result is the same as calling the SuspendThread function.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista
Minimum supported server	Windows Server 2008
Header	wow64apiset.h
Library	Kernel32.lib
DLL	Kernel32.dll

See also

ResumeThread

Feedback

Was this page helpful? Yes No

ZwQueryInformationProcess function

Article • 10/07/2021

[**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

```
C++

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.

Value	Meaning
	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.
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:

syntax

```
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.
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:

syntax

```
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:

syntax

```
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:

syntax

```
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

Feedback

Was this page helpful? Yes No

Get help at Microsoft Q&A

Process and Thread Macros

Article • 01/07/2021

The following macros are used with fibers:

- GetCurrentFiber
- GetFiberData

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

GetCurrentFiber function (winnt.h)

Article02/22/2024

Retrieves the address of the current fiber.

Syntax

```
C++

PVOID GetCurrentFiber();
```

Return value

The macro returns the address of the currently running fiber.

Remarks

The CreateFiber and ConvertThreadToFiber functions return the fiber address when the fiber is created. The **GetCurrentFiber** macro allows you to retrieve the address at any other time.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winnt.h (include Windows.h)

See also

ConvertThreadToFiber

CreateFiber

Fibers

GetFiberData function (winnt.h)

Article02/22/2024

Retrieves the fiber data associated with the current fiber.

Syntax

```
C++

PVOID GetFiberData();
```

Return value

The macro returns the fiber data for the currently running fiber.

Remarks

The fiber data is the value passed to the CreateFiber or ConvertThreadToFiber function in the *lpParameter* parameter. This value is also received as the parameter to the fiber function. It is stored as part of the fiber state information.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps | UWP apps]
Minimum supported server	Windows Server 2003 [desktop apps | UWP apps]
Target Platform	Windows
Header	winnt.h (include Windows.h)

See also

ConvertThreadToFiber

CreateFiber

Fibers

Process and Thread Structures

Article • 01/07/2021

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

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

DispatcherQueueOptions structure (dispatcherqueue.h)

Article02/22/2024

Represents options around threading affinity and type of COM apartment for a new DispatcherQueueController.

Syntax

```C++
struct DispatcherQueueOptions {
  DWORD                                dwSize;
  DISPATCHERQUEUE_THREAD_TYPE          threadType;
  DISPATCHERQUEUE_THREAD_APARTMENTTYPE apartmentType;
};
```

Members

`dwSize`

Size of this **DispatcherQueueOptions** structure.

`threadType`

Thread affinity for a new DispatcherQueueController.

`apartmentType`

Specifies whether to initialize the COM apartment on the new thread as an application single-threaded apartment (ASTA) or a single-threaded apartment (STA). This field is relevant only if **threadType** is **DQTYPE_THREAD_DEDICATED**. Use **DQTAT_COM_NONE** when **DispatcherQueueOptions.threadType** is **DQTYPE_THREAD_CURRENT**.

Remarks

Introduced in Windows 10, version 1709.

Requirements

Expand table

Requirement	Value
Header	dispatcherqueue.h

See also

CreateDispatcherQueueController

PROCESS_PROTECTION_LEVEL_INFORMATION structure (processthreadsapi.h)

Article02/22/2024

Specifies whether Protected Process Light (PPL) is enabled.

Syntax

```
C++

typedef struct PROCESS_PROTECTION_LEVEL_INFORMATION {
  DWORD ProtectionLevel;
} PROCESS_PROTECTION_LEVEL_INFORMATION;
```

Members

`ProtectionLevel`

The one of the following values.

Expand table

Value	Meaning
PROTECTION_LEVEL_WINTCB_LIGHT	For internal use only.
PROTECTION_LEVEL_WINDOWS	For internal use only.
PROTECTION_LEVEL_WINDOWS_LIGHT	For internal use only.
PROTECTION_LEVEL_ANTIMALWARE_LIGHT	For internal use only.
PROTECTION_LEVEL_LSA_LIGHT	For internal use only.
PROTECTION_LEVEL_WINTCB	Not implemented.
PROTECTION_LEVEL_CODEGEN_LIGHT	Not implemented.
PROTECTION_LEVEL_AUTHENTICODE	Not implemented.
PROTECTION_LEVEL_PPL_APP	The process is a third party app that is using process protection.
PROTECTION_LEVEL_NONE	The process is not protected.

Requirements

Expand table

Requirement	Value
Header	processthreadsapi.h

THREAD_POWER_THROTTLING_STATE structure (processthreadsapi.h)

Article02/22/2024

Specifies the throttling policies and how to apply them to a target thread when that thread is subject to power management. This structure is used by the SetThreadInformation function.

Syntax

C++

```
typedef struct _THREAD_POWER_THROTTLING_STATE {
  ULONG Version;
  ULONG ControlMask;
  ULONG StateMask;
} THREAD_POWER_THROTTLING_STATE;
```

Members

`Version`

The version of the **THREAD_POWER_THROTTLING_STATE** structure.

Expand table

Value	Meaning
THREAD_POWER_THROTTLING_CURRENT_VERSION	The current version.

`ControlMask`

This field enables the caller to take control of the power throttling mechanism.

Expand table

Value	Meaning
THREAD_POWER_THROTTLING_EXECUTION_SPEED	Manages the execution speed of the thread.

`StateMask`

Manages the power throttling mechanism on/off state.

Expand table

Value	Meaning
THREAD_POWER_THROTTLING_EXECUTION_SPEED	Manages the execution speed of the thread.

Requirements

Expand table

Requirement	Value
Header	processthreadsapi.h

Feedback

Was this page helpful? Yes No

PROCESS_POWER_THROTTLING_STATE structure (processthreadsapi.h)

Article11/01/2022

Specifies the throttling policies and how to apply them to a target process when that process is subject to power management. This structure is used by the SetProcessInformation function.

Syntax

```
C++

typedef struct _PROCESS_POWER_THROTTLING_STATE {
  ULONG Version;
  ULONG ControlMask;
  ULONG StateMask;
} PROCESS_POWER_THROTTLING_STATE, *PPROCESS_POWER_THROTTLING_STATE;
```

Members

`Version`

The version of the **PROCESS_POWER_THROTTLING_STATE** structure.

Expand table

Value	Meaning
PROCESS_POWER_THROTTLING_CURRENT_VERSION	The current version.

`ControlMask`

This field enables the caller to take control of the power throttling mechanism.

Expand table

Value	Meaning
PROCESS_POWER_THROTTLING_EXECUTION_SPEED	Manages the execution speed of the process.

`StateMask`

Manages the power throttling mechanism on/off state.

Expand table

Value	Meaning
PROCESS_POWER_THROTTLING_EXECUTION_SPEED	Manages the execution speed of the process.

Requirements

Expand table

Requirement	Value
Header	processthreadsapi.h

Feedback

Was this page helpful? Yes No

APP_MEMORY_INFORMATION structure (processthreadsapi.h)

Article02/22/2024

Represents app memory usage at a single point in time. This structure is used by the PROCESS_INFORMATION_CLASS enumeration.

Syntax

```C++
typedef struct _APP_MEMORY_INFORMATION {
  ULONG64 AvailableCommit;
  ULONG64 PrivateCommitUsage;
  ULONG64 PeakPrivateCommitUsage;
  ULONG64 TotalCommitUsage;
} APP_MEMORY_INFORMATION, *PAPP_MEMORY_INFORMATION;
```

Members

`AvailableCommit`

Total commit available to the app.

`PrivateCommitUsage`

The app's usage of private commit.

`PeakPrivateCommitUsage`

The app's peak usage of private commit.

`TotalCommitUsage`

The app's total usage of private plus shared commit.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 10, version 1511 [desktop apps | UWP apps]
Minimum supported server	Windows Server 2016 [desktop apps | UWP apps]
Header	processthreadsapi.h (include Windows.h)

See also

PROCESS_INFORMATION_CLASS enumeration

Feedback

Was this page helpful? Yes No

AR_STATE enumeration (winuser.h)

Article02/22/2024

Indicates the state of screen auto-rotation for the system. For example, whether auto-rotation is supported, and whether it is enabled by the user. This enum is a bitwise OR of one or more of the following values.

Syntax

C++

```
typedef enum tagAR_STATE {
  AR_ENABLED = 0x0,
  AR_DISABLED = 0x1,
  AR_SUPPRESSED = 0x2,
  AR_REMOTESESSION = 0x4,
  AR_MULTIMON = 0x8,
  AR_NOSENSOR = 0x10,
  AR_NOT_SUPPORTED = 0x20,
  AR_DOCKED = 0x40,
  AR_LAPTOP = 0x80
} AR_STATE, *PAR_STATE;
```

Constants

Expand table

`AR_ENABLED` Value: *0x0* Auto-rotation is enabled by the user.
`AR_DISABLED` Value: *0x1* Auto-rotation is disabled by the user.
`AR_SUPPRESSED` Value: *0x2* Auto-rotation is currently suppressed by one or more process auto-rotation preferences.
`AR_REMOTESESSION` Value: *0x4* The session is remote, and auto-rotation is temporarily disabled as a result.

`AR_MULTIMON` Value: *0x8* The system has multiple monitors attached, and auto-rotation is temporarily disabled as a result.	
`AR_NOSENSOR` Value: *0x10* The system does not have an auto-rotation sensor.	
`AR_NOT_SUPPORTED` Value: *0x20* Auto-rotation is not supported with the current system configuration.	
`AR_DOCKED` Value: *0x40* The device is docked, and auto-rotation is temporarily disabled as a result.	
`AR_LAPTOP` Value: *0x80* The device is in laptop mode, and auto-rotation is temporarily disabled as a result.	

Requirements

Expand table

Requirement	Value
Header	winuser.h

Feedback

Was this page helpful? Yes No

CACHE_DESCRIPTOR structure (winnt.h)

Article02/22/2024

Describes the cache attributes.

Syntax

C++

```
typedef struct _CACHE_DESCRIPTOR {
  BYTE                 Level;
  BYTE                 Associativity;
  WORD                 LineSize;
  DWORD                Size;
  PROCESSOR_CACHE_TYPE Type;
} CACHE_DESCRIPTOR, *PCACHE_DESCRIPTOR;
```

Members

`Level`

The cache level. This member can currently be one of the following values; other values may be supported in the future.

Expand table

Value	Meaning
1	L1
2	L2
3	L3

`Associativity`

The cache associativity. If this member is CACHE_FULLY_ASSOCIATIVE (0xFF), the cache is fully associative.

`LineSize`

The cache line size, in bytes.

`Size`

The cache size, in bytes.

`Type`

The cache type. This member is a PROCESSOR_CACHE_TYPE value.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista, Windows XP Professional x64 Edition [desktop apps only]
Minimum supported server	Windows Server 2008, Windows Server 2003 with SP1 [desktop apps only]
Header	winnt.h (include Windows.h)

See also

GetLogicalProcessorInformation

PROCESSOR_CACHE_TYPE

SYSTEM_LOGICAL_PROCESSOR_INFORMATION

Feedback

Was this page helpful? Yes No

CACHE_RELATIONSHIP structure (winnt.h)

Article02/22/2024

Describes cache attributes. This structure is used with the GetLogicalProcessorInformationEx function.

Syntax

C++

```
typedef struct _CACHE_RELATIONSHIP {
  BYTE                 Level;
  BYTE                 Associativity;
  WORD                 LineSize;
  DWORD                CacheSize;
  PROCESSOR_CACHE_TYPE Type;
  BYTE                 Reserved[18];
  WORD                 GroupCount;
  union {
    GROUP_AFFINITY GroupMask;
    GROUP_AFFINITY GroupMasks[ANYSIZE_ARRAY];
  } DUMMYUNIONNAME;
} CACHE_RELATIONSHIP, *PCACHE_RELATIONSHIP;
```

Members

`Level`

The cache level. This member can be one of the following values.

Expand table

Value	Meaning
1	L1
2	L2
3	L3

`Associativity`

The cache associativity. If this member is CACHE_FULLY_ASSOCIATIVE (0xFF), the cache is fully associative.

`LineSize`

The cache line size, in bytes.

`CacheSize`

The cache size, in bytes.

`Type`

The cache type. This member is a PROCESSOR_CACHE_TYPE value.

`Reserved[18]`

This member is reserved.

`GroupCount`

`DUMMYUNIONNAME`

`DUMMYUNIONNAME.GroupMask`

`DUMMYUNIONNAME.GroupMasks[ANYSIZE_ARRAY]`

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Header	winnt.h

See also

GROUP_AFFINITY

GetLogicalProcessorInformationEx

PROCESSOR_CACHE_TYPE

Feedback

Was this page helpful? Yes No

GROUP_AFFINITY structure (winnt.h)

Article02/22/2024

Represents a processor group-specific affinity, such as the affinity of a thread.

Syntax

```
C++

typedef struct _GROUP_AFFINITY {
  KAFFINITY Mask;
  WORD      Group;
  WORD      Reserved[3];
} GROUP_AFFINITY, *PGROUP_AFFINITY;
```

Members

`Mask`

A bitmap that specifies the affinity for zero or more processors within the specified group.

`Group`

The processor group number.

`Reserved[3]`

This member is reserved.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Header	winnt.h

See also

CACHE_RELATIONSHIP

NUMA_NODE_RELATIONSHIP

PROCESSOR_RELATIONSHIP

Feedback

Was this page helpful? Yes No

GROUP_RELATIONSHIP structure (winnt.h)

Article02/22/2024

Represents information about processor groups. This structure is used with the GetLogicalProcessorInformationEx function.

Syntax

```
C++

typedef struct _GROUP_RELATIONSHIP {
  WORD                 MaximumGroupCount;
  WORD                 ActiveGroupCount;
  BYTE                 Reserved[20];
  PROCESSOR_GROUP_INFO GroupInfo[ANYSIZE_ARRAY];
} GROUP_RELATIONSHIP, *PGROUP_RELATIONSHIP;
```

Members

`MaximumGroupCount`

The maximum number of processor groups on the system.

`ActiveGroupCount`

The number of active groups on the system. This member indicates the number of PROCESSOR_GROUP_INFO structures in the **GroupInfo** array.

`Reserved[20]`

This member is reserved.

`GroupInfo[ANYSIZE_ARRAY]`

An array of PROCESSOR_GROUP_INFO structures. Each structure represents the number and affinity of processors in an active group on the system.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Header	winnt.h

See also

GetLogicalProcessorInformationEx

PROCESSOR_GROUP_INFO

SYSTEM_LOGICAL_PROCESSOR_INFORMATION_EX

Feedback

Was this page helpful? Yes No

IO_COUNTERS structure (winnt.h)

Article04/02/2021

Contains I/O accounting information for a process or a job object. For a job object, the counters include all operations performed by all processes that have ever been associated with the job, in addition to all processes currently associated with the job.

Syntax

C++

```
typedef struct _IO_COUNTERS {
  ULONGLONG ReadOperationCount;
  ULONGLONG WriteOperationCount;
  ULONGLONG OtherOperationCount;
  ULONGLONG ReadTransferCount;
  ULONGLONG WriteTransferCount;
  ULONGLONG OtherTransferCount;
} IO_COUNTERS;
```

Members

`ReadOperationCount`

The number of read operations performed.

`WriteOperationCount`

The number of write operations performed.

`OtherOperationCount`

The number of I/O operations performed, other than read and write operations.

`ReadTransferCount`

The number of bytes read.

`WriteTransferCount`

The number of bytes written.

`OtherTransferCount`

The number of bytes transferred during operations other than read and write operations.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	winnt.h (include Windows.h)

See also

GetProcessIoCounters

JOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION

Feedback

Was this page helpful? Yes No

JOBOBJECT_ASSOCIATE_COMPLETION_PORT structure (winnt.h)

Article03/13/2025

Contains information used to associate a completion port with a job. You can associate one completion port with a job.

Syntax

```
C++

typedef struct _JOBOBJECT_ASSOCIATE_COMPLETION_PORT {
  PVOID  CompletionKey;
  HANDLE CompletionPort;
} JOBOBJECT_ASSOCIATE_COMPLETION_PORT,
*PJOBOBJECT_ASSOCIATE_COMPLETION_PORT;
```

Members

`CompletionKey`

The value to use in the *dwCompletionKey* parameter of PostQueuedCompletionStatus when messages are sent on behalf of the job.

`CompletionPort`

The completion port to use in the *CompletionPort* parameter of the PostQueuedCompletionStatus function when messages are sent on behalf of the job.

Windows 8 and newer, Windows Server 2012 and newer: Specify **NULL** to remove the association between the current completion port and the job.

Remarks

The system sends messages to the I/O completion port associated with a job when certain events occur. If the job is nested, 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. All messages are sent directly from the job as if the job had called the PostQueuedCompletionStatus function.

Note that, except for limits set with the **JobObjectNotificationLimitInformation** information class, messages are intended only as notifications and their delivery to the completion port is not guaranteed. The failure of a message to arrive at the completion port does not necessarily mean that the event did not occur. Notifications for limits set with **JobObjectNotificationLimitInformation** are guaranteed to arrive at the completion port.

A thread must monitor the completion port using the GetQueuedCompletionStatus function to pick up the messages. The thread receives information in the **GetQueuedCompletionStatus** parameters shown in the following table.

Expand table

Parameter	Information Received
lpCompletionKey	The value specified in *CompletionKey* during the completion-port association. If a completion port is associated with multiple jobs, *CompletionKey* should help the caller determine which completion port is sending a message.
lpOverlapped	Message-specific value. For more information, see the following table of message identifiers.
LpNumberOfBytes	The message identifier that indicates which job-related event occurred. For more information, see the following table of message identifiers.

The following messages can be sent to the completion port. Note that for messages that return a process identifier, you cannot guarantee that this process is still active or that the identifier has not been recycled (assigned to a new process after termination) unless you maintain an open handle to the process.

Expand table

Message Identifier	Description
JOB_OBJECT_MSG_ABNORMAL_EXIT_PROCESS	Indicates that a process associated with the job exited with an exit code that indicates an abnormal exit (see the list following this table). The value of *lpOverlapped* is the identifier of the exiting process.
JOB_OBJECT_MSG_ACTIVE_PROCESS_LIMIT	Indicates that the active process limit has been exceeded. The value of *lpOverlapped* is NULL.
JOB_OBJECT_MSG_ACTIVE_PROCESS_ZERO	Indicates that the active process count has been decremented to 0. For example, if the job

	currently has two active processes, the system sends this message after they both terminate. The value of *lpOverlapped* is NULL.
JOB_OBJECT_MSG_END_OF_JOB_TIME	Indicates that the JOB_OBJECT_POST_AT_END_OF_JOB option is in effect and the end-of-job time limit has been reached. Upon posting this message, the time limit is canceled and the job's processes can continue to run. The value of *lpOverlapped* is NULL.
JOB_OBJECT_MSG_END_OF_PROCESS_TIME	Indicates that a process has exceeded a per-process time limit. The system sends this message after the process termination has been requested. The value of *lpOverlapped* is the identifier of the process that exceeded its limit.
JOB_OBJECT_MSG_EXIT_PROCESS	Indicates that a process associated with the job has exited. The value of *lpOverlapped* is the identifier of the exiting process.
JOB_OBJECT_MSG_JOB_MEMORY_LIMIT	Indicates that a process associated with the job caused the job to exceed the job-wide memory limit (if one is in effect). The value of *lpOverlapped* specifies the identifier of the process that has attempted to exceed the limit. The system does not send this message if the process has not yet reported its process identifier.
JOB_OBJECT_MSG_NEW_PROCESS	Indicates that a process has been added to the job. Processes added to a job at the time a completion port is associated are also reported. The value of *lpOverlapped* is the identifier of the process added to the job.
JOB_OBJECT_MSG_NOTIFICATION_LIMIT	Indicates that a process associated with a job that has registered for resource limit notifications has exceeded one or more limits. Use the QueryInformationJobObject function with JobObjectLimitViolationInformation to determine which limit was exceeded. The value of *lpOverlapped* is the identifier of the process that has exceeded its limit. The system does not send this message if the process has not yet reported its process identifier.

JOB_OBJECT_MSG_PROCESS_MEMORY_LIMIT	Indicates that a process associated with the job has exceeded its memory limit (if one is in effect). The value of *lpOverlapped* is the identifier of the process that has exceeded its limit. The system does not send this message if the process has not yet reported its process identifier.

The following exit codes indicate an abnormal exit:

- STATUS_GUARD_PAGE_VIOLATION
- STATUS_DATATYPE_MISALIGNMENT
- STATUS_BREAKPOINT
- STATUS_SINGLE_STEP
- STATUS_ACCESS_VIOLATION
- STATUS_IN_PAGE_ERROR
- STATUS_ILLEGAL_INSTRUCTION
- STATUS_NONCONTINUABLE_EXCEPTION
- STATUS_INVALID_DISPOSITION
- STATUS_ARRAY_BOUNDS_EXCEEDED
- STATUS_FLOAT_DENORMAL_OPERAND
- STATUS_FLOAT_DIVIDE_BY_ZERO
- STATUS_FLOAT_INEXACT_RESULT
- STATUS_FLOAT_INVALID_OPERATION
- STATUS_FLOAT_OVERFLOW
- STATUS_FLOAT_STACK_CHECK
- STATUS_FLOAT_UNDERFLOW
- STATUS_INTEGER_DIVIDE_BY_ZERO
- STATUS_INTEGER_OVERFLOW
- STATUS_PRIVILEGED_INSTRUCTION
- STATUS_STACK_OVERFLOW
- STATUS_CONTROL_C_EXIT
- STATUS_FLOAT_MULTIPLE_FAULTS
- STATUS_FLOAT_MULTIPLE_TRAPS
- STATUS_REG_NAT_CONSUMPTION

You must be cautious when using the JOB_OBJECT_MSG_NEW_PROCESS and JOB_OBJECT_MSG_EXIT_PROCESS messages, as race conditions may occur. For instance, if processes are actively starting and exiting within a job, and you are in the process of assigning a completion port to the job, you may miss messages for processes whose

states change during the association of the completion port. For this reason, it is best to associate a completion port with a job when the job is inactive.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	winnt.h (include Windows.h)

See also

PostQueuedCompletionStatus

QueryInformationJobObject

SetInformationJobObject

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

JOBOBJECT_BASIC_ACCOUNTING_INFORMATION structure (winnt.h)

Article04/02/2021

Contains basic accounting information for a job object.

Syntax

