CreateThread

The **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.

```
HANDLE CreateThread(
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  SIZE_T dwStackSize,
  LPTHREAD_START_ROUTINE lpStartAddress,
  LPVOID lpParameter,
  DWORD dwCreationFlags,
  LPDWORD lpThreadId
);
```

Parameters

lpThreadAttributes
[in] 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 or impersonation token of the creator.

dwStackSize
[in] 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] Pointer to the application-defined function to be executed by the thread and represents the starting address of the thread. For more information on the thread function, see **ThreadProc**.

lpParameter
[in] Pointer to a variable to be passed to the thread.

dwCreationFlags
[in] Flags that control the creation of the thread. If the CREATE_SUSPENDED flag is specified, the thread is created in a suspended state, and will not run until the **ResumeThread** function is called. If this value is zero, the thread runs immediately after creation.

Windows Server 2003 and Windows XP: If the STACK_SIZE_PARAM_IS_A_RESERVATION flag is specified, the *dwStackSize* parameter specifies the initial reserve size of the stack. Otherwise, *dwStackSize* specifies the commit size.

lpThreadId
[out] Pointer to a variable that receives the thread identifier. If this parameter is NULL, the thread identifier is not returned.

Windows Me/98/95: This parameter may not be NULL.

Return Values

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).

Windows Me/98/95: **CreateThread** succeeds only when it is called in the context of a 32-bit program. A 32-bit DLL cannot create an additional thread when that DLL is being called by a 16-bit program.

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 2028 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, the handle can 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 impersonates a client, then calls **CreateThread** with a NULL security descriptor, the thread object created has a default security descriptor which allows access only to the impersonation token's TokenDefaultDacl owner or members. For more information, see Thread Security and Access Rights.

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:

Do not create a thread while impersonating another user. The call will succeed, however the newly created thread will have reduced access rights to itself when calling **GetCurrentThread**. The access rights granted are 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.

- 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 that uses functions from the static C run-time libraries should use the **beginthread** and **endthread** C run-time functions for thread management rather than **CreateThread** and **ExitThread**. Failure to do so results in small memory leaks when **ExitThread** is called. Note that this is not a problem with the C run-time in a DLL.

Example Code

For an example, see Creating Threads.

Requirements

Client: Included in Windows XP, Windows 2000 Professional, Windows NT Workstation, Windows Me, Windows 98, and Windows 95.
Server: Included in Windows Server 2003, Windows 2000 Server, and Windows NT Server.
Header: Declared in Winbase.h; include Windows.h.
Library: Use Kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, **CloseHandle**, **CreateProcess**, **CreateRemoteThread**, **ExitProcess**, **ExitThread**, **GetExitCodeThread**, **GetThreadPriority**, **ResumeThread**, **SetThreadPriority**, SECURITY_ATTRIBUTES, **ThreadProc**