

processthreadsapi.h header

Article01/24/2023

This header is used by multiple technologies. For more information, see:

- [Processes and threads](#)
- [Remote Desktop Services](#)
- [Security and Identity](#)

processthreadsapi.h contains the following programming interfaces:

Functions

[\[+\] Expand table](#)

CreateProcessA
Creates a new process and its primary thread. The new process runs in the security context of the calling process. (ANSI)
CreateProcessAsUserA
Creates a new process and its primary thread. The new process runs in the security context of the user represented by the specified token. (ANSI)
CreateProcessAsUserW
Creates a new process and its primary thread. The new process runs in the security context of the user represented by the specified token. (Unicode)
CreateProcessW
Creates a new process and its primary thread. The new process runs in the security context of the calling process. (Unicode)
CreateRemoteThread
Creates a thread that runs in the virtual address space of another process.
CreateRemoteThreadEx
Creates a thread that runs in the virtual address space of another process and optionally specifies extended attributes such as processor group affinity.

CreateThread
Creates a thread to execute within the virtual address space of the calling process.
DeleteProcThreadAttributeList
Deletes the specified list of attributes for process and thread creation.
ExitProcess
Ends the calling process and all its threads.
ExitThread
Ends the calling thread.
FlushInstructionCache
Flushes the instruction cache for the specified process.
FlushProcessWriteBuffers
Flushes the write queue of each processor that is running a thread of 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.
GetCurrentProcessorNumberEx
Retrieves the processor group and number of the logical processor in which the calling thread is running.
GetCurrentProcessToken
Retrieves a pseudo-handle that you can use as a shorthand way to refer to the access token associated with a process.
GetCurrentThread

	<p>Retrieves a pseudo handle for the calling thread.</p>
GetCurrentThreadEffectiveToken	<p>Retrieves a pseudo-handle that you can use as a shorthand way to refer to the token that is currently in effect for the thread, which is the thread token if one exists and the process token otherwise.</p>
GetCurrentThreadId	<p>Retrieves the thread identifier of the calling thread.</p>
GetCurrentThreadStackLimits	<p>Retrieves the boundaries of the stack that was allocated by the system for the current thread.</p>
GetCurrentThreadToken	<p>Retrieves a pseudo-handle that you can use as a shorthand way to refer to the impersonation token that was assigned to the current thread.</p>
GetExitCodeProcess	<p>Retrieves the termination status of the specified process.</p>
GetExitCodeThread	<p>Retrieves the termination status of the specified thread.</p>
GetMachineTypeAttributes	<p>Queries if the specified architecture is supported on the current system, either natively or by any form of compatibility or emulation layer.</p>
GetPriorityClass	<p>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.</p>
GetProcessDefaultCpuSetMasks	<p>Retrieves the list of CPU Sets in the process default set that was set by SetProcessDefaultCpuSetMasks or SetProcessDefaultCpuSets.</p>
GetProcessDefaultCpuSets	<p>Retrieves the list of CPU Sets in the process default set that was set by SetProcessDefaultCpuSets.</p>
GetProcessHandleCount	

	<p>Retrieves the number of open handles that belong to the specified process.</p>
GetProcessId	<p>Retrieves the process identifier of the specified process.</p>
GetProcessIdOfThread	<p>Retrieves the process identifier of the process associated with the specified thread.</p>
GetProcessInformation	<p>Retrieves information about the specified process. (GetProcessInformation)</p>
GetProcessMitigationPolicy	<p>Retrieves mitigation policy settings for the calling process.</p>
GetProcessPriorityBoost	<p>Retrieves the priority boost control state of the specified process.</p>
GetProcessShutdownParameters	<p>Retrieves the shutdown parameters for the currently calling process.</p>
GetProcessTimes	<p>Retrieves timing information for the specified process.</p>
GetProcessVersion	<p>Retrieves the major and minor version numbers of the system on which the specified process expects to run.</p>
GetStartupInfoW	<p>Retrieves the contents of the STARTUPINFO structure that was specified when the calling process was created.</p>
GetSystemCpuSetInformation	<p>Allows an application to query the available CPU Sets on the system, and their current state.</p>
GetSystemTimes	<p>Retrieves system timing information. On a multiprocessor system, the values returned are the sum of the designated times across all processors.</p>

GetThreadContext
Retrieves the context of the specified thread.
GetThreadDescription
Retrieves the description that was assigned to a thread by calling SetThreadDescription.
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. (GetThreadInformation)
GetThreadIOPendingFlag
Determines whether a specified thread has any I/O requests pending.
GetThreadPriority
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.
GetThreadPriorityBoost
Retrieves the priority boost control state of the specified thread.
GetThreadSelectedCpuSetMasks
Returns the explicit CPU Set assignment of the specified thread, if any assignment was set using SetThreadSelectedCpuSetMasks or SetThreadSelectedCpuSets.
GetThreadSelectedCpuSets
Returns the explicit CPU Set assignment of the specified thread, if any assignment was set using the SetThreadSelectedCpuSets API.
GetThreadTimes
Retrieves timing information for the specified thread.
InitializeProcThreadAttributeList

	Initializes the specified list of attributes for process and thread creation.
IsProcessCritical	Determines whether the specified process is considered critical.
IsProcessorFeaturePresent	Determines whether the specified processor feature is supported by the current computer.
OpenProcess	Opens an existing local process object.
OpenProcessToken	Opens the access token associated with a process.
OpenThread	Opens an existing thread object.
OpenThreadToken	Opens the access token associated with a thread.
ProcessIdToSessionId	Retrieves the Remote Desktop Services session associated with a specified process.
QueryProcessAffinityUpdateMode	Retrieves the affinity update mode of the specified process.
QueryProtectedPolicy	Queries the value associated with a protected policy.
QueueUserAPC	Adds a user-mode asynchronous procedure call (APC) object to the APC queue of the specified thread. (QueueUserAPC)
QueueUserAPC2	Adds a user-mode asynchronous procedure call (APC) object to the APC queue of the specified thread. (QueueUserAPC2)
ResumeThread	

Decrements a thread's suspend count. When the suspend count is decremented to zero, the execution of the thread is resumed.

[SetPriorityClass](#)

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.

[SetProcessAffinityUpdateMode](#)

Sets the affinity update mode of the specified process.

[SetProcessDefaultCpuSetMasks](#)

The SetProcessDefaultCpuSetMasks function (processthreadsapi.h) sets the default CPU Sets assignment for threads in the specified process.

[SetProcessDefaultCpuSets](#)

The SetProcessDefaultCpuSets function (processthreadsapi.h) sets the default CPU Sets assignment for threads in the specified process.

[SetProcessDynamicEHContinuationTargets](#)

Sets dynamic exception handling continuation targets for the specified process.

[SetProcessDynamicEnforcedCetCompatibleRanges](#)

Sets dynamic enforced CETCOMPAT ranges for the specified process.

[SetProcessInformation](#)

Sets information for the specified process.

[SetProcessMitigationPolicy](#)

Sets a mitigation policy for the calling process. Mitigation policies enable a process to harden itself against various types of attacks.

[SetProcessPriorityBoost](#)

Disables or enables the ability of the system to temporarily boost the priority of the threads of the specified process.

[SetProcessShutdownParameters](#)

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.

SetProtectedPolicy
Sets a protected policy.
SetThreadContext
Sets the context for the specified thread.
SetThreadDescription
Assigns a description to a thread.
SetThreadIdealProcessor
Sets a preferred processor for a thread. The system schedules threads on their preferred processors whenever possible.
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. This value, together with the priority class of the thread's process, determines the thread's base priority level.
SetThreadPriorityBoost
Disables or enables the ability of the system to temporarily boost the priority of a thread.
SetThreadSelectedCpuSetMasks
Sets the selected CPU Sets assignment for the specified thread. This assignment overrides the process default assignment, if one is set. (SetThreadSelectedCpuSetMasks)
SetThreadSelectedCpuSets
Sets the selected CPU Sets assignment for the specified thread. This assignment overrides the process default assignment, if one is set. (SetThreadSelectedCpuSets)
SetThreadStackGuarantee
Sets the minimum size of the stack associated with the calling thread or fiber that will be available during any stack overflow exceptions.

[SetThreadToken](#)

Assigns an impersonation token to a thread. The function can also cause a thread to stop using an impersonation token.

[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. The operating system selects the next thread to be executed.

[TerminateProcess](#)

Terminates the specified process and all of its threads.

[TerminateThread](#)

Terminates a thread.

[TlsAlloc](#)

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.

[TlsFree](#)

Releases a thread local storage (TLS) index, making it available for reuse.

[TlsGetValue](#)

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.

[TlsGetValue2](#)

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.

[TlsSetValue](#)

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.

[UpdateProcThreadAttribute](#)

Updates the specified attribute in a list of attributes for process and thread creation.

Structures

[\[+\] Expand table](#)

[APP_MEMORY_INFORMATION](#)

Represents app memory usage at a single point in time. This structure is used by the PROCESS_INFORMATION_CLASS class.

[MEMORY_PRIORITY_INFORMATION](#)

Specifies the memory priority for a thread or process.

[OVERRIDE_PREFETCH_PARAMETER](#)

[PROCESS_INFORMATION](#)

Contains information about a newly created process and its primary thread. It is used with the CreateProcess, CreateProcessAsUser, CreateProcessWithLogonW, or CreateProcessWithTokenW function.

[PROCESS_LEAP_SECOND_INFO](#)

Specifies how the system handles positive leap seconds.

[PROCESS_MACHINE_INFORMATION](#)

Specifies the architecture of a process and if that architecture of code can run in user mode, kernel mode, and/or under WoW64 on the host operating system.

[PROCESS_MEMORY_EXHAUSTION_INFO](#)

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.

[PROCESS_POWER_THROTTLING_STATE](#)

Specifies the throttling policies and how to apply them to a target process when that process is subject to power management.

[PROCESS_PROTECTION_LEVEL_INFORMATION](#)

Specifies whether Protected Process Light (PPL) is enabled.
STARTUPINFOA
Specifies the window station, desktop, standard handles, and appearance of the main window for a process at creation time. (ANSI)
STARTUPINFOW
Specifies the window station, desktop, standard handles, and appearance of the main window for a process at creation time. (Unicode)
THREAD_POWER_THROTTLING_STATE
Specifies the throttling policies and how to apply them to a target thread when that thread is subject to power management.

Enumerations

[\[+\] Expand table](#)

MACHINE_ATTRIBUTES
Specifies the ways in which an architecture of code can run on a host operating system. More than one bit may be set.
PROCESS_INFORMATION_CLASS
Indicates a specific class of process information.
PROCESS_MEMORY_EXHAUSTION_TYPE
Represents the different memory exhaustion types.
QUEUE_USER_AP_C_FLAGS
The QUEUE_USER_AP_C_FLAGS enumeration (processthreadsapi.h) specifies the modifier flags for user-mode asynchronous procedure call (APC) objects.
THREAD_INFORMATION_CLASS
The THREAD_INFORMATION_CLASS enumeration (processthreadsapi.h) specifies the collection of supported thread types.

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

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

CreateProcessA function (processsthreadsapi.h)

Article02/09/2023

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 determines whether the returned handle to the new thread object can be inherited by child processes. If *lpThreadAttributes* is NULL, the handle cannot be inherited.

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

[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 **BELLOW_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 = _tcstrup(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[] = _tcstrup(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 (processsthreadsapi.h)

Article 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 object and determines whether child processes can inherit the returned handle to the thread. 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[] = _tcscdup(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[] = _tcscdup(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	<code>processthreadsapi.h</code> (include <code>Windows.h</code>)
Library	<code>Advapi32.lib</code>
DLL	<code>Advapi32.dll</code>

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](#)

CreateProcessAsUserW function (processthreadsapi.h)

Article 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 CreateProcessAsUserW(
    [in, optional]     HANDLE          hToken,
    [in, optional]     LPCWSTR        lpApplicationName,
    [in, out, optional] LPWSTR         lpCommandLine,
    [in, optional]     LPSECURITY_ATTRIBUTES lpProcessAttributes,
    [in, optional]     LPSECURITY_ATTRIBUTES lpThreadAttributes,
    [in]               BOOL            bInheritHandles,
    [in]               DWORD           dwCreationFlags,
    [in, optional]     LPVOID          lpEnvironment,
    [in, optional]     LPCWSTR        lpCurrentDirectory,
    [in]               LPSTARTUPINFO   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 object and determines whether child processes can inherit the returned handle to the thread. 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 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 not encoding-neutral can lead to mismatches that

result in compilation or runtime errors. For more information, see [Conventions for Function Prototypes](#).