```C++
typedef struct _JOBOBJECT_BASIC_ACCOUNTING_INFORMATION {
  LARGE_INTEGER TotalUserTime;
  LARGE_INTEGER TotalKernelTime;
  LARGE_INTEGER ThisPeriodTotalUserTime;
  LARGE_INTEGER ThisPeriodTotalKernelTime;
  DWORD         TotalPageFaultCount;
  DWORD         TotalProcesses;
  DWORD         ActiveProcesses;
  DWORD         TotalTerminatedProcesses;
} JOBOBJECT_BASIC_ACCOUNTING_INFORMATION,
*PJOBOBJECT_BASIC_ACCOUNTING_INFORMATION;
```

Members

`TotalUserTime`

The total amount of user-mode execution time for all active processes associated with the job, as well as all terminated processes no longer associated with the job, in 100-nanosecond ticks.

`TotalKernelTime`

The total amount of kernel-mode execution time for all active processes associated with the job, as well as all terminated processes no longer associated with the job, in 100-nanosecond ticks.

`ThisPeriodTotalUserTime`

The total amount of user-mode execution time for all active processes associated with the job (as well as all terminated processes no longer associated with the job) since the last call that set a per-job user-mode time limit, in 100-nanosecond ticks.

This member is set to 0 on creation of the job, and each time a per-job user-mode time limit is established.

`ThisPeriodTotalKernelTime`

The total amount of kernel-mode execution time for all active processes associated with the job (as well as all terminated processes no longer associated with the job) since the last call that set a per-job kernel-mode time limit, in 100-nanosecond ticks.

This member is set to zero on creation of the job, and each time a per-job kernel-mode time limit is established.

`TotalPageFaultCount`

The total number of page faults encountered by all active processes associated with the job, as well as all terminated processes no longer associated with the job.

`TotalProcesses`

The total number of processes associated with the job during its lifetime, including those that have terminated. For example, when a process is associated with a job, but the association fails because of a limit violation, this value is incremented.

`ActiveProcesses`

The total number of processes currently associated with the job. When a process is associated with a job, but the association fails because of a limit violation, this value is temporarily incremented. When the terminated process exits and all references to the process are released, this value is decremented.

`TotalTerminatedProcesses`

The total number of processes terminated because of a limit violation.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	winnt.h (include Windows.h)

See also

QueryInformationJobObject

SetInformationJobObject

Feedback

Was this page helpful? Yes No

JOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION structure (winnt.h)

Article04/02/2021

Contains basic accounting and I/O accounting information for a job object.

Syntax

C++

```
typedef struct _JOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION {
  JOBOBJECT_BASIC_ACCOUNTING_INFORMATION BasicInfo;
  IO_COUNTERS                            IoInfo;
} JOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION,
*PJOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION;
```

Members

`BasicInfo`

A JOBOBJECT_BASIC_ACCOUNTING_INFORMATION structure that specifies the basic accounting information for the job.

`IoInfo`

An IO_COUNTERS structure that specifies the I/O accounting information for the job. The structure includes information for all processes that have ever been associated with the job, in addition to the information for all processes currently associated with the job.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	winnt.h (include Windows.h)

See also

IO_COUNTERS

JOBOBJECT_BASIC_ACCOUNTING_INFORMATION

QueryInformationJobObject

Feedback

Was this page helpful? Yes No

JOBOBJECT_BASIC_LIMIT_INFORMATION structure (winnt.h)

Article08/23/2022

Contains basic limit information for a job object.

Syntax

C++

```
typedef struct _JOBOBJECT_BASIC_LIMIT_INFORMATION {
  LARGE_INTEGER PerProcessUserTimeLimit;
  LARGE_INTEGER PerJobUserTimeLimit;
  DWORD         LimitFlags;
  SIZE_T        MinimumWorkingSetSize;
  SIZE_T        MaximumWorkingSetSize;
  DWORD         ActiveProcessLimit;
  ULONG_PTR     Affinity;
  DWORD         PriorityClass;
  DWORD         SchedulingClass;
} JOBOBJECT_BASIC_LIMIT_INFORMATION, *PJOBOBJECT_BASIC_LIMIT_INFORMATION;
```

Members

`PerProcessUserTimeLimit`

If **LimitFlags** specifies **JOB_OBJECT_LIMIT_PROCESS_TIME**, this member is the per-process user-mode execution time limit, in 100-nanosecond ticks. Otherwise, this member is ignored.

The system periodically checks to determine whether each process associated with the job has accumulated more user-mode time than the set limit. If it has, the process is terminated.

If the job is nested, the effective limit is the most restrictive limit in the job chain.

`PerJobUserTimeLimit`

If **LimitFlags** specifies **JOB_OBJECT_LIMIT_JOB_TIME**, this member is the per-job user-mode execution time limit, in 100-nanosecond ticks. Otherwise, this member is ignored.

The system adds the current time of the processes associated with the job to this limit. For example, if you set this limit to 1 minute, and the job has a process that has accumulated 5 minutes of user-mode time, the limit actually enforced is 6 minutes.

The system periodically checks to determine whether the sum of the user-mode execution time for all processes is greater than this end-of-job limit. If it is, the action specified in the **EndOfJobTimeAction** member of the JOBOBJECT_END_OF_JOB_TIME_INFORMATION structure is carried out. By default, all processes are terminated and the status code is set to **ERROR_NOT_ENOUGH_QUOTA**.

To register for notification when this limit is exceeded without terminating processes, use the SetInformationJobObject function with the **JobObjectNotificationLimitInformation** information class.

`LimitFlags`

The limit flags that are in effect. This member is a bitfield that determines whether other structure members are used. Any combination of the following values can be specified.

Expand table

Value	Meaning
JOB_OBJECT_LIMIT_ACTIVE_PROCESS 0x00000008	Establishes a maximum number of simultaneously active processes associated with the job. The **ActiveProcessLimit** member contains additional information.
JOB_OBJECT_LIMIT_AFFINITY 0x00000010	Causes all processes associated with the job to use the same processor affinity. The **Affinity** member contains additional information. If the job is nested, the specified processor affinity must be a subset of the effective affinity of the parent job. If the specified affinity a superset of the affinity of the parent job, it is ignored and the affinity of the parent job is used.
JOB_OBJECT_LIMIT_BREAKAWAY_OK 0x00000800	If any process associated with the job creates a child process using the **CREATE_BREAKAWAY_FROM_JOB** flag while this limit is in effect, the child process is not associated with the job. This limit requires use of a JOBOBJECT_EXTENDED_LIMIT_INFORMATION

	structure. Its **BasicLimitInformation** member is a **JOBOBJECT_BASIC_LIMIT_INFORMATION** structure.
JOB_OBJECT_LIMIT_DIE_ON_UNHANDLED_EXCEPTION 0x00000400	Forces a call to the SetErrorMode function with the **SEM_NOGPFAULTERRORBOX** flag for each process associated with the job. If an exception occurs and the system calls the UnhandledExceptionFilter function, the debugger will be given a chance to act. If there is no debugger, the functions returns **EXCEPTION_EXECUTE_HANDLER**. Normally, this will cause termination of the process with the exception code as the exit status. This limit requires use of a JOBOBJECT_EXTENDED_LIMIT_INFORMATION structure. Its **BasicLimitInformation** member is a **JOBOBJECT_BASIC_LIMIT_INFORMATION** structure.
JOB_OBJECT_LIMIT_JOB_MEMORY 0x00000200	Causes all processes associated with the job to limit the job-wide sum of their committed memory. When a process attempts to commit memory that would exceed the job-wide limit, it fails. If the job object is associated with a completion port, a **JOB_OBJECT_MSG_JOB_MEMORY_LIMIT** message is sent to the completion port. This limit requires use of a JOBOBJECT_EXTENDED_LIMIT_INFORMATION structure. Its **BasicLimitInformation** member is a **JOBOBJECT_BASIC_LIMIT_INFORMATION** structure. To register for notification when this limit is exceeded while allowing processes to continue to commit memory, use the SetInformationJobObject function with the **JobObjectNotificationLimitInformation** information class.
JOB_OBJECT_LIMIT_JOB_TIME 0x00000004	Establishes a user-mode execution time limit for the job. The **PerJobUserTimeLimit** member contains additional information. This flag cannot be used with **JOB_OBJECT_LIMIT_PRESERVE_JOB_TIME**.

JOB_OBJECT_LIMIT_KILL_ON_JOB_CLOSE 0x00002000	Causes all processes associated with the job to terminate when the last handle to the job is closed. This limit requires use of a JOBOBJECT_EXTENDED_LIMIT_INFORMATION structure. Its **BasicLimitInformation** member is a **JOBOBJECT_BASIC_LIMIT_INFORMATION** structure.
JOB_OBJECT_LIMIT_PRESERVE_JOB_TIME 0x00000040	Preserves any job time limits you previously set. As long as this flag is set, you can establish a per-job time limit once, then alter other limits in subsequent calls. This flag cannot be used with **JOB_OBJECT_LIMIT_JOB_TIME**.
JOB_OBJECT_LIMIT_PRIORITY_CLASS 0x00000020	Causes all processes associated with the job to use the same priority class. For more information, see Scheduling Priorities. The **PriorityClass** member contains additional information. If the job is nested, the effective priority class is the lowest priority class in the job chain.
JOB_OBJECT_LIMIT_PROCESS_MEMORY 0x00000100	Causes all processes associated with the job to limit their committed memory. When a process attempts to commit memory that would exceed the per-process limit, it fails. If the job object is associated with a completion port, a **JOB_OBJECT_MSG_PROCESS_MEMORY_LIMIT** message is sent to the completion port. If the job is nested, the effective memory limit is the most restrictive memory limit in the job chain. This limit requires use of a JOBOBJECT_EXTENDED_LIMIT_INFORMATION structure. Its **BasicLimitInformation** member is a **JOBOBJECT_BASIC_LIMIT_INFORMATION** structure.
JOB_OBJECT_LIMIT_PROCESS_TIME 0x00000002	Establishes a user-mode execution time limit for each currently active process and for all future processes associated with the job. The **PerProcessUserTimeLimit** member contains additional information.

JOB_OBJECT_LIMIT_SCHEDULING_CLASS 0x00000080	Causes all processes in the job to use the same scheduling class. The **SchedulingClass** member contains additional information. If the job is nested, the effective scheduling class is the lowest scheduling class in the job chain.
JOB_OBJECT_LIMIT_SILENT_BREAKAWAY_OK 0x00001000	Allows any process associated with the job to create child processes that are not associated with the job. If the job is nested and its 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. This limit requires use of a JOBOBJECT_EXTENDED_LIMIT_INFORMATION structure. Its **BasicLimitInformation** member is a **JOBOBJECT_BASIC_LIMIT_INFORMATION** structure.
JOB_OBJECT_LIMIT_SUBSET_AFFINITY 0x00004000	Allows processes to use a subset of the processor affinity for all processes associated with the job. This value must be combined with **JOB_OBJECT_LIMIT_AFFINITY**. **Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP:** This flag is supported starting with Windows 7 and Windows Server 2008 R2.
JOB_OBJECT_LIMIT_WORKINGSET 0x00000001	Causes all processes associated with the job to use the same minimum and maximum working set sizes. The **MinimumWorkingSetSize** and **MaximumWorkingSetSize** members contain additional information. If the job is nested, the effective working set size is the smallest working set size in the job chain.

`MinimumWorkingSetSize`

If **LimitFlags** specifies **JOB_OBJECT_LIMIT_WORKINGSET**, this member is the minimum working set size in bytes for each process associated with the job. Otherwise, this member is ignored.

If **MaximumWorkingSetSize** is nonzero, **MinimumWorkingSetSize** cannot be zero.

`MaximumWorkingSetSize`

If **LimitFlags** specifies **JOB_OBJECT_LIMIT_WORKINGSET**, this member is the maximum working set size in bytes for each process associated with the job. Otherwise, this member is ignored.

If **MinimumWorkingSetSize** is nonzero, **MaximumWorkingSetSize** cannot be zero.

`ActiveProcessLimit`

If **LimitFlags** specifies **JOB_OBJECT_LIMIT_ACTIVE_PROCESS**, this member is the active process limit for the job. Otherwise, this member is ignored.

If you try to associate a process with a job, and this causes the active process count to exceed this limit, the process is terminated and the association fails.

`Affinity`

If **LimitFlags** specifies **JOB_OBJECT_LIMIT_AFFINITY**, this member is the processor affinity for all processes associated with the job. Otherwise, this member is ignored.

The affinity must be a subset of the system affinity mask obtained by calling the GetProcessAffinityMask function. The affinity of each thread is set to this value, but threads are free to subsequently set their affinity, as long as it is a subset of the specified affinity mask. Processes cannot set their own affinity mask.

`PriorityClass`

If **LimitFlags** specifies **JOB_OBJECT_LIMIT_PRIORITY_CLASS**, this member is the priority class for all processes associated with the job. Otherwise, this member is ignored.

Processes and threads cannot modify their priority class. The calling process must enable the **SE_INC_BASE_PRIORITY_NAME** privilege.

`SchedulingClass`

If **LimitFlags** specifies **JOB_OBJECT_LIMIT_SCHEDULING_CLASS**, this member is the scheduling class for all processes associated with the job. Otherwise, this member is ignored.

The valid values are 0 to 9. Use 0 for the least favorable scheduling class relative to other threads, and 9 for the most favorable scheduling class relative to other threads. By default, this value is 5. To use a scheduling class greater than 5, the calling process must enable the **SE_INC_BASE_PRIORITY_NAME** privilege.

Remarks

Processes can still empty their working sets using the SetProcessWorkingSetSize function with (**SIZE_T**)-1, even when **JOB_OBJECT_LIMIT_WORKINGSET** is used. However, you cannot use **SetProcessWorkingSetSize** change the minimum or maximum working set size of a process in a job object.

The system increments the active process count when you attempt to associate a process with a job. If the limit is exceeded, the system decrements the active process count only when the process terminates and all handles to the process are closed. Therefore, if you have an open handle to a process that has been terminated in such a manner, you cannot associate any new processes until the handle is closed and the active process count is below the limit.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	winnt.h (include Windows.h)

See also

GetProcessAffinityMask

JOBOBJECT_END_OF_JOB_TIME_INFORMATION

JOBOBJECT_EXTENDED_LIMIT_INFORMATION

JOBOBJECT_NOTIFICATION_LIMIT_INFORMATION

QueryInformationJobObject

SetInformationJobObject

SetProcessWorkingSetSize

Feedback

Was this page helpful? Yes No

JOBOBJECT_BASIC_PROCESS_ID_LIST structure (winnt.h)

Article09/01/2022

Contains the process identifier list for a job object. If the job is nested, the process identifier list consists of all processes associated with the job and its child jobs.

Syntax

C++

```
typedef struct _JOBOBJECT_BASIC_PROCESS_ID_LIST {
  DWORD     NumberOfAssignedProcesses;
  DWORD     NumberOfProcessIdsInList;
  ULONG_PTR ProcessIdList[1];
} JOBOBJECT_BASIC_PROCESS_ID_LIST, *PJOBOBJECT_BASIC_PROCESS_ID_LIST;
```

Members

`NumberOfAssignedProcesses`

The number of process identifiers to be stored in **ProcessIdList**.

`NumberOfProcessIdsInList`

The number of process identifiers returned in the **ProcessIdList** buffer. If this number is less than **NumberOfAssignedProcesses**, increase the size of the buffer to accommodate the complete list.

`ProcessIdList[1]`

A variable-length array of process identifiers returned by this call. Array elements 0 through **NumberOfProcessIdsInList**– 1 contain valid process identifiers.

Requirements

Expand table

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

Requirement	Value
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	winnt.h (include Windows.h)

See also

QueryInformationJobObject

SetInformationJobObject

Feedback

Was this page helpful? Yes No

JOBOBJECT_BASIC_UI_RESTRICTIONS structure (winnt.h)

Article04/02/2021

Contains basic user-interface restrictions for a job object.

Syntax

C++

```
typedef struct _JOBOBJECT_BASIC_UI_RESTRICTIONS {
  DWORD UIRestrictionsClass;
} JOBOBJECT_BASIC_UI_RESTRICTIONS, *PJOBOBJECT_BASIC_UI_RESTRICTIONS;
```

Members

`UIRestrictionsClass`

The restriction class for the user interface. This member can be one or more of the following values.

Expand table

Value	Meaning
JOB_OBJECT_UILIMIT_DESKTOP 0x00000040	Prevents processes associated with the job from creating desktops and switching desktops using the CreateDesktop and SwitchDesktop functions.
JOB_OBJECT_UILIMIT_DISPLAYSETTINGS 0x00000010	Prevents processes associated with the job from calling the ChangeDisplaySettings function.
JOB_OBJECT_UILIMIT_EXITWINDOWS 0x00000080	Prevents processes associated with the job from calling the ExitWindows or ExitWindowsEx function.
JOB_OBJECT_UILIMIT_GLOBALATOMS 0x00000020	Prevents processes associated with the job from accessing global atoms. When this flag is used, each job has its own atom table.
JOB_OBJECT_UILIMIT_HANDLES 0x00000001	Prevents processes associated with the job from using USER handles owned by processes not associated with the same job.

JOB_OBJECT_UILIMIT_READCLIPBOARD 0x00000002	Prevents processes associated with the job from reading data from the clipboard.
JOB_OBJECT_UILIMIT_SYSTEMPARAMETERS 0x00000008	Prevents processes associated with the job from changing system parameters by using the SystemParametersInfo function.
JOB_OBJECT_UILIMIT_WRITECLIPBOARD 0x00000004	Prevents processes associated with the job from writing data to the clipboard.

Remarks

If you specify the JOB_OBJECT_UILIMIT_HANDLES flag, when a process associated with the job broadcasts messages, they are only sent to top-level windows owned by processes associated with the same job. In addition, hooks can be installed only on threads belonging to processes associated with the job.

To grant access to a User handle to a job that has a user-interface restriction, use the UserHandleGrantAccess function.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	winnt.h (include Windows.h)

See also

ExitWindows

ExitWindowsEx

QueryInformationJobObject

SetInformationJobObject

SystemParametersInfo

Feedback

Was this page helpful? Yes No

JOBOBJECT_CPU_RATE_CONTROL_INFORMATION structure (winnt.h)

Article04/02/2021

Contains CPU rate control information for a job object. This structure is used by the SetInformationJobObject and QueryInformationJobObject functions with the **JobObjectCpuRateControlInformation** information class.

Syntax

C++

