CreateThread function

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 WINAPI CreateThread(
  _In_opt_  LPSECURITY_ATTRIBUTES  lpThreadAttributes,
  _In_      SIZE_T                 dwStackSize,
  _In_      LPTHREAD_START_ROUTINE lpStartAddress,
  _In_opt_  LPVOID                 lpParameter,
  _In_      DWORD                  dwCreationFlags,
  _Out_opt_ LPDWORD                lpThreadId
);
```

Parameters

lpThreadAttributes [in, optional]

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.

dwStackSize [in]

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.

lpStartAddress [in]

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.

lpParameter [in, optional]

A pointer to a variable to be passed to the thread.

dwCreationFlags [in]

The flags that control the creation of the thread.

Value	Meaning
0	The thread runs immediately after creation.
CREATE_SUSPENDED 0x00000004	The thread is created in a suspended state, and does not run until the ResumeThread function is called.
STACK_SIZE_PARAM_IS_A_RESERVATION 0x00010000	The *dwStackSize* parameter specifies the initial reserve size of the stack. If this flag is not specified, *dwStackSize* specifies the commit size.

lpThreadId [out, optional]

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 can create at most 2,048 threads. 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

Minimum supported client	Windows XP [desktop apps only]
Minimum supported server	Windows Server 2003 [desktop apps only]

Minimum supported phone	Windows Phone 8.1
Header	WinBase.h on Windows XP, Windows Server 2003, Windows Vista, Windows 7, Windows Server 2008, and Windows Server 2008 R2 (include Windows.h); Processthreadsapi.h on Windows 8, Windows Server 2012, and Windows Phone 8.1
Library	Kernel32.lib; WindowsPhoneCore.lib on Windows Phone 8.1
DLL	Kernel32.dll; KernelBase.dll on Windows Phone 8.1

See also

CloseHandle
CreateProcess
CreateRemoteThread
ExitProcess
ExitThread
GetExitCodeThread
GetThreadPriority
Process and Thread Functions
ResumeThread
SetThreadPriority
SECURITY_ATTRIBUTES
SuspendThread
ThreadProc
Threads

Community Additions

description of argument lpParameter is incorrect

The description of lpParameter as "A pointer to a variable to be passed to the thread" is ambiguous. The value lpParameter is itself passed to the thread function. The variable that it points to is not passed. (How could it be? It's a (void *), so CreateThread can't know what it points to).

lpParameter can be NULL or point to a byte or to a struct. It needn't be a pointer at all. It can be any value that can be cast to and from a pointer.

The description of this argument to CreateThread is worded variously in the Win32, Windows CE and other operating systems' documentation, but in every case, the wording is imprecise and/or misleading.

Brian Knittel
9/25/2014

Maximum threads per process?

This link has a nice simple explanation about why a process can't create more than around 2000 threads!!

http://blogs.msdn.com/b/oldnewthing/archive/2005/07/29/444912.aspx

BJØRN78
8/14/2011

Closing the handle before the thread procedure returns

I have heard of this being done, although this page doesn't seem to mention if it's valid to do such a thing. I've even heard of those that call CloseHandle() on the thread handle immediately after creating the thread. So what's the deal here?

[edit] Yes, you can close the thread handle immediately, the thread object will exist until all handles are closed **and** the thread has terminated. Closing the handles before the thread terminates isn't explicitely allowed here - but also not explicitely forbidden, and numerous samples make use of these "one-shot threads". (ph)

unuseddisplayname
8/16/2010

Extra Print Line

Program -

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

DWORD ThreadProc (LPVOID lpdwThreadParam );

//Global variable Shared by all threads
int nGlobalCount = 0;
//Main function which starts out each thread
int __cdecl main( int argc, char **argv)
{
int i, nThreads = 5;
DWORD dwThreadId;
//Determine the number of threads to start
if (argc > 1) {
nThreads = atoi( argv[1]);
}

//Set the global count to number of threads
nGlobalCount = nThreads;
//Start the threads
for (i=1; i<= nThreads; i++) {
//printf("i - %d\n",i);
if (CreateThread(NULL, //Choose default security
0, //Default stack size
(LPTHREAD_START_ROUTINE)&ThreadProc,
//Routine to execute
(LPVOID) &i, //Thread parameter
0, //Immediately run the thread
&dwThreadId //Thread Id
) == NULL)
{
printf("Error Creating Thread#: %d\n",i);
return(1);
}
else
{
printf("Global Thread Count: %d %d %d\n", nGlobalCount, nThreads, i);
Sleep(1000);
}
}
return 0;
}

//Thread Routine
DWORD ThreadProc (LPVOID lpdwThreadParam )
{
//Print Thread Number
printf ("Thread #: %d\n", *((int*)lpdwThreadParam));
//Reduce the count
nGlobalCount--;
//ENd of thread
return 0;
}
```