Requirements

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](#)

Feedback

Was this page helpful?

 Yes

 No

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

CreateProcessW function (processthreadsapi.h)

Article02/09/2023

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 [CreateProcessAsUser](#) or [CreateProcessWithLogonW](#) function.

Syntax

C++

```
BOOL CreateProcessW(
    [in, optional]     LPCWSTR           lpApplicationName,
    [in, out, optional] LPWSTR            lpCommandLine,
    [in, optional]     LPSECURITY_ATTRIBUTES lpProcessAttributes,
    [in, optional]     LPSECURITY_ATTRIBUTES lpThreadAttributes,
    [in]               BOOL              bInheritHandles,
    [in]               DWORD             dwCreationFlags,
    [in, optional]     LPVOID            lpEnvironment,
    [in, optional]     LPCWSTR           lpCurrentDirectory,
    [in]               LPSTARTUPINFO     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.

`[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:

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. The name of this directory is System.
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 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 determines whether the returned handle to the new thread object can be inherited by child processes. If *lpThreadAttributes* is NULL, the handle cannot be inherited.

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

[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 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] lppProcessInformation

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 = _tcstrup(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[] = _tcstrup(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 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

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](#)

[ShellExecuteW](#)

[CreateProcessAsUser](#)

[CreateProcessWithLogonW](#)

[ExitProcess](#)

[GetCommandLine](#)

[GetEnvironmentStrings](#)

[GetExitCodeProcess](#)

[GetFullPathName](#)

[GetStartupInfo](#)

[OpenProcess](#)

[PROCESS_INFORMATION](#)

[Process and Thread Functions](#)

[Processes](#)

[SECURITY_ATTRIBUTES](#)

[STARTUPINFO](#)

[STARTUPINFOEX](#)

[SetErrorMode](#)

[TerminateProcess](#)

[WaitForInputIdle](#)

Feedback

Was this page helpful?

 Yes

 No

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

CreateRemoteThread function (processsthreadsapi.h)

Article11/01/2022

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.

Windows XP: The ACLs in the default security descriptor for a thread come from the primary or impersonation token of the creator. This behavior changed with Windows XP with SP2 and Windows Server 2003.

[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 <i>dwStackSize</i> parameter specifies the initial reserve size of the stack. If this flag is not specified, <i>dwStackSize</i> 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 an 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 (processsthreadsapi.h)

Article11/01/2022

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.
<code>CREATE_SUSPENDED</code> 0x00000004	The thread is created in a suspended state and does not run until the ResumeThread function is called.
<code>STACK_SIZE_PARAM_IS_A_RESERVATION</code> 0x00010000	The <code>dwStackSize</code> parameter specifies the initial reserve size of the stack. If this flag is not specified, <code>dwStackSize</code> 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 (processsthreadsapi.h)

Article 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 determines whether the returned handle can be inherited by child processes. If *lpThreadAttributes* is NULL, the handle cannot be inherited.

The **lpSecurityDescriptor** member of the structure specifies a security descriptor for the new thread. If *lpThreadAttributes* is NULL, the thread gets a default security descriptor. The ACLs 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 <i>dwStackSize</i> parameter specifies the initial reserve size of the stack. If this flag is not specified, <i>dwStackSize</i> 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

Requirement	Value
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](#)

DeleteProcThreadAttributeList function (processthreadsapi.h)

Article02/22/2024

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)
Library	Kernel32.lib

Requirement	Value
DLL	Kernel32.dll

See also

[InitializeProcThreadAttributeList](#)

[Process and Thread Functions](#)

ExitProcess function (processsthreadsapi.h)

Article 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 (processsthreadsapi.h)

Article 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]

Requirement	Value
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](#)

FlushInstructionCache function (processthreadsapi.h)

Article 02/22/2024

Flushes the instruction cache for the specified process.

Syntax

C++

```
BOOL FlushInstructionCache(
    [in] HANDLE hProcess,
    [in] LPCVOID lpBaseAddress,
    [in] SIZE_T dwSize
);
```

Parameters

[in] `hProcess`

A handle to a process whose instruction cache is to be flushed.

[in] `lpBaseAddress`

A pointer to the base of the region to be flushed. This parameter can be **NULL**.

[in] `dwSize`

The size of the region to be flushed if the *lpBaseAddress* parameter is not **NULL**, 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

Applications should call **FlushInstructionCache** if they generate or modify code in memory. The CPU cannot detect the change, and may execute the old code it cached.

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

[Debugging Functions](#)

FlushProcessWriteBuffers function (processsthreadsapi.h)

Article11/01/2022

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	processsthreadsapi.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](#)

GetCurrentProcess function (processsthreadsapi.h)

Article02/22/2024

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	processsthreadsapi.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 (processsthreadsapi.h)

Article02/22/2024

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	processsthreadsapi.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 (processsthreadsapi.h)

Article02/22/2024

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 (processsthreadsapi.h)

Article02/22/2024

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

GetCurrentProcessToken function (processsthreadsapi.h)

Article02/22/2024

Retrieves a pseudo-handle that you can use as a shorthand way to refer to the [access token](#) associated with a process.

Syntax

C++

```
HANDLE GetCurrentProcessToken();
```

Return value

A pseudo-handle that you can use as a shorthand way to refer to the [access token](#) associated with a process.

Remarks

A pseudo-handle is a special constant that can function as the access token for the current process. The calling process can use a pseudo-handle to specify the access token for that process whenever a token handle is required. Child processes do not inherit pseudo-handles.

Starting in Windows 8, this pseudo-handle has only TOKEN_QUERY and TOKEN_QUERY_SOURCE access rights.

The pseudo-handle cannot be duplicated by the [DuplicateHandle](#) function or the [DuplicateToken](#) function.

You do not need to close the pseudo-handle when you no longer need it. If you call the [CloseHandle](#) function with a pseudo-handle, the function has no effect.

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	

See also

[Access Rights for Access-Token Objects](#)

[OpenProcessToken](#)

GetCurrentThread function (processsthreadsapi.h)

Article 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](#)

Vert.dll APIs available in VBS enclaves

GetCurrentThreadEffectiveToken function (processthreadsapi.h)

Article02/22/2024

Retrieves a pseudo-handle that you can use as a shorthand way to refer to the token that is currently in effect for the thread, which is the thread token if one exists and the process token otherwise.

Syntax

C++

```
HANDLE GetCurrentThreadEffectiveToken();
```

Return value

A pseudo-handle that you can use as a shorthand way to refer to the token that is currently in effect for the thread.

Remarks

A pseudo-handle is a special constant that can function as the effective token for the current thread. The calling thread can use a pseudo-handle to specify the effective token for that thread whenever a token handle is required. Child processes do not inherit pseudo-handles.

Starting in Windows 8, this pseudo-handle has only TOKEN_QUERY and TOKEN_QUERY_SOURCE access rights.

The pseudo-handle cannot be duplicated by the [DuplicateHandle](#) function or the [DuplicateToken](#) function.

You do not need to close the pseudo-handle when you no longer need it. If you call the [CloseHandle](#) function with a pseudo-handle, the function has no effect.

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	

See also

[Access Rights for Access-Token Objects](#)

[GetCurrentProcessToken](#)

[GetCurrentThreadToken](#)

GetCurrentThreadId function (processsthreadsapi.h)

Article02/22/2024

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
Header	processsthreadsapi.h (include Windows.h on Windows Server 2003, Windows Vista,

Requirement	Value
	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 (processsthreadsapi.h)

Article02/22/2024

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](#)

GetCurrentThreadToken function (processsthreadsapi.h)

Article02/22/2024

Retrieves a pseudo-handle that you can use as a shorthand way to refer to the [impersonation token](#) that was assigned to the current thread.

Syntax

C++

```
HANDLE GetCurrentThreadToken();
```

Return value

A pseudo-handle that you can use as a shorthand way to refer to the [impersonation token](#) that was assigned to the current thread.

Remarks

A pseudo-handle is a special constant that can function as the impersonation token for the current thread. The calling thread can use a pseudo-handle to specify the impersonation token for that thread whenever a token handle is required. Child processes do not inherit pseudo-handles.

Starting in Windows 8, this pseudo-handle has only TOKEN_QUERY and TOKEN_QUERY_SOURCE access rights.

The pseudo-handle cannot be duplicated by the [DuplicateHandle](#) function or the [DuplicateToken](#) function.

You do not need to close the pseudo-handle when you no longer need it. If you call the [CloseHandle](#) function with a pseudo-handle, the function has no effect.

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	

See also

[Access Rights for Access-Token Objects](#)

[OpenThreadToken](#)

[SetThreadToken](#)

GetExitCodeProcess function (processsthreadsapi.h)

Article11/01/2022

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.

 [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 (processsthreadsapi.h)

Article 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)
Library	Kernel32.lib; WindowsPhoneCore.lib on Windows Phone 8.1

Requirement	Value
DLL	Kernel32.dll; KernelBase.dll on Windows Phone 8.1

See also

[ExitThread](#)

[GetExitCodeProcess](#)

[OpenThread](#)

[Process and Thread Functions](#)

[TerminateThread](#)

[Terminating a Thread](#)

GetMachineTypeAttributes function (processthreadsapi.h)

Article02/22/2024

Queries if the specified architecture is supported on the current system, either natively or by any form of compatibility or emulation layer.

Syntax

C++

```
HRESULT GetMachineTypeAttributes(
    USHORT           Machine,
    MACHINE_ATTRIBUTES *MachineTypeAttributes
);
```

Parameters

Machine

An IMAGE_FILE_MACHINE_* value corresponding to the architecture of code to be tested for supportability. See the list of architecture values in [Image File Machine Constants](#).

MachineTypeAttributes

Output parameter receives a pointer to a value from the [MACHINE_ATTRIBUTES](#) enumeration indicating if the specified code architecture can run in user mode, kernel mode, and/or under WOW64 on the host operating system.

Return value

If the function fails, the return value is a nonzero HRESULT value. If the function succeeds, the return value is zero.

Requirements

[+] [Expand table](#)

Requirement	Value
Minimum supported client	Windows Build 22000
Minimum supported server	Windows Build 22000
Header	processthreadsapi.h
Library	Kernel32.lib
DLL	Kernel32.dll

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

GetPriorityClass function (processthreadsapi.h)

Article11/01/2022

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	processsthreadsapi.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](#)

GetProcessDefaultCpuSetMasks function (processthreadsapi.h)

Article01/27/2022

Retrieves the list of CPU Sets in the process default set that was set by [SetProcessDefaultCpuSetMasks](#) or [SetProcessDefaultCpuSets](#).

Syntax

C++