```
typedef struct _JOBOBJECT_CPU_RATE_CONTROL_INFORMATION {
  DWORD ControlFlags;
  union {
    DWORD CpuRate;
    DWORD Weight;
    struct {
      WORD MinRate;
      WORD MaxRate;
    } DUMMYSTRUCTNAME;
  } DUMMYUNIONNAME;
} JOBOBJECT_CPU_RATE_CONTROL_INFORMATION,
*PJOBOBJECT_CPU_RATE_CONTROL_INFORMATION;
```

Members

`ControlFlags`

The scheduling policy for CPU rate control. This member can be one of the following values.

Expand table

Value	Meaning
JOB_OBJECT_CPU_RATE_CONTROL_ENABLE 0x1	This flag enables the job's CPU rate to be controlled based on weight or hard cap. You must set this value if you also set **JOB_OBJECT_CPU_RATE_CONTROL_WEIGHT_BASED**, **JOB_OBJECT_CPU_RATE_CONTROL_HARD_CAP**, or **JOB_OBJECT_CPU_RATE_CONTROL_MIN_MAX_RATE**.

JOB_OBJECT_CPU_RATE_CONTROL_WEIGHT_BASED 0x2	The job's CPU rate is calculated based on its relative weight to the weight of other jobs. If this flag is set, the **Weight** member contains more information. If this flag is clear, the **CpuRate** member contains more information. If you set **JOB_OBJECT_CPU_RATE_CONTROL_WEIGHT_BASED**, you cannot also set **JOB_OBJECT_CPU_RATE_CONTROL_MIN_MAX_RATE**.
JOB_OBJECT_CPU_RATE_CONTROL_HARD_CAP 0x4	The job's CPU rate is a hard limit. After the job reaches its CPU cycle limit for the current scheduling interval, no threads associated with the job will run until the next interval. If you set **JOB_OBJECT_CPU_RATE_CONTROL_HARD_CAP**, you cannot also set **JOB_OBJECT_CPU_RATE_CONTROL_MIN_MAX_RATE**.
JOB_OBJECT_CPU_RATE_CONTROL_NOTIFY 0x8	Sends messages when the CPU rate for the job exceeds the rate limits for the job during the tolerance interval.
JOB_OBJECT_ CPU_RATE_CONTROL_MIN_MAX_RATE 0x10	The CPU rate for the job is limited by minimum and maximum rates that you specify in the **MinRate** and **MaxRate** members. If you set **JOB_OBJECT_CPU_RATE_CONTROL_MIN_MAX_RATE**, you can set neither **JOB_OBJECT_CPU_RATE_CONTROL_WEIGHT_BASED** nor **JOB_OBJECT_CPU_RATE_CONTROL_HARD_CAP**.

`DUMMYUNIONNAME`

`DUMMYUNIONNAME.CpuRate`

Specifies the portion of processor cycles that the threads in a job object can use during each scheduling interval, as the number of cycles per 10,000 cycles. If the **ControlFlags** member specifies **JOB_OBJECT_CPU_RATE_WEIGHT_BASED** or **JOB_OBJECT_CPU_RATE_CONTROL_MIN_MAX_RATE**, this member is not used.

Set **CpuRate** to a percentage times 100. For example, to let the job use 20% of the CPU, set **CpuRate** to 20 times 100, or 2,000.

Do not set **CpuRate** to 0. If **CpuRate** is 0, SetInformationJobObject returns **INVALID_ARGS**.

`DUMMYUNIONNAME.Weight`

If the **ControlFlags** member specifies **JOB_OBJECT_CPU_RATE_WEIGHT_BASED**, this member specifies the scheduling weight of the job object, which determines the share of processor time given to the job relative to other workloads on the processor.

This member can be a value from 1 through 9, where 1 is the smallest share and 9 is the largest share. The default is 5, which should be used for most workloads.

If the **ControlFlags** member specifies **JOB_OBJECT_CPU_RATE_CONTROL_MIN_MAX_RATE**, this member is not used.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.MinRate`

Specifies the minimum portion of the processor cycles that the threads in a job object can reserve during each scheduling interval. Specify this rate as a percentage times 100. For example, to set a minimum rate of 50%, specify 50 times 100, or 5,000.

For the minimum rates to work correctly, the sum of the minimum rates for all of the job objects in the system cannot exceed 10,000, which is the equivalent of 100%.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.MaxRate`

Specifies the maximum portion of processor cycles that the threads in a job object can use during each scheduling interval. Specify this rate as a percentage times 100. For example, to set a maximum rate of 50%, specify 50 times 100, or 5,000.

After the job reaches this limit for a scheduling interval, no threads associated with the job can run until the next scheduling interval.

Remarks

You can set CPU rate control for multiple jobs in a hierarchy of nested jobs. When you set CPU rate control for a job object, the settings apply to the job and its child jobs in the hierarchy. When you set CPU rate control for a job in a nested hierarchy, the system calculates the corresponding quotas with respect to the CPU rate control of the immediate parent job for the job. In other words, the rates set for the job represent its portion of the CPU rate that is allocated to its parent job. If a job object does not have a parent with CPU rate control turned on in the chain of its parent jobs, the rate control for the job represents the portion of the CPU for the entire system.

CPU rate control cannot be used by job objects in applications running under Remote Desktop Services (formerly Terminal Services) if Dynamic Fair Share Scheduling (DFSS) is in effect.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]
Header	winnt.h

See also

JOBOBJECT_NOTIFICATION_LIMIT_INFORMATION

QueryInformationJobObject

SetInformationJobObject

Feedback

Was this page helpful? Yes No

JOBOBJECT_END_OF_JOB_TIME_INFORMATION structure (winnt.h)

Article02/22/2024

Specifies the action the system will perform when an end-of-job time limit is exceeded.

Syntax

C++

```
typedef struct _JOBOBJECT_END_OF_JOB_TIME_INFORMATION {
  DWORD EndOfJobTimeAction;
} JOBOBJECT_END_OF_JOB_TIME_INFORMATION,
*PJOBOBJECT_END_OF_JOB_TIME_INFORMATION;
```

Members

`EndOfJobTimeAction`

The action that the system will perform when the end-of-job time limit has been exceeded. This member can be one of the following values.

Expand table

Value	Meaning
JOB_OBJECT_TERMINATE_AT_END_OF_JOB 0	Terminates all processes and sets the exit status to ERROR_NOT_ENOUGH_QUOTA. The processes cannot prevent or delay their own termination. The job object is set to the signaled state and remains signaled until this limit is reset. No additional processes can be assigned to the job until the limit is reset. This is the default termination action.
JOB_OBJECT_POST_AT_END_OF_JOB 1	Posts a completion packet to the completion port using the PostQueuedCompletionStatus function. After the completion packet is posted, the system clears the end-of-job time limit, and processes in the job can continue their execution. If no completion port is associated with the job when the time limit has been exceeded, the action

	taken is the same as for JOB_OBJECT_TERMINATE_AT_END_OF_JOB.

Remarks

The end-of-job time limit is specified in the **PerJobUserTimeLimit** member of the JOBOBJECT_BASIC_LIMIT_INFORMATION structure.

To associate a completion port with a job, use the JOBOBJECT_ASSOCIATE_COMPLETION_PORT structure.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	winnt.h (include Windows.h)

See also

JOBOBJECT_ASSOCIATE_COMPLETION_PORT

JOBOBJECT_BASIC_LIMIT_INFORMATION

PostQueuedCompletionStatus

QueryInformationJobObject

SetInformationJobObject

Feedback

Was this page helpful? Yes No

JOBOBJECT_EXTENDED_LIMIT_INFORMATION structure (winnt.h)

Article04/02/2021

Contains basic and extended limit information for a job object.

Syntax

C++

```
typedef struct _JOBOBJECT_EXTENDED_LIMIT_INFORMATION {
  JOBOBJECT_BASIC_LIMIT_INFORMATION BasicLimitInformation;
  IO_COUNTERS                       IoInfo;
  SIZE_T                            ProcessMemoryLimit;
  SIZE_T                            JobMemoryLimit;
  SIZE_T                            PeakProcessMemoryUsed;
  SIZE_T                            PeakJobMemoryUsed;
} JOBOBJECT_EXTENDED_LIMIT_INFORMATION,
*PJOBOBJECT_EXTENDED_LIMIT_INFORMATION;
```

Members

`BasicLimitInformation`

A JOBOBJECT_BASIC_LIMIT_INFORMATION structure that contains basic limit information.

`IoInfo`

Reserved.

`ProcessMemoryLimit`

If the **LimitFlags** member of the JOBOBJECT_BASIC_LIMIT_INFORMATION structure specifies the **JOB_OBJECT_LIMIT_PROCESS_MEMORY** value, this member specifies the limit for the virtual memory that can be committed by a process. Otherwise, this member is ignored.

`JobMemoryLimit`

If the **LimitFlags** member of the JOBOBJECT_BASIC_LIMIT_INFORMATION structure specifies the **JOB_OBJECT_LIMIT_JOB_MEMORY** value, this member specifies the limit

for the virtual memory that can be committed for the job. Otherwise, this member is ignored.

`PeakProcessMemoryUsed`

The peak memory used by any process ever associated with the job.

`PeakJobMemoryUsed`

The peak memory usage of all processes currently associated with the job.

Remarks

The system tracks the value of **PeakProcessMemoryUsed** and **PeakJobMemoryUsed** constantly. This allows you know the peak memory usage of each job. You can use this information to establish a memory limit using the **JOB_OBJECT_LIMIT_PROCESS_MEMORY** or **JOB_OBJECT_LIMIT_JOB_MEMORY** value.

Note that the job memory and process memory limits are very similar in operation, but they are independent. You could set a job-wide limit of 100 MB with a per-process limit of 10 MB. In this scenario, no single process could commit more than 10 MB, and the set of processes associated with a job could never exceed 100 MB.

To register for notifications that a job has exceeded its peak memory limit while allowing processes to continue to commit memory, use the SetInformationJobObject function with the **JobObjectNotificationLimitInformation** information class.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	winnt.h (include Windows.h)

See also

JOBOBJECT_BASIC_LIMIT_INFORMATION

JOBOBJECT_NOTIFICATION_LIMIT_INFORMATION

QueryInformationJobObject

SetInformationJobObject

Feedback

Was this page helpful? Yes No

JOBOBJECT_IO_RATE_CONTROL_INFORMATION structure (jobapi2.h)

Article01/09/2025

Windows 10, version 1607, and newer: This structure is not supported.

Contains information used to control the I/O rate for a job. This structure is used by the SetIoRateControlInformationJobObject and QueryIoRateControlInformationJobObject functions.

Syntax

C++

```
typedef struct JOBOBJECT_IO_RATE_CONTROL_INFORMATION {
  LONG64 MaxIops;
  LONG64 MaxBandwidth;
  LONG64 ReservationIops;
  PCWSTR VolumeName;
  ULONG  BaseIoSize;
  ULONG  ControlFlags;
} JOBOBJECT_IO_RATE_CONTROL_INFORMATION;
```

Members

`MaxIops`

The maximum limit for the I/O rate in I/O operations per second (IOPS). Set to 0 if to specify no limit.

When you set both **MaxIops** and **MaxBandwith**, the operating system enforces the first limit that the I/O rate reaches.

`MaxBandwidth`

The maximum limit for the I/O rate in bytes per second. Set to 0 to specify no limit.

When you set both **MaxBandwith** and **MaxIops**, the operating system enforces the first limit that the I/O rate reaches.

`ReservationIops`

Sets a minimum I/O rate which the operating system reserves for the job. To make no reservation for the job, set this value to 0.

The operating system allows the job to perform I/O operations at this rate, if possible. If the sum of the minimum rates for all jobs exceeds the capacity of the operating system, the rate at which the operating system allows each job to perform I/O operations is proportional to the reservation for the job.

`VolumeName`

The NT device name for the volume to which you want to apply the policy for the I/O rate. For information about NT device names, see NT Device Names.

If this member is **NULL**, the policy for the I/O rate applies to all of the volumes for the operating system. For example, if this member is **NULL** and the **MaxIops** member is 100, the maximum limit for the I/O rate for each volume is set to 100 IOPS, instead of setting an aggregate limit for the I/O rate across all volumes of 100 IOPS.

`BaseIoSize`

The base size of the normalized I/O unit, in bytes. For example, if the **BaseIoSize** member is 8,000, every 8,000 bytes counts as one I/O unit. 4,000 bytes is also one I/O unit in this example, while 8,001 bytes is two I/O units.

You can set the value of this base I/O size by using the **StorageBaseIOSize** value of the **HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\QoS** registry key.

The value of the **BaseIoSize** member is subject to the following constraints:

- The SetIoRateControlInformationJobObject function requires that the **BaseIoSize** member of the **JOBOBJECT_IO_RATE_CONTROL_INFORMATION** structure that you pass to the function is 0.
- The QueryIoRateControlInformationJobObject method sets the **BaseIoSize** member of this structure to 0 if the volume that the **VolumeName** member specifies does not support the control of the I/O rate.
- The QueryIoRateControlInformationJobObject method sets the **BaseIoSize** member of this structure to the base size of the normalized I/O unit if the volume that the **VolumeName** member specifies does support the control of the I/O rate.

To query for the base size of the normalized I/O unit without creating a job, call QueryIoRateControlInformationJobObject with the *hJob* parameter set to **NULL** from a process that is not associated with a job.

`ControlFlags`

The policy for control of the I/O rate. This member can be one of the following values.

Expand table

Value	Meaning
JOB_OBJECT_IO_RATE_CONTROL_ENABLE 0x1	Turns on control of the I/O rate for the job when this structure is passed to the SetIoRateControlInformationJobObject function. Indicates that control of the I/O rate for the job is turned on when this structure is used with the QueryIoRateControlInformationJobObject function.

Remarks

You can only set one I/O rate control for a job in a hierarchy of nested jobs. The settings that you specify apply to that job and the child jobs in the hierarchy for that job. The settings do not apply to the chain of jobs from the parent job up to the top of the hierarchy. You still can change the settings for the original job in the hierarchy on which you set I/O rate control. However, attempts to set values for the control of the I/O rate for any other jobs in the hierarchy, including the parent jobs, fail.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 10 [desktop apps only]
Minimum supported server	Windows Server 2016 [desktop apps only]
Header	jobapi2.h

See also

QueryIoRateControlInformationJobObject

SetIoRateControlInformationJobObject

JOBOBJECT_LIMIT_VIOLATION_INFORMATION structure (winnt.h)

Article04/02/2021

Contains information about resource notification limits that have been exceeded for a job object. This structure is used with the QueryInformationJobObject function with the **JobObjectLimitViolationInformation** information class.

Syntax

C++

```
typedef struct _JOBOBJECT_LIMIT_VIOLATION_INFORMATION {
  DWORD                             LimitFlags;
  DWORD                             ViolationLimitFlags;
  DWORD64                           IoReadBytes;
  DWORD64                           IoReadBytesLimit;
  DWORD64                           IoWriteBytes;
  DWORD64                           IoWriteBytesLimit;
  LARGE_INTEGER                     PerJobUserTime;
  LARGE_INTEGER                     PerJobUserTimeLimit;
  DWORD64                           JobMemory;
  DWORD64                           JobMemoryLimit;
  JOBOBJECT_RATE_CONTROL_TOLERANCE RateControlTolerance;
  JOBOBJECT_RATE_CONTROL_TOLERANCE RateControlToleranceLimit;
} JOBOBJECT_LIMIT_VIOLATION_INFORMATION,
*PJOBOBJECT_LIMIT_VIOLATION_INFORMATION;
```

Members

`LimitFlags`

Flags that identify the notification limits in effect for the job. This member is a bitfield that determines whether other structure members are used. This member can be any combination of the following values.

Expand table

Value	Meaning
JOB_OBJECT_LIMIT_JOB_MEMORY 0x00000200	The job has a committed memory notification limit. The **JobMemoryLimit** member contains more information.

JOB_OBJECT_LIMIT_JOB_READ_BYTES 0x00010000	The job has an I/O read bytes notification limit. The **IoReadBytesLimit** member contains more information.
JOB_OBJECT_LIMIT_JOB_TIME 0x00000004	The job has a user-mode execution time notification limit. The **PerJobUserTimeLimit** member contains more information.
JOB_OBJECT_LIMIT_JOB_WRITE_BYTES 0x00020000	The job has an I/O write bytes notification limit. The **IoWriteBytesLimit** member contains more information.
JOB_OBJECT_LIMIT_RATE_CONTROL 0x00040000	The extent to which a job can exceed its CPU rate control limit. The **RateControlToleranceLimit** member contains more information.

`ViolationLimitFlags`

Flags that identify the notification limits that have been exceeded. This member is a bitfield that determines whether other structure members are used. This member can be any combination of the following values.

Expand table

Value	Meaning
JOB_OBJECT_LIMIT_READ_BYTES 0x00010000	The job's I/O read bytes notification limit has been exceeded. The **IoReadBytes** member contains more information.
JOB_OBJECT_LIMIT_WRITE_BYTES 0x00020000	The job's I/O write bytes notification limit has been exceeded. The **IoWriteBytes** member contains more information.
JOB_OBJECT_LIMIT_JOB_TIME 0x00000004	The job's user-mode execution time notification limit has been exceeded. The **PerJobUserTime** member contains more information.
JOB_OBJECT_LIMIT_JOB_MEMORY 0x00000200	The job's committed memory notification limit has been exceeded. The **JobMemory** member contains more information.
JOB_OBJECT_LIMIT_RATE_CONTROL 0x00040000	The job's CPU rate control limit has been exceeded. The **RateControlTolerance** member contains more information.

`IoReadBytes`

If the ViolationLimitFlags member specifies JOB_OBJECT_LIMIT_READ_BYTES, this member contains the total I/O read bytes for all processes in the job at the time the

notification was sent.

`IoReadBytesLimit`

If the LimitFlags member specifies JOB_OBJECT_LIMIT_READ_BYTES, this member contains the I/O read bytes notification limit in effect for the job.

`IoWriteBytes`

If the ViolationLimitFlags member specifies JOB_OBJECT_LIMIT_WRITE_BYTES, this member contains the total I/O write bytes for all processes in the job at the time the notification was sent.

`IoWriteBytesLimit`

If the LimitFlags member specifies JOB_OBJECT_LIMIT_WRITE_BYTES, this member contains the I/O write bytes notification limit in effect for the job.

`PerJobUserTime`

If the ViolationLimitFlags member specifies JOB_OBJECT_LIMIT_JOB_TIME, this member contains the total user-mode execution time for all processes in the job at the time the notification was sent.

`PerJobUserTimeLimit`

If the LimitFlags member specifies JOB_OBJECT_LIMIT_JOB_TIME, this member contains the user-mode execution notification limit in effect for the job.

`JobMemory`

If the ViolationLimitFlags member specifies JOB_OBJECT_LIMIT_JOB_MEMORY, this member contains the committed memory for all processes in the job at the time the notification was sent.

`JobMemoryLimit`

If the LimitFlags member specifies JOB_OBJECT_LIMIT_JOB_MEMORY, this member contains the committed memory limit in effect for the job.

`RateControlTolerance`

If the *LimitFlags* parameter specifies JOB_OBJECT_LIMIT_RATE_CONTROL, this member specifies the extent to which the job exceeded its CPU rate control limits at the time the notification was sent. This member can be one of the following values.

Expand table

Value	Meaning
ToleranceLow 1	The job exceeded its CPU rate control limits for 20% of the tolerance interval.
ToleranceMedium 2	The job exceeded its CPU rate control limits for 40% of the tolerance interval.
ToleranceHigh 3	The job exceeded its CPU rate control limits for 60% of the tolerance interval.

`RateControlToleranceLimit`

If the *LimitFlags* parameter specifies JOB_OBJECT_LIMIT_RATE_CONTROL, this member contains the CPU rate control notification limits specified for the job.

Expand table

Value	Meaning
ToleranceIntervalShort 1	The tolerance interval is 10 seconds.
ToleranceIntervalMedium 2	The tolerance interval is one minute.
ToleranceIntervalLong 3	The tolerance interval is 10 minutes.

Remarks

When any notification limit specified in a JOBOBJECT_NOTIFICATION_LIMIT_INFORMATION structure is exceeded, the system sends a JOB_OBJECT_MSG_NOTIFICATION_LIMIT message to the I/O completion port associated with the job.

To retrieve information about the limits that were exceeded, the application monitoring the I/O completion port must call the QueryInformationJobObject function with the **JobObjectLimitViolationInformation** information class and a pointer to a **JOBOBJECT_LIMIT_VIOLATION_INFORMATION** structure.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]
Header	winnt.h

See also

QueryInformationJobObject

SetInformationJobObject

Feedback

Was this page helpful? Yes No

JOBOBJECT_LIMIT_VIOLATION_INFORMATION_2 structure (winnt.h)

Article07/27/2022

Contains extended information about resource notification limits that have been exceeded for a job object. This structure is used with the QueryInformationJobObject function with the **JobObjectLimitViolationInformation2** information class.

Syntax

C++