Output in VC6.0 -

```
Global Thread Count: 5 5 1
Thread #: 1
Global Thread Count: 4 5 2
Global Thread Count: 4 5 2
Thread #: 2
Global Thread Count: 3 5 3
Global Thread Count: 3 5 3
Thread #: 3
Global Thread Count: 2 5 4
Thread #: 4
```

Global Thread Count: 1 5 5
Global Thread Count: 1 5 5
Thread #: 5

Check the extra printlines..

How is it possible?

[tfl - 13 07 10] Hi - and thanks for your post. You should post questions like this to the MSDN Forums at http://forums.microsoft.com/msdn or the MSDN Newsgroups at http://www.microsoft.com/communities/newsgroups/en-us/. You are much more likely get a quicker response using the forums than through the Community Content. For specific help about:

```
Visual Studio  : http://groups.google.com/groups/dir?sel=usenet%3Dmicrosoft.public.vstudio%2C&
SQL Server     : http://groups.google.com/groups/dir?sel=usenet%3Dmicrosoft.public.sqlserver%2C&
.NET Framework : http://groups.google.com/groups/dir?sel=usenet%3Dmicrosoft.public.dotnet.framework
PowerShell     : http://groups.google.com/group/microsoft.public.windows.powershell/topics?pli=1
All Public     : http://groups.google.com/groups/dir?sel=usenet%3Dmicrosoft.public%2C&
```

Thomas Lee
7/13/2010

Windows 98

Passing NULL for the 'lpThreadId' parameter causes the function to fail.

Thomas Lee
7/13/2010

Threads and DLL -- BUG?

I encountered this strange thing in my simple program. At this point, I am not sure if this is a bug or not!

I am just posting here as it could help some one!

My application delay loads my custom (non-MFC) DLL! The "dllMain()" function prints the cause of the delay-load (Process attach, thread attach, detach......) apart from other things! The application first creates a thread which inputs from the user(using scanf)! The application then calls a DLL function which actually causes the DLL to be loaded. In this case, I found that the DLL is NOT attached to the thread because the thread was created before the DLL was actually loaded! This is in sync with what I read in MSDN!

However, I also found that when the process exits, "dllMain" is called with "threadDetach" as well - although NO attach was performed earlier!

Not sure if this is regular behaviour!

MonkeyManAtHCL
8/12/2008

_beginthread vs CreateThread: which should you use?

A quick browse of ...\VC\crt\src\tidtable.c shows that CRT (the C runtime library) keeps a per-thread data structure, pointed to from a TLS slot.

The per-thread data structure keeps thread-local copies of errno, pointers to some char buffers (eg for strerror(), asctime() etc), floating-point state and a smattering of other stuff.

This is dynamically allocated and initialised by _beginthread(), but obviously CreateThread can't do this since it knows nothing of the CRT. All CRT routines which access this per-thread data structure will lazily create it if it doesn't yet exist, however there's always the risk that this dynamic allocation may fail. This explains the comment "the CRT may terminate the process in low-memory conditions".

So if you don't plan on running out of memory (and who does!) then you can use either _beginthread() or CreateThread(). Do you feel lucky?

Austin Donnelly MSFT

10/4/2007

Creating a thread during DLL Initialization

It is not clear whether a thread can be created during a DLL initialization (DllMain process attach) routine.

The DllMain documentation says:

> *For information on best practices when writing a DLL, see*
>
> *http://www.microsoft.com/whdc/driver/kernel/DLL_bestprac.mspx*
>
> .

That document says not to call CreateThread "from within DllMain" but it also says:

> "*Creating a thread can work if you do not synchronize with other threads, but it is risky.*"

The CreateRemoteThread documentation says:

> "*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.*".

I know I read somewhere else (other than CreateRemoteThread) that if a thread is created during DLL initialization then the thread won't execute until after the DLL initialization has completed, but I can't find that now. It probably is in the Best Practices document but I don't find it now.

Simple Samples
8/14/2007