```
BOOL GetProcessDefaultCpuSetMasks(
    HANDLE           Process,
    PGROUP_AFFINITY CpuSetMasks,
    USHORT          CpuSetMaskCount,
    PUSHORT         RequiredMaskCount
);
```

Parameters

Process

Specifies a process handle for the process to query. This handle must have the [PROCESS_QUERY_LIMITED_INFORMATION](#) access right. The value returned by [GetCurrentProcess](#) can also be specified here.

CpuSetMasks

Specifies an optional buffer to retrieve a list of [GROUP_AFFINITY](#) structures representing the process default CPU Sets.

CpuSetMaskCount

Specifies the size of the *CpuSetMasks* array, in elements.

RequiredMaskCount

On successful return, specifies the number of affinity structures written to the array. If the *CpuSetMasks* array is too small, the function fails with [ERROR_INSUFFICIENT_BUFFER](#) and sets the *RequiredMaskCount* parameter to the

number of elements required. The number of required elements is always less than or equal to the maximum group count returned by [GetMaximumProcessorGroupCount](#).

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero and extended error information can be retrieved by calling [GetLastError](#).

If the array supplied is too small, the error value is **ERROR_INSUFFICIENT_BUFFER** and the *RequiredMaskCount* is set to the number of elements required.

Remarks

If no default CPU Sets are set for a given process, then the *RequiredMaskCount* parameter is set to 0 and the function succeeds.

This function is analogous to [GetProcessDefaultCpuSets](#), except that it uses group affinities as opposed to CPU Set IDs to represent a list of CPU sets. This means that the process default CPU Sets are mapped to their home processors, and those processors are retrieved in the resulting list of group affinities.

Requirements

Minimum supported client	Windows 11
Minimum supported server	Windows Server 2022
Header	processthreadsapi.h
DLL	kernel32.dll

Feedback

Was this page helpful?

 Yes

 No

Get help at Microsoft Q&A

GetProcessDefaultCpuSets function (processthreadsapi.h)

Article02/22/2024

Retrieves the list of CPU Sets in the process default set that was set by [SetProcessDefaultCpuSets](#). If no default CPU Sets are set for a given process, then the **RequiredIdCount** is set to 0 and the function succeeds.

Syntax

C++

```
BOOL GetProcessDefaultCpuSets(
    HANDLE Process,
    PULONG CpuSetIds,
    ULONG CpuSetIdCount,
    PULONG RequiredIdCount
);
```

Parameters

Process

Specifies a process handle for the process to query. This handle must have the PROCESS_QUERY_LIMITED_INFORMATION access right. The value returned by [GetCurrentProcess](#) 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 process default 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 API returns FALSE, and 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

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

GetProcessHandleCount function (processsthreadsapi.h)

Article02/22/2024

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)

Article02/22/2024

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

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)

Article02/22/2024

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

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 (processsthreadsapi.h)

Article01/31/2024

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 [OVERLAY_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(PERSONAL_POWER_THROTTLING_STATE)`.

If the *ProcessInformationClass* parameter is **ProcessProtectionLevelInfo**, this parameter must be `sizeof(PROTECTION_LEVEL_INFORMATION)`.

If the *ProcessInformationClass* parameter is **ProcessLeapSecondInfo**, this parameter must be `sizeof(PERSONAL_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]

Requirement	Value
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](#)

GetProcessMitigationPolicy function (processthreadsapi.h)

Article11/01/2022

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 <i>lpBuffer</i> 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 <i>lpBuffer</i> parameter points to a PROCESS_MITIGATION_ASLR_POLICY structure that specifies the ASLR policy flags.

Value	Meaning
ProcessDynamicCodePolicy	<p>The dynamic code policy of the process. When turned on, the process cannot generate dynamic code or modify existing executable code.</p> <p>The <i>lpBuffer</i> parameter points to a PROCESS_MITIGATION_DYNAMIC_CODE_POLICY structure that specifies the dynamic code policy flags.</p>
ProcessStrictHandleCheckPolicy	<p>The process will receive a fatal error if it manipulates a handle that is not valid.</p> <p>The <i>lpBuffer</i> parameter points to a PROCESS_MITIGATION_STRICT_HANDLE_CHECK_POLICY structure that specifies the handle check policy flags.</p>
ProcessSystemCallDisablePolicy	<p>Disables the ability to use NTUser/GDI functions at the lowest layer.</p> <p>The <i>lpBuffer</i> parameter points to a PROCESS_MITIGATION_SYSTEM_CALL_DISABLE_POLICY structure that specifies the system call disable policy flags.</p>
ProcessMitigationOptionsMask	<p>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.</p> <p>The <i>lpBuffer</i> parameter points to a ULONG64 bit vector for the mask, or a two-element array of ULONG64 bit vectors.</p>
ProcessExtensionPointDisablePolicy	<p>Prevents certain built-in third party extension points from being enabled, preventing legacy extension point DLLs from being loaded into the process.</p> <p>The <i>lpBuffer</i> parameter points to a PROCESS_MITIGATION_EXTENSION_POINT_DISABLE_POLICY structure that specifies the extension point disable policy flags.</p>
ProcessControlFlowGuardPolicy	<p>The Control Flow Guard (CFG) policy of the process.</p> <p>The <i>lpBuffer</i> parameter points to a PROCESS_MITIGATION_CONTROL_FLOW_GUARD_POLICY structure that specifies the CFG policy flags.</p>
ProcessSignaturePolicy	<p>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).</p> <p>The <i>lpBuffer</i> parameter points to a PROCESS_MITIGATION_BINARY_SIGNATURE_POLICY structure that specifies the signature policy flags.</p>
ProcessFontDisablePolicy	<p>The policy regarding font loading for the process. When turned on, the process cannot load non-system fonts.</p>

Value	Meaning
	The <i>lpBuffer</i> parameter points to a PROCESS_MITIGATION_FONT_DISABLE_POLICY structure that specifies the policy flags for font loading.
ProcessImageLoadPolicy	<p>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 as remote devices or files that have the low mandatory label.</p> <p>The <i>lpBuffer</i> parameter points to a PROCESS_MITIGATION_IMAGE_LOAD_POLICY structure that specifies the policy flags for image loading.</p>
ProcessRedirectionTrustPolicy	The RedirectionGuard policy of a process. The <i>lpBuffer</i> parameter points to a PROCESS_MITIGATION_REDIRECTION_TRUST_POLICY structure that specifies the mitigation mode.
ProcessSideChannelIsolationPolicy	<p>Windows 10, version 1809 and above: The policy regarding isolation of side channels for the specified process.</p> <p>The <i>lpBuffer</i> parameter points to a PROCESS_MITIGATION_SIDE_CHANNEL_ISOLATION_POLICY structure that specifies the policy flags for side channel isolation.</p>
ProcessUserShadowStackPolicy	<p>Windows 10, version 2004 and above: The policy regarding user-mode Hardware-enforced Stack Protection for the specified process.</p> <p>The <i>lpBuffer</i> parameter points to a PROCESS_MITIGATION_USER_SHADOW_STACK_POLICY structure that specifies the policy flags for user-mode Hardware-enforced Stack Protection.</p>

[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)

Article02/22/2024

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

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 (processsthreadsapi.h)

Article 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)

Article11/01/2022

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	processsthreadsapi.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)

Article11/01/2022

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](#)

GetStartupInfoW function (processthreadsapi.h)

Article 02/22/2024

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 (processthreadsapi.h)

Article02/22/2024

Allows an application to query the available CPU Sets on the system, and their current state.

Syntax

C++

```
BOOL GetSystemCpuSetInformation(
    PSYSTEM_CPU_SET_INFORMATION Information,
    ULONG BufferLength,
    PULONG ReturnedLength,
    HANDLE Process,
    ULONG Flags
);
```

Parameters

Information

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

The length, in bytes, of the output buffer passed as the Information argument.

ReturnedLength

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

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

[\[+\] Expand table](#)

Requirement	Value
Minimum supported client	Windows 10
Minimum supported server	Windows Server 2016
Header	processthreadsapi.h

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

GetSystemTimes function (processsthreadsapi.h)

Article02/22/2024

Retrieves system timing information. On a multiprocessor system, the values returned are the sum of the designated times across all processors.

Syntax

C++

```
BOOL GetSystemTimes(
    [out, optional] PFILETIME lpIdleTime,
    [out, optional] PFILETIME lpKernelTime,
    [out, optional] PFILETIME lpUserTime
);
```

Parameters

[out, optional] lpIdleTime

A pointer to a [FILETIME](#) structure that receives the amount of time that the system has been idle.

[out, optional] lpKernelTime

A pointer to a [FILETIME](#) structure that receives the amount of time that the system has spent executing in Kernel mode (including all threads in all processes, on all processors). This time value also includes the amount of time the system has been idle.

[out, optional] lpUserTime

A pointer to a [FILETIME](#) structure that receives the amount of time that the system has spent executing in User mode (including all threads in all processes, on all processors).

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 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

[FILETIME](#)

[System Time](#)

[Time Functions](#)

GetThreadContext function (processsthreadsapi.h)

Article 02/02/2023

Retrieves the context of the specified thread.

A 64-bit application can retrieve the context of a WOW64 thread using the [Wow64GetThreadContext](#).

Syntax

C++

```
BOOL GetThreadContext(
    [in]      HANDLE     hThread,
    [in, out] LPCONTEXT lpContext
);
```

Parameters

[in] hThread

A handle to the thread whose context is to be retrieved. The handle must have **THREAD_GET_CONTEXT** access to the thread. For more information, see [Thread Security and Access Rights](#).

Windows XP or Windows Server 2003: The handle must also have **THREAD_QUERY_INFORMATION** access.

[in, out] lpContext

A pointer to a **CONTEXT** structure (such as [ARM64_NT_CONTEXT](#)) that receives the appropriate context of the specified thread. The value of the **ContextFlags** member of this structure specifies which portions of a thread's context are retrieved. The **CONTEXT** structure is highly processor specific. Refer to the WinNT.h header file for processor-specific definitions of this structures and any alignment requirements.

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 is used to retrieve the thread context of the specified thread. The function retrieves a selective context based on the value of the **ContextFlags** member of the context structure. The thread identified by the *hThread* parameter is typically being debugged, but the function can also operate when the thread is not being debugged.

You cannot get a valid context for a running thread. Use the [SuspendThread](#) function to suspend the thread before calling [GetThreadContext](#).

If you call [GetThreadContext](#) for the current thread, the function returns successfully; however, the context returned is not valid.

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

- [CONTEXT](#)
- [ARM64_NT_CONTEXT](#)
- [Debugging Functions](#)
- [GetXStateFeaturesMask](#)
- [SetThreadContext](#)
- [SuspendThread](#)
- [Wow64GetThreadContext](#)

GetThreadDescription function (processsthreadsapi.h)

Article11/01/2022

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](#)

GetThreadId function (processsthreadsapi.h)

Article02/22/2024

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 (processsthreadsapi.h)

Article11/01/2022

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

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 (processsthreadsapi.h)

Article 02/22/2024

Retrieves information about the specified thread.

Syntax

C++

```
BOOL GetThreadInformation(
    [in] HANDLE             hThread,
    [in] THREAD_INFORMATION_CLASS ThreadInformationClass,
    [out] 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 (processsthreadsapi.h)

Article11/01/2022

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 (processsthreadsapi.h)

Article11/01/2022

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)

Article02/22/2024

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](#)

GetThreadSelectedCpuSetMasks function (processthreadsapi.h)

Article 02/22/2024

Returns the explicit CPU Set assignment of the specified thread, if any assignment was set using [SetThreadSelectedCpuSetMasks](#) or [SetThreadSelectedCpuSets](#).

Syntax

C++

```
BOOL GetThreadSelectedCpuSetMasks(
    HANDLE           Thread,
    PGROUP_AFFINITY CpuSetMasks,
    USHORT          CpuSetMaskCount,
    PUSHORT         RequiredMaskCount
);
```

Parameters

Thread

Specifies the thread for which to query the selected CPU Sets. This handle must have the [PROCESS_QUERY_LIMITED_INFORMATION](#) access right. The value returned by [GetCurrentProcess](#) can also be specified here.

CpuSetMasks

Specifies an optional buffer to retrieve a list of [GROUP_AFFINITY](#) structures representing the thread selected CPU Sets.

CpuSetMaskCount

Specifies the size of the *CpuSetMasks* array, in elements.

RequiredMaskCount

On successful return, specifies the number of affinity structures written to the array. If the array is too small, the function fails with [ERROR_INSUFFICIENT_BUFFER](#) and sets the *RequiredMaskCount* parameter to the number of elements required. The number of

required elements is always less than or equal to the maximum group count returned by [GetMaximumProcessorGroupCount](#).