```
typedef struct JOBOBJECT_LIMIT_VIOLATION_INFORMATION_2 {
  DWORD                           LimitFlags;
  DWORD                           ViolationLimitFlags;
  DWORD64                         IoReadBytes;
  DWORD64                         IoReadBytesLimit;
  DWORD64                         IoWriteBytes;
  DWORD64                         IoWriteBytesLimit;
  LARGE_INTEGER                   PerJobUserTime;
  LARGE_INTEGER                   PerJobUserTimeLimit;
  DWORD64                         JobMemory;
  union {
    DWORD64 JobHighMemoryLimit;
    DWORD64 JobMemoryLimit;
  } DUMMYUNIONNAME;
  union {
    JOBOBJECT_RATE_CONTROL_TOLERANCE RateControlTolerance;
    JOBOBJECT_RATE_CONTROL_TOLERANCE CpuRateControlTolerance;
  } DUMMYUNIONNAME2;
  union {
    JOBOBJECT_RATE_CONTROL_TOLERANCE RateControlToleranceLimit;
    JOBOBJECT_RATE_CONTROL_TOLERANCE CpuRateControlToleranceLimit;
  } DUMMYUNIONNAME3;
  DWORD64                         JobLowMemoryLimit;
  JOBOBJECT_RATE_CONTROL_TOLERANCE IoRateControlTolerance;
  JOBOBJECT_RATE_CONTROL_TOLERANCE IoRateControlToleranceLimit;
  JOBOBJECT_RATE_CONTROL_TOLERANCE NetRateControlTolerance;
  JOBOBJECT_RATE_CONTROL_TOLERANCE NetRateControlToleranceLimit;
} JOBOBJECT_LIMIT_VIOLATION_INFORMATION_2;
```

Members

`LimitFlags`

Flags that identify the notification limits in effect for the job. This member is a bitfield that determines whether other structure members are used. This member can be any combination of the following values.

Expand table

Value	Meaning
JOB_OBJECT_LIMIT_JOB_MEMORY_HIGH 0x00000200	The job has a committed memory notification limit. The **JobHighMemoryLimit** member contains more information.
JOB_OBJECT_LIMIT_JOB_MEMORY_LOW 0x00008000	The job has a committed minimum memory notification limit. The **JobLowMemoryLimit** member contains more information.
JOB_OBJECT_LIMIT_JOB_READ_BYTES 0x00010000	The job has an I/O read bytes notification limit. The **IoReadBytesLimit** member contains more information.
JOB_OBJECT_LIMIT_JOB_WRITE_BYTES 0x00020000	The job has an I/O write bytes notification limit. The **IoWriteBytesLimit** member contains more information.
JOB_OBJECT_LIMIT_JOB_TIME 0x00000004	The job has a user-mode execution time notification limit. The **PerJobUserTimeLimit** member contains more information.
JOB_OBJECT_LIMIT_RATE_CONTROL 0x00040000	The job has notification limit for the extent to which a job can exceed its CPU rate control limit. The **RateControlToleranceLimit** member contains more information.
JOB_OBJECT_LIMIT_CPU_RATE_CONTROL 0x00040000	The job has notification limit for the extent to which a job can exceed its CPU rate control limit. The **CpuRateControlToleranceLimit** member contains more information.
JOB_OBJECT_LIMIT_IO_RATE_CONTROL 0x00080000	The job has notification limit for the extent to which a job can exceed its I/O rate control limit. The **IoRateControlToleranceLimit** member contains more information.
JOB_OBJECT_LIMIT_NET_RATE_CONTROL 0x00100000	The job has notification limit for the extent to which a job can exceed its network rate control limit. The **NetRateControlToleranceLimit** member contains more information.

`ViolationLimitFlags`

Flags that identify the notification limits that have been exceeded. This member is a bitfield that determines whether other structure members are used. This member can be any

combination of the following values.

Expand table

Value	Meaning
JOB_OBJECT_LIMIT_JOB_READ_BYTES 0x00010000	The job's I/O read bytes notification limit has been exceeded. The **IoReadBytes** member contains more information.
JOB_OBJECT_LIMIT_JOB_WRITE_BYTES 0x00020000	The job's I/O write bytes notification limit has been exceeded. The **IoWriteBytes** member contains more information.
JOB_OBJECT_LIMIT_JOB_TIME 0x00000004	The job's user-mode execution time notification limit has been exceeded. The **PerJobUserTime** member contains more information.
JOB_OBJECT_LIMIT_JOB_MEMORY_HIGH 0x00000200	The job's committed maximum memory notification limit has been exceeded. The **JobMemory** member contains more information.
JOB_OBJECT_LIMIT_JOB_MEMORY_LOW 0x00008000	The job's committed memory has fallen below its minimum notification limit. The **JobMemory** member contains more information.
JOB_OBJECT_LIMIT_RATE_CONTROL 0x00040000	The job's CPU rate control limit has been exceeded. The **RateControlTolerance** member contains more information.
JOB_OBJECT_LIMIT_CPU_RATE_CONTROL 0x00040000	The job's CPU rate control limit has been exceeded. The **CpuRateControlTolerance** member contains more information.
JOB_OBJECT_LIMIT_IO_RATE_CONTROL 0x00080000	The job's I/O rate control limit has been exceeded. The **IoRateControlTolerance** member contains more information.
JOB_OBJECT_LIMIT_NET_RATE_CONTROL 0x00100000	The job's network rate control limit has been exceeded. The **NetworkRateControlTolerance** member contains more information.

`IoReadBytes`

If the **ViolationLimitFlags** member specifies **JOB_OBJECT_LIMIT_JOB_READ_BYTES**, this member contains the total I/O read bytes for all processes in the job at the time the notification was sent.

`IoReadBytesLimit`

If the **LimitFlags** member specifies **JOB_OBJECT_LIMIT_JOB_READ_BYTES**, this member contains the I/O read bytes notification limit in effect for the job.

`IoWriteBytes`

If the **ViolationLimitFlags** member specifies **JOB_OBJECT_LIMIT_JOB_WRITE_BYTES**, this member contains the total I/O write bytes for all processes in the job at the time the notification was sent.

`IoWriteBytesLimit`

If the **LimitFlags** member specifies **JOB_OBJECT_LIMIT_JOB_WRITE_BYTES**, this member contains the I/O write bytes notification limit in effect for the job.

`PerJobUserTime`

If the **ViolationLimitFlags** member specifies **JOB_OBJECT_LIMIT_JOB_TIME**, this member contains the total user-mode execution time for all processes in the job at the time the notification was sent.

`PerJobUserTimeLimit`

If the **LimitFlags** member specifies **JOB_OBJECT_LIMIT_JOB_TIME**, this member contains the user-mode execution notification limit in effect for the job.

`JobMemory`

If the **ViolationLimitFlags** member specifies **JOB_OBJECT_LIMIT_JOB_MEMORY_HIGH** or **JOB_OBJECT_LIMIT_JOB_MEMORY_LOW**, this member contains the committed memory for all processes in the job at the time the notification was sent.

`DUMMYUNIONNAME`

`DUMMYUNIONNAME.JobHighMemoryLimit`

If the **LimitFlags** member specifies **JOB_OBJECT_LIMIT_JOB_MEMORY_HIGH**, this member contains the committed maximum memory limit in effect for the job.

`DUMMYUNIONNAME.JobMemoryLimit`

If the **LimitFlags** member specifies **JOB_OBJECT_LIMIT_JOB_MEMORY**, this member contains the committed maximum memory limit in effect for the job.

`DUMMYUNIONNAME2`

`DUMMYUNIONNAME2.RateControlTolerance`

If the **LimitFlags** member specifies **JOB_OBJECT_LIMIT_RATE_CONTROL**, this member specifies the extent to which the job exceeded its CPU rate control limits at the time the notification was

sent. This member can be one of the following values.

Expand table

Value	Meaning
ToleranceLow 1	The job exceeded its CPU rate control limits for 20% of the tolerance interval.
ToleranceMedium 2	The job exceeded its CPU rate control limits for 40% of the tolerance interval.
ToleranceHigh 3	The job exceeded its CPU rate control limits for 60% of the tolerance interval.

`DUMMYUNIONNAME2.CpuRateControlTolerance`

If the **LimitFlags** member specifies **JOB_OBJECT_LIMIT_CPU_RATE_CONTROL**, this member specifies the extent to which the job exceeded its CPU rate control limits at the time the notification was sent. This member can be one of the following values.

Expand table

Value	Meaning
ToleranceLow 1	The job exceeded its CPU rate control limits for 20% of the tolerance interval.
ToleranceMedium 2	The job exceeded its CPU rate control limits for 40% of the tolerance interval.
ToleranceHigh 3	The job exceeded its CPU rate control limits for 60% of the tolerance interval.

`DUMMYUNIONNAME3`

`DUMMYUNIONNAME3.RateControlToleranceLimit`

If the **LimitFlags** member specifies **JOB_OBJECT_LIMIT_RATE_CONTROL**, this member contains the CPU rate control notification limits specified for the job.

Expand table

Value	Meaning
ToleranceLow 1	The job can exceed its CPU rate control limits for 20% of the tolerance interval.

ToleranceMedium 2	The job can exceed its CPU rate control limits for 40% of the tolerance interval.
ToleranceHigh 3	The job can exceed its CPU rate control limits for 60% of the tolerance interval.

`DUMMYUNIONNAME3.CpuRateControlToleranceLimit`

If the *LimitFlags* parameter specifies **JOB_OBJECT_LIMIT_CPU_RATE_CONTROL**, this member contains the CPU rate control notification limits specified for the job.

Expand table

Value	Meaning
ToleranceLow 1	The job can exceed its CPU rate control limits for 20% of the tolerance interval.
ToleranceMedium 2	The job can exceed its CPU rate control limits for 40% of the tolerance interval.
ToleranceHigh 3	The job can exceed its CPU rate control limits for 60% of the tolerance interval.

`JobLowMemoryLimit`

If the **LimitFlags** member specifies **JOB_OBJECT_LIMIT_JOB_MEMORY_LOW**, this member contains the committed minimum memory limit in effect for the job.

`IoRateControlTolerance`

If the **LimitFlags** member specifies **JOB_OBJECT_LIMIT_IO_RATE_CONTROL**, this member specifies the extent to which the job exceeded its I/O rate control limits at the time the notification was sent. This member can be one of the following values.

Expand table

Value	Meaning
ToleranceLow 1	The job exceeded its I/O rate control limits for 20% of the tolerance interval.
ToleranceMedium 2	The job exceeded its I/O rate control limits for 40% of the tolerance interval.
ToleranceHigh 3	The job exceeded its I/O rate control limits for 60% of the tolerance interval.

`IoRateControlToleranceLimit`

If the *LimitFlags* parameter specifies **JOB_OBJECT_LIMIT_IO_RATE_CONTROL**, this member contains the I/O rate control notification limits specified for the job.

Expand table

Value	Meaning
ToleranceLow 1	The job can exceed its I/O rate control limits for 20% of the tolerance interval.
ToleranceMedium 2	The job can exceed its I/O rate control limits for 40% of the tolerance interval.
ToleranceHigh 3	The job can exceed its I/O rate control limits for 60% of the tolerance interval.

`NetRateControlTolerance`

If the **LimitFlags** member specifies **JOB_OBJECT_LIMIT_NET_RATE_CONTROL**, this member specifies the extent to which the job exceeded its network rate control limits at the time the notification was sent. This member can be one of the following values.

Expand table

Value	Meaning
ToleranceLow 1	The job exceeded its network rate control limits for 20% of the tolerance interval.
ToleranceMedium 2	The job exceeded its network rate control limits for 40% of the tolerance interval.
ToleranceHigh 3	The job exceeded its network rate control limits for 60% of the tolerance interval.

`NetRateControlToleranceLimit`

If the *LimitFlags* parameter specifies **JOB_OBJECT_LIMIT_NETWORK_RATE_CONTROL**, this member contains the network rate control notification limits specified for the job.

Expand table

Value	Meaning

ToleranceLow 1	The job can exceed its network rate control limits for 20% of the tolerance interval.
ToleranceMedium 2	The job can exceed its network rate control limits for 40% of the tolerance interval.
ToleranceHigh 3	The job can exceed its network rate control limits for 60% of the tolerance interval.

Remarks

When any notification limit specified in a JOBOBJECT_NOTIFICATION_LIMIT_INFORMATION_2 structure is exceeded, the system sends a **JOB_OBJECT_MSG_NOTIFICATION_LIMIT** message to the I/O completion port associated with the job.

To retrieve information about the limits that were exceeded, the application monitoring the I/O completion port must call the QueryInformationJobObject function with the **JobObjectLimitViolationInformation2** information class and a pointer to a **JOBOBJECT_LIMIT_VIOLATION_INFORMATION_2** structure.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 10 [desktop apps only]
Minimum supported server	Windows Server 2016 [desktop apps only]
Header	winnt.h

See also

JOBOBJECT_NOTIFICATION_LIMIT_INFORMATION_2

QueryInformationJobObject

SetInformationJobObject

JOBOBJECT_NET_RATE_CONTROL_INFORMATION structure (winnt.h)

Article02/22/2024

Contains information used to control the network traffic for a job. This structure is used by the SetInformationJobObject and QueryInformationJobObject functions with the **JobObjectNetRateControlInformation** information class.

Syntax

```
C++

typedef struct JOBOBJECT_NET_RATE_CONTROL_INFORMATION {
  DWORD64                           MaxBandwidth;
  JOB_OBJECT_NET_RATE_CONTROL_FLAGS ControlFlags;
  BYTE                              DscpTag;
} JOBOBJECT_NET_RATE_CONTROL_INFORMATION;
```

Members

`MaxBandwidth`

The maximum bandwidth for outgoing network traffic for the job, in bytes.

`ControlFlags`

A combination of JOB_OBJECT_NET_RATE_CONTROL_FLAGS enumeration values that specify the scheduling policy for network rate control.

`DscpTag`

The value to use for the Differentiated Service code point (DSCP) field to turn on network quality of service (QoS) for all outgoing network traffic generated by the processes of the job object. The valid range is from 0x00 through 0x3F. For information about DSCP, see Differentiated Services.

Remarks

You can only set the control of the network traffic on one job in a hierarchy of nested jobs. The settings that you specify apply to that job and the child jobs in the hierarchy for that job. The settings do not apply to the chain of jobs from the parent job up to the top of the hierarchy.

You can change the settings on the original job in the hierarchy on which you set rate control. However, attempts to set values for the control of the network rate for any other jobs in the hierarchy, including the parent jobs, fail.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 10 [desktop apps only]
Minimum supported server	Windows Server 2016 [desktop apps only]
Header	winnt.h

See also

JOB_OBJECT_NET_RATE_CONTROL_FLAGS

QueryInformationJobObject

SetInformationJobObject

JOBOBJECT_NOTIFICATION_LIMIT_INFORMATION structure (winnt.h)

Article04/02/2021

Contains information about notification limits for a job object. This structure is used by the SetInformationJobObject and QueryInformationJobObject functions with the **JobObjectNotificationLimitInformation** information class.

Syntax

C++

```
typedef struct _JOBOBJECT_NOTIFICATION_LIMIT_INFORMATION {
  DWORD64                                   IoReadBytesLimit;
  DWORD64                                   IoWriteBytesLimit;
  LARGE_INTEGER                             PerJobUserTimeLimit;
  DWORD64                                   JobMemoryLimit;
  JOBOBJECT_RATE_CONTROL_TOLERANCE          RateControlTolerance;
  JOBOBJECT_RATE_CONTROL_TOLERANCE_INTERVAL RateControlToleranceInterval;
  DWORD                                     LimitFlags;
} JOBOBJECT_NOTIFICATION_LIMIT_INFORMATION,
*PJOBOBJECT_NOTIFICATION_LIMIT_INFORMATION;
```

Members

`IoReadBytesLimit`

If the *LimitFlags* member specifies JOB_OBJECT_LIMIT_JOB_READ_BYTES, this member is the notification limit for total I/O bytes read by all processes in the job. Otherwise, this member is ignored.

`IoWriteBytesLimit`

If the *LimitFlags* parameter specifies JOB_OBJECT_LIMIT_JOB_WRITE_BYTES, this member is the notification limit for total I/O bytes written by all processes in the job. Otherwise, this member is ignored.

`PerJobUserTimeLimit`

If the *LimitFlags* parameter specifies JOB_OBJECT_LIMIT_JOB_TIME, this member is the notification limit for per-job user-mode execution time, in 100-nanosecond ticks. Otherwise, this member is ignored.

The system adds the accumulated execution time of processes associated with the job to this limit when the limit is set. For example, if a process associated with the job has already accumulated 5 minutes of user-mode execution time and the limit is set to 1 minute, the limit actually enforced is 6 minutes.

To specify **PerJobUserTimeLimit** as an enforceable limit and terminate processes in jobs that exceed the limit, see the JOBOBJECT_BASIC_LIMIT_INFORMATION structure.

`JobMemoryLimit`

If the *LimitFlags* parameter specifies JOB_OBJECT_LIMIT_JOB_MEMORY, this member is the notification limit for total virtual memory that can be committed by all processes in the job, in bytes. Otherwise, this member is ignored.

To specify **JobMemoryLimit** as an enforceable limit and prevent processes in jobs that exceed the limit from continuing to commit memory, see the JOBOBJECT_EXTENDED_LIMIT_INFORMATION structure.

`RateControlTolerance`

If the *LimitFlags* parameter specifies JOB_OBJECT_LIMIT_RATE_CONTROL, this member specifies the extent to which a job can exceed its CPU rate control limits during the interval specified by the **RateControlToleranceInterval** member. Otherwise, this member is ignored.

This member can be one of the following values. If no value is specified, **ToleranceHigh** is used.

Expand table

Value	Meaning
ToleranceLow 1	The job can exceed its CPU rate control limits for 20% of the tolerance interval.
ToleranceMedium 2	The job can exceed its CPU rate control limits for 40% of the tolerance interval.
ToleranceHigh 3	The job can exceed its CPU rate control limits for 60% of the tolerance interval.

`RateControlToleranceInterval`

If the *LimitFlags* parameter specifies JOB_OBJECT_LIMIT_RATE_CONTROL, this member specifies the interval during which a job's CPU usage is monitored to determine whether the job has exceeded its CPU rate control limits. Otherwise, this member is ignored.

This member can be one of the following values. If no value is specified, **ToleranceIntervalShort** is used.

Expand table

Value	Meaning
ToleranceIntervalShort 1	The tolerance interval is 10 seconds.
ToleranceIntervalMedium 2	The tolerance interval is one minute.
ToleranceIntervalLong 3	The tolerance interval is 10 minutes.

`LimitFlags`

The limit flags that are in effect. This member is a bitfield that determines whether other structure members are used. Any combination of the following values can be specified.

Expand table

Value	Meaning
JOB_OBJECT_LIMIT_JOB_MEMORY 0x00000200	Establishes the committed memory limit to the job-wide sum of committed memory for all processes associated with the job. The **JobMemoryLimit** member contains additional information.
JOB_OBJECT_LIMIT_JOB_READ_BYTES 0x00010000	Establishes the I/O read bytes limit to the job-wide sum of I/O bytes read by all processes associated with the job. The **IoReadBytesLimit** member contains more information.
JOB_OBJECT_LIMIT_JOB_WRITE_BYTES 0x00020000	Establishes the I/O write bytes limit to the job-wide sum of I/O bytes written by all processes associated with the job. The **IoWriteBytesLimit** member contains more information.
JOB_OBJECT_LIMIT_JOB_TIME 0x00000004	Establishes the limit for user-mode execution time for the job. The **PerJobUserTimeLimit** member contains additional information.
JOB_OBJECT_LIMIT_RATE_CONTROL 0x00040000	Establishes the notification threshold for the CPU rate control limits established for the job. The **RateControlTolerance** and **RateControlToleranceInterval** members contain additional information.

	CPU rate control limits are established by calling SetInformationJobObject with the **JobObjectCpuRateInformationClass** information class.

Remarks

When a notification limit is exceeded, the system sends a JOB_OBJECT_MSG_NOTIFICATION_LIMIT message to the I/O completion port associated with the job. Processes in the job continue to run and can continue to allocate memory or transmit read or write bytes beyond the specified limits.

When the application monitoring the I/O completion port receives a JOB_OBJECT_MSG_NOTIFICATION_LIMIT message, it must call QueryInformationJobObject with the **JobObjectLimitViolationInformation** information class. Limit violation information is received in a JOBOBJECT_LIMIT_VIOLATION_STRUCTURE that contains information about all notification limits that were exceeded at the time of the query. The system will not send another JOB_OBJECT_MSG_NOTIFICATION_LIMIT message until after **QueryInformationJobObject** is called.

CPU rate control limits for a job are established in a JOBOBJECT_CPU_RATE_CONTROL_INFORMATION structure. The CPU rate control values in the **JOBOBJECT_NOTIFICATION_LIMIT_INFORMATION** structure specify how much the job can exceed its established CPU rate control limits before notification is sent.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]
Header	winnt.h

See also

JOBOBJECT_CPU_RATE_CONTROL_INFORMATION

JOBOBJECT_LIMIT_VIOLATION_INFORMATION

QueryInformationJobObject

SetInformationJobObject

Feedback

Was this page helpful? Yes No

JOBOBJECT_NOTIFICATION_LIMIT_INFORMATION_2 structure (winnt.h)

Article04/02/2021

Contains extended information about notification limits for a job object. This structure is used by the SetInformationJobObject and QueryInformationJobObject functions with the **JobObjectNotificationLimitInformation2** information class.

Syntax

C++

```
typedef struct JOBOBJECT_NOTIFICATION_LIMIT_INFORMATION_2 {
  DWORD64                                   IoReadBytesLimit;
  DWORD64                                   IoWriteBytesLimit;
  LARGE_INTEGER                             PerJobUserTimeLimit;
  union {
    DWORD64 JobHighMemoryLimit;
    DWORD64 JobMemoryLimit;
  } DUMMYUNIONNAME;
  union {
    JOBOBJECT_RATE_CONTROL_TOLERANCE RateControlTolerance;
    JOBOBJECT_RATE_CONTROL_TOLERANCE CpuRateControlTolerance;
  } DUMMYUNIONNAME2;
  union {
    JOBOBJECT_RATE_CONTROL_TOLERANCE_INTERVAL RateControlToleranceInterval;
    JOBOBJECT_RATE_CONTROL_TOLERANCE_INTERVAL CpuRateControlToleranceInterval;
  } DUMMYUNIONNAME3;
  DWORD                                     LimitFlags;
  JOBOBJECT_RATE_CONTROL_TOLERANCE          IoRateControlTolerance;
  DWORD64                                   JobLowMemoryLimit;
  JOBOBJECT_RATE_CONTROL_TOLERANCE_INTERVAL IoRateControlToleranceInterval;
  JOBOBJECT_RATE_CONTROL_TOLERANCE          NetRateControlTolerance;
  JOBOBJECT_RATE_CONTROL_TOLERANCE_INTERVAL NetRateControlToleranceInterval;
} JOBOBJECT_NOTIFICATION_LIMIT_INFORMATION_2;
```

Members

`IoReadBytesLimit`

If the *LimitFlags* member specifies **JOB_OBJECT_LIMIT_JOB_READ_BYTES**, this member is the notification limit for the total I/O bytes read by all processes in the job. Otherwise, this member is ignored.

`IoWriteBytesLimit`

If the *LimitFlags* parameter specifies **JOB_OBJECT_LIMIT_JOB_WRITE_BYTES**, this member is the notification limit for the total I/O bytes written by all processes in the job. Otherwise, this member is ignored.

`PerJobUserTimeLimit`

If the *LimitFlags* parameter specifies **JOB_OBJECT_LIMIT_JOB_TIME**, this member is the notification limit for per-job user-mode execution time, in 100-nanosecond ticks. Otherwise, this member is ignored.

The system adds the accumulated execution time of processes associated with the job to this limit when the limit is set. For example, if a process associated with the job has already accumulated 5 minutes of user-mode execution time and the limit is set to 1 minute, the limit actually enforced is 6 minutes.

To specify **PerJobUserTimeLimit** as an enforceable limit and terminate processes in jobs that exceed the limit, see the JOBOBJECT_BASIC_LIMIT_INFORMATION structure.

`DUMMYUNIONNAME`

`DUMMYUNIONNAME.JobHighMemoryLimit`

If the *LimitFlags* parameter specifies **JOB_OBJECT_LIMIT_JOB_MEMORY_HIGH**, this member is the notification maximum limit for total virtual memory that can be committed by all processes in the job, in bytes. Otherwise, this member is ignored.

`DUMMYUNIONNAME.JobMemoryLimit`

If the *LimitFlags* parameter specifies **JOB_OBJECT_LIMIT_JOB_MEMORY**, this member is the notification maximum limit for total virtual memory that can be committed by all processes in the job, in bytes. Otherwise, this member is ignored.

`DUMMYUNIONNAME2`

`DUMMYUNIONNAME2.RateControlTolerance`

If the *LimitFlags* parameter specifies **JOB_OBJECT_LIMIT_RATE_CONTROL**, this member specifies the extent to which a job can exceed its CPU rate control limits during the interval specified by the **RateControlToleranceInterval** member. Otherwise, this member is ignored.

This member can be one of the following values. If no value is specified, **ToleranceHigh** is used.

Expand table

Value	Meaning

ToleranceLow 1	The job can exceed its CPU rate control limits for 20% of the tolerance interval.
ToleranceMedium 2	The job can exceed its CPU rate control limits for 40% of the tolerance interval.
ToleranceHigh 3	The job can exceed its CPU rate control limits for 60% of the tolerance interval.

`DUMMYUNIONNAME2.CpuRateControlTolerance`

If the *LimitFlags* parameter specifies **JOB_OBJECT_LIMIT_CPU_RATE_CONTROL**, this member specifies the extent to which a job can exceed its CPU rate control limits during the interval specified by the **CpuRateControlToleranceInterval** member. Otherwise, this member is ignored.

This member can be one of the following values. If no value is specified, **ToleranceHigh** is used.

Expand table

Value	Meaning
ToleranceLow 1	The job can exceed its CPU rate control limits for 20% of the tolerance interval.
ToleranceMedium 2	The job can exceed its CPU rate control limits for 40% of the tolerance interval.
ToleranceHigh 3	The job can exceed its CPU rate control limits for 60% of the tolerance interval.

`DUMMYUNIONNAME3`

`DUMMYUNIONNAME3.RateControlToleranceInterval`

If the *LimitFlags* parameter specifies **JOB_OBJECT_LIMIT_RATE_CONTROL**, this member specifies the interval during which a job's CPU usage is monitored to determine whether the job has exceeded its CPU rate control limits. Otherwise, this member is ignored.

This member can be one of the following values. If no value is specified, **ToleranceIntervalShort** is used.

Expand table

Value	Meaning
ToleranceIntervalShort	The tolerance interval is 10 seconds.

1	
ToleranceIntervalMedium 2	The tolerance interval is one minute.
ToleranceIntervalLong 3	The tolerance interval is 10 minutes.

`DUMMYUNIONNAME3.CpuRateControlToleranceInterval`

If the *LimitFlags* parameter specifies **JOB_OBJECT_CPU_LIMIT_RATE_CONTROL**, this member specifies the interval during which a job's CPU usage is monitored to determine whether the job has exceeded its CPU rate control limits. Otherwise, this member is ignored.

This member can be one of the following values. If no value is specified, **ToleranceIntervalShort** is used.

Expand table

Value	Meaning
ToleranceIntervalShort 1	The tolerance interval is 10 seconds.
ToleranceIntervalMedium 2	The tolerance interval is one minute.
ToleranceIntervalLong 3	The tolerance interval is 10 minutes.

`LimitFlags`

The limit flags that are in effect. This member is a bitfield that determines whether other structure members are used. Any combination of the following values can be specified.

Expand table

Value	Meaning
JOB_OBJECT_LIMIT_JOB_MEMORY_HIGH 0x00000200	Establishes the notification threshold for the job-wide sum of private committed memory for all processes associated with the job. The **JobHighMemoryLimit** member contains additional information.
JOB_OBJECT_LIMIT_JOB_MEMORY_LOW 0x00008000	Establishes the notification minimum for the job-wide sum of private committed memory for all processes associated with the job. If this value is set, a notification is sent when the amount of private committed memory falls below this

	threshold. The **JobLowMemoryLimit** member contains additional information.
JOB_OBJECT_LIMIT_JOB_READ_BYTES 0x00010000	Establishes the I/O read bytes limit to the job-wide sum of I/O bytes read by all processes associated with the job. The **IoReadBytesLimit** member contains more information.
JOB_OBJECT_LIMIT_JOB_WRITE_BYTES 0x00020000	Establishes the I/O write bytes limit to the job-wide sum of I/O bytes written by all processes associated with the job. The **IoWriteBytesLimit** member contains more information.
JOB_OBJECT_LIMIT_JOB_TIME 0x00000004	Establishes the limit for user-mode execution time for the job. The **PerJobUserTimeLimit** member contains additional information.
JOB_OBJECT_LIMIT_CPU_RATE_CONTROL 0x00040000	Establishes the notification threshold for the CPU rate control limits established for the job. The **CpuRateControlTolerance** and **CpuRateControlToleranceInterval** members contain additional information. CPU rate control limits are established by calling SetInformationJobObject with the **JobObjectCpuRateInformationClass** information class.
JOB_OBJECT_LIMIT_RATE_CONTROL 0x00040000	Establishes the notification threshold for the CPU rate control limits established for the job. The **RateControlTolerance** and **RateControlToleranceInterval** members contain additional information. CPU rate control limits are established by calling SetInformationJobObject with the **JobObjectCpuRateInformationClass** information class.
JOB_OBJECT_LIMIT_IO_RATE_CONTROL 0x00080000	Establishes the notification threshold for the I/O rate control limits established for the job. The **IoRateControlTolerance** and **IoRateControlToleranceInterval** members contain additional information. I/O rate control limits are established by calling SetIoRateControlInformationJobObject.
JOB_OBJECT_LIMIT_NET_RATE_CONTROL 0x00100000	Establishes the notification threshold for the network rate control limits established for the job. The **NetRateControlTolerance** and **NetRateControlToleranceInterval** members contain additional information. Network rate control limits are established by calling SetInformationJobObject with the **JobObjectNetRateInformationClass** information class.

`IoRateControlTolerance`

If the *LimitFlags* parameter specifies **JOB_OBJECT_LIMIT_IO_RATE_CONTROL**, this member specifies the extent to which a job can exceed its I/O rate control limits during the interval specified by the **IoRateControlToleranceInterval** member. Otherwise, this member is ignored.

This member can be one of the following values. If no value is specified, **ToleranceHigh** is used.

Expand table

Value	Meaning
ToleranceLow 1	The job can exceed its I/O rate control limits for 20% of the tolerance interval.
ToleranceMedium 2	The job can exceed its I/O rate control limits for 40% of the tolerance interval.
ToleranceHigh 3	The job can exceed its I/O rate control limits for 60% of the tolerance interval.

`JobLowMemoryLimit`

If the *LimitFlags* parameter specifies **JOB_OBJECT_LIMIT_JOB_MEMORY_LOW**, this member is the notification limit minimum for the total virtual memory that can be committed by all processes in the job, in bytes. Otherwise, this member is ignored.

`IoRateControlToleranceInterval`

If the *LimitFlags* parameter specifies **JOB_OBJECT_IO_LIMIT_RATE_CONTROL**, this member specifies the interval during which a job's I/O usage is monitored to determine whether the job has exceeded its I/O rate control limits. Otherwise, this member is ignored.

This member can be one of the following values. If no value is specified, **ToleranceIntervalShort** is used.

Expand table

Value	Meaning
ToleranceIntervalShort 1	The tolerance interval is 10 seconds.
ToleranceIntervalMedium 2	The tolerance interval is one minute.
ToleranceIntervalLong	The tolerance interval is 10 minutes.

3

`NetRateControlTolerance`

If the *LimitFlags* parameter specifies **JOB_OBJECT_LIMIT_IO_RATE_CONTROL**, this member specifies the extent to which a job can exceed its network rate control limits during the interval specified by the **NetRateControlToleranceInterval** member. Otherwise, this member is ignored.

This member can be one of the following values. If no value is specified, **ToleranceHigh** is used.

Expand table

Value	Meaning
ToleranceLow 1	The job can exceed its network rate control limits for 20% of the tolerance interval.
ToleranceMedium 2	The job can exceed its network rate control limits for 40% of the tolerance interval.
ToleranceHigh 3	The job can exceed its network rate control limits for 60% of the tolerance interval.

`NetRateControlToleranceInterval`

If the *LimitFlags* parameter specifies **JOB_OBJECT_NET_LIMIT_RATE_CONTROL**, this member specifies the interval during which a job's network usage is monitored to determine whether the job has exceeded its network rate control limits. Otherwise, this member is ignored.

This member can be one of the following values. If no value is specified, **ToleranceIntervalShort** is used.

Expand table

Value	Meaning
ToleranceIntervalShort 1	The tolerance interval is 10 seconds.
ToleranceIntervalMedium 2	The tolerance interval is one minute.
ToleranceIntervalLong 3	The tolerance interval is 10 minutes.

Remarks

When a notification limit is exceeded, the system sends a **JOB_OBJECT_MSG_NOTIFICATION_LIMIT** message to the I/O completion port associated with the job. Processes in the job continue to run and can continue to allocate memory or transmit read or write bytes beyond the specified limits.

When the application monitoring the I/O completion port receives a **JOB_OBJECT_MSG_NOTIFICATION_LIMIT** message, it must call QueryInformationJobObject with the **JobObjectLimitViolationInformation2** information class. Limit violation information is received in a JOBOBJECT_LIMIT_VIOLATION_INFORMATION_2 structure that contains information about all notification limits that were exceeded at the time of the query. The system will not send another **JOB_OBJECT_MSG_NOTIFICATION_LIMIT** message until after **QueryInformationJobObject** is called.

CPU rate control limits for a job are established in a JOBOBJECT_CPU_RATE_CONTROL_INFORMATION structure. The CPU rate control values in the **JOBOBJECT_NOTIFICATION_LIMIT_INFORMATION_2** structure specify how much the job can exceed its established CPU rate control limits before notification is sent.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 10 [desktop apps only]
Minimum supported server	Windows Server 2016 [desktop apps only]
Header	winnt.h

See also

JOBOBJECT_CPU_RATE_CONTROL_INFORMATION

JOBOBJECT_IO_RATE_CONTROL_INFORMATION

JOBOBJECT_LIMIT_VIOLATION_INFORMATION_2

JOBOBJECT_NET_RATE_CONTROL_INFORMATION

QueryInformationJobObject

SetInformationJobObject

SetIoRateControlInformationJobObject

JOBOBJECT_SECURITY_LIMIT_INFORMATION structure (winnt.h)

Article04/02/2021

[JOBOBJECT_SECURITY_LIMIT_INFORMATION is available for use in the operating systems specified in the Requirements section. Support for this structure was removed starting with Windows Vista. For information, see Remarks.]

Contains the security limitations for a job object.

Syntax

C++

```
typedef struct _JOBOBJECT_SECURITY_LIMIT_INFORMATION {
  DWORD             SecurityLimitFlags;
  HANDLE            JobToken;
  PTOKEN_GROUPS     SidsToDisable;
  PTOKEN_PRIVILEGES PrivilegesToDelete;
  PTOKEN_GROUPS     RestrictedSids;
} JOBOBJECT_SECURITY_LIMIT_INFORMATION,
*PJOBOBJECT_SECURITY_LIMIT_INFORMATION;
```

Members

`SecurityLimitFlags`

The security limitations for the job. This member can be one or more of the following values.

Expand table

Value	Meaning
JOB_OBJECT_SECURITY_FILTER_TOKENS 0x00000008	Applies a filter to the token when a process impersonates a client. Requires at least one of the following members to be set: **SidsToDisable**, **PrivilegesToDelete**, or **RestrictedSids**.
JOB_OBJECT_SECURITY_NO_ADMIN 0x00000001	Prevents any process in the job from using a token that specifies the local administrators group.

JOB_OBJECT_SECURITY_ONLY_TOKEN 0x00000004	Forces processes in the job to run under a specific token. Requires a token handle in the **JobToken** member.
JOB_OBJECT_SECURITY_RESTRICTED_TOKEN 0x00000002	Prevents any process in the job from using a token that was not created with the CreateRestrictedToken function.

`JobToken`

A handle to the primary token that represents a user. The handle must have TOKEN_ASSIGN_PRIMARY access.

If the token was created with CreateRestrictedToken, all processes in the job are limited to that token or a further restricted token. Otherwise, the caller must have the SE_ASSIGNPRIMARYTOKEN_NAME privilege.

`SidsToDisable`

A pointer to a TOKEN_GROUPS structure that specifies the SIDs to disable for access checking, if **SecurityLimitFlags** is JOB_OBJECT_SECURITY_FILTER_TOKENS.

This member can be NULL if you do not want to disable any SIDs.

`PrivilegesToDelete`

A pointer to a TOKEN_PRIVILEGES structure that specifies the privileges to delete from the token, if **SecurityLimitFlags** is JOB_OBJECT_SECURITY_FILTER_TOKENS.

This member can be NULL if you do not want to delete any privileges.

`RestrictedSids`

A pointer to a TOKEN_GROUPS structure that specifies the deny-only SIDs that will be added to the access token, if **SecurityLimitFlags** is JOB_OBJECT_SECURITY_FILTER_TOKENS.

This member can be NULL if you do not want to specify any deny-only SIDs.

Remarks

After security limitations are placed on processes in a job, they cannot be revoked.

Starting with Windows Vista, you must set security limitations individually for each process associated with a job object, rather than setting them for the job object by

using SetInformationJobObject. For information, see Process Security and Access Rights.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	winnt.h (include Windows.h)

See also

CreateRestrictedToken

QueryInformationJobObject

SetInformationJobObject

TOKEN_GROUPS

TOKEN_PRIVILEGES

Feedback

Was this page helpful? Yes No

MEMORY_PRIORITY_INFORMATION structure (processthreadsapi.h)

Article02/22/2024

Specifies the memory priority for a thread or process. This structure is used by the GetProcessInformation, SetProcessInformation, GetThreadInformation, and SetThreadInformation functions.

Syntax

C++

```
typedef struct _MEMORY_PRIORITY_INFORMATION {
  ULONG MemoryPriority;
} MEMORY_PRIORITY_INFORMATION, *PMEMORY_PRIORITY_INFORMATION;
```

Members

`MemoryPriority`

The memory priority for the thread or process. This member can be one of the following values.

Expand table

Value	Meaning
MEMORY_PRIORITY_VERY_LOW 1	Very low memory priority.
MEMORY_PRIORITY_LOW 2	Low memory priority.
MEMORY_PRIORITY_MEDIUM 3	Medium memory priority.
MEMORY_PRIORITY_BELOW_NORMAL 4	Below normal memory priority.
MEMORY_PRIORITY_NORMAL 5	Normal memory priority. This is the default priority for all threads and processes on the system.

Remarks

The memory priority of a thread or process serves as a hint to the memory manager when it trims pages from the working set. Other factors being equal, pages with lower memory priority are trimmed before pages with higher memory priority. For more information, see Working Set.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]
Header	processthreadsapi.h (include Windows.h)

See also

GetProcessInformation

GetThreadInformation

SetProcessInformation

SetThreadInformation

Feedback

Was this page helpful? Yes No

NUMA_NODE_RELATIONSHIP structure (winnt.h)

Article11/20/2024

Represents information about a NUMA node in a processor group. This structure is used with the GetLogicalProcessorInformationEx function.

Syntax

C++

```
typedef struct _NUMA_NODE_RELATIONSHIP {
  DWORD NodeNumber;
  BYTE  Reserved[18];
  WORD  GroupCount;
  union {
    GROUP_AFFINITY GroupMask;
    GROUP_AFFINITY GroupMasks[ANYSIZE_ARRAY];
  } DUMMYUNIONNAME;
} NUMA_NODE_RELATIONSHIP, *PNUMA_NODE_RELATIONSHIP;
```

Members

`NodeNumber`

The number of the NUMA node.

`Reserved[18]`

This member is reserved.

`GroupCount`

The number of groups included in the *GroupMasks* array. This field was introduced in Windows 20H2. On earlier versions, this value is always 0.

`DUMMYUNIONNAME`

`DUMMYUNIONNAME.GroupMask`

A GROUP_AFFINITY structure that specifies a group number and processor affinity within the group.

`DUMMYUNIONNAME.GroupMasks[ANYSIZE_ARRAY]`

An array of GROUP_AFFINITY structure that specifies a group number and processor affinity within the group.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Header	winnt.h

See also

GROUP_AFFINITY

GetLogicalProcessorInformationEx

SYSTEM_LOGICAL_PROCESSOR_INFORMATION_EX

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

ORIENTATION_PREFERENCE enumeration (winuser.h)

Article02/22/2024

Indicates the screen orientation preference for a desktop app process.

Syntax

```C++
typedef enum ORIENTATION_PREFERENCE {
  ORIENTATION_PREFERENCE_NONE = 0x0,
  ORIENTATION_PREFERENCE_LANDSCAPE = 0x1,
  ORIENTATION_PREFERENCE_PORTRAIT = 0x2,
  ORIENTATION_PREFERENCE_LANDSCAPE_FLIPPED = 0x4,
  ORIENTATION_PREFERENCE_PORTRAIT_FLIPPED = 0x8
} ;
```

Constants

Expand table

`ORIENTATION_PREFERENCE_NONE` Value: *0x0* The process has no device orientation preferences. The system may choose any available setting.
`ORIENTATION_PREFERENCE_LANDSCAPE` Value: *0x1* The process represents a desktop app that can be used in landscape mode.
`ORIENTATION_PREFERENCE_PORTRAIT` Value: *0x2* The process represents a desktop app that can be used in portrait mode.
`ORIENTATION_PREFERENCE_LANDSCAPE_FLIPPED` Value: *0x4* The process represents a desktop app that can be used in flipped landscape mode.
`ORIENTATION_PREFERENCE_PORTRAIT_FLIPPED` Value: *0x8* The process represents a desktop app that can be used in flipped portrait mode.

Requirements

Expand table

Requirement	Value
Header	winuser.h

PEB structure (winternl.h)

Article09/01/2022

[This structure may be altered in future versions of Windows.]

Contains process information.

Syntax

C++

```
typedef struct _PEB {
  BYTE                          Reserved1[2];
  BYTE                          BeingDebugged;
  BYTE                          Reserved2[1];
  PVOID                         Reserved3[2];
  PPEB_LDR_DATA                 Ldr;
  PRTL_USER_PROCESS_PARAMETERS  ProcessParameters;
  PVOID                         Reserved4[3];
  PVOID                         AtlThunkSListPtr;
  PVOID                         Reserved5;
  ULONG                         Reserved6;
  PVOID                         Reserved7;
  ULONG                         Reserved8;
  ULONG                         AtlThunkSListPtr32;
  PVOID                         Reserved9[45];
  BYTE                          Reserved10[96];
  PPS_POST_PROCESS_INIT_ROUTINE PostProcessInitRoutine;
  BYTE                          Reserved11[128];
  PVOID                         Reserved12[1];
  ULONG                         SessionId;
} PEB, *PPEB;
```

Members

`Reserved1[2]`

Reserved for internal use by the operating system.

`BeingDebugged`

Indicates whether the specified process is currently being debugged. The **PEB** structure, however, is an internal operating-system structure whose layout may change in the future. It is best to use the CheckRemoteDebuggerPresent function instead.

`Reserved2[1]`

Reserved for internal use by the operating system.

`Reserved3[2]`

Reserved for internal use by the operating system.

`Ldr`

A pointer to a PEB_LDR_DATA structure that contains information about the loaded modules for the process.

`ProcessParameters`

A pointer to an RTL_USER_PROCESS_PARAMETERS structure that contains process parameter information such as the command line.

`Reserved4[3]`

Reserved for internal use by the operating system.

`AtlThunkSListPtr`

`Reserved5`

Reserved for internal use by the operating system.

`Reserved6`

Reserved for internal use by the operating system.

`Reserved7`

Reserved for internal use by the operating system.

`Reserved8`

`AtlThunkSListPtr32`

`Reserved9[45]`

`Reserved10[96]`

`PostProcessInitRoutine`

Not supported.

`Reserved11[128]`

`Reserved12[1]`

`SessionId`

The Terminal Services session identifier associated with the current process.

Remarks

The syntax for this structure on 64-bit Windows is as follows:

syntax

```
typedef struct _PEB {
    BYTE Reserved1[2];
    BYTE BeingDebugged;
    BYTE Reserved2[21];
    PPEB_LDR_DATA LoaderData;
    PRTL_USER_PROCESS_PARAMETERS ProcessParameters;
    BYTE Reserved3[520];
    PPS_POST_PROCESS_INIT_ROUTINE PostProcessInitRoutine;
    BYTE Reserved4[136];
    ULONG SessionId;
} PEB;
```

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	winternl.h

See also

NtQueryInformationProcess

ZwQueryInformationProcess

TEB

PEB_LDR_DATA

RTL_USER_PROCESS_PARAMETERS

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

PEB_LDR_DATA structure (winternl.h)

Article02/22/2024

[This structure may be altered in future versions of Windows.]

Contains information about the loaded modules for the process.

Syntax

C++

```
typedef struct _PEB_LDR_DATA {
  BYTE       Reserved1[8];
  PVOID      Reserved2[3];
  LIST_ENTRY InMemoryOrderModuleList;
} PEB_LDR_DATA, *PPEB_LDR_DATA;
```

Members

`Reserved1[8]`

Reserved for internal use by the operating system.

`Reserved2[3]`

Reserved for internal use by the operating system.

`InMemoryOrderModuleList`

The head of a doubly-linked list that contains the loaded modules for the process. Each item in the list is a pointer to an **LDR_DATA_TABLE_ENTRY** structure. For more information, see Remarks.

Remarks

The **LIST_ENTRY** structure is defined as follows:

syntax