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero and extended error information can be retrieved by calling [GetLastError](#).

If the array supplied is too small, the error value is **ERROR_INSUFFICIENT_BUFFER** and the RequiredMaskCount is set to the number of elements required.

Remarks

If no explicit assignment is set, *RequiredMaskCount* is set to 0 and the function succeeds.

This function is analogous to [GetThreadSelectedCpuSets](#), except that it uses group affinities as opposed to CPU Set IDs to represent a list of CPU sets. This means that the thread selected CPU Sets are mapped to their home processors, and those processors are retrieved in the resulting list of group affinities.

Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 11
Minimum supported server	Windows Server 2022
Header	processthreadsapi.h
DLL	kernel32.dll

Feedback

Was this page helpful?

 Yes

 No

GetThreadSelectedCpuSets function (processthreadsapi.h)

Article 08/23/2022

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 (processsthreadsapi.h)

Article11/01/2022

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](#)

InitializeProcThreadAttributeList function (processthreadsapi.h)

Article 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,
    [in]           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](#)

IsProcessCritical function (processsthreadsapi.h)

Article 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](#)

IsProcessorFeaturePresent function (processsthreadsapi.h)

Article02/02/2024

Determines whether the specified processor feature is supported by the current computer.

Syntax

C++

```
BOOL IsProcessorFeaturePresent(
    [in] DWORD ProcessorFeature
);
```

Parameters

[in] ProcessorFeature

The processor feature to be tested. This parameter can be one of the following values.

 Expand table

Value	Meaning
PF_ARM_64BIT_LOADSTORE_ATOMIC 25	The 64-bit load/store atomic instructions are available.
PF_ARM_DIVIDE_INSTRUCTION_AVAILABLE 24	The divide instructions are available.
PF_ARM_EXTERNAL_CACHE_AVAILABLE 26	The external cache is available.
PF_ARM_FMAC_INSTRUCTIONS_AVAILABLE 27	The floating-point multiply-accumulate instruction is available.
PF_ARM_VFP_32_REGISTERS_AVAILABLE 18	The VFP/Neon: 32 x 64bit register bank is present. This flag has the same meaning as PF_ARM_VFP_EXTENDED_REGISTERS .
PF_3DNOW_INSTRUCTIONS_AVAILABLE 7	The 3D-Now instruction set is available.
PF_CHANNELS_ENABLED 16	The processor channels are enabled.

PF_COMPARE_EXCHANGE_DOUBLE 2	The atomic compare and exchange operation (cmpxchg) is available.
PF_COMPARE_EXCHANGE128 14	The atomic compare and exchange 128-bit operation (cmpxchg16b) is available. Windows Server 2003 and Windows XP/2000: This feature is not supported.
PF_COMPARE64_EXCHANGE128 15	The atomic compare 64 and exchange 128-bit operation (cmp8xchg16) is available. Windows Server 2003 and Windows XP/2000: This feature is not supported.
PF_FASTFAIL_AVAILABLE 23	_fastfail() is available.
PF_FLOATING_POINT_EMULATED 1	Floating-point operations are emulated using a software emulator. This function returns a nonzero value if floating-point operations are emulated; otherwise, it returns zero.
PF_FLOATING_POINT_PRECISION_ERRATA 0	On a Pentium, a floating-point precision error can occur in rare circumstances.
PF_MMX_INSTRUCTIONS_AVAILABLE 3	The MMX instruction set is available.
PF_NX_ENABLED 12	Data execution prevention is enabled. Windows XP/2000: This feature is not supported until Windows XP with SP2 and Windows Server 2003 with SP1.
PF_PAE_ENABLED 9	The processor is PAE-enabled. For more information, see Physical Address Extension . All x64 processors always return a nonzero value for this feature.
PF_RDTSC_INSTRUCTION_AVAILABLE 8	The RDTSC instruction is available.
PF_RDWRFSGSBASE_AVAILABLE 22	RDFSBASE, RDGSBASE, WRFSBASE, and WRGSBASE instructions are available.
PF_SECOND_LEVEL_ADDRESS_TRANSLATION 20	Second Level Address Translation is supported by the hardware.
PF_SSE3_INSTRUCTIONS_AVAILABLE 13	The SSE3 instruction set is available.

		Windows Server 2003 and Windows XP/2000: This feature is not supported.
PF_SSSE3_INSTRUCTIONS_AVAILABLE 36		The SSSE3 instruction set is available.
PF_SSE4_1_INSTRUCTIONS_AVAILABLE 37		The SSE4_1 instruction set is available.
PF_SSE4_2_INSTRUCTIONS_AVAILABLE 38		The SSE4_2 instruction set is available.
PF_AVX_INSTRUCTIONS_AVAILABLE 39		The AVX instruction set is available.
PF_AVX2_INSTRUCTIONS_AVAILABLE 40		The AVX2 instruction set is available.
PF_AVX512F_INSTRUCTIONS_AVAILABLE 41		The AVX512F instruction set is available.
PF_VIRT_FIRMWARE_ENABLED 21		Virtualization is enabled in the firmware and made available by the operating system.
PF_XMMI_INSTRUCTIONS_AVAILABLE 6		The SSE instruction set is available.
PF_XMMI64_INSTRUCTIONS_AVAILABLE 10		The SSE2 instruction set is available. Windows 2000: This feature is not supported.
PF_XSAVE_ENABLED 17		The processor implements the XSAVE and XRSTOR instructions. Windows Server 2008, Windows Vista, Windows Server 2003 and Windows XP/2000: This feature is not supported until Windows 7 and Windows Server 2008 R2.
PF_ARM_V8_INSTRUCTIONS_AVAILABLE 29		This Arm processor implements the Arm v8 instructions set.
PF_ARM_V8_CRYPTO_INSTRUCTIONS_AVAILABLE 30		This Arm processor implements the Arm v8 extra cryptographic instructions (for example, AES, SHA1 and SHA2).
PF_ARM_V8_CRC32_INSTRUCTIONS_AVAILABLE 31		This Arm processor implements the Arm v8 extra CRC32 instructions.
PF_ARM_V81_ATOMIC_INSTRUCTIONS_AVAILABLE 34		This Arm processor implements the Arm v8.1 atomic instructions (for example, CAS, SWP).
PF_ARM_V82_DP_INSTRUCTIONS_AVAILABLE 43		This Arm processor implements the Arm v8.2 DP instructions (for example, SDOT, UDOT). This

	feature is optional in Arm v8.2 implementations and mandatory in Arm v8.4 implementations.
<code>PF_ARM_V83_JSCVT_INSTRUCTIONS_AVAILABLE</code> 44	This Arm processor implements the Arm v8.3 JSCVT instructions (for example, FJCVTZS).
<code>PF_ARM_V83_LRCPC_INSTRUCTIONS_AVAILABLE</code> 45	This Arm processor implements the Arm v8.3 LRCPC instructions (for example, LDAPR). Note that certain Arm v8.2 CPUs may optionally support the LRCPC instructions.

Return value

If the feature is supported, the return value is a nonzero value.

If the feature is not supported, the return value is zero.

If the HAL does not support detection of the feature, whether or not the hardware supports the feature, the return value is also zero.

Remarks

Support for `PF_SSSE3_INSTRUCTIONS_AVAILABLE` through `PF_AVX512F_INSTRUCTIONS_AVAILABLE` were added in the Windows SDK (19041) and are supported by Windows 10, Version 2004 (May 2020 Update) or later.

Support for `PF_ERMS_AVAILABLE`, `PF_ARM_V82_DP_INSTRUCTIONS_AVAILABLE`, and `PF_ARM_V83_JSCVT_INSTRUCTIONS_AVAILABLE` were added in the Windows SDK (20348) and are supported by Windows 11 and Windows Server 2022.

The define `PF_ARM_V83_LRCPC_INSTRUCTIONS_AVAILABLE` was added in the Windows SDK (22621) and is supported by Windows 11, Version 22H2.

Requirements

 Expand table

Requirement	Value
Minimum supported client	Windows 2000 Professional [desktop apps UWP apps]
Minimum supported server	Windows 2000 Server [desktop apps UWP apps]
Target Platform	Windows

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

See also

[Vertdll APIs available in VBS enclaves](#)

[System Information Functions](#)

MACHINE_ATTRIBUTES enumeration (processthreadsapi.h)

Article 11/01/2022

Specifies the ways in which an architecture of code can run on a host operating system. More than one bit may be set.

Syntax

C++

```
typedef enum _MACHINE_ATTRIBUTES {
    UserEnabled = 0x00000001,
    KernelEnabled = 0x00000002,
    Wow64Container = 0x00000004
} MACHINE_ATTRIBUTES;
```

Constants

`UserEnabled`

The specified architecture of code can run in user mode.

`KernelEnabled`

The specified architecture of code can run in kernel mode.

`Wow64Container`

The specified architecture of code runs by relying on WOW64's namespace [File System Redirector](#) and [Registry Redirector](#). This bit will be set, for example, on x86 code running on a host operating system that is x64 or ARM64. When the compatibility layer does not use WOW64 style filesystem and registry namespaces, like x64 on ARM64 which runs on the root namespace of the OS, this bit will be reset.

Requirements

Minimum supported client

Windows Build 22000

Minimum supported server

Windows Build 22000

Feedback

Was this page helpful?

 Yes

 No

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

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

OpenProcess function (processsthreadsapi.h)

Article 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](#)

OpenProcessToken function (processsthreadsapi.h)

Article 09/23/2022

The **OpenProcessToken** function opens the [access token](#) associated with a process.

Syntax

C++

```
BOOL OpenProcessToken(
    [in]  HANDLE ProcessHandle,
    [in]  DWORD  DesiredAccess,
    [out] PHANDLE TokenHandle
);
```

Parameters

[in] ProcessHandle

A handle to the process whose access token is opened. The process must have the **PROCESS_QUERY_LIMITED_INFORMATION** access permission. See [Process Security and Access Rights](#) for more info.

[in] DesiredAccess

Specifies an [access mask](#) that specifies the requested types of access to the access token. These requested access types are compared with the [discretionary access control list](#) (DACL) of the token to determine which accesses are granted or denied.

For a list of access rights for access tokens, see [Access Rights for Access-Token Objects](#).

[out] TokenHandle

A pointer to a handle that identifies the newly opened access token when the function returns.

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 get a handle to an elevated process from within a non-elevated process, both processes must be started from the same account.

If the process being checked was started by a different account, the checking process needs to have the SE_DEBUG_NAME privilege enabled. See [Privilege Constants \(Authorization\)](#) for more info.

To close the access token handle returned through the *TokenHandle* parameter, call [CloseHandle](#).

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	Advapi32.lib
DLL	Advapi32.dll

See also

[Access Control](#)

[Basic Access Control Functions](#)

[AccessCheck](#)

[AdjustTokenGroups](#)

[AdjustTokenPrivileges](#)

[CloseHandle](#)

[GetCurrentProcessToken](#)

[GetCurrentThreadEffectiveToken](#)

[GetCurrentThreadToken](#)

[GetTokenInformation](#)

[OpenThreadToken](#)

[SetTokenInformation](#)

OpenThread function (processthreadsapi.h)

Article02/22/2024

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](#)

OpenThreadToken function (processthreadsapi.h)

Article10/13/2021

The **OpenThreadToken** function opens the [access token](#) associated with a thread.

Syntax

C++

```
BOOL OpenThreadToken(
    [in]  HANDLE ThreadHandle,
    [in]  DWORD  DesiredAccess,
    [in]  BOOL   OpenAsSelf,
    [out] PHANDLE TokenHandle
);
```

Parameters

[in] **ThreadHandle**

A handle to the thread whose access token is opened.

[in] **DesiredAccess**

Specifies an [access mask](#) that specifies the requested types of access to the access token. These requested access types are reconciled against the token's [discretionary access control list](#) (DACL) to determine which accesses are granted or denied.

For a list of access rights for access tokens, see [Access Rights for Access-Token Objects](#).

[in] **OpenAsSelf**

TRUE if the access check is to be made against the process-level [security context](#).

FALSE if the access check is to be made against the current security context of the thread calling the **OpenThreadToken** function.

The *OpenAsSelf* parameter allows the caller of this function to open the access token of a specified thread when the caller is impersonating a token at **SecurityIdentification** level. Without this parameter, the calling thread cannot open the access token on the specified thread because it is impossible to open executive-level objects by using the **SecurityIdentification** impersonation level.

[out] *TokenHandle*

A pointer to a variable that receives the handle to the newly opened access token.

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 token has the anonymous impersonation level, the token will not be opened and [OpenThreadToken](#) sets ERROR_CANT_OPEN_ANONYMOUS as the error.

Remarks

Tokens with the anonymous impersonation level cannot be opened.

Close the access token handle returned through the *TokenHandle* parameter by calling [CloseHandle](#).

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	Advapi32.lib
DLL	Advapi32.dll

See also

[Access Control Overview](#)

[AccessCheck](#)

[AdjustTokenGroups](#)

[AdjustTokenPrivileges](#)

[Basic Access Control Functions](#)

[CloseHandle](#)

[GetCurrentThreadToken](#)

[GetTokenInformation](#)

[OpenProcessToken](#)

[SECURITY_IMPERSONATION_LEVEL](#)

[SetThreadToken](#)

[SetTokenInformation](#)

OVERRIDE_PREFETCH_PARAMETER structure (processthreadsapi.h)

Article03/08/2024

Provides additional control over App Launch Prefetch (ALPF) functionality.

Syntax

C++

```
typedef struct OVERRIDE_PREFETCH_PARAMETER {
    UINT32 Value;
} OVERRIDE_PREFETCH_PARAMETER;
```

Members

Value

A unique identifier for differentiating an application view or mode.

Remarks

App Launch Prefetch (ALPF) brings data and code pages into memory from disk before it's demanded. Prefetching monitors the data and code accessed during application startups and uses that information at the beginning of subsequent startups to read the code and data proactively in an efficient manner to improve performance.

If ALPF predicts incorrectly, the wrong pages may be fetched, slowing app launches. Applications with different "Views", such as Outlook Mail View or Calendar View, may cause this issue by needing different pages of memory depending on the View. To solve this, applications can pass a prefetch parameter to their launch through the command line, which will provide a unique identifier to differentiate between Views or other scenarios that cause the ALPF standard prediction to fail.

In some cases, however, prefetching doesn't always resolve a failure successfully. For example, failures can happen when different code paths in the same executable require different pages but those startups were launched with the same prefetch parameter. To resolve these types of situations, the OVERRIDE_PREFETCH_PARAMETER can be used by an app to override the system prefetch parameter (see [SetProcessInformation function](#)).

Examples

The following code example shows how to use the prefetch override APIs when an app launches with a command-line prefetch parameter of 2.

1. Assume this is the first launch of an app, so the app instance designates itself as the primary process.
2. This primary process queries the maximum allowable value of an **OVERRIDE_PREFETCH_PARAMETER**.
3. Once confirmed that the override value is less than this maximum (when the app launches with a prefetch parameter of 2), it is overridden with a value of 9 by a call to the [SetProcessInformation function](#) using a **ProcessInformation** value of **ProcessOverrideSubsequentPrefetchParameter**.
4. ALPF knows that an Override Prefetch Parameter has been set.
5. Another instance of GenericApp.exe is launched with a command-line prefetch parameter of 2. This instance will be transient as a primary process already exists.
6. Because an override from 2 to 9 has been set for this executable, ALPF will force this transient instance to launch in scenario 9 instead of 2.
7. ALPF now fetches appropriate pages for the primary process under scenario 2 and a separate set of pages for the other processes under scenario 9.
8. When the primary process of the app closes, the override will be removed, allowing the next launch of GenericApp.exe to become primary.

C++

```
int main (int argc, char *argv[]) {

    BOOL IsThisProcessPrimary;

    IsThisProcessPrimary = CheckIfProcessPrimary();

    if (!IsThisProcessPrimary) {
        // This process is transient; it does not call the Override Prefetch
        // Parameter API.

        PassTransientDataToPrimary(argc, argv);
        goto Exit;
    } else {
        // This process is primary; attempt to call Override Prefetch Parameter
        // before doing primary initialization.
        SetOverridePrefetchParameter(9);

        InitializeThisProcessAsPrimary(argc, argv);
        DisplayToUserAndWait();
    }

Exit:
```

```

        return 0;
    }

DWORD SetOverridePrefetchParameter (UINT32 OverrideParameter) {

    OVERRIDE_PREFETCH_PARAMETER ParamInfo;
    DWORD ErrorCode;
    BOOL Win32Success;

    ZeroMemory(&ParamInfo, sizeof(ParamInfo));

    // Get the maximum Override Prefetch Parameter from
    // GetProcessInformation.

    Win32Success = GetProcessInformation(GetCurrentProcess(),
        ProcessOverrideSubsequentPrefetchParameter,
        &ParamInfo,
        sizeof(ParamInfo));

    if (!Win32Success) {
        ErrorCode = GetLastError();
        goto Exit;
    }

    if (OverrideParameter <= ParamInfo.OverrideScenarioId) {
        ParamInfo.Value = OverrideParameter;
    } else {
        // The Override Prefetch Parameter requested isn't valid on this
        // system.
        // Continue to launch without setting an Override Prefetch
        // Parameter.
        ErrorCode = ERROR_INVALID_PARAMETER;
        goto Exit;
    }

    Win32Success = SetProcessInformation(GetCurrentProcess(),
        ProcessOverrideSubsequentPrefetchParameter,
        &ParamInfo,
        sizeof(ParamInfo));

    if (!Win32Success) {
        ErrorCode = GetLastError();
        goto Exit;
    }

Exit:
    ErrorCode = ERROR_SUCCESS;
    return ErrorCode;
}

```

Requirements

Requirement	Value
Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]
Header	processthreadsapi.h

See also

[PROCESS_INFORMATION_CLASS enumeration](#), [GetProcessInformation function](#)

Feedback

Was this page helpful?



[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

PROCESS_INFORMATION structure (processthreadsapi.h)

Article 02/22/2024

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	processsthreadsapi.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_INFORMATION_CLASS enumeration (processthreadsapi.h)

Article01/31/2024

Indicates a specific class of process information. Values from this enumeration are passed into the [GetProcessInformation](#) and [SetProcessInformation](#) functions to specify the type of process information passed in the void pointer argument of the function call.

Syntax

C++

```
typedef enum _PROCESS_INFORMATION_CLASS {
    ProcessMemoryPriority,
    ProcessMemoryExhaustionInfo,
    ProcessAppMemoryInfo,
    ProcessInPrivateInfo,
    ProcessPowerThrottling,
    ProcessReservedValue1,
    ProcessTelemetryCoverageInfo,
    ProcessProtectionLevelInfo,
    ProcessLeapSecondInfo,
    ProcessMachineTypeInfo,
    ProcessOverrideSubsequentPrefetchParameter,
    ProcessMaxOverridePrefetchParameter,
    ProcessInformationClassMax
} PROCESS_INFORMATION_CLASS;
```

Constants

[+] Expand table

ProcessMemoryPriority

The process information is represented by a [MEMORY_PRIORITY_INFORMATION](#) structure. Allows applications 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.

ProcessMemoryExhaustionInfo

The process information is represented by a [PROCESS_MEMORY_EXHAUSTION_INFO](#) structure. Allows applications to configure a process to terminate if an allocation fails to commit memory.

`ProcessAppMemoryInfo`

The process information is represented by a [APP_MEMORY_INFORMATION](#) structure. Allows applications to query the commit usage and the additional commit available to this process. Does not allow the caller to actually get a commit limit.

`ProcessInPrivateInfo`

If a process is set to `ProcessInPrivate` mode, and a trace session has set the [EVENT_ENABLE_PROPERTY_EXCLUDE_INPRIVATE](#) flag, then the trace session will drop all events from that process.

`ProcessPowerThrottling`

The process information is represented by a [PROCESS_POWER_THROTTLING_STATE](#) structure. Allows applications to configure how the system should throttle the target process's activity when managing power.

`ProcessReservedValue1`

Reserved.

`ProcessTelemetryCoverageInfo`

Reserved.

`ProcessProtectionLevelInfo`

The process information is represented by a [PROCESS_PROTECTION_LEVEL_INFORMATION](#) structure.

`ProcessLeapSecondInfo`

The process information is represented by a [PROCESS_LEAP_SECOND_INFO](#) structure.

`ProcessMachineTypeInfo`

The process is represented by a [PROCESS_MACHINE_INFORMATION](#) structure.

`ProcessOverrideSubsequentPrefetchParameter`

Can be used in a call to the [SetProcessInformation function](#) to set an [OVERRIDE_PREFETCH_PARAMETER structure](#) for the application that called it. The prefetch parameter is used to differentiate different file access patterns for the same process name.

`ProcessMaxOverridePrefetchParameter`

Can be used in a call to the [GetProcessInformation function](#) to query the maximum allowable value (inclusive) for an [OVERRIDE_PREFETCH_PARAMETER structure](#). (The prefetch parameter is used to differentiate different file access patterns for the same process name.)

`ProcessInformationClassMax`

The maximum value for this enumeration. This value may change in a future version.

Requirements

Minimum supported client	Windows Build 22000
Minimum supported server	Windows Build 22000
Header	processthreadsapi.h

See also

[GetProcessInformation function](#), [SetProcessInformation function](#),
[APP_MEMORY_INFORMATION structure](#), [PROCESS_MACHINE_INFORMATION structure](#),
[PROCESS_MEMORY_EXHAUSTION_INFO structure](#)

Feedback

Was this page helpful?

 Yes

 No

PROCESS_LEAP_SECOND_INFO structure (processthreadsapi.h)

Article11/01/2022

Specifies how the system handles positive leap seconds.

Syntax

C++

```
typedef struct _PROCESS_LEAP_SECOND_INFO {
    ULONG Flags;
    ULONG Reserved;
} PROCESS_LEAP_SECOND_INFO, *PPROCESS_LEAP_SECOND_INFO;
```

Members

Flags

Currently, the only valid flag is

`PROCESS_LEAP_SECOND_INFO_FLAG_ENABLE_SIXTY_SECOND`. That flag is described below.

Value	Meaning
<code>PROCESS_LEAP_SECOND_INFO_FLAG_ENABLE_SIXTY_SECOND</code>	This value changes the way positive leap seconds are handled by system. Specifically, it changes how the seconds field during a positive leap second is handled by the system. If this value is used, then the positive leap second will be shown (For example: 23:59:59 -> 23:59:60 -> 00:00:00. If this value is not used, then "sixty seconds" is disabled, and the 59th second preceding a positive leap second will be shown for 2 seconds with the milliseconds value ticking twice as slow. So 23:59:59 -> 23:59:59.500 ->

00:00:00, which takes 2 seconds in wall clock time. Disabling "sixty second" can help with legacy apps that do not support seeing the seconds value as 60 during the positive leap second. Such apps may crash or misbehave. Therefore, in these cases, we display the 59th second for twice as long during the positive leap second. Note that this setting is per-process, and does not persist if the process is restarted. Developers should test their app for compatibility with seeing the system return "60", and add a call to their app startup routines to either enable or disable "sixty seconds". "Sixty seconds" is disabled by default for each process. Obviously, this setting has no effect if leap seconds are disabled system-wide, because then the system will never even encounter a leap second.

Reserved

Reserved for future use

Requirements

Header	processthreadsapi.h
--------	---------------------

Feedback

Was this page helpful?



Get help at Microsoft Q&A

PROCESS_MACHINE_INFORMATION structure (processthreadsapi.h)

Article02/22/2024

Specifies the architecture of a process and if that architecture of code can run in user mode, kernel mode, and/or under WoW64 on the host operating system.