```
typedef struct _LIST_ENTRY {
   struct _LIST_ENTRY *Flink;
```

```
    struct _LIST_ENTRY *Blink;
} LIST_ENTRY, *PLIST_ENTRY, *RESTRICTED_POINTER PRLIST_ENTRY;
```

The **LDR_DATA_TABLE_ENTRY** structure is defined as follows:

syntax

```
typedef struct _LDR_DATA_TABLE_ENTRY {
    PVOID Reserved1[2];
    LIST_ENTRY InMemoryOrderLinks;
    PVOID Reserved2[2];
    PVOID DllBase;
    PVOID EntryPoint;
    PVOID Reserved3;
    UNICODE_STRING FullDllName;
    BYTE Reserved4[8];
    PVOID Reserved5[3];
    union {
        ULONG CheckSum;
        PVOID Reserved6;
    };
    ULONG TimeDateStamp;
} LDR_DATA_TABLE_ENTRY, *PLDR_DATA_TABLE_ENTRY;
```

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	winternl.h

See also

PEB

Feedback

Was this page helpful?

Yes No

PROCESS_INFORMATION structure (processthreadsapi.h)

Article02/22/2024

Contains information about a newly created process and its primary thread. It is used with the CreateProcess, CreateProcessAsUser, CreateProcessWithLogonW, or CreateProcessWithTokenW function.

Syntax

C++

```
typedef struct _PROCESS_INFORMATION {
  HANDLE hProcess;
  HANDLE hThread;
  DWORD  dwProcessId;
  DWORD  dwThreadId;
} PROCESS_INFORMATION, *PPROCESS_INFORMATION, *LPPROCESS_INFORMATION;
```

Members

`hProcess`

A handle to the newly created process. The handle is used to specify the process in all functions that perform operations on the process object.

`hThread`

A handle to the primary thread of the newly created process. The handle is used to specify the thread in all functions that perform operations on the thread object.

`dwProcessId`

A value that can be used to identify a process. The value is valid from the time the process is created until all handles to the process are closed and the process object is freed; at this point, the identifier may be reused.

`dwThreadId`

A value that can be used to identify a thread. The value is valid from the time the thread is created until all handles to the thread are closed and the thread object is freed; at this

point, the identifier may be reused.

Remarks

If the function succeeds, be sure to call the CloseHandle function to close the **hProcess** and **hThread** handles when you are finished with them. Otherwise, when the child process exits, the system cannot clean up the process structures for the child process because the parent process still has open handles to the child process. However, the system will close these handles when the parent process terminates, so the structures related to the child process object would be cleaned up at this point.

Examples

For an example, see Creating Processes.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	processthreadsapi.h (include Windows.h on Windows Server 2003, Windows Vista, Windows 7, Windows Server 2008 Windows Server 2008 R2)

See also

CreateProcess

CreateProcessAsUser

CreateProcessWithLogonW

CreateProcessWithTokenW

Feedback

Was this page helpful? Yes No

PROCESS_MEMORY_EXHAUSTION_INFO structure (processthreadsapi.h)

Article02/22/2024

Allows applications to configure a process to terminate if an allocation fails to commit memory. This structure is used by the PROCESS_INFORMATION_CLASS class.

Syntax

```
C++

typedef struct _PROCESS_MEMORY_EXHAUSTION_INFO {
  USHORT                          Version;
  USHORT                          Reserved;
  PROCESS_MEMORY_EXHAUSTION_TYPE Type;
  ULONG_PTR                       Value;
} PROCESS_MEMORY_EXHAUSTION_INFO, *PPROCESS_MEMORY_EXHAUSTION_INFO;
```

Members

`Version`

Version should be set to **PME_CURRENT_VERSION**.

`Reserved`

Reserved.

`Type`

Type of failure.

Type should be set to **PMETypeFailFastOnCommitFailure** (this is the only type available).

`Value`

Used to turn the feature on or off.

Expand table

Function	Setting

Enable	PME_FAILFAST_ON_COMMIT_FAIL_ENABLE
Disable	PME_FAILFAST_ON_COMMIT_FAIL_DISABLE

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 10, version 1511 [desktop apps | UWP apps]
Minimum supported server	Windows Server 2016 [desktop apps | UWP apps]
Header	processthreadsapi.h (include Windows.h)

See also

PROCESS_INFORMATION_CLASS

PROCESS_MEMORY_EXHAUSTION_TYPE

Feedback

Was this page helpful? Yes No

PROCESS_MITIGATION_ASLR_POLICY structure (winnt.h)

Article04/02/2021

Contains process mitigation policy settings for Address Space Randomization Layout (ASLR). The GetProcessMitigationPolicy and SetProcessMitigationPolicy functions use this structure.

Syntax

C++

```
typedef struct _PROCESS_MITIGATION_ASLR_POLICY {
  union {
    DWORD Flags;
    struct {
      DWORD EnableBottomUpRandomization : 1;
      DWORD EnableForceRelocateImages : 1;
      DWORD EnableHighEntropy : 1;
      DWORD DisallowStrippedImages : 1;
      DWORD ReservedFlags : 28;
    } DUMMYSTRUCTNAME;
  } DUMMYUNIONNAME;
} PROCESS_MITIGATION_ASLR_POLICY, *PPROCESS_MITIGATION_ASLR_POLICY;
```

Members

`DUMMYUNIONNAME`

`DUMMYUNIONNAME.Flags`

This member is reserved for system use.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.EnableBottomUpRandomization`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.EnableForceRelocateImages`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.EnableHighEntropy`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.DisallowStrippedImages`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.ReservedFlags`

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]
Header	winnt.h (include Windows.h)

Feedback

Was this page helpful? Yes No

PROCESS_MITIGATION_BINARY_SIGNATURE_POLICY structure (winnt.h)

Article04/02/2021

Contains process mitigation policy settings for the loading of images depending on the signatures for the image.

Syntax

C++

```
typedef struct _PROCESS_MITIGATION_BINARY_SIGNATURE_POLICY {
  union {
    DWORD Flags;
    struct {
      DWORD MicrosoftSignedOnly : 1;
      DWORD StoreSignedOnly : 1;
      DWORD MitigationOptIn : 1;
      DWORD AuditMicrosoftSignedOnly : 1;
      DWORD AuditStoreSignedOnly : 1;
      DWORD ReservedFlags : 27;
    } DUMMYSTRUCTNAME;
  } DUMMYUNIONNAME;
} PROCESS_MITIGATION_BINARY_SIGNATURE_POLICY,
*PPROCESS_MITIGATION_BINARY_SIGNATURE_POLICY;
```

Members

`DUMMYUNIONNAME`

`DUMMYUNIONNAME.Flags`

Reserved for system use.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.MicrosoftSignedOnly`

Set (0x1) to prevent the process from loading images that are not signed by Microsoft; otherwise leave unset (0x0).

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.StoreSignedOnly`

Set (0x1) to prevent the process from loading images that are not signed by the Windows Store; otherwise leave unset (0x0).

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.MitigationOptIn`

Set (0x1) to prevent the process from loading images that are not signed by Microsoft, the Windows Store and the Windows Hardware Quality Labs (WHQL); otherwise leave unset (0x0).

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.AuditMicrosoftSignedOnly`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.AuditStoreSignedOnly`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.ReservedFlags`

Reserved for system use.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 10 [desktop apps only]
Minimum supported server	Windows Server 2016 [desktop apps only]
Header	winnt.h

See also

GetProcessMitigationPolicy

SetProcessMitigationPolicy

Feedback

Was this page helpful? Yes No

PROCESS_MITIGATION_CONTROL_FLOW_GUARD_POLICY structure (winnt.h)

Article02/22/2024

Contains process mitigation policy settings for Control Flow Guard (CFG). The GetProcessMitigationPolicy and SetProcessMitigationPolicy functions use this structure.

Syntax

C++

```
typedef struct _PROCESS_MITIGATION_CONTROL_FLOW_GUARD_POLICY {
  union {
    DWORD Flags;
    struct {
      DWORD EnableControlFlowGuard : 1;
      DWORD EnableExportSuppression : 1;
      DWORD StrictMode : 1;
      DWORD EnableXfg : 1;
      DWORD EnableXfgAuditMode : 1;
      DWORD ReservedFlags : 27;
    } DUMMYSTRUCTNAME;
  } DUMMYUNIONNAME;
} PROCESS_MITIGATION_CONTROL_FLOW_GUARD_POLICY,
*PPROCESS_MITIGATION_CONTROL_FLOW_GUARD_POLICY;
```

Members

`DUMMYUNIONNAME`

`DUMMYUNIONNAME.Flags`

This member is reserved for system use.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.EnableControlFlowGuard`

CFG is enabled for the process if this flag is set. This field cannot be changed via SetProcessMitigationPolicy.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.EnableExportSuppression`

If TRUE, exported functions will be treated as invalid indirect call targets by default. Exported functions only become valid indirect call targets if they are dynamically resolved via GetProcAddress. This field cannot be changed via SetProcessMitigationPolicy.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.StrictMode`

If TRUE, all DLLs that are loaded must enable CFG. If a DLL does not enable CFG then the image will fail to load. This policy can be enabled after a process has started by calling SetProcessMitigationPolicy. It cannot be disabled once enabled.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.EnableXfg`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.EnableXfgAuditMode`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.ReservedFlags`

This member is reserved for system use.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8.1 [desktop apps only]
Minimum supported server	Windows Server 2016 [desktop apps only]
Header	winnt.h

Feedback

Was this page helpful? Yes No

PROCESS_MITIGATION_DEP_POLICY structure (winnt.h)

Article02/22/2024

Contains process mitigation policy settings for data execution prevention (DEP). The GetProcessMitigationPolicy and SetProcessMitigationPolicy functions use this structure.

Syntax

C++

```
typedef struct _PROCESS_MITIGATION_DEP_POLICY {
  union {
    DWORD Flags;
    struct {
      DWORD Enable : 1;
      DWORD DisableAtlThunkEmulation : 1;
      DWORD ReservedFlags : 30;
    } DUMMYSTRUCTNAME;
  } DUMMYUNIONNAME;
  BOOLEAN Permanent;
} PROCESS_MITIGATION_DEP_POLICY, *PPROCESS_MITIGATION_DEP_POLICY;
```

Members

`DUMMYUNIONNAME`

`DUMMYUNIONNAME.Flags`

This member is reserved for system use.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.Enable`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.DisableAtlThunkEmulation`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.ReservedFlags`

`Permanent`

DEP is permanently enabled and cannot be disabled if this field is set to TRUE.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]
Header	winnt.h

Feedback

Was this page helpful? Yes No

PROCESS_MITIGATION_DYNAMIC_CODE_POLICY structure (winnt.h)

Article02/22/2024

Contains process mitigation policy settings for restricting dynamic code generation and modification.

Syntax

```C++
typedef struct _PROCESS_MITIGATION_DYNAMIC_CODE_POLICY {
  union {
    DWORD Flags;
    struct {
      DWORD ProhibitDynamicCode : 1;
      DWORD AllowThreadOptOut : 1;
      DWORD AllowRemoteDowngrade : 1;
      DWORD AuditProhibitDynamicCode : 1;
      DWORD ReservedFlags : 28;
    } DUMMYSTRUCTNAME;
  } DUMMYUNIONNAME;
} PROCESS_MITIGATION_DYNAMIC_CODE_POLICY,
*PPROCESS_MITIGATION_DYNAMIC_CODE_POLICY;
```

Members

`DUMMYUNIONNAME`

`DUMMYUNIONNAME.Flags`

Reserved for system use.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.ProhibitDynamicCode`

Set (0x1) to prevent the process from generating dynamic code or modifying existing executable code; otherwise leave unset (0x0).

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.AllowThreadOptOut`

Set (0x1) to allow threads to opt out of the restrictions on dynamic code generation by calling the SetThreadInformation function with the *ThreadInformation* parameter set to **ThreadDynamicCodePolicy**; otherwise leave unset (0x0). You should not use the **AllowThreadOptOut** and **ThreadDynamicCodePolicy** settings together to provide strong security. These settings are only intended to enable applications to adapt their code more easily for full dynamic code restrictions.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.AllowRemoteDowngrade`

Set (0x1) to allow non-AppContainer processes to modify all of the dynamic code settings for the calling process, including relaxing dynamic code restrictions after they have been set.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.AuditProhibitDynamicCode`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.ReservedFlags`

Reserved for system use.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8.1 [desktop apps only]
Minimum supported server	Windows Server 2016 [desktop apps only]
Header	winnt.h

See also

GetProcessMitigationPolicy

SetProcessMitigationPolicy

Feedback

Was this page helpful? Yes No

PROCESS_MITIGATION_EXTENSION_POINT_DISABLE_POLICY structure (winnt.h)

Article02/22/2024

Contains process mitigation policy settings for legacy extension point DLLs. The GetProcessMitigationPolicy and SetProcessMitigationPolicy functions use this structure.

Syntax

C++

```
typedef struct _PROCESS_MITIGATION_EXTENSION_POINT_DISABLE_POLICY {
  union {
    DWORD Flags;
    struct {
      DWORD DisableExtensionPoints : 1;
      DWORD ReservedFlags : 31;
    } DUMMYSTRUCTNAME;
  } DUMMYUNIONNAME;
} PROCESS_MITIGATION_EXTENSION_POINT_DISABLE_POLICY,
*PPROCESS_MITIGATION_EXTENSION_POINT_DISABLE_POLICY;
```

Members

`DUMMYUNIONNAME`

`DUMMYUNIONNAME.Flags`

This member is reserved for system use.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.DisableExtensionPoints`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.ReservedFlags`

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]
Header	winnt.h

Feedback

Was this page helpful? Yes No

PROCESS_MITIGATION_FONT_DISABLE_POLICY structure (winnt.h)

Article02/22/2024

Contains process mitigation policy settings for the loading of non-system fonts.

Syntax

C++

```
typedef struct _PROCESS_MITIGATION_FONT_DISABLE_POLICY {
  union {
    DWORD Flags;
    struct {
      DWORD DisableNonSystemFonts : 1;
      DWORD AuditNonSystemFontLoading : 1;
      DWORD ReservedFlags : 30;
    } DUMMYSTRUCTNAME;
  } DUMMYUNIONNAME;
} PROCESS_MITIGATION_FONT_DISABLE_POLICY,
*PPROCESS_MITIGATION_FONT_DISABLE_POLICY;
```

Members

`DUMMYUNIONNAME`

`DUMMYUNIONNAME.Flags`

Reserved for system use.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.DisableNonSystemFonts`

Set (0x1) to prevent the process from loading non-system fonts; otherwise leave unset (0x0).

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.AuditNonSystemFontLoading`

Set (0x1) to indicate that an Event Tracing for Windows (ETW) event should be logged when the process attempts to load a non-system font; leave unset (0x0) to indicate that an ETW event should not be logged.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.ReservedFlags`

Reserved for system use.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 10 [desktop apps only]
Minimum supported server	Windows Server 2016 [desktop apps only]
Header	winnt.h

See also

GetProcessMitigationPolicy

SetProcessMitigationPolicy

Feedback

Was this page helpful? Yes No

PROCESS_MITIGATION_IMAGE_LOAD_POLICY structure (winnt.h)

Article02/22/2024

Contains process mitigation policy settings for the loading of images from a remote device.

Syntax

C++

```
typedef struct _PROCESS_MITIGATION_IMAGE_LOAD_POLICY {
  union {
    DWORD Flags;
    struct {
      DWORD NoRemoteImages : 1;
      DWORD NoLowMandatoryLabelImages : 1;
      DWORD PreferSystem32Images : 1;
      DWORD AuditNoRemoteImages : 1;
      DWORD AuditNoLowMandatoryLabelImages : 1;
      DWORD ReservedFlags : 27;
    } DUMMYSTRUCTNAME;
  } DUMMYUNIONNAME;
} PROCESS_MITIGATION_IMAGE_LOAD_POLICY,
*PPROCESS_MITIGATION_IMAGE_LOAD_POLICY;
```

Members

`DUMMYUNIONNAME`

`DUMMYUNIONNAME.Flags`

Reserved for system use.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.NoRemoteImages`

Set (0x1) to prevent the process from loading images from a remote device, such as a UNC share; otherwise leave unset (0x0).

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.NoLowMandatoryLabelImages`

Set (0x1) to prevent the process from loading images that have a Low mandatory label, as written by low IL; otherwise leave unset (0x0).

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.PreferSystem32Images`

Set (0x1) to search for images to load in the System32 subfolder of the folder in which Windows is installed first, then in the application directory in the standard DLL search order; otherwise leave unset (0x0).

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.AuditNoRemoteImages`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.AuditNoLowMandatoryLabelImages`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.ReservedFlags`

Reserved for system use.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 10 [desktop apps only]
Minimum supported server	Windows Server 2016 [desktop apps only]
Header	winnt.h

See also

GetProcessMitigationPolicy

SetProcessMitigationPolicy

Feedback

Was this page helpful? Yes No

PROCESS_MITIGATION_STRICT_HANDLE_CHECK_POLICY structure (winnt.h)

Article04/02/2021

Used to impose new behavior on handle references that are not valid.

Syntax

C++

```
typedef struct _PROCESS_MITIGATION_STRICT_HANDLE_CHECK_POLICY {
  union {
    DWORD Flags;
    struct {
      DWORD RaiseExceptionOnInvalidHandleReference : 1;
      DWORD HandleExceptionsPermanentlyEnabled : 1;
      DWORD ReservedFlags : 30;
    } DUMMYSTRUCTNAME;
  } DUMMYUNIONNAME;
} PROCESS_MITIGATION_STRICT_HANDLE_CHECK_POLICY,
*PPROCESS_MITIGATION_STRICT_HANDLE_CHECK_POLICY;
```

Members

`DUMMYUNIONNAME`

`DUMMYUNIONNAME.Flags`

This member is reserved for system use.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.RaiseExceptionOnInvalidHandleReference`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.HandleExceptionsPermanentlyEnabled`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.ReservedFlags`

Remarks

As a general rule, strict handle checking cannot be turned off once it is turned on. Therefore, when calling the SetProcessMitigationPolicy function with this policy, the

values of the **RaiseExceptionOnInvalidHandleReference** and **HandleExceptionsPermanentlyEnabled** substructure members must be the same. It is not possible to enable invalid handle exceptions only temporarily.

The exception to the general rule about strict handle checking always being a permanent state is that debugging tools such as Application Verifier can cause the operating system to enable invalid handle exceptions temporarily. Under those cases, it is possible for the GetProcessMitigationPolicy function to return with **RaiseExceptionOnInvalidHandleReference** set to 1, but **HandleExceptionsPermanentlyEnabled** set to 0.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]
Header	winnt.h (include Windows.h)

Feedback

Was this page helpful? Yes No

PROCESS_MITIGATION_SYSTEM_CALL_DISABLE_POLICY structure (winnt.h)

Article02/22/2024

Used to impose restrictions on what system calls can be invoked by a process.

Syntax

```C++
typedef struct _PROCESS_MITIGATION_SYSTEM_CALL_DISABLE_POLICY {
  union {
    DWORD Flags;
    struct {
      DWORD DisallowWin32kSystemCalls : 1;
      DWORD AuditDisallowWin32kSystemCalls : 1;
      DWORD DisallowFsctlSystemCalls : 1;
      DWORD AuditDisallowFsctlSystemCalls : 1;
      DWORD ReservedFlags : 28;
    } DUMMYSTRUCTNAME;
  } DUMMYUNIONNAME;
} PROCESS_MITIGATION_SYSTEM_CALL_DISABLE_POLICY,
*PPROCESS_MITIGATION_SYSTEM_CALL_DISABLE_POLICY;
```

Members

`DUMMYUNIONNAME`

`DUMMYUNIONNAME.Flags`

This member is reserved for system use.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.DisallowWin32kSystemCalls`

When set to 1, the process is not permitted to perform GUI system calls.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.AuditDisallowWin32kSystemCalls`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.DisallowFsctlSystemCalls`

When set to 1, the process is not be able to successfully make NtFsControlFile system calls, with the following FsControlCode exceptions.

- FSCTL_IS_VOLUME_MOUNTED
- FSCTL_PIPE_IMPERSONATE
- FSCTL_PIPE_LISTEN
- FSCTL_PIPE_DISCONNECT
- FSCTL_PIPE_TRANSCEIVE
- FSCTL_PIPE_WAIT
- FSCTL_PIPE_GET_PIPE_ATTRIBUTE
- FSCTL_PIPE_GET_CONNECTION_ATTRIBUTE
- FSCTL_PIPE_GET_HANDLE_ATTRIBUTE
- FSCTL_PIPE_PEEK
- FSCTL_PIPE_EVENT_SELECT
- FSCTL_PIPE_EVENT_ENUM

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.AuditDisallowFsctlSystemCalls`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.ReservedFlags`

This member is reserved for system use.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8 [desktop apps only]
Minimum supported server	Windows Server 2012 [desktop apps only]
Header	winnt.h (include Windows.h)

Feedback

Was this page helpful? Yes No

PROCESSOR_GROUP_INFO structure (winnt.h)

Article02/22/2024

Represents the number and affinity of processors in a processor group.

Syntax

C++