Syntax

C++

```
typedef struct _PROCESS_MACHINE_INFORMATION {
    USHORT          ProcessMachine;
    USHORT          Res0;
    MACHINE_ATTRIBUTES MachineAttributes;
} PROCESS_MACHINE_INFORMATION;
```

Members

ProcessMachine

An IMAGE_FILE_MACHINE_* value indicating the architecture of the associated process. See the list of architecture values in [Image File Machine Constants](#).

Res0

Reserved.

MachineAttributes

A value from the [MACHINE_ATTRIBUTES](#) enumeration indicating if the process's architecture can run in user mode, kernel mode, and/or under WOW64 on the host operating system.

Requirements

[+] [Expand table](#)

Requirement	Value
Minimum supported client	Windows Build 22000
Minimum supported server	Windows Build 22000
Header	processthreadsapi.h

See also

[PROCESS_INFORMATION_CLASS](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

PROCESS_MEMORY_EXHAUSTION_INFO structure (processthreadsapi.h)

Article 02/22/2024

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_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_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

PROCESS_PROTECTION_LEVEL_INFORMATION structure (processsthreadsapi.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

ProcessIdToSessionId function (processsthreadsapi.h)

Article02/22/2024

Retrieves the Remote Desktop Services session associated with a specified process.

Syntax

C++

```
BOOL ProcessIdToSessionId(
    [in]  DWORD dwProcessId,
    [out] DWORD *pSessionId
);
```

Parameters

[in] dwProcessId

Specifies a process identifier. Use the [GetCurrentProcessId](#) function to retrieve the process identifier for the current process.

[out] pSessionId

Pointer to a variable that receives the identifier of the Remote Desktop Services session under which the specified process is running. To retrieve the identifier of the session currently attached to the console, use the [WTSGetActiveConsoleSessionId](#) function.

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

Callers must hold the **PROCESS_QUERY_INFORMATION** access right for the specified process. For more information, see [Process Security and Access Rights](#).

Requirements

 [Expand table](#)

Requirement	Value
Minimum supported client	Windows Vista
Minimum supported server	Windows Server 2008
Target Platform	Windows
Header	processsthreadsapi.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

See also

[OSVERSIONINFOEX](#)

[WTSQuerySessionInformation](#)

QueryProcessAffinityUpdateMode function (processthreadsapi.h)

Article02/22/2024

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](#)

QueryProtectedPolicy function (processthreadsapi.h)

Article02/22/2024

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

QUEUE_USER_AP_C_FLAGS enumeration (processsthreadsapi.h)

Article 02/22/2024

Specifies the modifier flags for user-mode asynchronous procedure call (APC) objects.

Syntax

C++

```
typedef enum _QUEUE_USER_AP_C_FLAGS {
    QUEUE_USER_AP_C_FLAGS_NONE,
    QUEUE_USER_AP_C_FLAGS_SPECIAL_USER_AP_C,
    QUEUE_USER_AP_C_CALLBACK_DATA_CONTEXT
} QUEUE_USER_AP_C_FLAGS;
```

Constants

[+] Expand table

QUEUE_USER_AP_C_FLAGS_NONE

No flags are passed. Behavior is identical to [QueueUserAPC function](#).

QUEUE_USER_AP_C_FLAGS_SPECIAL_USER_AP_C

Queue a special user-mode APC instead of a regular user-mode APC.

QUEUE_USER_AP_C_CALLBACK_DATA_CONTEXT

Receive the processor context that was interrupted when the thread was directed to call the APC function.

Remarks

The *Parameter* argument of the [PAPCFUNC callback function](#) is modified to point to an APC_CALLBACK_DATA structure (see below), which contains the original *Parameter* argument, a pointer to the interrupted processor context, and reserved fields.

C++

```
typedef struct _APC_CALLBACK_DATA {
    ULONG_PTR Parameter;
    PCONTEXT ContextRecord;
    ULONG_PTR Reserved0;
    ULONG_PTR Reserved1;
} APC_CALLBACK_DATA, *PAPC_CALLBACK_DATA;
```

Requirements

[+] Expand table

Requirement	Value
Minimum supported client	Windows Build 22000
Minimum supported server	Windows Build 22000
Header	processthreadsapi.h (include Windows.h)

See also

[QueueUserAPC2](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

QueueUserAPC function (processsthreadsapi.h)

Article11/01/2022

Adds a user-mode [asynchronous procedure call](#) (APC) object to the APC queue of the specified thread.

Syntax

C++

```
DWORD QueueUserAPC(
    [in] PAPCFUNC  pfnAPC,
    [in] HANDLE     hThread,
    [in] ULONG_PTR  dwData
);
```

Parameters

[in] *pfnAPC*

A pointer to the application-supplied APC function to be called when the specified thread performs an alertable wait operation. For more information, see [PAPCFUNC callback function](#).

[in] *hThread*

A handle to the thread. The handle must have the **THREAD_SET_CONTEXT** access right. For more information, see [Synchronization Object Security and Access Rights](#).

[in] *dwData*

A single value that is passed to the APC function pointed to by the *pfnAPC* parameter.

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 are no error values defined for this function that can be retrieved by calling [GetLastError](#).

Remarks

See [QueueUserAPC2 function](#) for information on special user-mode APCs.

The APC support provided in the operating system allows an application to queue an APC object to a thread. To ensure successful execution of functions used by the APC, APCs should be queued only to threads in the caller's process.

Note

Queuing APCs to threads outside the caller's process is not recommended for a number of reasons. DLL rebasing can cause the addresses of functions used by the APC to be incorrect when the functions are executed outside the caller's process. Similarly, if a 64-bit process queues an APC to a 32-bit process or vice versa, addresses will be incorrect and the application will crash. Other factors can prevent successful function execution, even if the address is known.

Each thread has its own APC queue. The queuing of an APC is a request for the thread to call the APC function. The operating system issues a software interrupt to direct the thread to call the APC function.

When a user-mode APC is queued, the thread is not directed to call the APC function unless it is in an alertable state. After the thread is in an alertable state, the thread handles all pending APCs in first in, first out (FIFO) order, and the wait operation returns `WAIT_IO_COMPLETION`. A thread enters an alertable state by using [SleepEx function](#), [SignalObjectAndWait function](#), [WaitForSingleObjectEx function](#), [WaitForMultipleObjectsEx function](#), or [MsgWaitForMultipleObjectsEx function](#).

If an application queues an APC before the thread begins running, the thread begins by calling the APC function. After the thread calls an APC function, it calls the APC functions for all APCs in its APC queue.

It is possible to sleep or wait for an object within the APC. If you perform an alertable wait inside an APC, it will recursively dispatch the APCs. This can cause a stack overflow.

When the thread is terminated using the [ExitThread function](#) or [TerminateThread function](#) function, the APCs in its APC queue are lost. The APC functions are not called.

When the thread is in the process of being terminated, calling `QueueUserAPC` to add to the thread's APC queue will fail with (31) `ERROR_GEN_FAILURE`.

Note that the [ReadFileEx function](#), [SetWaitableTimer function](#), and [WriteFileEx function](#) functions are implemented using an APC as the completion notification callback mechanism.

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

- [PAPCFUNC callback function](#)
- [Asynchronous Procedure Calls](#)
- [Synchronization Functions](#)

QueueUserAPC2 function (processsthreadsapi.h)

Article03/17/2023

Adds a user-mode [asynchronous procedure call](#) (APC) object to the APC queue of the specified thread.

Syntax

C++

```
BOOL QueueUserAPC2(
    PAPCFUNC          ApcRoutine,
    HANDLE            Thread,
    ULONG_PTR         Data,
    QUEUE_USER_AP_C_FLAGS Flags
);
```

Parameters

ApcRoutine

A pointer to the application-supplied APC function to be called when the specified thread performs an alertable wait operation. For more information, see [APCProc](#).

For special user-mode APCs, an alertable wait is not required. See [Remarks](#) for more information about special user-mode APCs.

Thread

A handle to the thread. The handle must have **THREAD_SET_CONTEXT** access permission. For more information, see [Synchronization Object Security and Access Rights](#).

Data

A single value that is passed to the APC function pointed to by the *ApcRoutine* parameter.

Flags

A value from [QUEUE_USER_APCT FLAGS](#) enumeration that modifies the behavior of the user-mode APC.

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

Regular user-mode APCs are only executed if the target thread is in an alertable state. See [QueueUserAPC function](#) for additional remarks on regular user-mode APCs.

Special user-mode APCs always execute, even if the target thread is not in an alertable state. For example, if the target thread is currently executing user-mode code, or if the target thread is currently performing an alertable wait, the target thread will be interrupted immediately for APC execution. If the target thread is executing a system call, or performing a non-alertable wait, the APC will be executed after the system call or non-alertable wait finishes (the wait is not interrupted).

Since the execution of the special user-mode APC is not synchronized with the target thread, particular care must be taken (beyond the normal requirements for multithreading and synchronization). For example, when acquiring any locks, the interrupted target thread may already own the lock or be in the process of acquiring or releasing the lock. In addition, because there are no facilities to block a thread from receiving special user-mode APCs, a special user-mode APC can be executed on a target thread that is already executing a special user-mode APC.

Currently, special user-mode APCs are only supported on native architectures, and not when running under WoW.

Requirements

Minimum supported client	Windows 11 (Build 22000)
Minimum supported server	Windows Server 2022 (Build 20348)
Target Platform	Windows

Header	processthreadsapi.h (include Windows.h)
Library	Kernel32.lib
DLL	Kernel32.dll

Feedback

Was this page helpful?

 Yes

 No

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

ResumeThread function (processsthreadsapi.h)

Article 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](#)

SetPriorityClass function (processsthreadsapi.h)

Article11/01/2022

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

	<p>system. Use extreme care when using the high-priority class, because a high-priority class application can use nearly all available CPU time.</p>
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	<p>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.</p> <p>This value can be specified only if <i>hProcess</i> is a handle to the current process. The function fails if the process is already in background processing mode.</p> <p>Windows Server 2003 and Windows XP: This value is not supported.</p>
PROCESS_MODE_BACKGROUND_END 0x00200000	<p>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.</p> <p>This value can be specified only if <i>hProcess</i> is a handle to the current process. The function fails if the process is not in background processing mode.</p> <p>Windows Server 2003 and Windows XP: This value is not supported.</p>
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](#)

SetProcessAffinityUpdateMode function (processthreadsapi.h)

Article 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.
<code>PROCESS_AFFINITY_ENABLE_AUTO_UPDATE</code> 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](#)

SetProcessDefaultCpuSetMasks function (processthreadsapi.h)

Article02/22/2024

Sets the default CPU Sets assignment for threads in the specified process.

Syntax

C++

```
BOOL SetProcessDefaultCpuSetMasks(
    HANDLE      Process,
    PGROUP_AFFINITY CpuSetMasks,
    USHORT      CpuSetMaskCount
);
```

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.

CpuSetMasks

Specifies an optional buffer of [GROUP_AFFINITY](#) structures representing the CPU Sets to set as the process default CPU set. If this is NULL, the [SetProcessDefaultCpuSetMasks](#) function clears out any assignment.

CpuSetMaskCount

Specifies the size of the *CpuSetMasks* array, in elements. If the buffer is NULL, this value must be zero.

Return value

This function cannot fail when passed valid parameters.

Remarks

Threads belonging to this process which don't have CPU Sets explicitly set using [SetThreadSelectedCpuSetMasks](#) or [SetThreadSelectedCpuSets](#), will inherit the sets specified by [SetProcessDefaultCpuSetMasks](#) automatically.

This function is analogous to [SetProcessDefaultCpuSets](#), except that it uses group affinities as opposed to CPU Set IDs to represent a list of CPU sets. This means that the resulting process default CPU Set assignment is the set of all CPU sets with a home processor in the provided list of group affinities.

Requirements

[Expand table](#)

Requirement	Value
Minimum supported client	Windows 11
Minimum supported server	Windows Server 2022
Header	processthreadsapi.h
DLL	kernel32.dll

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

SetProcessDefaultCpuSets function (processthreadsapi.h)

Article 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

SetProcessDynamicEHContinuationTargets function (processsthreadsapi.h)

Article 10/20/2021

Sets dynamic exception handling continuation targets for the specified process.

Syntax

C++

```
BOOL SetProcessDynamicEHContinuationTargets(
    HANDLE Process,
    USHORT NumberOfTargets,
    PPROCESS_DYNAMIC_EH_CONTINUATION_TARGET Targets
);
```

Parameters

Process

A handle to the process. This handle must have the **PROCESS_SET_INFORMATION** access right. For more information, see [Process Security and Access Rights](#).

NumberOfTargets

Supplies the number of dynamic exception handling continuation targets to set.

Targets

A pointer to an array of dynamic exception handling continuation targets. For more information on this structure, see [PROCESS_DYNAMIC_EH_CONTINUATION_TARGET](#).

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 even if the function fails, a portion of the supplied continuation targets may have been successfully processed. The caller needs to check the flags in

each individual continuation target specified via *Targets* to determine if it was successfully processed.

Remarks

If user-mode Hardware-enforced Stack Protection is enabled for a process, when calling APIs that modify the execution context of a thread such as [RtlRestoreContext](#) and [SetThreadContext](#), validation is performed on the Instruction Pointer specified in the new execution context. *RtlRestoreContext* is used during [Structured Exception Handling](#) (SEH) exception unwinding to unwind to the target frame that contains the `_except` block and to start executing code at the continuation target. Therefore, the operating system needs to know the instruction addresses of all the valid continuation targets in order to allow the unwind operation via *RtlRestoreContext*. For compiled binaries, the list of continuation targets is generated by the linker and stored in the binary image. For dynamic code, the continuation targets need to be specified using [SetProcessDynamicEHContinuationTargets](#).

Requirements

Minimum supported client	Windows 10 Build 20348
Minimum supported server	Windows 10 Build 20348
Header	processthreadsapi.h

Feedback

Was this page helpful?



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

SetProcessDynamicEnforcedCetCompatibleRanges function (processthreadsapi.h)

Article02/22/2024

ⓘ Note

This API was added to the 19041 SDK in an update released in November 2020.

Sets dynamic enforced CETCOMPAT ranges for the specified process.

Syntax

C++

```
BOOL SetProcessDynamicEnforcedCetCompatibleRanges(
    HANDLE             Process,
    USHORT            NumberOfRanges,
    PPROCESS_DYNAMIC_ENFORCED_ADDRESS_RANGE Ranges
);
```

Parameters

Process

A handle to the process. This handle must have the **PROCESS_SET_INFORMATION** access right. For more information, see [Process Security and Access Rights](#).

NumberOfRanges

Supplies the number of dynamic enforced CETCOMPAT ranges to set.

Ranges

A pointer to an array of dynamic enforced CETCOMPAT ranges. For more information on this structure, see [PROCESS_DYNAMIC_ENFORCED_ADDRESS_RANGE](#).

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 even if the function fails, a portion of the supplied CETCOMPAT ranges may have been successfully processed. The caller needs to check the flags in each individual CETCOMPAT range specified via *Ranges* to determine if it was successfully processed.

Remarks

User-mode Hardware-enforced Stack Protection (HSP) is a security feature where the CPU verifies function return addresses at runtime by employing a shadow stack mechanism, if supported by the hardware. In HSP compatibility mode, only shadow stack violations occurring in modules that are considered compatible with shadow stacks (CETCOMPAT) are fatal. For a module to be considered CETCOMPAT, it needs to be either compiled with [CETCOMPAT](#) for binaries, or marked using [SetProcessDynamicEnforcedCetCompatibleRanges](#) for dynamic code. In HSP strict mode, all shadow stack violations are fatal.

Requirements

[+] Expand table

Requirement	Value
Minimum supported client	Windows 10, version 2004 (10.0; Build 19041.662)
Minimum supported server	Windows Server, version 2004 (10.0; Build 19041.662)
Header	processthreadsapi.h

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

SetProcessInformation function (processsthreadsapi.h)

Article10/31/2024

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 [OVERSIZE_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(PERSON_POWER_THROTTLING_STATE)`.

If the *ProcessInformationClass* parameter is **ProcessLeapSecondInfo**, this parameter must be `sizeof(PERSON_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)

Article11/01/2022

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 <i>lpBuffer</i> 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 <i>lpBuffer</i> 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 <i>lpBuffer</i> parameter points to a PROCESS_MITIGATION_DYNAMIC_CODE_POLICY structure that specifies the dynamic code policy flags.

Value	Meaning
ProcessStrictHandleCheckPolicy	<p>The process will receive a fatal error if it manipulates a handle that is not valid.</p> <p>The <i>lpBuffer</i> parameter points to a PROCESS_MITIGATION_STRICT_HANDLE_CHECK_POLICY structure that specifies the handle check policy flags.</p>
ProcessSystemCallDisablePolicy	<p>Disables the ability to use NTUser/GDI functions at the lowest layer.</p> <p>The <i>lpBuffer</i> parameter points to a PROCESS_MITIGATION_SYSTEM_CALL_DISABLE_POLICY structure that specifies the system call disable policy flags.</p>
ProcessMitigationOptionsMask	<p>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.</p> <p>The <i>lpBuffer</i> 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.</p>
ProcessExtensionPointDisablePolicy	<p>The <i>lpBuffer</i> parameter points to a PROCESS_MITIGATION_EXTENSION_POINT_DISABLE_POLICY structure that specifies the extension point disable policy flags.</p>
ProcessControlFlowGuardPolicy	<p>The Control Flow Guard (CFG) policy of the process.</p> <p>The <i>lpBuffer</i> parameter points to a PROCESS_MITIGATION_CONTROL_FLOW_GUARD_POLICY structure that specifies the CFG policy flags.</p>
ProcessSignaturePolicy	<p>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).</p> <p>The <i>lpBuffer</i> parameter points to a PROCESS_MITIGATION_BINARY_SIGNATURE_POLICY structure that specifies the signature policy flags.</p>
ProcessFontDisablePolicy	<p>The policy regarding font loading for the process. When turned on, the process cannot load non-system fonts.</p> <p>The <i>lpBuffer</i> parameter points to a PROCESS_MITIGATION_FONT_DISABLE_POLICY structure that specifies the policy flags for font loading.</p>
ProcessImageLoadPolicy	<p>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 as remote devices or files that have the low mandatory label.</p>

Value	Meaning
	The <i>lpBuffer</i> 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 <i>lpBuffer</i> 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 <i>lpBuffer</i> 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 <i>lpBuffer</i> 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)

Article 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](#)

SetProcessShutdownParameters function (processsthreadsapi.h)

Article11/01/2022

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 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](#)

SetProtectedPolicy function (processthreadsapi.h)

Article02/22/2024

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

SetThreadContext function (processsthreadsapi.h)

Article 08/23/2022

Sets the context for the specified thread.

A 64-bit application can set the context of a WOW64 thread using the [Wow64SetThreadContext](#) function.

Syntax

C++

```
BOOL SetThreadContext(
    [in] HANDLE      hThread,
    [in] const CONTEXT *lpContext
);
```

Parameters

[in] hThread

A handle to the thread whose context is to be set. The handle must have the **THREAD_SET_CONTEXT** access right to the thread. For more information, see [Thread Security and Access Rights](#).

[in] lpContext

A pointer to a **CONTEXT** structure that contains the context to be set in the specified thread. The value of the **ContextFlags** member of this structure specifies which portions of a thread's context to set. Some values in the **CONTEXT** structure that cannot be specified are silently set to the correct value. This includes bits in the CPU status register that specify the privileged processor mode, global enabling bits in the debugging register, and other states that must be controlled by the operating system.

Return value

If the context was set, the return value is nonzero.

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

Remarks

The function sets the thread context based on the value of the `ContextFlags` member of the context structure. The thread identified by the `hThread` parameter is typically being debugged, but the function can also operate even when the thread is not being debugged.

Do not try to set the context for a running thread; the results are unpredictable. Use the [SuspendThread](#) function to suspend the thread before calling [SetThreadContext](#).

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

[CONTEXT](#)

[Debugging Functions](#)

[GetThreadContext](#)

[GetXStateFeaturesMask](#)

[SetXStateFeaturesMask](#)

[SuspendThread](#)

SetThreadDescription function (processsthreadsapi.h)

Article 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](#)

SetThreadIdealProcessor function (processsthreadsapi.h)

Article02/22/2024

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 (processsthreadsapi.h)

Article11/01/2022

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 (processsthreadsapi.h)

Article11/01/2022

Sets information for the specified thread.

Syntax

C++

```
BOOL SetThreadInformation(
    [in] HANDLE             hThread,
    [in] THREAD_INFORMATION_CLASS ThreadInformationClass,
    [in] 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

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

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

SetThreadPriority function (processsthreadsapi.h)

Article 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 <i>hThread</i> 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 <i>hThread</i> 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**, **BELLOW_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)

Article 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](#)

SetThreadSelectedCpuSetMasks function (processthreadsapi.h)

Article 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 SetThreadSelectedCpuSetMasks(
    HANDLE Thread,
    PGROUP_AFFINITY CpuSetMasks,
    USHORT CpuSetMaskCount
);
```

Parameters

Thread

Specifies the thread on which to set the CPU Set assignment.

[PROCESS_SET_LIMITED_INFORMATION](#) access right. The value returned by [GetCurrentProcess](#) can also be specified here.

CpuSetMasks

Specifies an optional buffer of [GROUP_AFFINITY](#) structures representing the CPU Sets to set as the thread selected CPU set. If this is NULL, the **SetThreadSelectedCpuSetMasks** function clears out any assignment, reverting to process default assignment if one is set.

CpuSetMaskCount

Specifies the number of [GROUP_AFFINITY](#) structures in the list passed in the GroupCpuSets argument. If the buffer is NULL, this value must be zero.

Return value

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero and extended error information can be retrieved by calling [GetLastError](#).

Remarks

This function is analogous to [SetThreadSelectedCpuSets](#), except that it uses group affinities as opposed to CPU Set IDs to represent a list of CPU sets. This means that the resulting thread selected CPU Set assignment is the set of all CPU sets with a home processor in the provided list of group affinities.

Requirements

 [Expand table](#)

Requirement	Value
Minimum supported client	Windows 11
Minimum supported server	Windows Server 2022
Header	processthreadsapi.h
DLL	kernel32.dll

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

SetThreadSelectedCpuSets function (processthreadsapi.h)

Article02/22/2024

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 (processsthreadsapi.h)

Article02/06/2024

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](#)

SetThreadToken function (processsthreadsapi.h)

Article02/22/2024

The **SetThreadToken** function assigns an [impersonation token](#) to a thread. The function can also cause a thread to stop using an impersonation token.

Syntax

C++

```
BOOL SetThreadToken(
    [in, optional] PHANDLE Thread,
    [in, optional] HANDLE Token
);
```

Parameters

[in, optional] Thread

A pointer to a handle to the thread to which the function assigns the impersonation token.

If *Thread* is **NULL**, the function assigns the impersonation token to the calling thread.

[in, optional] Token

A handle to the impersonation token to assign to the thread. This handle must have been opened with **TOKEN_IMPERSONATE** access rights. For more information, see [Access Rights for Access-Token Objects](#).

If *Token* is **NULL**, the function causes the thread to stop using an impersonation token.

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 using the **SetThreadToken** function to impersonate, you must have the impersonate privileges and make sure that the **SetThreadToken** function succeeds before calling the [RevertToSelf](#) 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)
Library	Advapi32.lib
DLL	Advapi32.dll

See also

[Access Control Overview](#)

[Basic Access Control Functions](#)

[OpenThreadToken](#)

STARTUPINFOA structure (processsthreadsapi.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).