```
typedef struct _PROCESSOR_GROUP_INFO {
  BYTE      MaximumProcessorCount;
  BYTE      ActiveProcessorCount;
  BYTE      Reserved[38];
  KAFFINITY ActiveProcessorMask;
} PROCESSOR_GROUP_INFO, *PPROCESSOR_GROUP_INFO;
```

Members

`MaximumProcessorCount`

The maximum number of processors in the group.

`ActiveProcessorCount`

The number of active processors in the group.

`Reserved[38]`

This member is reserved.

`ActiveProcessorMask`

A bitmap that specifies the affinity for zero or more active processors within the group.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Header	winnt.h

See also

GROUP_RELATIONSHIP

Feedback

Was this page helpful? Yes No

PROCESSOR_NUMBER structure (winnt.h)

Article02/22/2024

Represents a logical processor in a processor group.

Syntax

C++

```
typedef struct _PROCESSOR_NUMBER {
  WORD Group;
  BYTE Number;
  BYTE Reserved;
} PROCESSOR_NUMBER, *PPROCESSOR_NUMBER;
```

Members

`Group`

The processor group to which the logical processor is assigned.

`Number`

The number of the logical processor relative to the group.

`Reserved`

This parameter is reserved.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Header	winnt.h

See also

Processor Groups

Feedback

Was this page helpful? Yes No

PROCESSOR_RELATIONSHIP structure (winnt.h)

Article09/01/2022

Represents information about affinity within a processor group. This structure is used with the GetLogicalProcessorInformationEx function.

Syntax

C++

```
typedef struct _PROCESSOR_RELATIONSHIP {
  BYTE           Flags;
  BYTE           EfficiencyClass;
  BYTE           Reserved[20];
  WORD           GroupCount;
  GROUP_AFFINITY GroupMask[ANYSIZE_ARRAY];
} PROCESSOR_RELATIONSHIP, *PPROCESSOR_RELATIONSHIP;
```

Members

`Flags`

If the **Relationship** member of the SYSTEM_LOGICAL_PROCESSOR_INFORMATION_EX structure is **RelationProcessorCore**, this member is LTP_PC_SMT if the core has more than one logical processor, or 0 if the core has one logical processor.

If the **Relationship** member of the SYSTEM_LOGICAL_PROCESSOR_INFORMATION_EX structure is **RelationProcessorPackage**, this member is always 0.

`EfficiencyClass`

If the **Relationship** member of the SYSTEM_LOGICAL_PROCESSOR_INFORMATION_EX structure is **RelationProcessorCore**, **EfficiencyClass** specifies the intrinsic tradeoff between performance and power for the applicable core. A core with a higher value for the efficiency class has intrinsically greater performance and less efficiency than a core with a lower value for the efficiency class. **EfficiencyClass** is only nonzero on systems with a heterogeneous set of cores.

If the **Relationship** member of the SYSTEM_LOGICAL_PROCESSOR_INFORMATION_EX structure is **RelationProcessorPackage**, **EfficiencyClass** is always 0.

The minimum operating system version that supports this member is Windows 10.

`Reserved[20]`

This member is reserved.

`GroupCount`

This member specifies the number of entries in the **GroupMask** array. For more information, see Remarks.

`GroupMask[ANYSIZE_ARRAY]`

An array of GROUP_AFFINITY structures. The **GroupCount** member specifies the number of structures in the array. Each structure in the array specifies a group number and processor affinity within the group.

Remarks

The **PROCESSOR_RELATIONSHIP** structure describes the logical processors associated with either a processor core or a processor package.

If the **PROCESSOR_RELATIONSHIP** structure represents a processor core, the **GroupCount** member is always 1.

If the **PROCESSOR_RELATIONSHIP** structure represents a processor package, the **GroupCount** member is 1 only if all processors are in the same processor group. If the package contains more than one NUMA node, the system might assign different NUMA nodes to different processor groups. In this case, the **GroupCount** member is the number of groups to which NUMA nodes in the package are assigned.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Header	winnt.h

See also

GROUP_AFFINITY

GetLogicalProcessorInformationEx

SYSTEM_LOGICAL_PROCESSOR_INFORMATION_EX

Feedback

Was this page helpful? Yes No

RTL_USER_PROCESS_PARAMETERS structure (winternl.h)

Article02/22/2024

[This structure may be altered in future versions of Windows.]

Contains process parameter information.

Syntax

C++

```
typedef struct _RTL_USER_PROCESS_PARAMETERS {
  BYTE           Reserved1[16];
  PVOID          Reserved2[10];
  UNICODE_STRING ImagePathName;
  UNICODE_STRING CommandLine;
} RTL_USER_PROCESS_PARAMETERS, *PRTL_USER_PROCESS_PARAMETERS;
```

Members

`Reserved1[16]`

Reserved for internal use by the operating system.

`Reserved2[10]`

Reserved for internal use by the operating system.

`ImagePathName`

The path of the image file for the process.

`CommandLine`

The command-line string passed to the process.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	winternl.h

See also

PEB

Feedback

Was this page helpful? Yes No

STARTUPINFOA structure (processthreadsapi.h)

Article11/01/2022

Specifies the window station, desktop, standard handles, and appearance of the main window for a process at creation time.

Syntax

C++

```
typedef struct _STARTUPINFOA {
  DWORD  cb;
  LPSTR  lpReserved;
  LPSTR  lpDesktop;
  LPSTR  lpTitle;
  DWORD  dwX;
  DWORD  dwY;
  DWORD  dwXSize;
  DWORD  dwYSize;
  DWORD  dwXCountChars;
  DWORD  dwYCountChars;
  DWORD  dwFillAttribute;
  DWORD  dwFlags;
  WORD   wShowWindow;
  WORD   cbReserved2;
  LPBYTE lpReserved2;
  HANDLE hStdInput;
  HANDLE hStdOutput;
  HANDLE hStdError;
} STARTUPINFOA, *LPSTARTUPINFOA;
```

Members

`cb`

The size of the structure, in bytes.

`lpReserved`

Reserved; must be NULL.

`lpDesktop`

The name of the desktop, or the name of both the desktop and window station for this process. A backslash in the string indicates that the string includes both the desktop and window station names.

For more information, see Thread Connection to a Desktop.

`lpTitle`

For console processes, this is the title displayed in the title bar if a new console window is created. If NULL, the name of the executable file is used as the window title instead. This parameter must be NULL for GUI or console processes that do not create a new console window.

`dwX`

If **dwFlags** specifies STARTF_USEPOSITION, this member is the x offset of the upper left corner of a window if a new window is created, in pixels. Otherwise, this member is ignored.

The offset is from the upper left corner of the screen. For GUI processes, the specified position is used the first time the new process calls CreateWindow to create an overlapped window if the *x* parameter of **CreateWindow** is CW_USEDEFAULT.

`dwY`

If **dwFlags** specifies STARTF_USEPOSITION, this member is the y offset of the upper left corner of a window if a new window is created, in pixels. Otherwise, this member is ignored.

The offset is from the upper left corner of the screen. For GUI processes, the specified position is used the first time the new process calls CreateWindow to create an overlapped window if the *y* parameter of **CreateWindow** is CW_USEDEFAULT.

`dwXSize`

If **dwFlags** specifies STARTF_USESIZE, this member is the width of the window if a new window is created, in pixels. Otherwise, this member is ignored.

For GUI processes, this is used only the first time the new process calls CreateWindow to create an overlapped window if the *nWidth* parameter of **CreateWindow** is CW_USEDEFAULT.

`dwYSize`

If **dwFlags** specifies STARTF_USESIZE, this member is the height of the window if a new window is created, in pixels. Otherwise, this member is ignored.

For GUI processes, this is used only the first time the new process calls CreateWindow to create an overlapped window if the *nHeight* parameter of **CreateWindow** is CW_USEDEFAULT.

`dwXCountChars`

If **dwFlags** specifies STARTF_USECOUNTCHARS, if a new console window is created in a console process, this member specifies the screen buffer width, in character columns. Otherwise, this member is ignored.

`dwYCountChars`

If **dwFlags** specifies STARTF_USECOUNTCHARS, if a new console window is created in a console process, this member specifies the screen buffer height, in character rows. Otherwise, this member is ignored.

`dwFillAttribute`

If **dwFlags** specifies STARTF_USEFILLATTRIBUTE, this member is the initial text and background colors if a new console window is created in a console application. Otherwise, this member is ignored.

This value can be any combination of the following values: FOREGROUND_BLUE, FOREGROUND_GREEN, FOREGROUND_RED, FOREGROUND_INTENSITY, BACKGROUND_BLUE, BACKGROUND_GREEN, BACKGROUND_RED, and BACKGROUND_INTENSITY. For example, the following combination of values produces red text on a white background:

`FOREGROUND_RED| BACKGROUND_RED| BACKGROUND_GREEN| BACKGROUND_BLUE`

`dwFlags`

A bitfield that determines whether certain **STARTUPINFO** members are used when the process creates a window. This member can be one or more of the following values.

Expand table

Value	Meaning
STARTF_FORCEONFEEDBACK 0x00000040	Indicates that the cursor is in feedback mode for two seconds after CreateProcess is called. The Working in Background cursor is displayed (see the Pointers tab in the Mouse control panel utility).

	If during those two seconds the process makes the first GUI call, the system gives five more seconds to the process. If during those five seconds the process shows a window, the system gives five more seconds to the process to finish drawing the window. The system turns the feedback cursor off after the first call to GetMessage, regardless of whether the process is drawing.
STARTF_FORCEOFFFEEDBACK 0x00000080	Indicates that the feedback cursor is forced off while the process is starting. The Normal Select cursor is displayed.
STARTF_PREVENTPINNING 0x00002000	Indicates that any windows created by the process cannot be pinned on the taskbar. This flag must be combined with STARTF_TITLEISAPPID.
STARTF_RUNFULLSCREEN 0x00000020	Indicates that the process should be run in full-screen mode, rather than in windowed mode. This flag is only valid for console applications running on an x86 computer.
STARTF_TITLEISAPPID 0x00001000	The **lpTitle** member contains an AppUserModelID. This identifier controls how the taskbar and **Start** menu present the application, and enables it to be associated with the correct shortcuts and Jump Lists. Generally, applications will use the SetCurrentProcessExplicitAppUserModelID and **GetCurrentProcessExplicitAppUserModelID** functions instead of setting this flag. For more information, see Application User Model IDs. If STARTF_PREVENTPINNING is used, application windows cannot be pinned on the taskbar. The use of any AppUserModelID-related window properties by the application overrides this setting for that window only. This flag cannot be used with STARTF_TITLEISLINKNAME.
STARTF_TITLEISLINKNAME 0x00000800	The **lpTitle** member contains the path of the shortcut file (.lnk) that the user invoked to start this process. This is typically set by the shell when a .lnk file pointing to the launched application is invoked. Most applications will not need to set this value. This flag cannot be used with STARTF_TITLEISAPPID.
STARTF_UNTRUSTEDSOURCE 0x00008000	The command line came from an untrusted source. For more information, see Remarks.

STARTF_USECOUNTCHARS 0x00000008	The **dwXCountChars** and **dwYCountChars** members contain additional information.
STARTF_USEFILLATTRIBUTE 0x00000010	The **dwFillAttribute** member contains additional information.
STARTF_USEHOTKEY 0x00000200	The **hStdInput** member contains additional information. This flag cannot be used with **STARTF_USESTDHANDLES**.
STARTF_USEPOSITION 0x00000004	The **dwX** and **dwY** members contain additional information.
STARTF_USESHOWWINDOW 0x00000001	The **wShowWindow** member contains additional information.
STARTF_USESIZE 0x00000002	The **dwXSize** and **dwYSize** members contain additional information.
STARTF_USESTDHANDLES 0x00000100	The **hStdInput**, **hStdOutput**, and **hStdError** members contain additional information. If this flag is specified when calling one of the process creation functions, the handles must be inheritable and the function's *bInheritHandles* parameter must be set to TRUE. For more information, see Handle Inheritance. If this flag is specified when calling the GetStartupInfo function, these members are either the handle value specified during process creation or INVALID_HANDLE_VALUE. Handles must be closed with CloseHandle when they are no longer needed. This flag cannot be used with **STARTF_USEHOTKEY**.

`wShowWindow`

If **dwFlags** specifies STARTF_USESHOWWINDOW, this member can be any of the values that can be specified in the *nCmdShow* parameter for the ShowWindow function, except for SW_SHOWDEFAULT. Otherwise, this member is ignored.

For GUI processes, the first time ShowWindow is called, its *nCmdShow* parameter is ignored **wShowWindow** specifies the default value. In subsequent calls to ShowWindow, the **wShowWindow** member is used if the *nCmdShow* parameter of **ShowWindow** is set to SW_SHOWDEFAULT.

`cbReserved2`

Reserved for use by the C Run-time; must be zero.

`lpReserved2`

Reserved for use by the C Run-time; must be NULL.

`hStdInput`

If **dwFlags** specifies STARTF_USESTDHANDLES, this member is the standard input handle for the process. If STARTF_USESTDHANDLES is not specified, the default for standard input is the keyboard buffer.

If **dwFlags** specifies STARTF_USEHOTKEY, this member specifies a hotkey value that is sent as the *wParam* parameter of a WM_SETHOTKEY message to the first eligible top-level window created by the application that owns the process. If the window is created with the WS_POPUP window style, it is not eligible unless the WS_EX_APPWINDOW extended window style is also set. For more information, see CreateWindowEx.

Otherwise, this member is ignored.

`hStdOutput`

If **dwFlags** specifies STARTF_USESTDHANDLES, this member is the standard output handle for the process. Otherwise, this member is ignored and the default for standard output is the console window's buffer.

If a process is launched from the taskbar or jump list, the system sets **hStdOutput** to a handle to the monitor that contains the taskbar or jump list used to launch the process. For more information, see Remarks.**Windows 7, Windows Server 2008 R2, Windows Vista, Windows Server 2008, Windows XP and Windows Server 2003:** This behavior was introduced in Windows 8 and Windows Server 2012.

`hStdError`

If **dwFlags** specifies STARTF_USESTDHANDLES, this member is the standard error handle for the process. Otherwise, this member is ignored and the default for standard error is the console window's buffer.

Remarks

For graphical user interface (GUI) processes, this information affects the first window created by the CreateWindow function and shown by the ShowWindow function. For console processes, this information affects the console window if a new console is

created for the process. A process can use the GetStartupInfo function to retrieve the **STARTUPINFO** structure specified when the process was created.

If a GUI process is being started and neither STARTF_FORCEONFEEDBACK or STARTF_FORCEOFFFEEDBACK is specified, the process feedback cursor is used. A GUI process is one whose subsystem is specified as "windows."

If a process is launched from the taskbar or jump list, the system sets GetStartupInfo to retrieve the **STARTUPINFO** structure and check that **hStdOutput** is set. If so, use GetMonitorInfo to check whether **hStdOutput** is a valid monitor handle (HMONITOR). The process can then use the handle to position its windows.

If the **STARTF_UNTRUSTEDSOURCE** flag is specified, the application should be aware that the command line is untrusted. If this flag is set, applications should disable potentially dangerous features such as macros, downloaded content, and automatic printing. This flag is optional, but applications that call CreateProcess are encouraged to set this flag when launching a program with untrusted command line arguments (such as those provided by web content) so that the newly created process can apply appropriate policy.

The **STARTF_UNTRUSTEDSOURCE** flag is supported starting in Windows Vista, but it is not defined in the SDK header files prior to the Windows 10 SDK. To use the flag in versions prior to Windows 10, you can define it manually in your program.

Examples

The following code example shows the use of **StartUpInfoA**.

C++

```
#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 );
}
```

For more information about this example, see Creating Processes.

> **Note**
>
> The processthreadsapi.h header defines STARTUPINFO as an alias that automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Expand table

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

Requirement	Value
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	processthreadsapi.h (include Windows.h on Windows Server 2003, Windows Vista, Windows 7, Windows Server 2008 Windows Server 2008 R2)

See also

CreateProcess

CreateProcessAsUser

CreateProcessWithLogonW

CreateProcessWithTokenW

GetStartupInfo

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

STARTUPINFOEXA structure (winbase.h)

Article02/22/2024

Specifies the window station, desktop, standard handles, and attributes for a new process. It is used with the CreateProcess and CreateProcessAsUser functions.

Syntax

C++

```
typedef struct _STARTUPINFOEXA {
  STARTUPINFOA                 StartupInfo;
  LPPROC_THREAD_ATTRIBUTE_LIST lpAttributeList;
} STARTUPINFOEXA, *LPSTARTUPINFOEXA;
```

Members

`StartupInfo`

A STARTUPINFO structure.

`lpAttributeList`

An attribute list. This list is created by the InitializeProcThreadAttributeList function.

Remarks

Be sure to set the **cb** member of the STARTUPINFO structure to `sizeof(STARTUPINFOEX)`.

> **Note**
>
> The winbase.h header defines STARTUPINFOEX as an alias that automatically selects the ANSI or Unicode version of this function based on the definition of the UNICODE preprocessor constant. Mixing usage of the encoding-neutral alias with code that is not encoding-neutral can lead to mismatches that result in compilation or runtime errors. For more information, see Conventions for Function Prototypes.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista [desktop apps only]
Minimum supported server	Windows Server 2008 [desktop apps only]
Header	winbase.h (include Windows.h)

See also

CreateProcess

CreateProcessAsUser

InitializeProcThreadAttributeList

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

SYSTEM_CPU_SET_INFORMATION structure (winnt.h)

Article04/02/2021

This structure is returned by GetSystemCpuSetInformation. It is used to enumerate the CPU Sets on the system and determine their current state.

This is a variable-sized structure designed for future expansion. When iterating over this structure, use the size field to determine the offset to the next structure.

Syntax

C++

```
typedef struct _SYSTEM_CPU_SET_INFORMATION {
  DWORD                    Size;
  CPU_SET_INFORMATION_TYPE Type;
  union {
    struct {
      DWORD   Id;
      WORD    Group;
      BYTE    LogicalProcessorIndex;
      BYTE    CoreIndex;
      BYTE    LastLevelCacheIndex;
      BYTE    NumaNodeIndex;
      BYTE    EfficiencyClass;
      union {
        BYTE AllFlags;
        struct {
          BYTE Parked : 1;
          BYTE Allocated : 1;
          BYTE AllocatedToTargetProcess : 1;
          BYTE RealTime : 1;
          BYTE ReservedFlags : 4;
        } DUMMYSTRUCTNAME;
      } DUMMYUNIONNAME2;
      union {
        DWORD Reserved;
        BYTE  SchedulingClass;
      };
      DWORD64 AllocationTag;
    } CpuSet;
  } DUMMYUNIONNAME;
} SYSTEM_CPU_SET_INFORMATION, *PSYSTEM_CPU_SET_INFORMATION;
```

Members

`Size`

This is the size, in bytes, of this information structure.

`Type`

This is the type of information in the structure. Applications should skip any structures with unrecognized types.

`DUMMYUNIONNAME`

`DUMMYUNIONNAME.CpuSet`

`DUMMYUNIONNAME.CpuSet.Id`

The ID of the specified CPU Set. This identifier can be used with SetProcessDefaultCpuSets or SetThreadSelectedCpuSets when specifying a list of CPU Sets to affinitize to.

`DUMMYUNIONNAME.CpuSet.Group`

Specifies the Processor Group of the CPU Set. All other values in the CpuSet structure are relative to the processor group.

`DUMMYUNIONNAME.CpuSet.LogicalProcessorIndex`

Specifies the group-relative index of the home processor of the CPU Set. Unless the CPU Set is parked for thermal or power management reasons or assigned for exclusive use to another application, threads will run on the home processor of one of their CPU Sets. The **Group** and **LogicalProcessorIndex** fields are the same as the ones found in the PROCESSOR_NUMBER structure and they correspond to the **Group** field and **Mask** field of the GROUP_AFFINITY structure.

`DUMMYUNIONNAME.CpuSet.CoreIndex`

A group-relative value indicating which "Core" has the home processor of the CPU Set. This number is the same for all CPU Sets in the same group that share significant execution resources with each other, such as different hardware threads on a single core that supports simultaneous multi-threading.

`DUMMYUNIONNAME.CpuSet.LastLevelCacheIndex`

A group-relative value indicating which CPU Sets share at least one level of cache with each other. This value is the same for all CPU Sets in a group that are on processors that share cache with each other.

`DUMMYUNIONNAME.CpuSet.NumaNodeIndex`

A group-relative value indicating which NUMA node a CPU Set is on. All CPU Sets in a given group that are on the same NUMA node will have the same value for this field.

`DUMMYUNIONNAME.CpuSet.EfficiencyClass`

A value indicating the intrinsic energy efficiency of a processor for systems that support heterogeneous processors (such as ARM big.LITTLE systems). CPU Sets with higher numerical values of this field have home processors that are faster but less power-efficient than ones with lower values.

`DUMMYUNIONNAME.CpuSet.DUMMYUNIONNAME2`

`DUMMYUNIONNAME.CpuSet.DUMMYUNIONNAME2.AllFlags`

`DUMMYUNIONNAME.CpuSet.DUMMYUNIONNAME2.DUMMYSTRUCTNAME`

`DUMMYUNIONNAME.CpuSet.DUMMYUNIONNAME2.DUMMYSTRUCTNAME.Parked`

`DUMMYUNIONNAME.CpuSet.DUMMYUNIONNAME2.DUMMYSTRUCTNAME.Allocated`

`DUMMYUNIONNAME.CpuSet.DUMMYUNIONNAME2.DUMMYSTRUCTNAME.AllocatedToTargetProcess`

`DUMMYUNIONNAME.CpuSet.DUMMYUNIONNAME2.DUMMYSTRUCTNAME.RealTime`

`DUMMYUNIONNAME.CpuSet.DUMMYUNIONNAME2.DUMMYSTRUCTNAME.ReservedFlags`

`DUMMYUNIONNAME.CpuSet.Reserved`

Reserved.

`DUMMYUNIONNAME.CpuSet.SchedulingClass`

`DUMMYUNIONNAME.CpuSet.AllocationTag`

Specifies a tag used by Core Allocation to communicate a given allocated CPU Set between threads in different components.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 10 [desktop apps | UWP apps]
Minimum supported server	Windows Server 2016 [desktop apps | UWP apps]
Header	winnt.h (include Windows.h)

Feedback

Was this page helpful? Yes No

SYSTEM_LOGICAL_PROCESSOR_INFORMATION structure (winnt.h)

Article09/01/2022

Describes the relationship between the specified processor set. This structure is used with the GetLogicalProcessorInformation function.

Syntax

C++

```
typedef struct _SYSTEM_LOGICAL_PROCESSOR_INFORMATION {
  ULONG_PTR                      ProcessorMask;
  LOGICAL_PROCESSOR_RELATIONSHIP Relationship;
  union {
    struct {
      BYTE Flags;
    } ProcessorCore;
    struct {
      DWORD NodeNumber;
    } NumaNode;
    CACHE_DESCRIPTOR Cache;
    ULONGLONG        Reserved[2];
  } DUMMYUNIONNAME;
} SYSTEM_LOGICAL_PROCESSOR_INFORMATION,
*PSYSTEM_LOGICAL_PROCESSOR_INFORMATION;
```

Members

`ProcessorMask`

The processor mask identifying the processors described by this structure. A processor mask is a bit vector in which each set bit represents an active processor in the relationship. At least one bit will be set.

On a system with more than 64 processors, the processor mask identifies processors in a single processor group.

`Relationship`

The relationship between the processors identified by the value of the **ProcessorMask** member. This member can be one of the following LOGICAL_PROCESSOR_RELATIONSHIP values.

Expand table

Value	Meaning
RelationCache 2	The specified logical processors share a cache. The **Cache** member contains additional information. **Windows Server 2003:** This value is not supported until Windows Server 2003 with SP1 and Windows XP Professional x64 Edition.
RelationNumaNode 1	The specified logical processors are part of the same NUMA node. The **NumaNode** member contains additional information.
RelationProcessorCore 0	The specified logical processors share a single processor core. The **ProcessorCore** member contains additional information.
RelationProcessorPackage 3	The specified logical processors share a physical package. There is no additional information available. **Windows Server 2003 and Windows XP Professional x64 Edition:** This value is not supported until Windows Server 2003 with SP1 and Windows XP with SP3.

Future versions of Windows may support additional values for the **Relationship** member.

`DUMMYUNIONNAME`

`DUMMYUNIONNAME.ProcessorCore`

This structure contains valid data only if the **Relationship** member is RelationProcessorCore.

`DUMMYUNIONNAME.ProcessorCore.Flags`

If the value of this member is 1, the logical processors identified by the value of the **ProcessorMask** member share functional units, as in Hyperthreading or SMT. Otherwise, the identified logical processors do not share functional units.

Windows Server 2003 and Windows XP Professional x64 Edition: This member is also 1 for cores that share a physical package. Therefore, to determine whether the processor supports multiple cores or hyperthreading on systems prior to Windows Vista, use the CPUID instruction.

`DUMMYUNIONNAME.NumaNode`

This structure contains valid data only if the **Relationship** member is RelationNumaNode.

`DUMMYUNIONNAME.NumaNode.NodeNumber`

Identifies the NUMA node. The valid values of this parameter are 0 to the highest NUMA node number inclusive. A non-NUMA multiprocessor system will report that all processors belong to one NUMA node.

`DUMMYUNIONNAME.Cache`

A CACHE_DESCRIPTOR structure that identifies the characteristics of a particular cache. There is one record returned for each cache reported. Some or all caches may not be reported, depending on the mechanism used by the processor to identify its caches. Therefore, do not assume the absence of any particular caches. Caches are not necessarily shared among logical processors.

This structure contains valid data only if the **Relationship** member is RelationCache.

Windows Server 2003: This member is not supported until Windows Server 2003 with SP1 and Windows XP Professional x64 Edition.

`DUMMYUNIONNAME.Reserved[2]`

Reserved. Do not use.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows Vista, Windows XP Professional x64 Edition [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	winnt.h (include Windows.h)

See also

CACHE_DESCRIPTOR

GetLogicalProcessorInformation

GetLogicalProcessorInformationEx

LOGICAL_PROCESSOR_RELATIONSHIP

SYSTEM_LOGICAL_PROCESSOR_INFORMATION_EX

Feedback

Was this page helpful? Yes No

SYSTEM_LOGICAL_PROCESSOR_INFORMATION_EX structure (winnt.h)

Article02/22/2024

Contains information about the relationships of logical processors and related hardware. The GetLogicalProcessorInformationEx function uses this structure.

Syntax

```
C++

typedef struct _SYSTEM_LOGICAL_PROCESSOR_INFORMATION_EX {
  LOGICAL_PROCESSOR_RELATIONSHIP Relationship;
  DWORD                          Size;
  union {
    PROCESSOR_RELATIONSHIP Processor;
    NUMA_NODE_RELATIONSHIP NumaNode;
    CACHE_RELATIONSHIP     Cache;
    GROUP_RELATIONSHIP     Group;
  } DUMMYUNIONNAME;
} SYSTEM_LOGICAL_PROCESSOR_INFORMATION_EX,
*PSYSTEM_LOGICAL_PROCESSOR_INFORMATION_EX;
```

Members

`Relationship`

The type of relationship between the logical processors. This parameter can be one of the LOGICAL_PROCESSOR_RELATIONSHIP enumeration values.

`Size`

The size of the structure.

`DUMMYUNIONNAME`

`DUMMYUNIONNAME.Processor`

A PROCESSOR_RELATIONSHIP structure that describes processor affinity. This structure contains valid data only if the **Relationship** member is **RelationProcessorCore**, **RelationProcessorDie**, **RelationProcessorModule** or **RelationProcessorPackage**.

`DUMMYUNIONNAME.NumaNode`

A NUMA_NODE_RELATIONSHIP structure that describes a NUMA node. This structure contains valid data only if the **Relationship** member is **RelationNumaNode** or **RelationNumaNodeEx**.

`DUMMYUNIONNAME.Cache`

A CACHE_RELATIONSHIP structure that describes cache attributes. This structure contains valid data only if the **Relationship** member is **RelationCache**.

`DUMMYUNIONNAME.Group`

A GROUP_RELATIONSHIP structure that contains information about the processor groups. This structure contains valid data only if the **Relationship** member is **RelationGroup**.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Header	winnt.h

See also

CACHE_RELATIONSHIP

GROUP_RELATIONSHIP

NUMA_NODE_RELATIONSHIP

PROCESSOR_RELATIONSHIP

Feedback

Was this page helpful? Yes No

TEB structure (winternl.h)

Article05/29/2024

[This structure may be altered in future versions of Windows. Applications should use the alternate functions listed in this topic.]

The Thread Environment Block (TEB) structure describes the state of a thread.

Syntax

C++

```
typedef struct _TEB {
  PVOID Reserved1[12];
  PPEB  ProcessEnvironmentBlock;
  PVOID Reserved2[399];
  BYTE  Reserved3[1952];
  PVOID TlsSlots[64];
  BYTE  Reserved4[8];
  PVOID Reserved5[26];
  PVOID ReservedForOle;
  PVOID Reserved6[4];
  PVOID TlsExpansionSlots;
} TEB, *PTEB;
```

Members

`Reserved1[12]`

Reserved for internal use by the operating system.

`ProcessEnvironmentBlock`

A pointer to the PEB structure that contains information for the process as a whole.

`Reserved2[399]`

Reserved for internal use by the operating system.

`Reserved3[1952]`

Reserved for internal use by the operating system.

`TlsSlots[64]`

Data for Thread Local Storage. Call the TlsGetValue function to access it.

`Reserved4[8]`

Reserved for internal use by the operating system.

`Reserved5[26]`

Reserved for internal use by the operating system.

`ReservedForOle`

Do not use. Call CoGetContextToken instead.

`Reserved6[4]`

Reserved for internal use by the operating system.

`TlsExpansionSlots`

Additional data for Thread Local Storage. Call the TlsGetValue function to access it.

Remarks

The definition of this structure may change from one version of Windows to the next. Do not assume a maximum size for this structure. To see the members of this structure, refer to winternal.h.

You should not directly access this structure. To access the values of the *TlsSlots* and *TlsExpansionSlots* fields, call TlsGetValue. To access the value of the *ReservedForOle* field, call CoGetContextToken.

In the following versions of Windows, the offset of the 32-bit TEB address within the 64-bit TEB is 0. This can be used to directly access the 32-bit TEB of a WOW64 thread. This might change in later versions of Windows.

Expand table

Windows Vista	Windows Server 2008
Windows 7	Windows Server 2008 R2
Windows 8	Windows Server 2012
Windows 8.1	Windows Server 2012 R2

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	winternl.h

See also

TlsGetValue

Feedback

Was this page helpful? Yes No

Provide product feedback | Get help at Microsoft Q&A

UMS_CREATE_THREAD_ATTRIBUTES structure (winnt.h)

Article04/02/2021

Specifies attributes for a user-mode scheduling (UMS) worker thread.

This structure is used with the UpdateProcThreadAttribute function.

Syntax

```
C++

typedef struct _UMS_CREATE_THREAD_ATTRIBUTES {
  DWORD UmsVersion;
  PVOID UmsContext;
  PVOID UmsCompletionList;
} UMS_CREATE_THREAD_ATTRIBUTES, *PUMS_CREATE_THREAD_ATTRIBUTES;
```

Members

`UmsVersion`

The UMS version for which the application was built. This parameter must be **UMS_VERSION**.

`UmsContext`

A pointer to a UMS thread context for the worker thread to be created. This pointer is provided by the CreateUmsThreadContext function.

`UmsCompletionList`

A pointer to a UMS completion list. This pointer is provided by the CreateUmsCompletionList function. The newly created worker thread is queued to the specified completion list.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Header	winnt.h

Feedback

Was this page helpful? Yes No

UMS_SCHEDULER_STARTUP_INFO structure (winbase.h)

Article03/18/2022

Specifies attributes for a user-mode scheduling (UMS) scheduler thread. The EnterUmsSchedulingMode function uses this structure.

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

Syntax

```C++
typedef struct _UMS_SCHEDULER_STARTUP_INFO {
  ULONG                       UmsVersion;
  PUMS_COMPLETION_LIST        CompletionList;
  PUMS_SCHEDULER_ENTRY_POINT  SchedulerProc;
  PVOID                       SchedulerParam;
} UMS_SCHEDULER_STARTUP_INFO, *PUMS_SCHEDULER_STARTUP_INFO;
```

Members

`UmsVersion`

The UMS version for which the application was built. This parameter must be **UMS_VERSION**.

`CompletionList`

A pointer to a UMS completion list to associate with the calling thread.

`SchedulerProc`

A pointer to an application-defined UmsSchedulerProc entry point function. The system calls this function when the calling thread has been converted to UMS and is ready to run UMS worker threads. Subsequently, it calls this function when a UMS worker thread running on the calling thread yields or blocks.

`SchedulerParam`

An application-defined parameter to pass to the specified UmsSchedulerProc function.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 [desktop apps only]
Minimum supported server	Windows Server 2008 R2 [desktop apps only]
Header	winbase.h (include Windows.h)

Feedback

Was this page helpful? Yes No

UMS_SYSTEM_THREAD_INFORMATION structure (winbase.h)

Article02/22/2024

Specifies a UMS scheduler thread, UMS worker thread, or non-UMS thread.

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

Syntax

C++

```
typedef struct _UMS_SYSTEM_THREAD_INFORMATION {
  ULONG UmsVersion;
  union {
    struct {
      ULONG IsUmsSchedulerThread : 1;
      ULONG IsUmsWorkerThread : 1;
    } DUMMYSTRUCTNAME;
    ULONG ThreadUmsFlags;
  } DUMMYUNIONNAME;
} UMS_SYSTEM_THREAD_INFORMATION, *PUMS_SYSTEM_THREAD_INFORMATION;
```

Members

`UmsVersion`

The UMS version.

You must set this member to UMS_VERSION before calling the GetUmsSystemThreadInformation function.

`DUMMYUNIONNAME`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME`

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.IsUmsSchedulerThread`

A bitfield that specifies that the thread is a UMS scheduler thread.

`DUMMYUNIONNAME.DUMMYSTRUCTNAME.IsUmsWorkerThread`

A bitfield that specifies that the thread is a UMS worker thread.

`DUMMYUNIONNAME.ThreadUmsFlags`

Remarks

Used by the GetUmsSystemThreadInformation function.

At most one of **IsUmsSchedulerThread** and **IsUmsWorkerThread** will be set.

If both **IsUmsSchedulerThread** and **IsUmsWorkerThread** are clear then the thread is a non-UMS thread.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 7 with SP1 [desktop apps only],Windows 7 (64-bit only) and Windows Server 2008 R2 (64-bit only) with KB977165 installed
Minimum supported server	Windows Server 2008 R2 with SP1 [desktop apps only]
Header	winbase.h (include Windows.h)

Feedback

Was this page helpful? Yes No

GetProcessMitigationPolicy function (processthreadsapi.h)

11/01/2022

Retrieves mitigation policy settings for the calling process.

Syntax

```C++
BOOL GetProcessMitigationPolicy(
  [in]  HANDLE                    hProcess,
  [in]  PROCESS_MITIGATION_POLICY MitigationPolicy,
  [out] PVOID                     lpBuffer,
  [in]  SIZE_T                    dwLength
);
```

Parameters

`[in] hProcess`

A handle to the process. This handle must have the PROCESS_QUERY_INFORMATION access right. For more information, see Process Security and Access Rights.

`[in] MitigationPolicy`

The mitigation policy to retrieve. This parameter can be one of the following values.

Expand table

Value	Meaning
ProcessDEPPolicy	The data execution prevention (DEP) policy of the process. The *lpBuffer* parameter points to a PROCESS_MITIGATION_DEP_POLICY structure that specifies the DEP policy flags.
ProcessASLRPolicy	The Address Space Layout Randomization (ASLR) policy of the process. The *lpBuffer* parameter points to a PROCESS_MITIGATION_ASLR_POLICY structure that specifies the ASLR policy flags.

Value	Meaning
ProcessDynamicCodePolicy	The dynamic code policy of the process. When turned on, the process cannot generate dynamic code or modify existing executable code. The *lpBuffer* parameter points to a PROCESS_MITIGATION_DYNAMIC_CODE_POLICY structure that specifies the dynamic code policy flags.
ProcessStrictHandleCheckPolicy	The process will receive a fatal error if it manipulates a handle that is not valid. The *lpBuffer* parameter points to a PROCESS_MITIGATION_STRICT_HANDLE_CHECK_POLICY structure that specifies the handle check policy flags.
ProcessSystemCallDisablePolicy	Disables the ability to use NTUser/GDI functions at the lowest layer. The *lpBuffer* parameter points to a PROCESS_MITIGATION_SYSTEM_CALL_DISABLE_POLICY structure that specifies the system call disable policy flags.
ProcessMitigationOptionsMask	Returns the mask of valid bits for all the mitigation options on the system. An application can set many mitigation options without querying the operating system for mitigation options by combining bitwise with the mask to exclude all non-supported bits at once. The *lpBuffer* parameter points to a **ULONG64** bit vector for the mask, or a two-element array of **ULONG64** bit vectors.
ProcessExtensionPointDisablePolicy	Prevents certain built-in third party extension points from being enabled, preventing legacy extension point DLLs from being loaded into the process. The *lpBuffer* parameter points to a PROCESS_MITIGATION_EXTENSION_POINT_DISABLE_POLICY structure that specifies the extension point disable policy flags.
ProcessControlFlowGuardPolicy	The Control Flow Guard (CFG) policy of the process. The *lpBuffer* parameter points to a PROCESS_MITIGATION_CONTROL_FLOW_GUARD_POLICY structure that specifies the CFG policy flags.
ProcessSignaturePolicy	The policy of a process that can restrict image loading to those images that are either signed by Microsoft, by the Windows Store, or by Microsoft, the Windows Store and the Windows Hardware Quality Labs (WHQL). he *lpBuffer* parameter points to a PROCESS_MITIGATION_BINARY_SIGNATURE_POLICY structure that specifies the signature policy flags.
ProcessFontDisablePolicy	The policy regarding font loading for the process. When turned on, the process cannot load non-system fonts.

Value	Meaning
	The *lpBuffer* parameter points to a PROCESS_MITIGATION_FONT_DISABLE_POLICY structure that specifies the policy flags for font loading.
ProcessImageLoadPolicy	The policy regarding image loading for the process, which determines the types of executable images that are allowed to be mapped into the process. When turned on, images cannot be loaded from some locations, such a remote devices or files that have the low mandatory label. The *lpBuffer* parameter points to a PROCESS_MITIGATION_IMAGE_LOAD_POLICY structure that specifies the policy flags for image loading.
ProcessRedirectionTrustPolicy	The RedirectionGuard policy of a process. The *lpBuffer* parameter points to a PROCESS_MITIGATION_REDIRECTION_TRUST_POLICY structure that specifies the mitigation mode.
ProcessSideChannelIsolationPolicy	Windows 10, version 1809 and above: The policy regarding isolation of side channels for the specified process. The *lpBuffer* parameter points to a PROCESS_MITIGATION_SIDE_CHANNEL_ISOLATION_POLICY structure that specifies the policy flags for side channel isolation.
ProcessUserShadowStackPolicy	Windows 10, version 2004 and above: The policy regarding user-mode Hardware-enforced Stack Protection for the specified process. The *lpBuffer* parameter points to a PROCESS_MITIGATION_USER_SHADOW_STACK_POLICY structure that specifies the policy flags for user-mode Hardware-enforced Stack Protection.

`[out] lpBuffer`

If the *MitigationPolicy* parameter is **ProcessDEPPolicy**, this parameter points to a PROCESS_MITIGATION_DEP_POLICY structure that receives the DEP policy flags.

If the *MitigationPolicy* parameter is **ProcessASLRPolicy**, this parameter points to a PROCESS_MITIGATION_ASLR_POLICY structure that receives the ASLR policy flags.

If the *MitigationPolicy* parameter is **ProcessDynamicCodePolicy**, this parameter points to a PROCESS_MITIGATION_DYNAMIC_CODE_POLICY structure that receives the dynamic code policy flags.

If the *MitigationPolicy* parameter is **ProcessStrictHandleCheckPolicy**, this parameter points to a PROCESS_MITIGATION_STRICT_HANDLE_CHECK_POLICY structure that specifies the handle check policy flags.

If the *MitigationPolicy* parameter is **ProcessSystemCallDisablePolicy**, this parameter points to a PROCESS_MITIGATION_SYSTEM_CALL_DISABLE_POLICY structure that specifies the system call disable policy flags.

If the *MitigationPolicy* parameter is **ProcessMitigationOptionsMask**, this parameter points to a **ULONG64** bit vector for the mask or a two-element array of **ULONG64** bit vectors.

If the *MitigationPolicy* parameter is **ProcessExtensionPointDisablePolicy**, this parameter points to a PROCESS_MITIGATION_EXTENSION_POINT_DISABLE_POLICY structure that specifies the extension point disable policy flags.

If the *MitigationPolicy* parameter is **ProcessControlFlowGuardPolicy**, this parameter points to a PROCESS_MITIGATION_CONTROL_FLOW_GUARD_POLICY structure that specifies the CFG policy flags.

If the *MitigationPolicy* parameter is **ProcessSignaturePolicy**, this parameter points to a PROCESS_MITIGATION_BINARY_SIGNATURE_POLICY structure that receives the signature policy flags.

If the *MitigationPolicy* parameter is **ProcessFontDisablePolicy**, this parameter points to a PROCESS_MITIGATION_FONT_DISABLE_POLICY structure that receives the policy flags for font loading.

If the *MitigationPolicy* parameter is **ProcessImageLoadPolicy**, this parameter points to a PROCESS_MITIGATION_IMAGE_LOAD_POLICY structure that receives the policy flags for image loading.

If the *MitigationPolicy* parameter is **ProcessRedirectionTrustPolicy**, this parameter points to a PROCESS_MITIGATION_REDIRECTION_TRUST_POLICY structure that specifies the mitigation mode.

If the *MitigationPolicy* parameter is **ProcessUserShadowStackPolicy**, this parameter points to a PROCESS_MITIGATION_USER_SHADOW_STACK_POLICY structure that receives the policy flags for user-mode Hardware-enforced Stack Protection.

`[in] dwLength`

The size of *lpBuffer*, in bytes.

Return value

If the function succeeds, it returns **TRUE**. If the function fails, it returns **FALSE**. To retrieve error values defined for this function, call GetLastError.

Remarks

To compile an application that uses this function, set _WIN32_WINNT >= 0x0602. For more information, see Using the Windows Headers.

Requirements

Expand table

Requirement	Value
Minimum supported client	Windows 8 [desktop apps | UWP apps]
Minimum supported server	Windows Server 2012 [desktop apps | UWP apps]
Target Platform	Windows
Header	processthreadsapi.h
Library	Kernel32.lib
DLL	Kernel32.dll

Process Creation Flags

Article • 04/07/2022

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

C

```
     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.

Constant/value	Description
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**.
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**.

Constant/value	Description
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.
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.

Constant/value	Description
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.
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)

Feedback

Was this page helpful? Yes No

Get help at Microsoft Q&A