	<p>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.</p>
	<p>The system turns the feedback cursor off after the first call to GetMessage, regardless of whether the process is drawing.</p>
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	<p>Indicates that any windows created by the process cannot be pinned on the taskbar.</p> <p>This flag must be combined with STARTF_TITLEISAPPID.</p>
STARTF_RUNFULLSCREEN 0x00000020	<p>Indicates that the process should be run in full-screen mode, rather than in windowed mode.</p> <p>This flag is only valid for console applications running on an x86 computer.</p>
STARTF_TITLEISAPPID 0x00001000	<p>The IpTitle 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.</p> <p>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.</p> <p>This flag cannot be used with STARTF_TITLEISLINKNAME.</p>
STARTF_TITLEISLINKNAME 0x00000800	<p>The IpTitle 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.</p> <p>This flag cannot be used with STARTF_TITLEISAPPID.</p>
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.
STARTFUSESIZE 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 <i>bInheritHandles</i> 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;
    }
    else
    {
        StartProcess( argv[0], argv );
    }
}
```

```

}

// 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](#)

STARTUPINFO structure (processthreadsapi.h)

Article 11/01/2022

Specifies the window station, desktop, standard handles, and appearance of the main window for a process at creation time.

Syntax

C++

```
typedef struct _STARTUPINFO {
    DWORD    cb;
    LPWSTR   lpReserved;
    LPWSTR   lpDesktop;
    LPWSTR   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;
} STARTUPINFO, *LPSTARTUPINFO;
```

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.

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).

	<p>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.</p>
	<p>The system turns the feedback cursor off after the first call to GetMessage, regardless of whether the process is drawing.</p>
STARTF_FORCEOFFFEEDBACK 0x00000080	<p>Indicates that the feedback cursor is forced off while the process is starting. The Normal Select cursor is displayed.</p>
STARTF_PREVENTPINNING 0x00002000	<p>Indicates that any windows created by the process cannot be pinned on the taskbar. This flag must be combined with STARTF_TITLEISAPPID.</p>
STARTF_RUNFULLSCREEN 0x00000020	<p>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.</p>
STARTF_TITLEISAPPID 0x00001000	<p>The IpTitle 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.</p> <p>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.</p> <p>This flag cannot be used with STARTF_TITLEISLINKNAME.</p>
STARTF_TITLEISLINKNAME 0x00000800	<p>The IpTitle 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.</p> <p>This flag cannot be used with STARTF_TITLEISAPPID.</p>
STARTF_UNTRUSTEDSOURCE 0x00008000	<p>The command line came from an untrusted source. For more information, see Remarks.</p>
STARTF_USECOUNTCHARS	<p>The dwXCountChars and dwYCountChars members</p>

0x00000008	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	<p>The hStdInput, hStdOutput, and hStdError members contain additional information.</p> <p>If this flag is specified when calling one of the process creation functions, the handles must be inheritable and the function's <i>bInheritHandles</i> parameter must be set to TRUE. For more information, see Handle Inheritance.</p> <p>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.</p> <p>Handles must be closed with CloseHandle when they are no longer needed.</p> <p>This flag cannot be used with STARTF_USEHOTKEY.</p>

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. Applications that call [CreateProcess](#) are encouraged to set this flag when launching a program with untrusted command line arguments (e.g. 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 **StartUpInfoW**.

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 `processsthreadsapi.h` header defines `STARTUPINFO` 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

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](#)

[GetStartupInfo](#)

Feedback

Was this page helpful?

 Yes

 No

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

SuspendThread function (processsthreadsapi.h)

Article02/22/2024

Suspends the specified thread.

A 64-bit application can suspend a WOW64 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

SwitchToThread function (processsthreadsapi.h)

Article02/22/2024

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](#)

TerminateProcess function (processsthreadsapi.h)

Article02/06/2024

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 (processsthreadsapi.h)

Article02/22/2024

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]
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

[CreateProcess](#)

[CreateThread](#)

[ExitThread](#)

[GetExitCodeThread](#)

[OpenThread](#)

[Process and Thread Functions](#)

[Terminating a Thread](#)

[Threads](#)

THREAD_INFORMATION_CLASS enumeration (processthreadsapi.h)

Article02/22/2024

Specifies the collection of supported thread types.

Syntax

C++

```
typedef enum _THREAD_INFORMATION_CLASS {
    ThreadMemoryPriority,
    ThreadAbsoluteCpuPriority,
    ThreadDynamicCodePolicy,
    ThreadPowerThrottling,
    ThreadInformationClassMax
} THREAD_INFORMATION_CLASS;
```

Constants

[] Expand table

ThreadMemoryPriority Lower the memory priority of threads that perform background operations or access files and data that are not expected to be accessed frequently.
ThreadAbsoluteCpuPriority CPU priority.
ThreadDynamicCodePolicy Generate dynamic code or modify executable code.
ThreadPowerThrottling Throttle the target process activity for power management.
ThreadInformationClassMax

Requirements

Requirement	Value
Minimum supported client	Windows Build 22000
Minimum supported server	Windows Build 22000
Header	processthreadsapi.h (include Windows.h)

See also

[UnmapViewOfFile2 function](#), [UnmapViewOfFileEx function](#), [GetThreadInformation function](#), [SetThreadInformation function](#),
[PROCESS_MITIGATION_DYNAMIC_CODE_POLICY structure](#),

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

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

TlsAlloc function (processsthreadsapi.h)

Article 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 (processsthreadsapi.h)

Article02/06/2024

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](#)

Vert.dll APIs available in VBS enclaves

TlsGetValue function (processsthreadsapi.h)

Article 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](#)

TlsGetValue2 function (processthreadsapi.h)

Article 08/14/2024

ⓘ Important

Some information relates to a prerelease product which may be substantially modified before it's commercially released. Microsoft makes no warranties, express or implied, with respect to the information provided here.

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 TlsGetValue2(
    [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.

Remarks

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 [TlsGetValue2](#) to retrieve the stored value.

[TlsGetValue2](#) 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 [TlsGetValue2](#).

This function is identical to [TlsGetValue](#) except that it doesn't set the thread's last error. Applications calling this function should avoid using 0 as a valid value, because [GetLastError](#) cannot be called to check if the function failed.

Examples

See [Using Thread Local Storage](#) or [Using Thread Local Storage in a Dynamic-Link Library](#).

Requirements

[+] [Expand table](#)

Requirement	Value
Minimum supported client	Windows 11, version 24H2
Target Platform	Windows
Header	processthreadsapi.h
Library	Kernel32.lib
DLL	Kernel32.dll
API set	api-ms-win-core-processthreads-l1-1-8

See also

[Process and Thread Functions](#)

[Thread Local Storage](#)

[TlsAlloc](#)

[TlsFree](#)

[TlsSetValue](#)

[Vertdll APIs available in VBS enclaves](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

TlsSetValue function (processsthreadsapi.h)

Article02/22/2024

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](#)

UpdateProcThreadAttribute function (processsthreadsapi.h)

Article 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
<code>PROC_THREAD_ATTRIBUTE_GROUP_AFFINITY</code>	The <code>lpValue</code> 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.
<code>PROC_THREAD_ATTRIBUTE_HANDLE_LIST</code>	The <code>lpValue</code> parameter is a pointer to a list of handles to be inherited by the child

	<p>process.</p> <p>These handles must be created as inheritable handles and must not include pseudo handles such as those returned by the GetCurrentProcess or GetCurrentThread function.</p> <div style="border: 1px solid #ccc; padding: 10px; margin-top: 10px;"> <p>Note if you use this attribute, pass in a value of TRUE for the <i>bInheritHandles</i> parameter of the CreateProcess function.</p> </div>
<code>PROC_THREAD_ATTRIBUTE_IDEAL_PROCESSOR</code>	<p>The <i>lpValue</i> parameter is a pointer to a PROCESSOR_NUMBER structure that specifies the ideal processor for the new thread.</p> <p>Supported in Windows 7 and newer and Windows Server 2008 R2 and newer.</p>
<code>PROC_THREAD_ATTRIBUTE_MACHINE_TYPE</code>	<p>The <i>lpValue</i> parameter is a pointer to a WORD that specifies the machine architecture of the child process.</p> <p>Supported in Windows 11 and newer.</p> <p>The WORD pointed to by <i>lpValue</i> can be a value listed on IMAGE FILE MACHINE CONSTANTS.</p>
<code>PROC_THREAD_ATTRIBUTE_MITIGATION_POLICY</code>	<p>The <i>lpValue</i> 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.</p> <p>The specified policy overrides the policies set for the application and the system and cannot be changed after the child process starts running.</p> <p>The DWORD or DWORD64 pointed to by <i>lpValue</i> can be one or more of the values listed in the remarks.</p> <p>Supported in Windows 7 and newer and Windows Server 2008 R2 and newer.</p>
<code>PROC_THREAD_ATTRIBUTE_PARENT_PROCESS</code>	<p>The <i>lpValue</i> parameter is a pointer to the handle of a process to use (instead of the calling process) as the parent for the</p>

	<p>process being created. The handle for the process used must have the PROCESS_CREATE_PROCESS access right.</p> <p>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.)</p>
PROC_THREAD_ATTRIBUTE_PREFERRED_NODE	<p>The <i>lpValue</i> parameter is a pointer to the node number of the preferred NUMA node for the new process.</p> <p>Supported in Windows 7 and newer and Windows Server 2008 R2 and newer.</p>
PROC_THREAD_ATTRIBUTE_UMS_THREAD	<p>The <i>lpValue</i> 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.</p> <p>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.</p> <p>Supported in Windows 7 and newer and Windows Server 2008 R2 and newer.</p> <p>Not supported in Windows 11 and newer (see User-Mode Scheduling).</p>
PROC_THREAD_ATTRIBUTE_SECURITY_CAPABILITIES	<p>The <i>lpValue</i> 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.</p> <p>Supported in Windows 8 and newer and Windows Server 2012 and newer.</p>
PROC_THREAD_ATTRIBUTE_PROTECTION_LEVEL	<p>The <i>lpValue</i> parameter is a pointer to a DWORD value of PROTECTION_LEVEL_SAME. This specifies the protection level of the child process to</p>

	<p>be the same as the protection level of its parent process.</p>
	<p>Supported in Windows 8.1 and newer and Windows Server 2012 R2 and newer.</p>
PROC_THREAD_ATTRIBUTE_CHILD_PROCESS_POLICY	<p>The <i>lpValue</i> 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.</p> <p>For information on the possible values for the DWORD to which <i>lpValue</i> points, see Remarks.</p> <p>Supported in Windows 10 and newer and Windows Server 2016 and newer.</p>
PROC_THREAD_ATTRIBUTE_DESKTOP_APP_POLICY	<p>This attribute is relevant only to win32 applications that have been converted to UWP packages by using the Desktop Bridge.</p> <p>The <i>lpValue</i> 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.</p> <p>For information about the possible values for the DWORD to which <i>lpValue</i> points, see Remarks.</p> <p>Supported in Windows 10 Version 1703 and newer and Windows Server Version 1709 and newer.</p>
PROC_THREAD_ATTRIBUTE_JOB_LIST	<p>The <i>lpValue</i> parameter is a pointer to a list of job handles to be assigned to the child process, in the order specified.</p> <p>Supported in Windows 10 and newer and Windows Server 2016 and newer.</p>
PROC_THREAD_ATTRIBUTE_ENABLE_OPTIONAL_XSTATE_FEATURES	<p>The <i>lpValue</i> parameter is a pointer to a DWORD64 value that specifies the set of optional XState features to enable for the new thread.</p> <p>Supported in Windows 11 and newer and Windows Server 2022 and newer.</p>

[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_ASRL_ALWAYS_ON (0x00000001 << 16)

PROCESS_CREATION_MITIGATION_POLICY_BOTTOM_UP_ASRL_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_ASRL_ALWAYS_ON (0x00000001 << 20)

PROCESS_CREATION_MITIGATION_POLICY_HIGH_ENTROPY_ASRL_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](#)